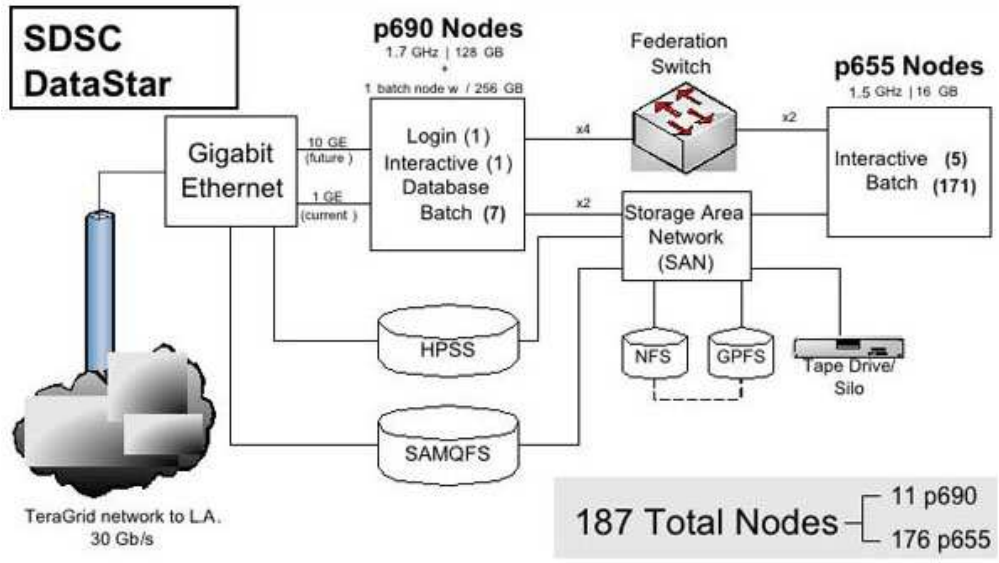



Figure 110.3: System Configuration of DataStar http://www.sdsc.edu/user_services/datastar/ (2020)

SDSC's TeraGrid cluster currently consists of 256 IBM cluster nodes, each with dual 1.5 GHz Intel Itanium 2 processors, for a peak performance of 3.1 teraflops. The nodes are equipped with four gigabytes (GBs) of physical memory per node. The cluster is running SuSE Linux and is using Myricom's Myrinet cluster interconnect network. Table 110.2 shows the technical configuration of the IA64 cluster, on which the second part of the performance study has been done.

### 110.4.3 Soil-Foundation Interaction Model

A soil-shallow-foundation interaction model as shown in Figure 110.4 has been set up to study the parallel performance. 3D brick element with 8 integration (Gaussian) points is used. The soil is modeled by Template3D elasto-plastic material model (Drucker-Prager model with Armstrong Frederick nonlinear kinematic hardening rule) and linear elasticity is assumed for the foundation. More advanced constitutive laws can be applied through Template3D model although the model used here suffices the purpose of this research to show repartitioning triggered by plastification. It is shown in this research that the speedup by adaptive load balancing is significant even for seemingly simple constitutive model. The material properties are shown in Table 110.3 and the vertical loading is applied at 5.0kN increments. The performance analysis has been carried out on DataStar supercomputer at San Diego Supercomputing Center (P655+ 8-way nodes).

Table 110.2: Technical Information of IA64 TeraGrid Cluster at SDSC

| IA-64 Cluster (tg-login.sdsc.teragrid.org) | |
|---|---|
| COMPONENT | DESCRIPTION |
| Architecture | Linux Cluster |
| Access Nodes | ⋆ quad-processor<br>⋆ ECC SDRAM memory: 8 GB<br>⋆ 2 nodes (8 processors) |
| Compute Nodes | ⋆ dual-processor<br>⋆ ECC SDRAM memory: 4 GB<br>⋆ 262 nodes (524 processors) |
| Processor | ⋆ Intel Itanium 2, 1.5 GHz<br>⋆ Integrated 6 MB L3 cache<br>⋆ Peak performance 3.1 Tflops |
| Network Interconnect | Myrinet 2000, Gigabit Ethernet, Fiber Channel |
| Disk | 1.7 TB of NFS, 50 TB of GPFS (Parallel File System) |
| Operating System | Linux 2.4-SMP (SuSE SLES 8.0) |
| Compilers | ⋆ Intel: Fortran77/90/95 C C++<br>⋆ GNU: Fortran77 C C++ |
| Batch System | Portable Batch System (PBS) with Catalina Scheduler |

Table 110.3: Material Constants for Soil-Foundation Interaction Model

| Soil | |
|---|---|
| Elastic modulus | $E$ = 17400kPa |
| Poisson ratio | $\nu$ = 0.35 |
| Friction angle | $\phi$ = 37.1° |
| Cohesion | $c$ = 0 |
| Isotropic Hardening | Linear |
| Kinematic Hardening | A/F nonlinear ($h_a$ = 116.0, $C_r$ = 80.0) |
| **Foundation** | |
| Elastic modulus | $E$ = 21GPa |
| Poisson ratio | $\nu$ = 0.2 |

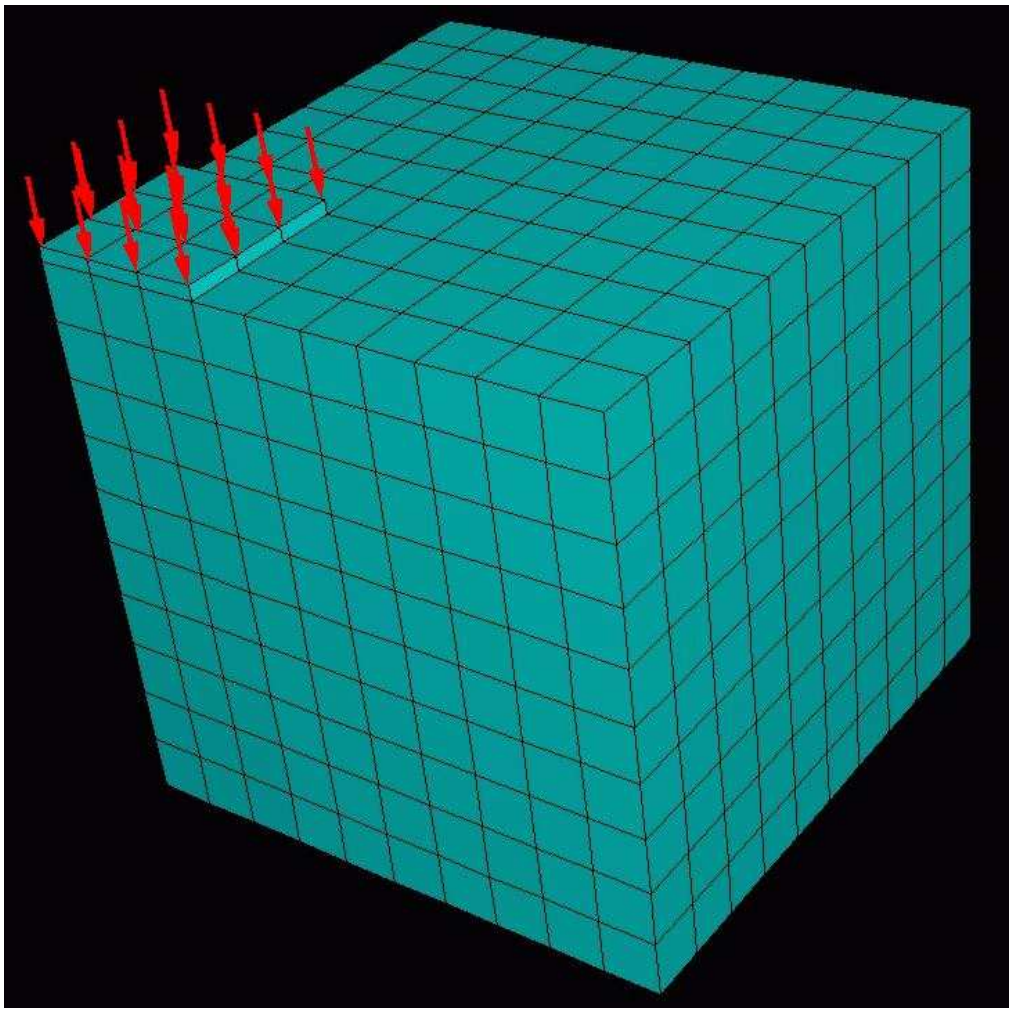


Figure 110.4: Example Finite Element Model of Soil-Foundation Interaction (Indication Only, Real Model Shown in Each Individual Section)

### 110.4.4 Numerical Study for ITR

As described in Section 110.3.3.2, the parameter ITR in ParMETIS describes the ratio between the time required for performing the inter-processor communications incurred during parallel processing compared to the time to perform the data redistribution associated with balancing the load. It acts like a switch on algorithmic approaches of ParMETIS repartitioning kernel. With ITR factor being very small, the ParMETIS tends to do that repartitioning which can minimize data redistribution cost. If the ITR factor is set to be very large, ParMETIS tends to minimize edge-cut of the final repartition.

In parallel design of PDD, if repartitioning is necessary to achieve load balance after each load increment, the whole AnalysisModel McKenna (1997) has to be wiped off thus a new analysis container

can be defined to reload subsequent analysis steps. The data redistribution cost can be much higher than communication overhead only. In order to determine an adequate ITR value for our application, preliminary study needs to be performed to investigate the effectiveness of the URA. In this research, two extreme values of the ITR ($0.001$ and $1,000,000$) are prescribed and then parallel analysis is carried out on 2, 4 and 8 processors to see how the partition/repartition algorithm behaves. Two soil-structure interaction models as shown in Figure 110.6 have been used in this parametric study. Timing data and partition figures have been collected to investigate the performance of different approaches. The one that tends to bring better performance will be adopted in subsequent parallel analysis for prototype 3D soil structure interaction problems. Figure 110.7 to Figure 110.12 shows the initial partition and final repartition figures for two different types of algorithms. With ITR factor to be very small, the URA tends to present results that minimize data redistribution cost, in which diffusive repartitioning approach is used. On the other hand, if the ITR factor is set to be very large, then the URA algorithm tends to give repartitioning with lowest edge cut but with considerably higher data redistribution cost.

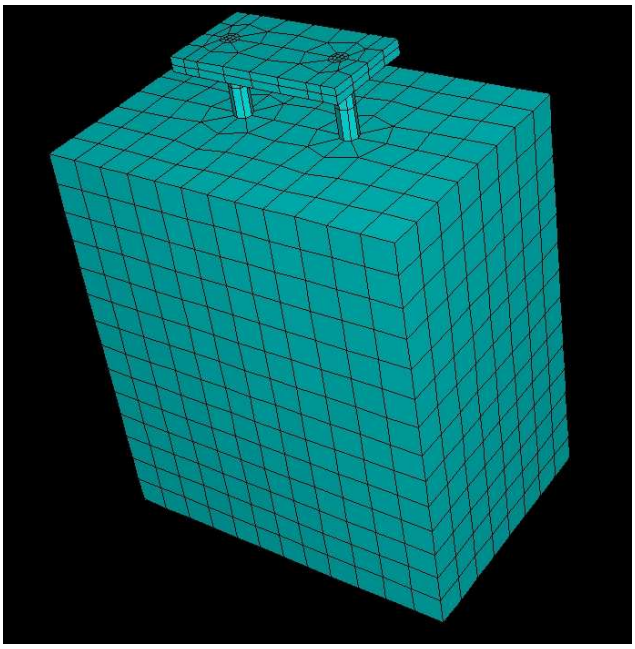


Figure 110.5: FE Models (1,968 Elements, 7,500 DOFs) for Studying Soil-Foundation Interaction Problems

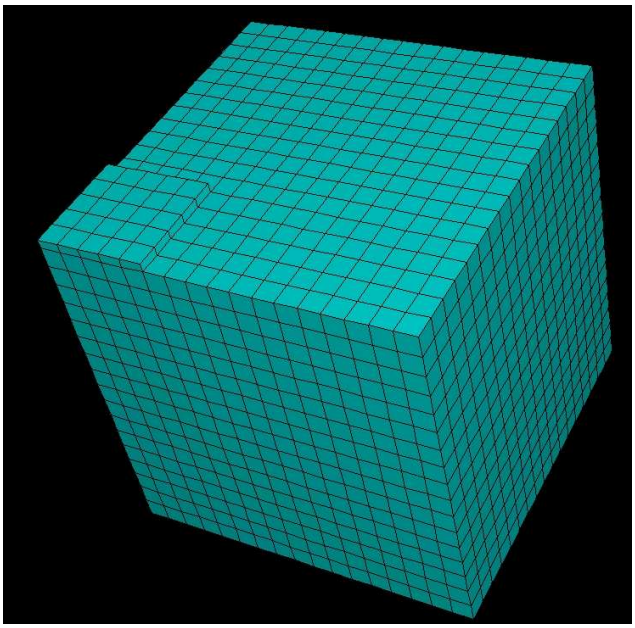


Figure 110.6: FE Models (4,938 Elements, 17,604 DOFs) for Studying Soil-Foundation Interaction Problems

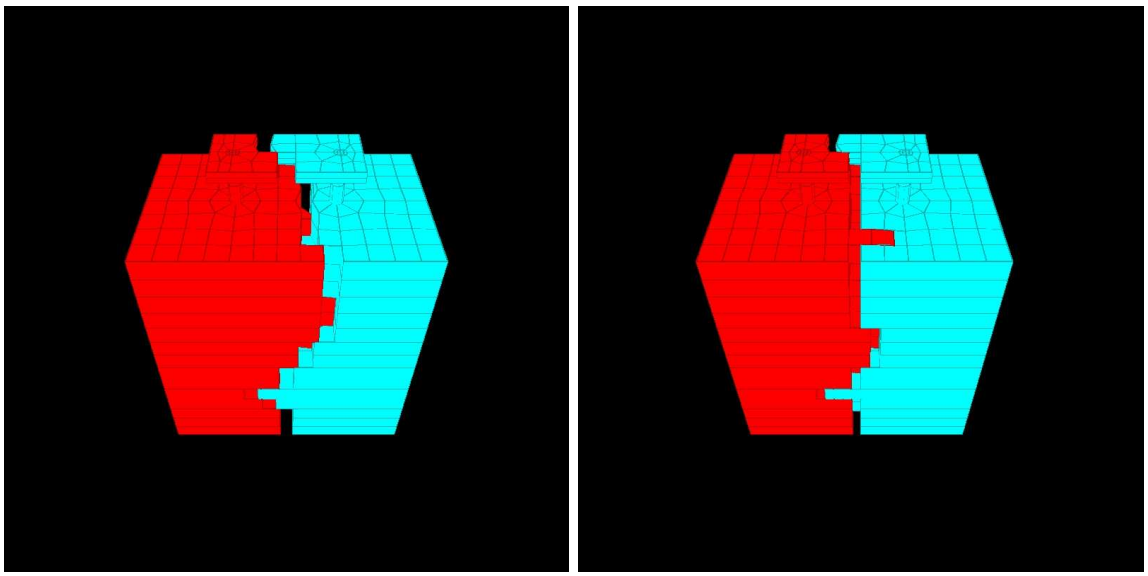


Figure 110.7: Partition and Repartition on 2 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

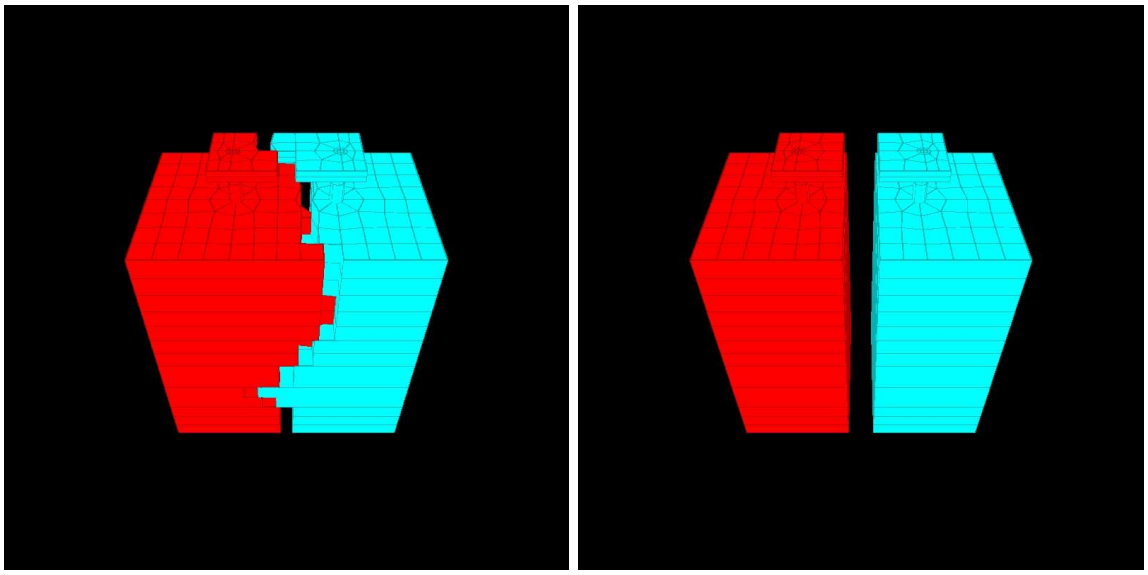


Figure 110.8: Partition and Repartition on 2 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

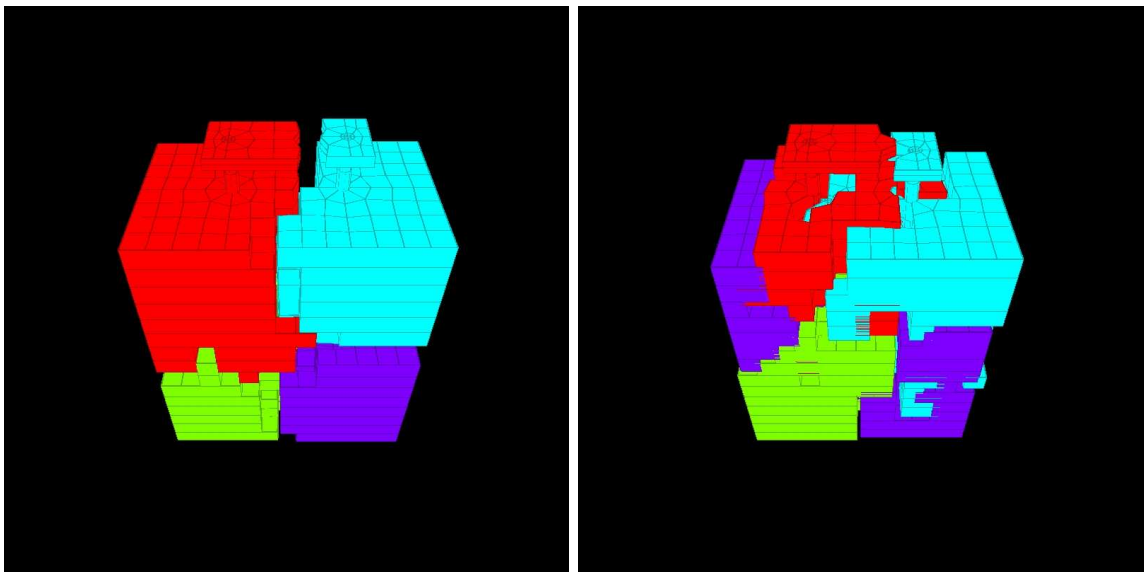


Figure 110.9: Partition and Repartition on 4 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

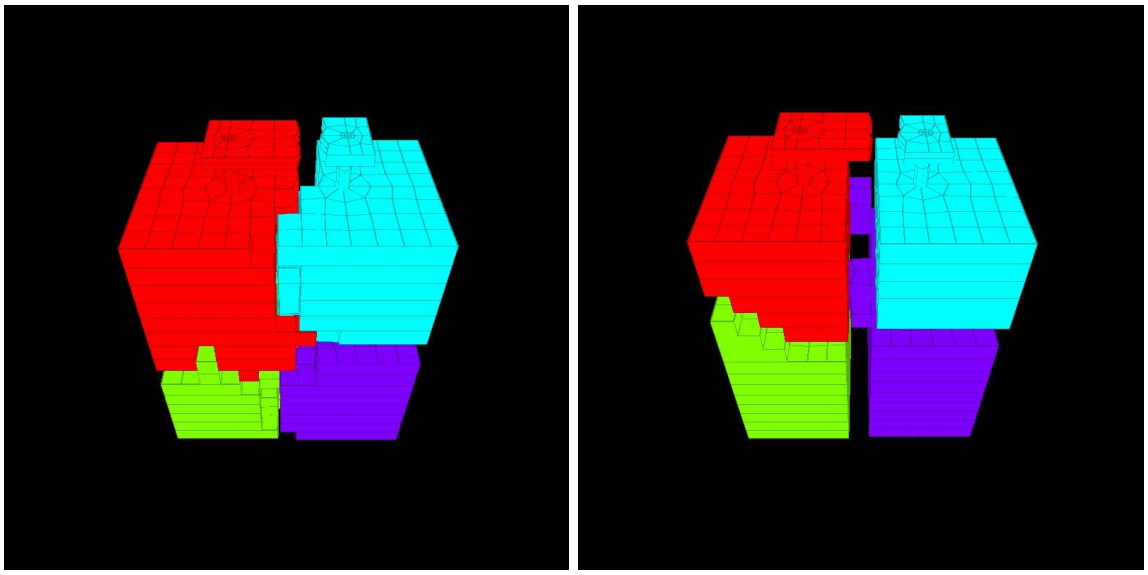


Figure 110.10: Partition and Repartition on 4 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

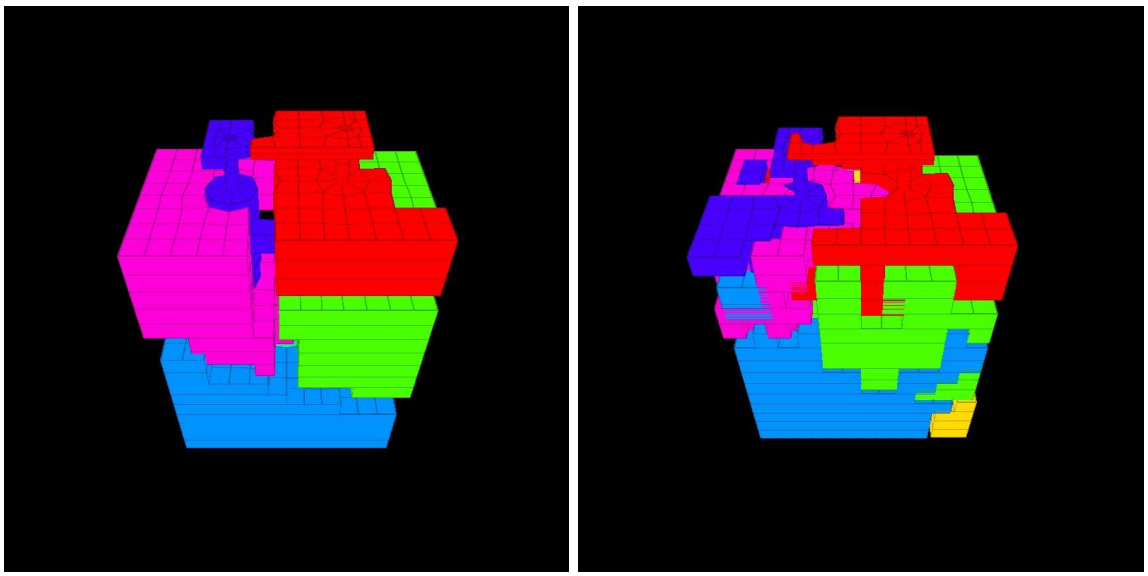


Figure 110.11: Partition and Repartition on 7 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

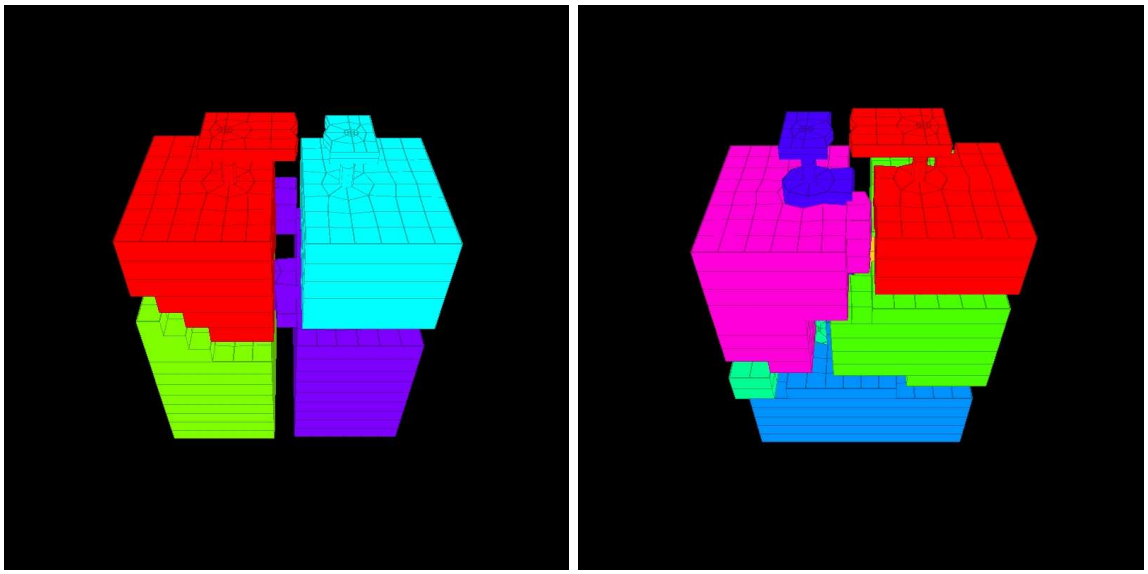


Figure 110.12: Partition and Repartition on 7 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (1,968 Elements, 7,500 DOFs)

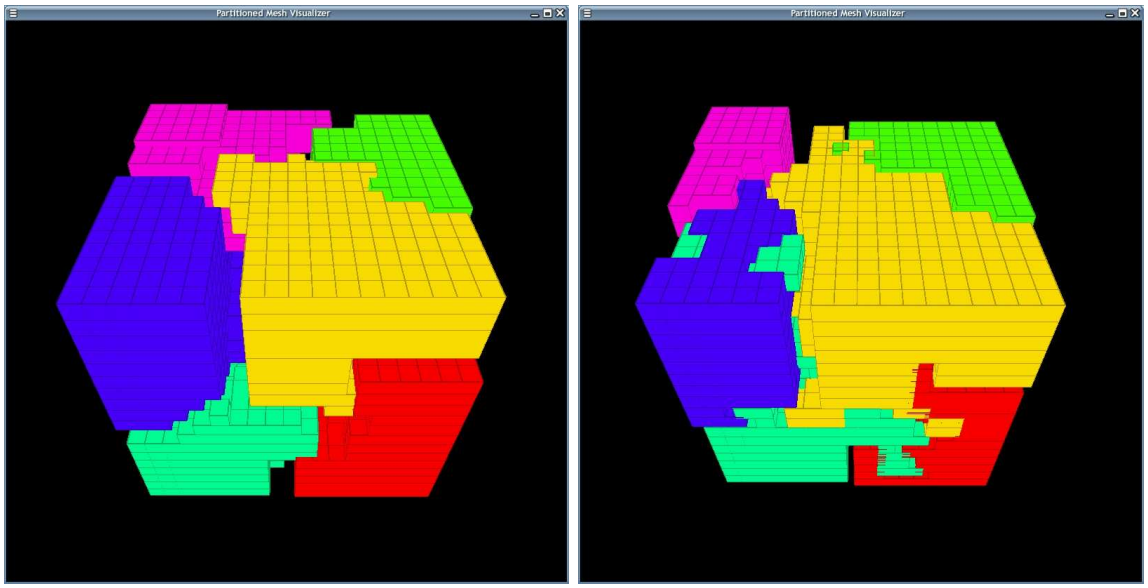



Figure 110.13: Partition and Repartition on 7 CPUs (ITR=1e-3, Imbal. Tol. 5%), FE Model (4,938 Elements, 17,604 DOFs)

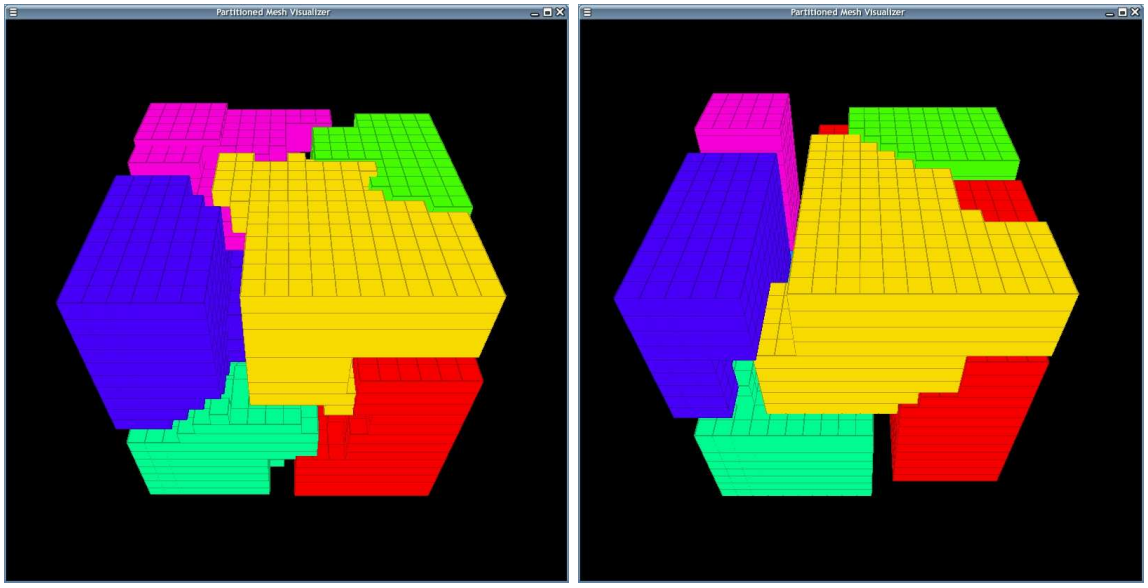



Figure 110.14: Partition and Repartition on 7 CPUs (ITR=1e6, Imbal. Tol. 5%), FE Model (4,938 Elements, 17,604 DOFs)

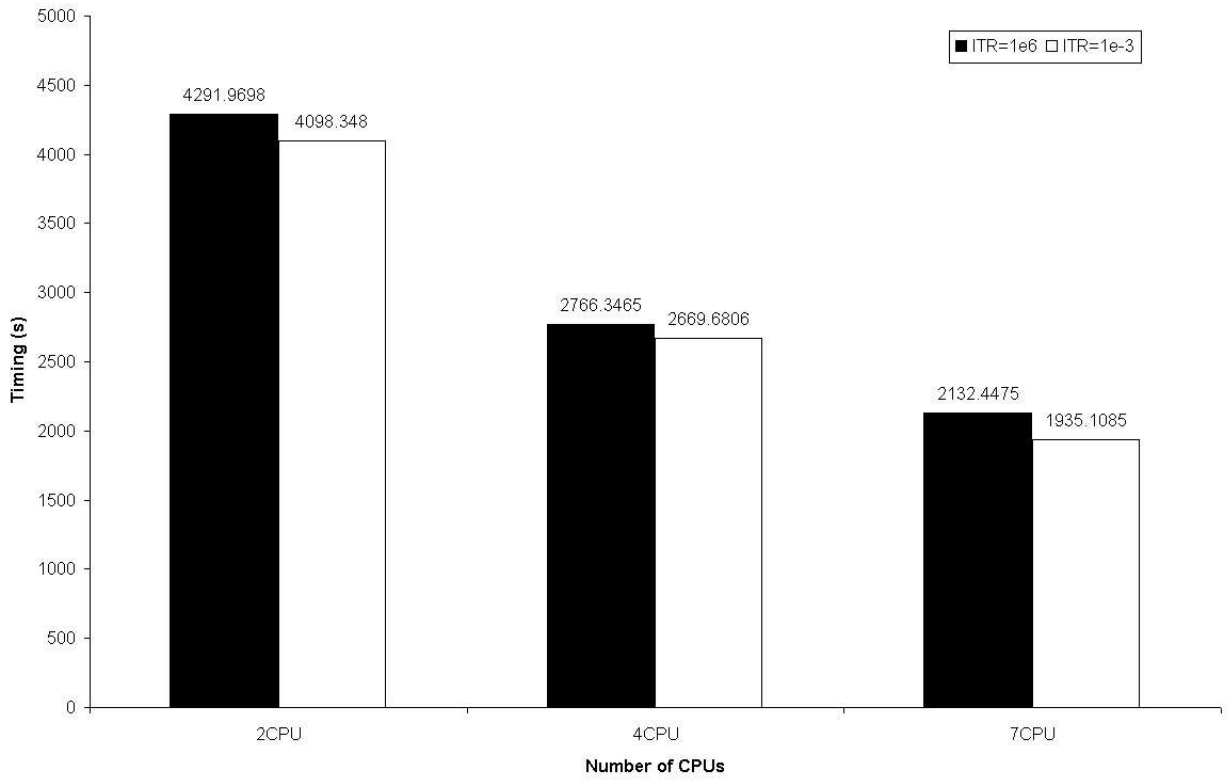



Figure 110.15: Timing Data of ITR Parametric Studies (1,968 Elements, 7,500 DOFs, Imbal. Tol. 5%)

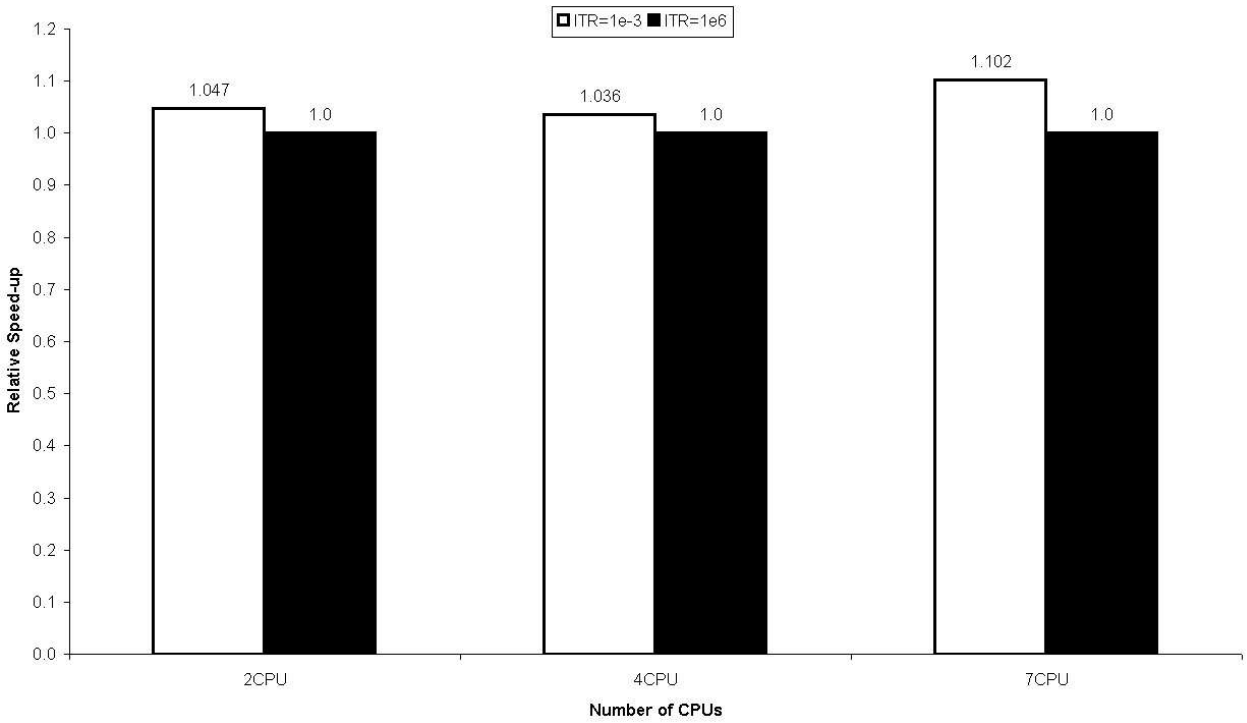



Figure 110.16: Relative Speedup of ITR=1e-3 over ITR=1e6 (1,968 Elements, 7,500 DOFs, Imbal. Tol. 5%)

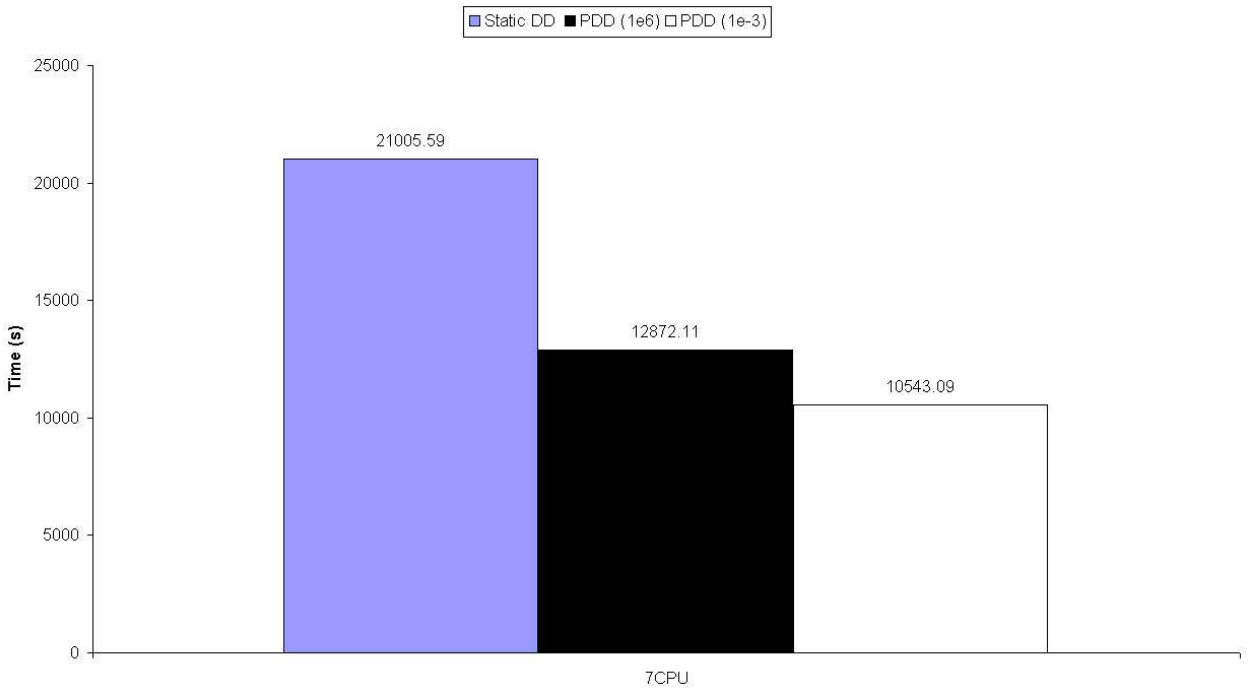



Figure 110.17: Timing Data of ITR Parametric Studies (4,938 Elements, 17,604 DOFs, Imbal. Tol. 5%)

Figures 110.15, 110.16 and 110.17 show the speedup data of parametric study on ITR factors. The purpose is to expose the the more efficient approach to do repartitioning for our specific parallel SFSI simulations, either scratch-remap approach ($ITR = 1e6$) or diffusive approach ($ITR = 1e-3$). Through the study of this chapter, some conclusions can be drawn.

1. Smaller value of ITR (1e-3) outperforms larger value (1e6). The performance gain is up to 22.1% for 7 processors. As the model gets larger, the speedup tends to get better.

2. With small ITR value, the URA algorithm tends to give results for diffusive partition/repartitioning scheme, which is good for performance for our application in overall due to the fact that the overhead associated with data redistribution in this research is very high. Diffusive approach minimizes possible data movement thus delivers better performance. The drawback is the diffusive approach typically gives very bad or even disconnected graphs with very high edge-cut as shown in Figures 110.7, 110.9 and 110.11. So careful attention must be paid to these graph structures when programing the finite element calculation. In this sense, the diffusive algorithm is not as robust as scratch/remapping. One very important observation was that repetitive repartitionings tend to yield totally ill-connected graph.

3. With large ITR value, the URA algorithm adopts the scratch/remapping scheme which inevitably introduce huge data redistribution cost. But this approach gives high quality graph and the integrity of original graph is well preserved as shown Figures 110.8, 110.10 and 110.12. This will be of great meaning for parallel finite element method based on substructure-type methods. Another important observation was, the scratch/remapping approach performed much more repartitionings than diffusive approach for same analysis. Repetitive repartitionings by scratch/remapping method tends to totally migrate all elements out of their initial partitioning and repartitioning never stops even though the computation is stabilized (in the sense of formation of plastic zones). This also explains in part why the diffusive approach can substantially outperform scratch/remapping.

4. Based on the timing analysis performed in this chapter, ITR=1e-3 is the best choice that brings substantially better performance over large ITR values. With the increase of number of processing units or the model size, the performance gain is more significant as shown in Figures 110.15, 110.16 and 110.17. Robustness of the diffusive approach has not caused much trouble in our application.

### 110.4.5 Parallel Performance Analysis

Timing routines have been implemented in PDD (MOSS and other used libraries, such as Template3DEP/NewTemplate3D to study the parallel performance. The preprocessing unit, like reading model data from file, has not

been timed so the speed up here reflects only algorithmic gain by graph partitioning. In the current phase of this research, the equation solving problem has not been addressed yet. More meaningful perspective would be to consider performance gains by simply switching from plain graph partitioning to adaptive graph partitioning, which is also the basic aim of this research. As we can see from the results below, adaptive graph partitioning improves the overall performance of elasto-plastic finite element computations. The partitioning/repartitioning overhead has been minimized by using parallel partitioner.

As stated in previous sections of this chapter, there are a couple of key parameters that control performance of the adaptive load balancing algorithm. One is the ITR factor, and the other is the computational load imbalance tolerance.

1. ITR is the key parameter which determines the algorithmic approach of the adaptive load balancing scheme. Depending on different applications and network interconnections, this value can be set to very small (0.001) or very large (up to 1,000,000) and algorithm focus will be set to minimizing data redistribution or edge-cut respectively as explained in previous sections.

2. Computational Imbalance Load Tolerance is the other key factor affecting greatly the overall performance of the whole application codes. Basically speaking, with larger finite element model, the tolerance should be set higher due to the fact that data redistribution and subsequent analysis-restarting overhead can be substantially higher as the finite element model size increases.

The performance tunings on ITR factor tend to yield consistent results as stated previously that smaller ITR (0.001) brings better performance over large ITR values. Diffusive repartitioning algorithm outperforms scratch/remapping in our application.

While on the other hand, tuning on load imbalance studies has been more illusive. The first conclusion is that load imbalance tolerance larger than 5% was not able to work robustly as the size of finite element model increases in the application study of this chapter.

Detailed parametric studies have been performed on DataStar IBM Power4 and IA64 Intel clusters to indicate the effectiveness of the proposed adaptive PDD algorithm. Models with different sizes have been tested on various number of processors to show the scalability of computational performance. All results will be compared with static one-step Domain Decomposition approach to investigate the advantage of proposed PDD algorithm in nonlinear elastic-plastic finite element calculations.

In the following sections, timing data and partition/repartition figures will be presented and results will be discussed at the end of this chapter.

#### 110.4.5.1 Soil-Foundation Model with 4,035 DOFs

The partition/repartition figures by PDD have been shown in Figure 110.18, 110.19, 110.20.

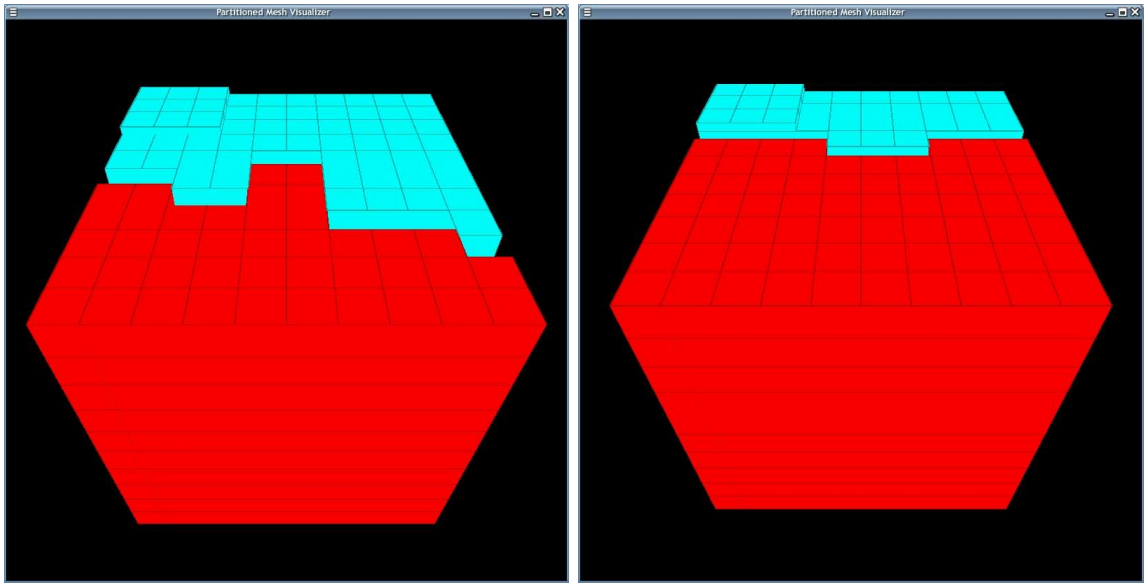



Figure 110.18: 4,035 DOFs Model, 2 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition

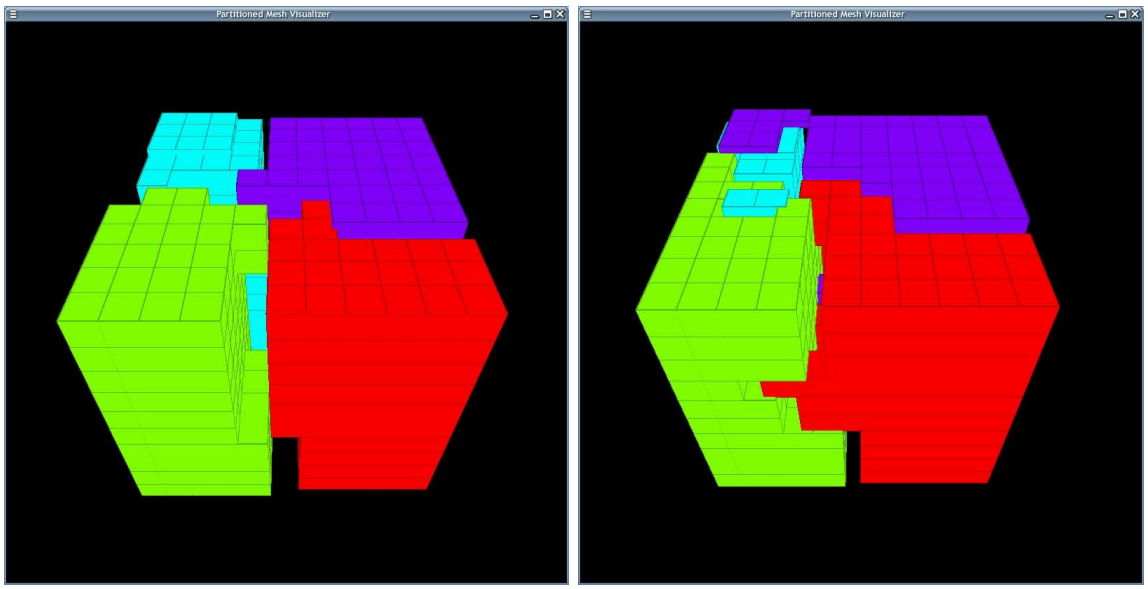



Figure 110.19: 4,035 DOFs Model, 4 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition

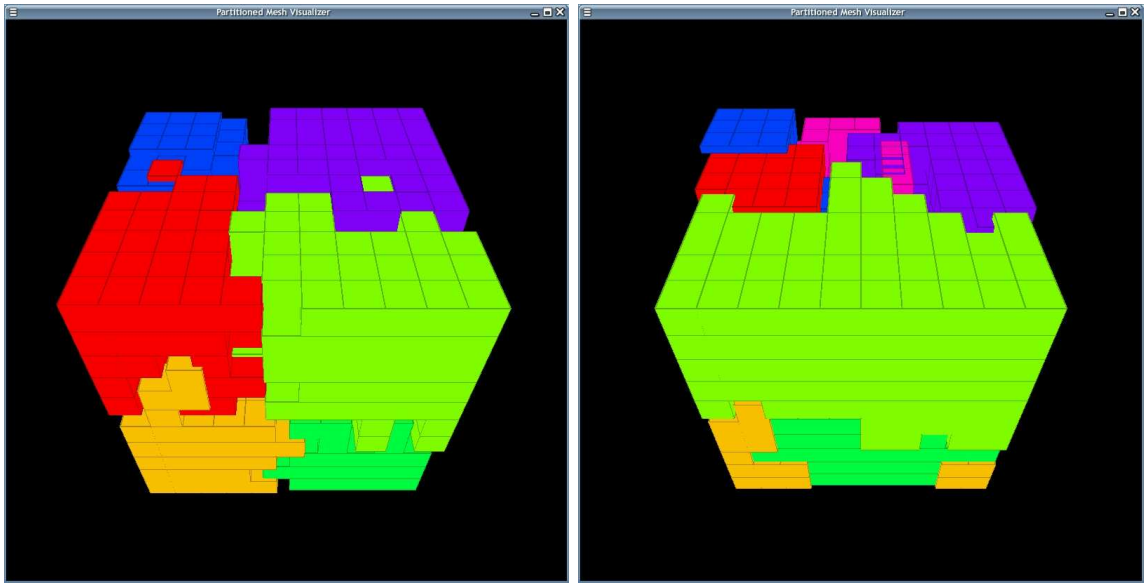



Figure 110.20: 4,035 DOFs Model, 8 CPUs, ITR=1e-3, Imbal Tol 5%, PDD Partition/Repartition

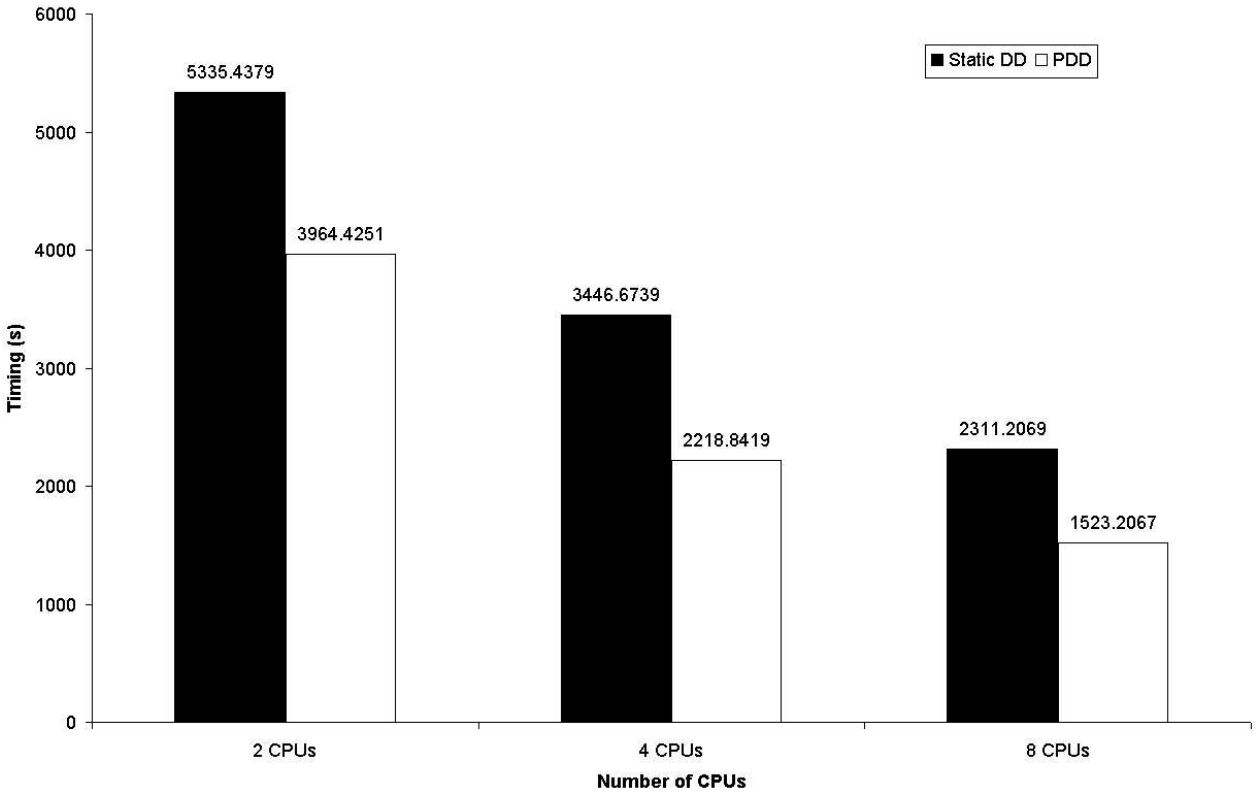



Figure 110.21: Timing Data of Parallel Runs on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5%

Table 110.4: Test Cases of Performance Studies

| | |
|---|---|
| Model Sizes (DOF) | 4,035, 17,604, 32,091, 68,451 |
| # of CPUs | 3, 5, 7, 16, 32, 64 |
| ITR Factors | 0.001, 1,000,000 |
| Imbalance Tolerance | 5%, 10%, 20% |

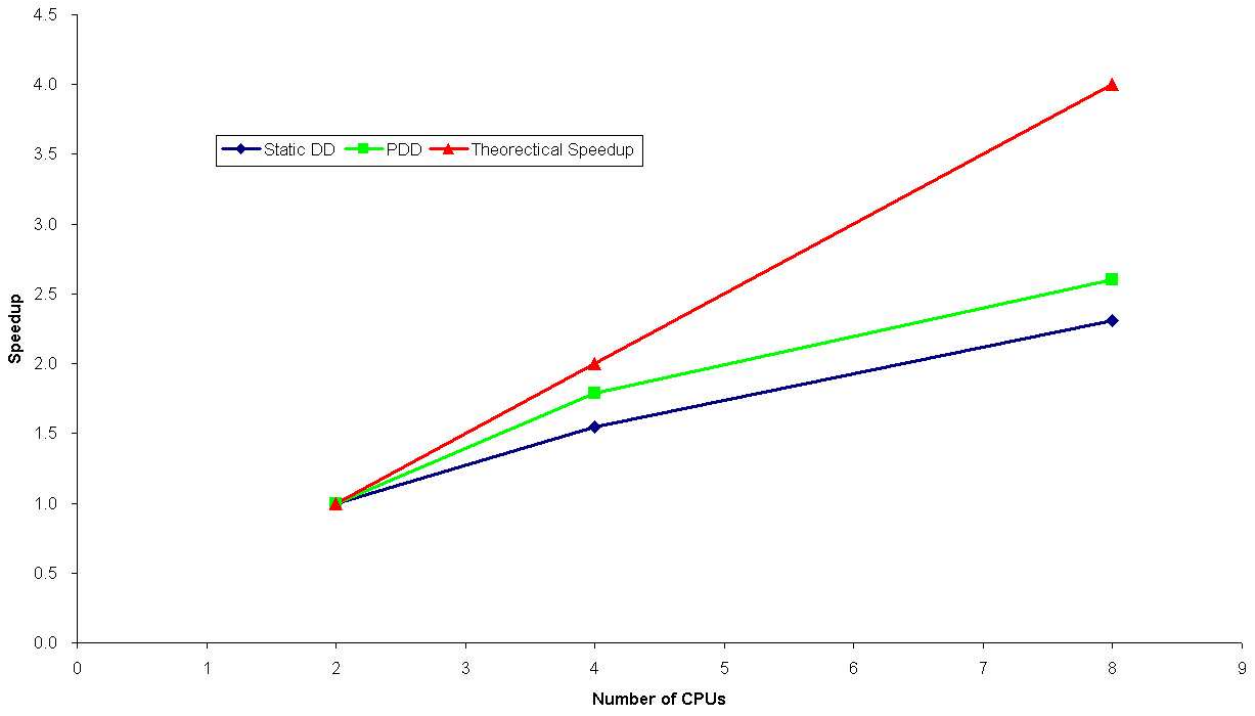



Figure 110.22: Absolute Speedup Data of Parallel Runs on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5%

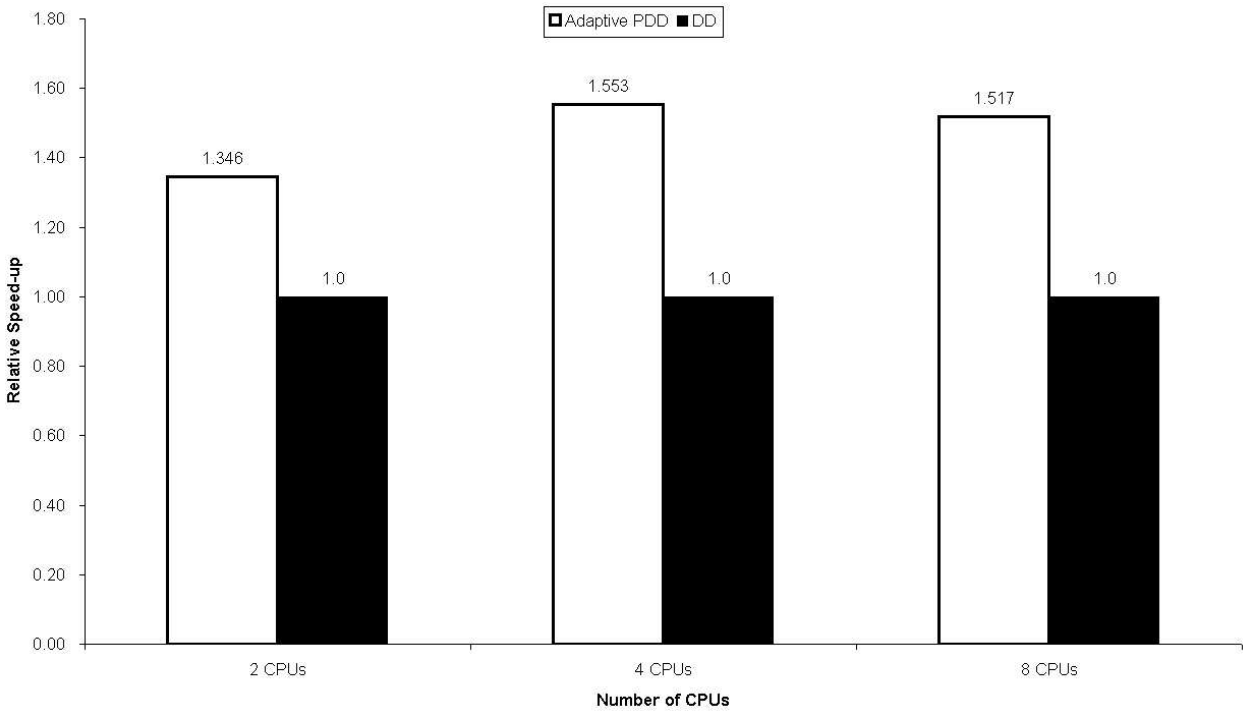



Figure 110.23: Relative Speedup of PDD over Static DD on 4,035 DOFs Model, ITR=1e-3, Imbal Tol 5%

#### 110.4.5.2 Soil-Foundation Model with 4,938 Elements, 17,604 DOFs

This is the same model as described before but with more elements as shown in 110.24. Timing data has been collected to indicate performance gains by adaptive load balancing Partition and repartition figures are shown from Figure 110.28 to 110.30. The partition/repartition figures by PDD have been

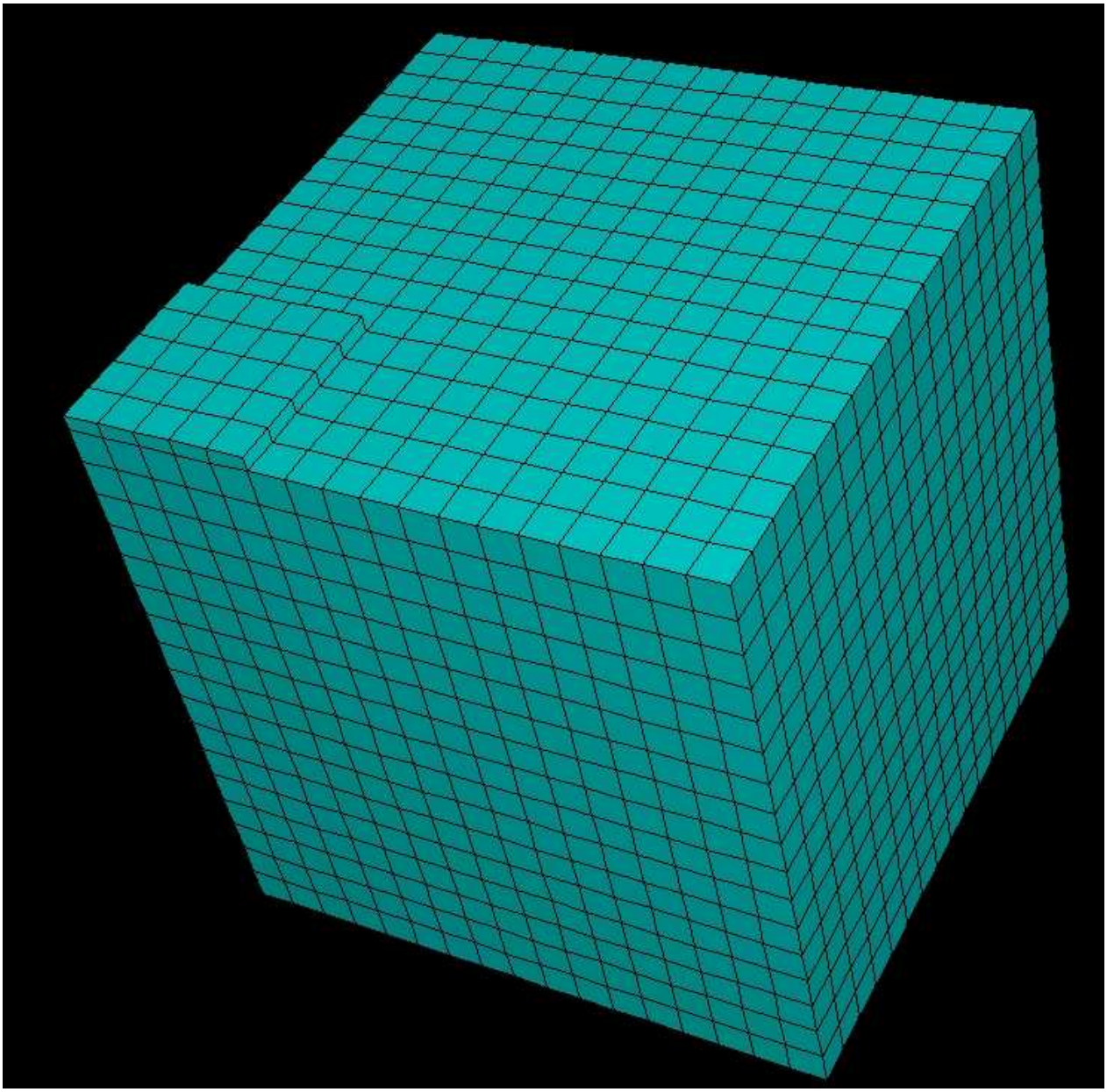


Figure 110.24: Finite Element Model of Soil-Foundation Interaction (4,938 Elements, 17,604 DOFs)

shown in Figure 110.28, 110.29, 110.30.

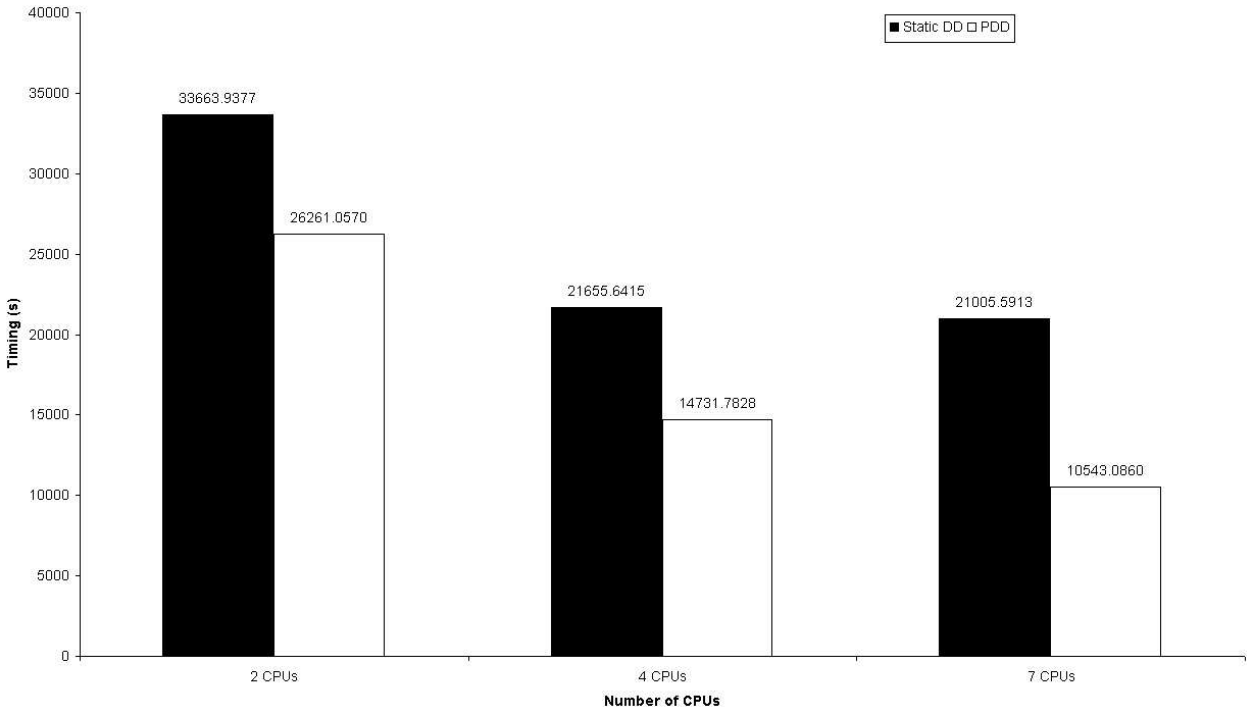



Figure 110.25: Timing Data of Parallel Runs on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

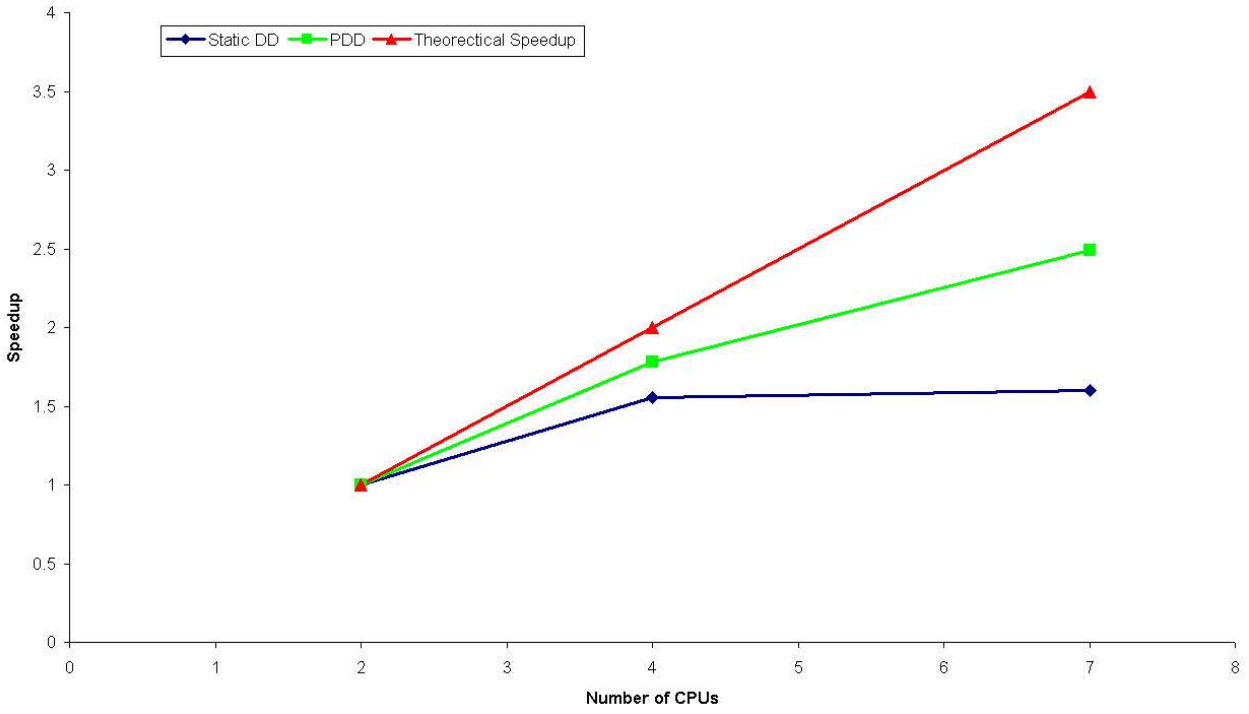



Figure 110.26: Absolute Speedup Data of Parallel Runs on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

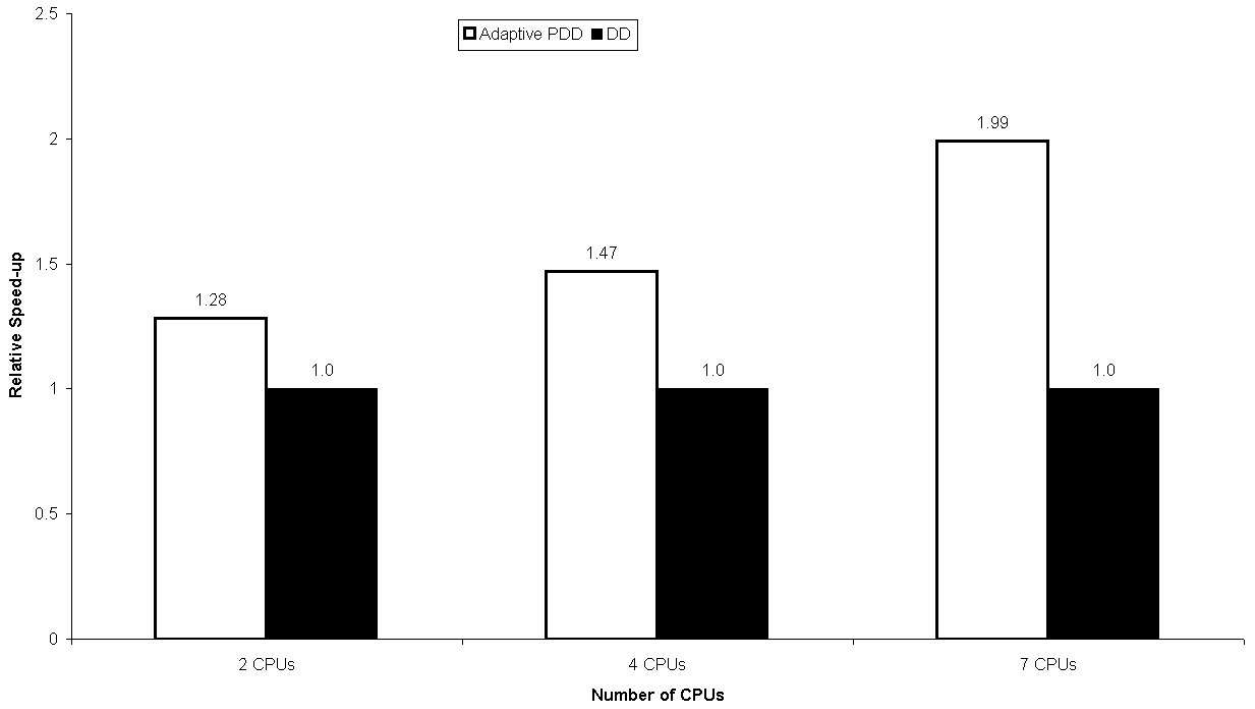



Figure 110.27: Relative Speedup of PDD over Static DD on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

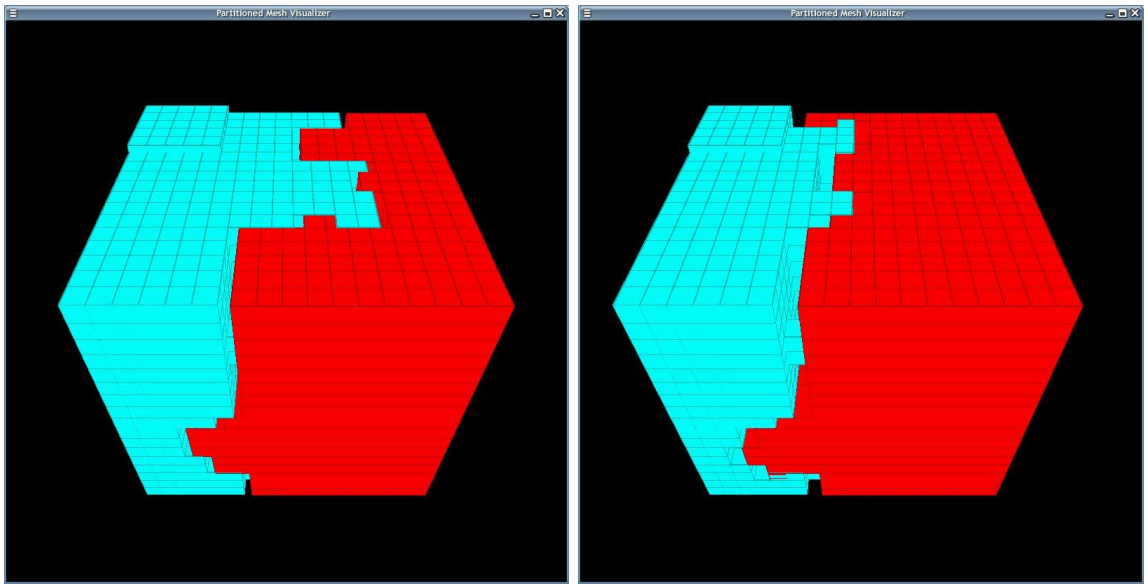



Figure 110.28: 4,938 Elements, 17,604 DOFs Model, 2 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

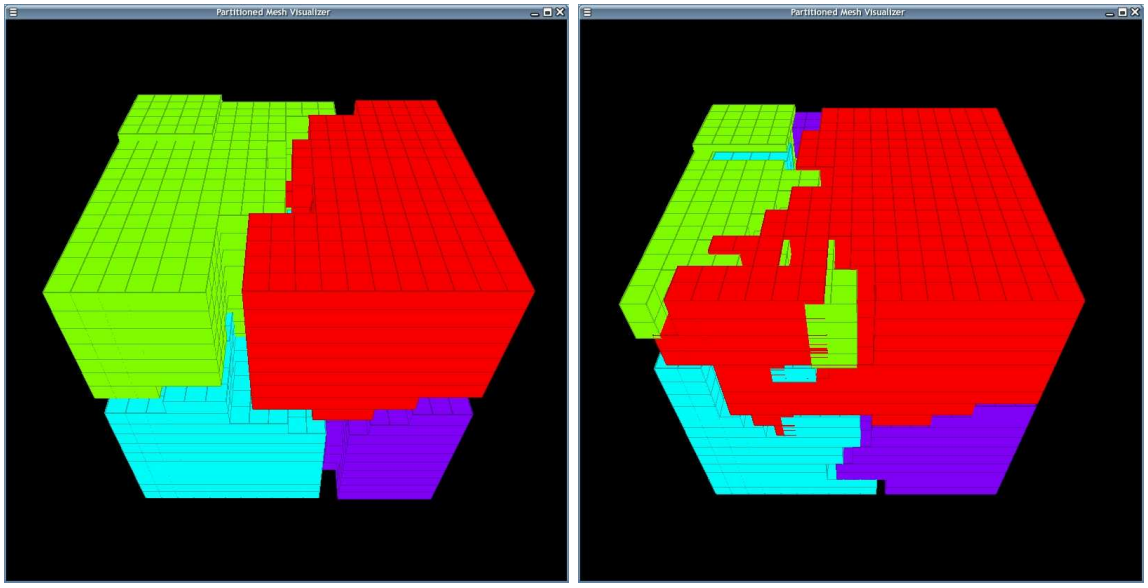



Figure 110.29: 4,938 Elements, 17,604 DOFs Model, 4 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

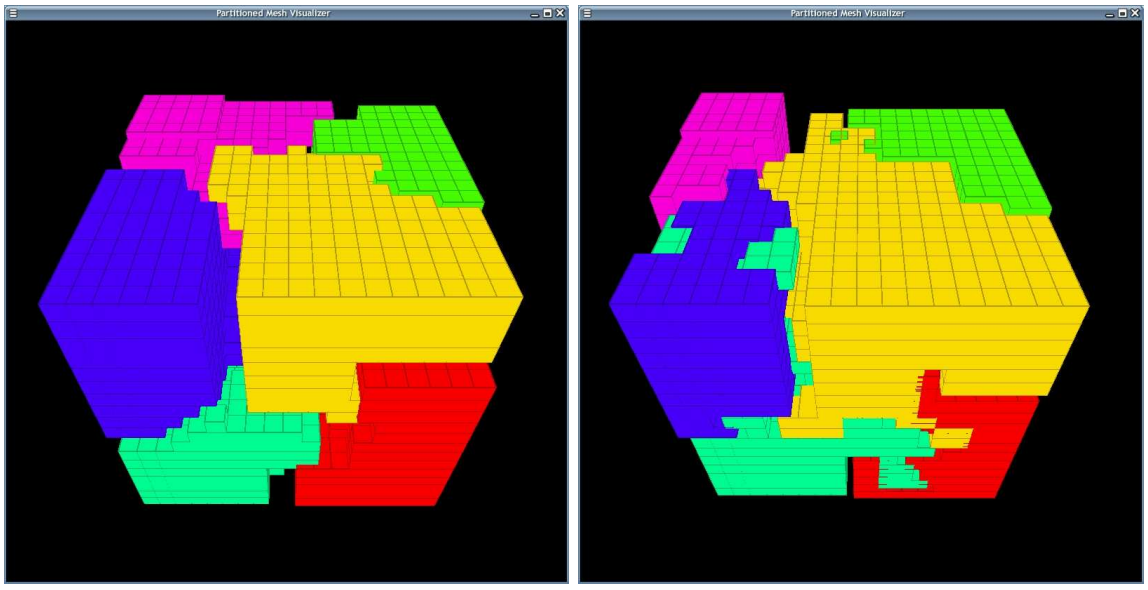



Figure 110.30: 4,938 Elements, 17,604 DOFs Model, 8 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

#### 110.4.5.3 Soil-Foundation Model with 9,297 Elements, 32,091 DOFs

The mesh is shown in Figure 110.31. Speed up results are shown from Figure 110.32 to Figure 110.34. Partition and repartition figures are shown from Figure 110.35 to Figure 110.39.

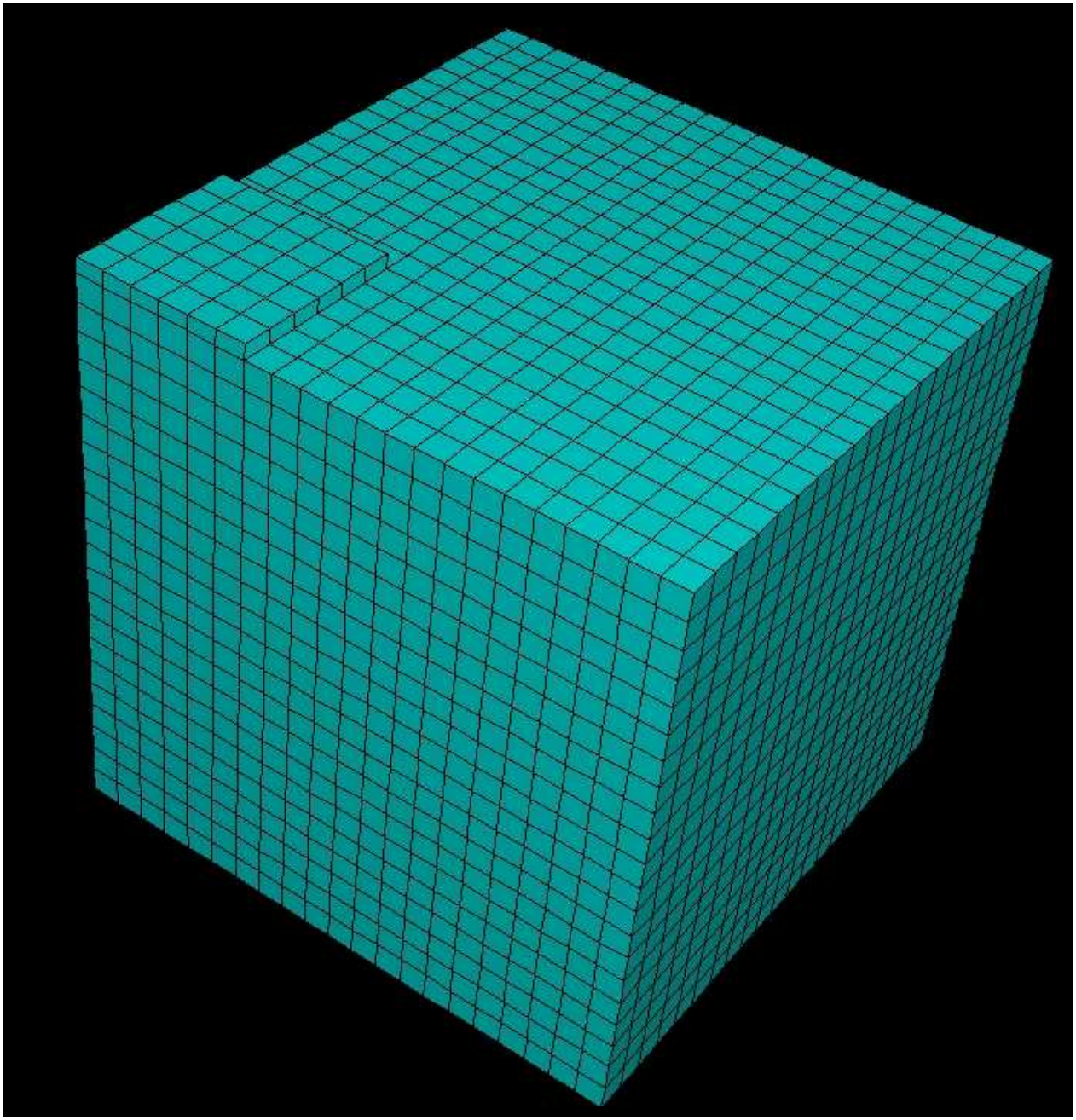


Figure 110.31: Finite Element Model of Soil-Foundation Interaction (9,297 Elements, 32,091 DOFs)

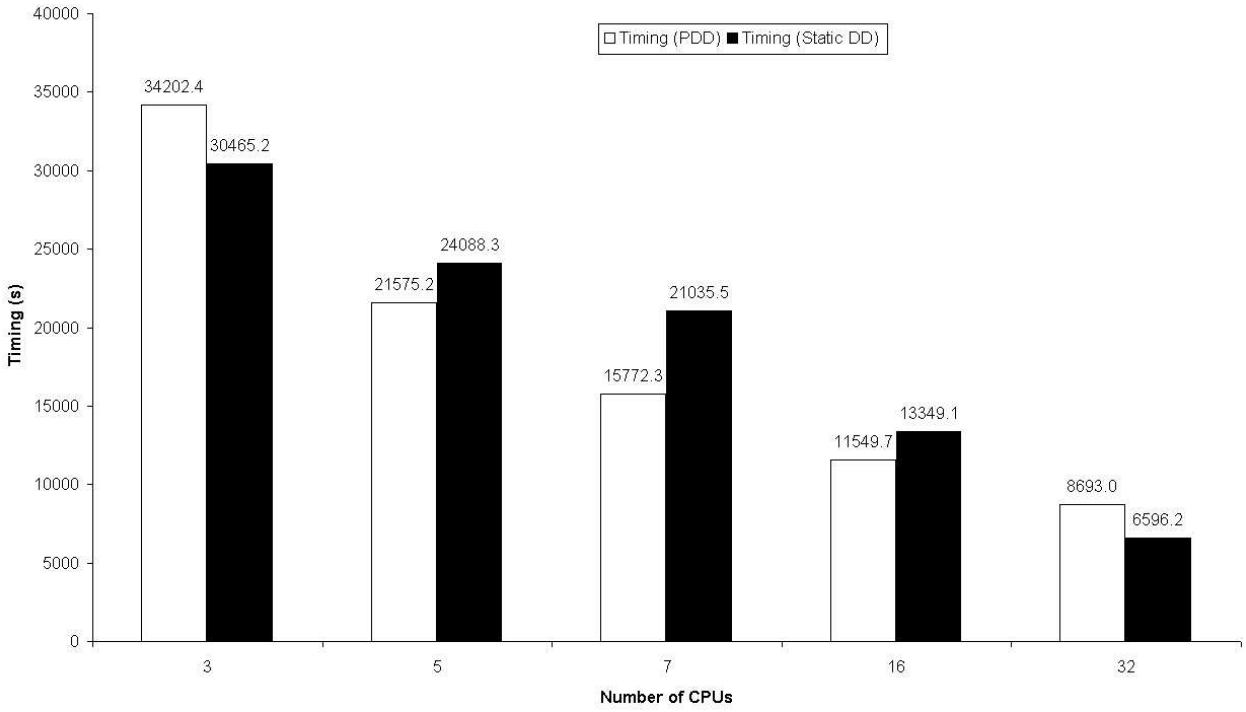



Figure 110.32: Timing Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

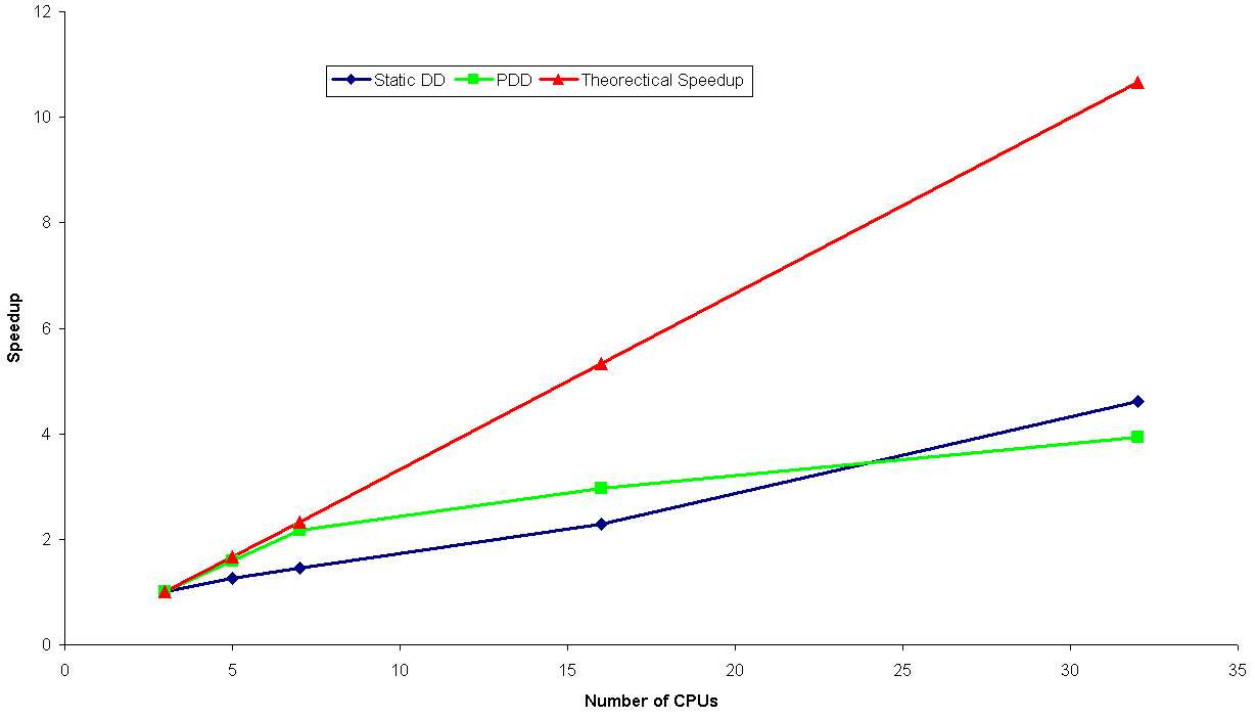



Figure 110.33: Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

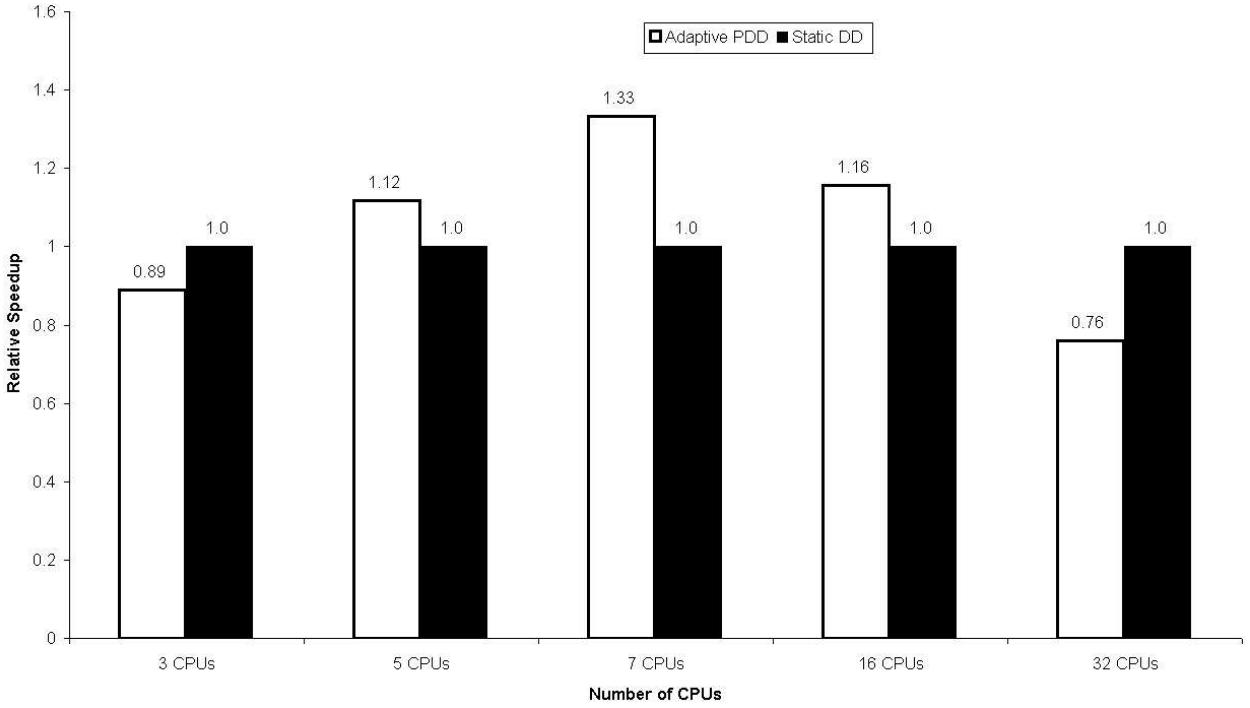



Figure 110.34: Relative Speedup of PDD over Static DD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

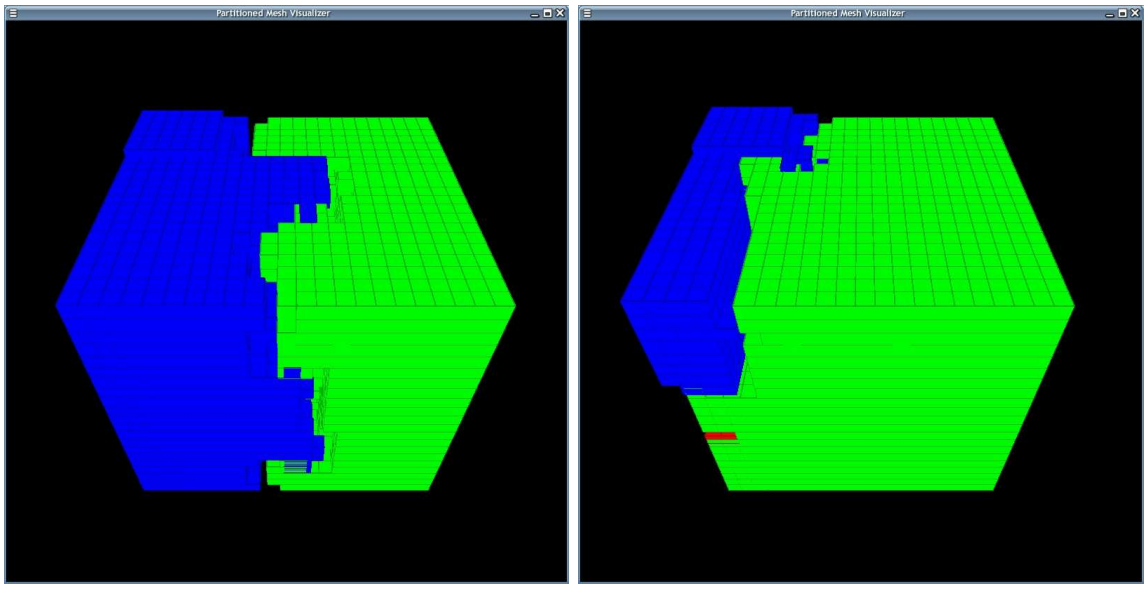



Figure 110.35: 9,297 Elements, 32,091 DOFs Model, 3 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

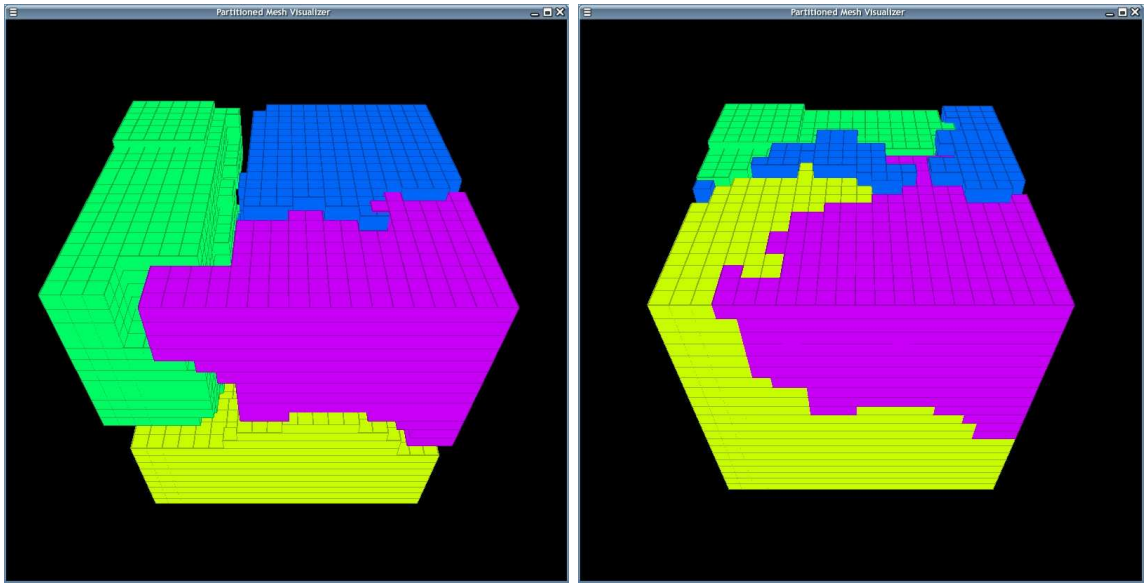



Figure 110.36: 9,297 Elements, 32,091 DOFs Model, 5 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

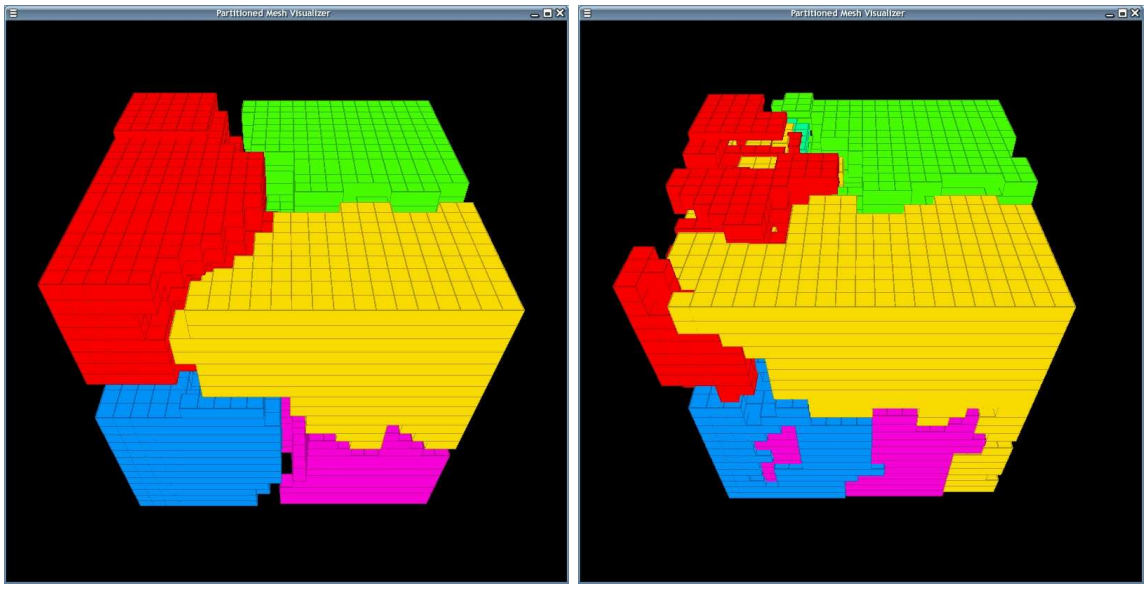



Figure 110.37: 9,297 Elements, 32,091 DOFs Model, 7 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

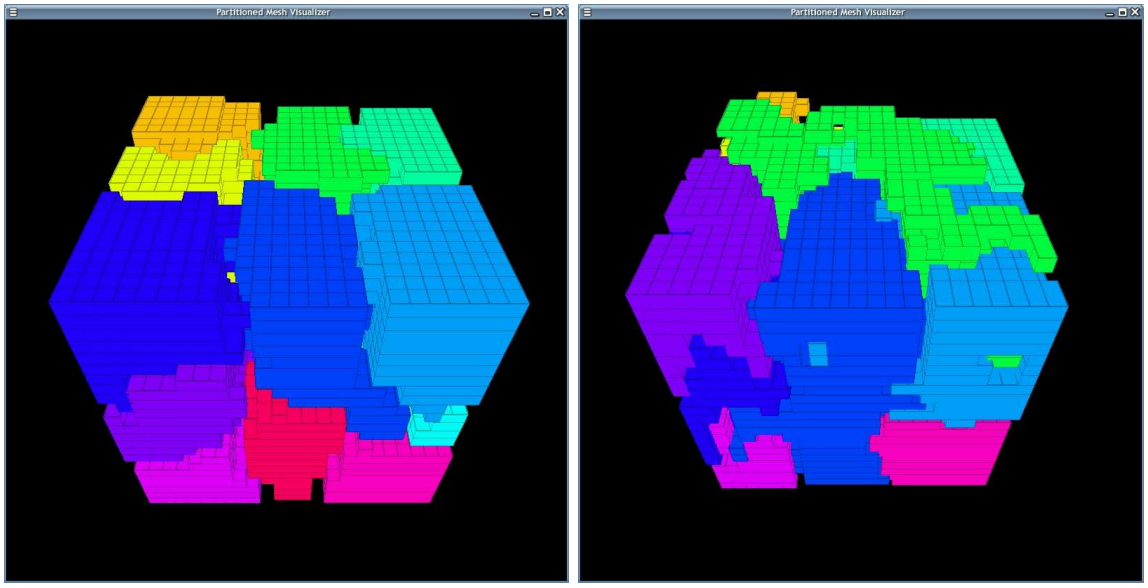



Figure 110.38: 9,297 Elements, 32,091 DOFs Model, 16 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

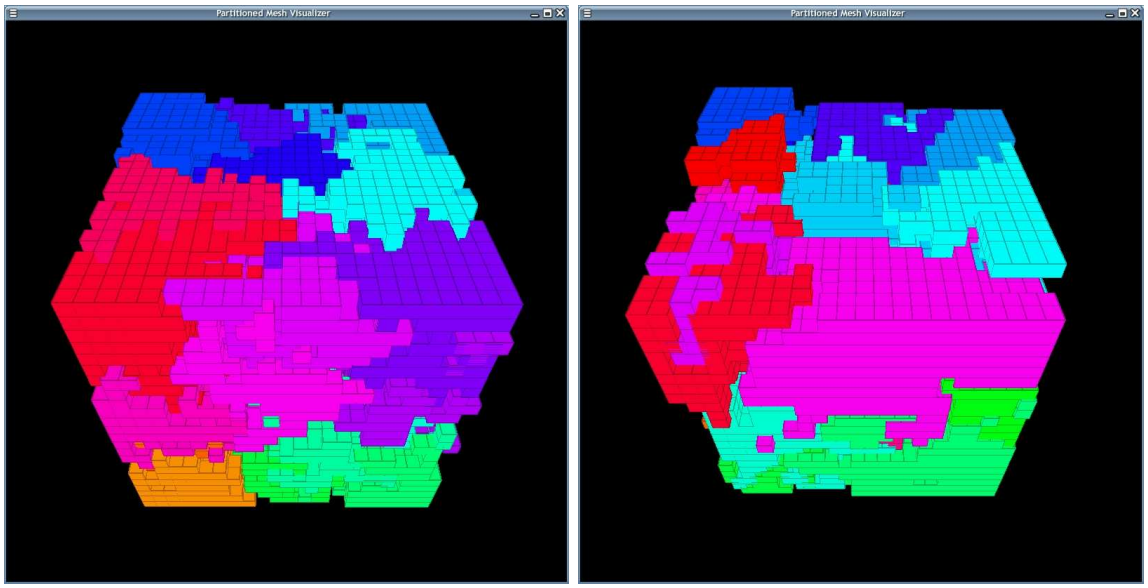



Figure 110.39: 9,297 Elements, 32,091 DOFs Model, 32 CPUs, PDD Partition/Repartition, ITR=1e-3, Imbal Tol 5%

### 110.4.6 Algorithm Fine-Tuning

From performance analysis results in previous sections, it has been shown that adaptive graph partitioning algorithm based on element graph can improve overall load balance for nonlinear elastic-plastic finite element calculations. Speed up has been observed on example problems. While on the other hand, we can also see as the model size increases, the efficiency of proposed PDD algorithm dropped sharply as shown in Figures 110.33 and 110.34.

So the naive implementation of PDD does not work as expected. With load balancing, one expects that the performance of PDD should not be worse than the DD case. It otherwise implies that the PDD does not bring performance gain that can completely offset its own extra load balancing operations-related overheads.

In this chapter, more detailed algorithm fine-tuning has been performed to address the problems we had in previous sections of the naive PDD implementation.

In order to improve the overall efficiency of proposed PDD algorithm. we have to consider two levels of costs when one wishes to balance the computational load among processing units. One is the data communication cost, and the other one is finite element model regeneration overhead associated with specific application problems.

Currently the adaptive graph partitioning algorithm does not consider the fact that the network communication patterns might differ much among processing nodes. The single $ITR$ value indicates the algorithmic approach of the graph partitioning algorithm, but the real communication performance has not been addressed in the implementation.

On the other hand, certain applications impose extra problem-dependent overhead to repartitioning operations. For example, whenever data communications happen, the finite element model has to be wiped off and regenerated. This is not inherent with the graph partitioning algorithm but still needs to be addressed in order to get the best performance. As observed in this chapter, model regeneration overhead increases when the finite element model becomes bigger.

In order to improve the overall performance of our application, we hope to consider both data communication and model regeneration cost and create a new strategy through which we can adaptively monitor the extra overheads to assure that load balancing operation can offset both costs.

This chapter will first investigate the effect of load balance tolerance on performance and then a new globally adaptive strategy will be proposed to handle both communication and model regeneration overhead. Speedup analysis have been done to show performance gains.

### 110.4.7 Fine Tuning on Load Imbalance Tolerance

If one finds out that the application-associated overhead (say, model regeneration cost) overwhelms when repartitioning happens, the most natural way to improve performance is to increase the load imbalance tolerance of the adaptive repartition routine. In this way, one hopes to increase the critical load imbalance that can trigger the balancing routine and so that the repartition counts can be reduced. As a result, model regeneration cost can do less harm to the overall performance.

This should rather viewed as a work-around and has not been effective in our application.

The tuning approach aims at improving efficiency of previous runs that failed showing speedup over static domain decomposition method. Shallow foundation model with 9,297 Elements, 32,091 DOFs has been chosen to study the effect of imbalance tolerance on parallel performance. Model setup has been the same as in previous sections.

Speedup analysis results have been shown in Figures 110.40, 110.41 and 110.42.

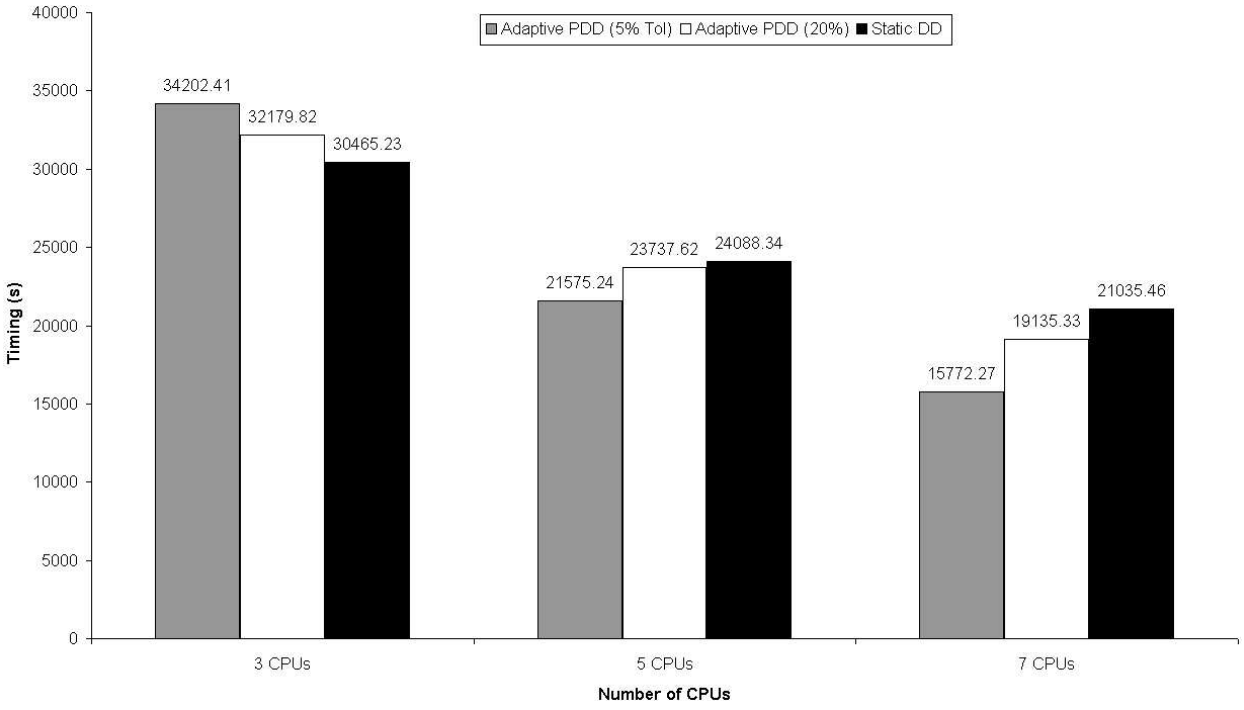



Figure 110.40: Timing Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20%

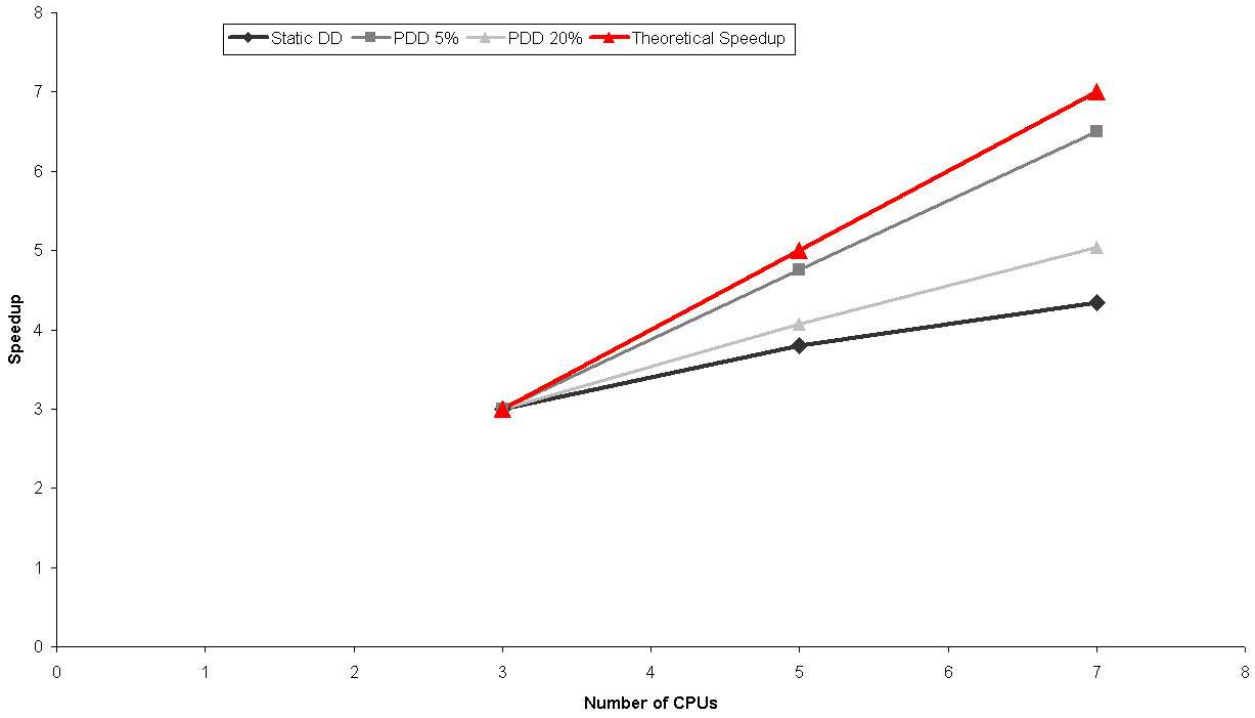



Figure 110.41: Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20%

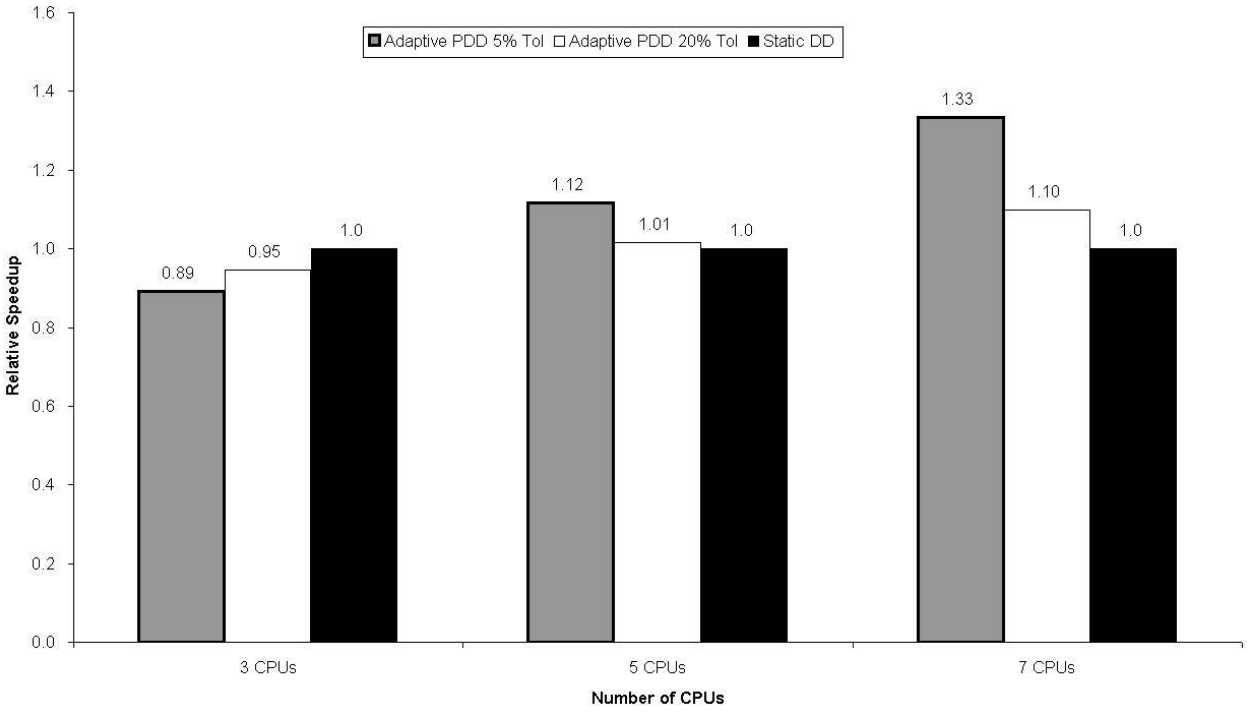



Figure 110.42: Relative Speedup of PDD over Static DD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 20%

From the performance results, we can see that increasing load imbalance tolerance does not lead to efficiency for our application. As the number of processing units increases, the whole performance of application codes deteriorates. It is also important to note that the adaptive graph partitioning/repartitioning kernel in ParMETIS has not been capable of producing adequate partitions for finite element calculations when the load imbalance tolerance is larger than the recommended 5% Karypis et al. (2003). The application crushed with 20% imbalance tolerance for same models tested in previous sections.

The conclusion reached for the application in this chapter is that load imbalance tolerance larger than 5% has not been proved more efficient. This can also be explained in more details.

In the implementation of ParMETIS, load imbalance tolerance is one of the most important parameters in the sense that this value determines whether repartition will be switched on. The other equally significant implication of this value comes from the fact that it also establishes target load imbalance residual to be achieved after adaptive load balancing. That means for each repartition, the ParMETIS will only reduce the load imbalance to the provided tolerance.

In current implementation, the load imbalance tolerance is set to be the same for both switch-on and target values, which is not capable of bringing the best performance into our application due to the fact that aside from data redistribution cost, analysis model reconstruction is equally expensive. The dilemma is described by numerical example as shown in Table 110.5.

Table 110.5: Observation on Load Imbalance Tolerance %5

| | |
|---|---|
| Model | 20,476 Elements, 68,451 DOFs |
| CPUs | 32 |
| Imbalance Before | 7.018% |
| Imbalance After | 4.9% |
| Model Regeneration | 57.2934 seconds |
| Total Step Time | 140.961 seconds |

We can easily see that tiny portion of data movement to balance out $7.018 - 4.9 = 2.228\%$ loads still invoked analysis model regeneration, which accounts for extra overhead that is about 40.6% of total step time.

Because the load balance tolerance is also the target value that the repartitioning operation hopes to achieve. The implication is that after repartitioning, the load distribution among processing units is barely under this acceptable tolerance. The performance study conducted so far showed that continuous plastification can easily creates load imbalance over this tolerance so another round of repartitioning would be launched again. It greatly brings down the performance of the whole application when the

huge data redistribution overhead is taken just to overcome a tiny imbalance. This explains why changing the tolerance was not able to bring better performance in our application.

In order to improve performance while still minimizing load imbalance, we hope to maximize the efficiency of model regeneration routine in our application. This is a two-fold statement, firstly, we don't want to blindly increase the load imbalance because it basically claims we fail our adaptive PDD algorithm by not switching on repartitioning (5% is suggested by the author of ParMETIS Karypis et al. (2003) and has been proved to be the most stable value in this chapter), secondly, with each repartitioning, we hope to achieve "perfect balance" as much as possible and in this way, the huge model regeneration cost can be offset by performance gain. What was proposed as future extension of this chapter is the idea of dual load imbalance tolerances. Load balancing triggering tolerance and the target tolerance can be defined separately. We can set higher triggering tolerance to reduce the number of repartition counts, while on the other hand a strict target tolerance can be set close to 1.0 to get better load distribution out of the balancing routine. With proposed approach, our application in this chapter will be able to fully take advantage of the repartition routines without sacrificing too much on model regenerations.

### 110.4.8 Globally Adaptive PDD Algorithm

One significant drawback of current implementation is that neither network communication nor model regeneration cost has been considered in element-graph-based type domain decomposition algorithm. Element graph only records computational load carried by each element. Only one $ITR$ factor characterizes algorithmic approach of the load balancing operation and this is apparently too crude for complicated network/hardware configurations. The ignorance of the repartitioning-associated overheads inherent with application codes can lead to serious performance drop of the proposed PDD algorithm as shown in Figure 110.43.

This drawback can harm the overall performance of the whole application code more seriously when the simulation is to be run on heterogeneous networks, which means we can have different network connections and nodes with varied computational power. The dilemma is, without exact monitoring of network communication and local model regeneration costs, we can easily sacrifice the performance gain by load balancing operations.

A second approach proposed in this chapter was the idea of modified Globally Adaptive PDD algorithm. The novelty comes from the fact that both data redistribution and analysis model regeneration costs will be monitored during execution. Load balancing will be triggered only when the performance gain necessarily offset the extra cost associated with the whole program. Domain graph structures will be kept intact till successful repartitioning happens. Meanwhile all elemental calculations will be timed to provide graph vertex weights. Data will be accumulated till algorithm restart happens, when all analysis

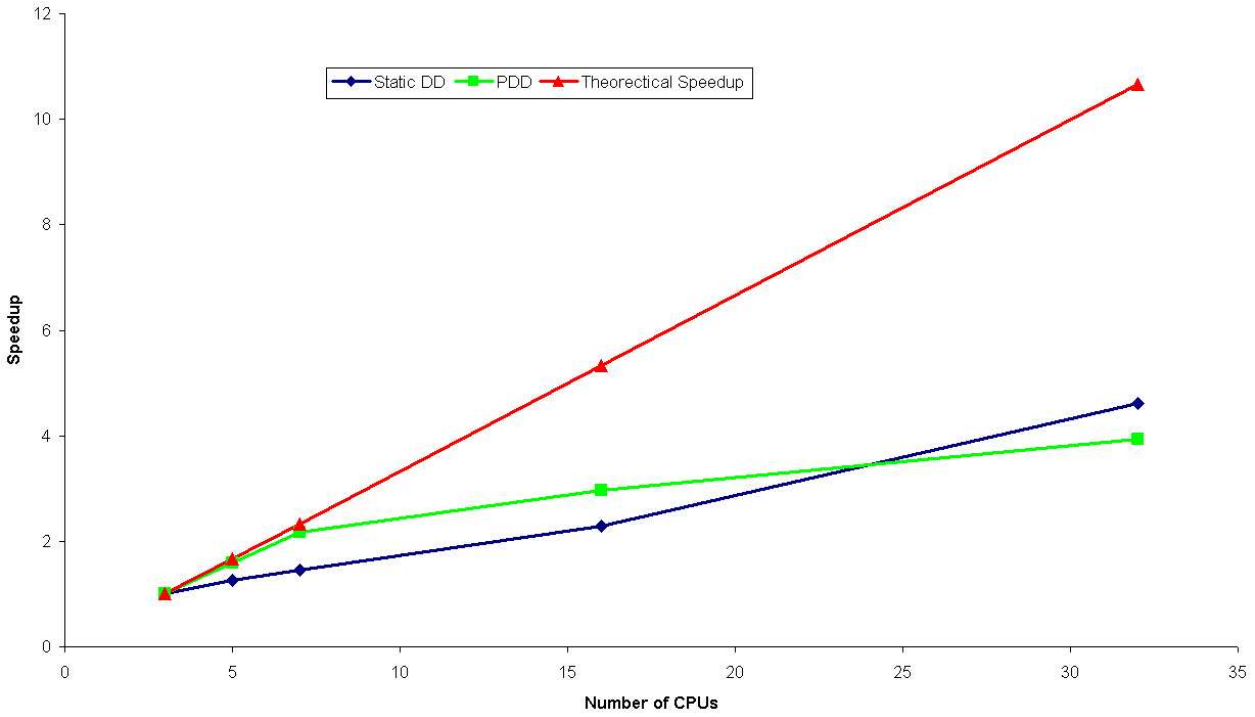



Figure 110.43: Absolute Speedup Data of Parallel Runs on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

model and vertex weights will be nullified.

This improvement aims at handling network communication and any specific application-associated overheads automatically at the global level in order to remedy the drawback that the element graph repartitioning kernel currently supported by ParMETIS is not capable of directly reflecting this application level overheads. The new strategy is to automatically monitor network communication and local model regeneration timings which will be integrated to the entry of load balancing routines to act as additional triggers of the operation along with the load imbalance tolerance.

Performance study shows that PDD algorithm with the new additions significantly improve performance even when the number of processing units is large. This modification fixes the drawback shown in previous sections that the performance of PDD was beaten by static domain decomposition when the number of processors increases.

This strategy is called to be globally adaptive because both data communication and model regeneration costs are monitored at the application level, which tells best how the real application performs on all kinds of networks. Whatever the network/hardware configurations might be, real application runs always deliver the most accurate performance counters. This information can be applied on top of graph

partitioning algorithm as a supplement to account for the drawback that the algorithm kernel is not capable of integrating global data communication costs.

#### 110.4.8.1 Implementations

We can define the global overhead associated with load balancing operation as two parts, data communication cost $T_{comm}$ and finite element model regeneration cost $T_{regen}$,

$$T_{overhead} := T_{comm} + T_{regen} \tag{110.5}$$

Performance counters have been setup to study both.

- $T_{comm}$
  Data communication patterns characterizing the network configuration can be readily measured as the program runs the initial partitioning. As described in previous sections, initial domain decomposition needs to be done to send elements over to processing nodes. This step is necessary for parallel finite element processing and it provides perfect initial estimate how the communication pattern of the application performs on specific networks. Timing routines have been added to automatically measure the communication cost. This cost is inherently changing as the network condition might vary as simulation progresses, so whenever data redistribution happens, this metric will be automatically updated to reflect the network conditions.

- $T_{regen}$
  Model regeneration cost basically comes from the fact that if data redistribution happens, the analysis model needs to be regenerated to reflect changes of nodes and elements inside the domain. Detailed operations include renumbering DOFs and rehandling constraints. This part of cost is application-dependent. In current implementation of PDD, efforts have been made to set up timing stop at the entry and exit of model regeneration routines to get the accurate data for the extra overhead. It is also important to note that model regeneration happens when the initial data distribution finishes, again the initial domain decomposition phase provides perfect initial estimate of the model regeneration cost on any specific hardware configurations.

Naturally, for the load balancing operations to pay off, the $T_{overhead}$ has to be offset by the performance gain $T_{gain}$. This chapter also creates a strategy to estimate the performance gain $T_{gain}$ even before the load balancing operation happens and this metric provides global control on top of the existing graph repartitioning algorithm.

As implemented in previous sections, the computational load on each element is represented by the associated vertex weight $vwgt[i]$. If the $SUM$ operation is applied on every single processing node, the

exact computational distribution among processors can be obtained as total wall clock time for each CPU as shown in Equation 110.6,

$$T_j := \sum_{i=1}^{n} vwgt[i],\ j = 1, 2, \ldots, np \tag{110.6}$$

in which $n$ is the number of elements on each processing domain and $np$ is the number of CPUs.

If we define,

$$T_{sum} := sum(T_j),\ T_{max} := max(T_j), \text{ and } T_{min} := min(T_j),\ j = 1, 2, \ldots, np \tag{110.7}$$

one always hope to minimize $T_{max}$ because in parallel processing, $T_{max}$ controls the total wall clock time. By load balancing operations, we mean to deliver evenly distributed computational loads among processors. So theoretically, the best execution time is,

$$T_{best} := T_{sum}/np, \text{ and } T_j \equiv T_{best},\ j = 1, 2, \ldots, np \tag{110.8}$$

if the perfect load balance is to be achieved.

Based on definitions above, the best performance gain $T_{gain}$ one can obtain from load balancing operations can be calculated as,

$$T_{gain} := T_{max} - T_{best} \tag{110.9}$$

Finally, the load balancing operation will be beneficial IF AND ONLY IF

$$T_{gain} \geq T_{overhead} = T_{comm} + T_{regen} \tag{110.10}$$

#### 110.4.8.2 Performance Results

The newly improved design has been compared to the old design to see the effectiveness of the globally adaptive switch of PDD algorithm.

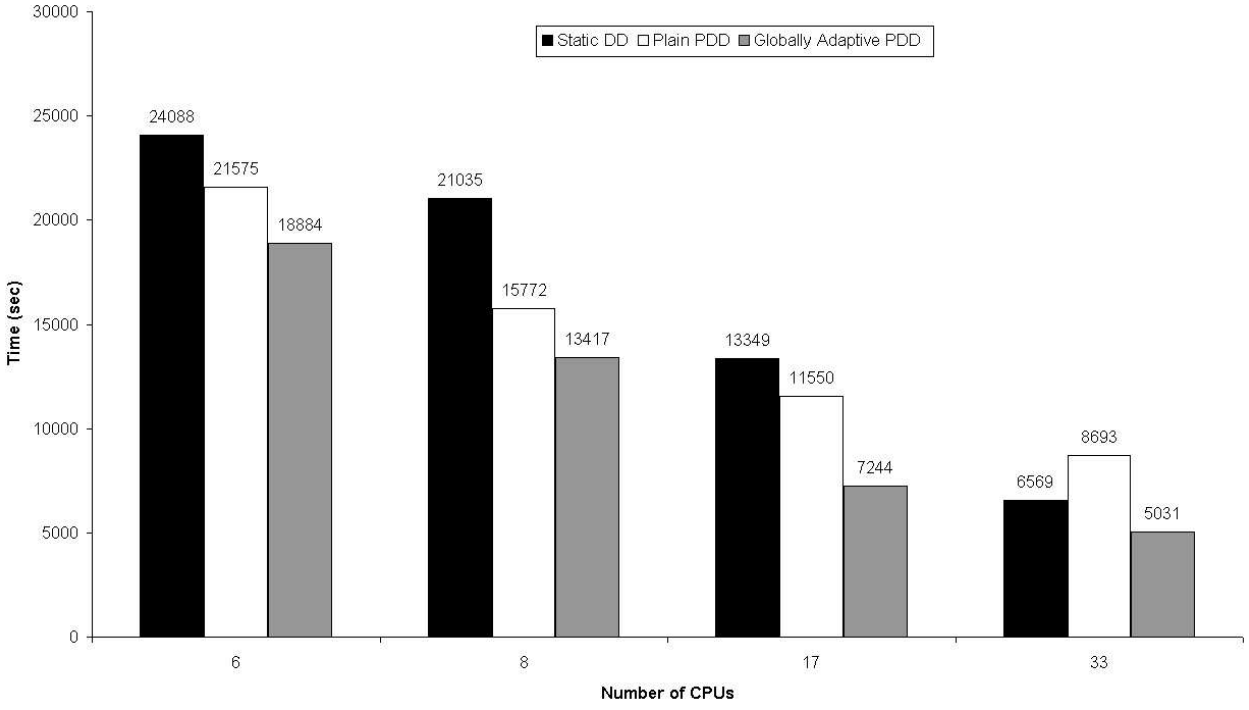



Figure 110.44: Performance of Globally Adaptive PDD on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

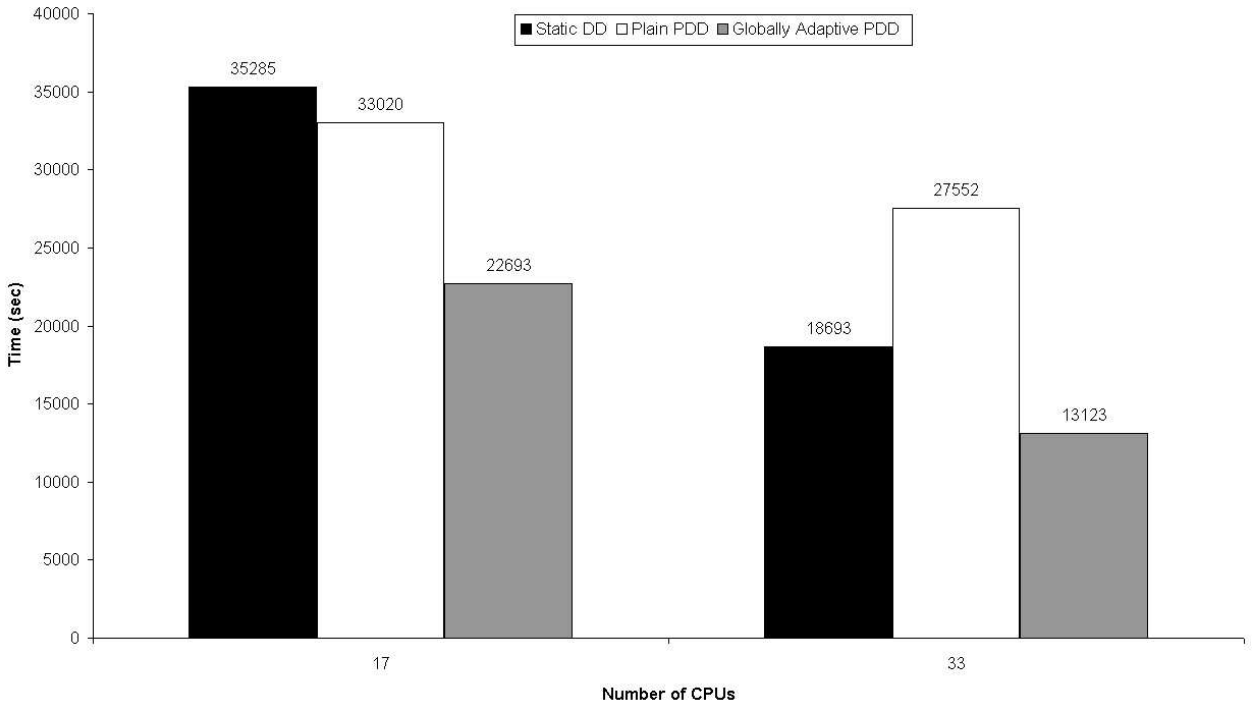



Figure 110.45: Performance of Globally Adaptive PDD on 20,476 Elements, 68,451 DOFs Model, ITR=1e-3, Imbal Tol 5%

From Figures 110.44 and 110.45, advantage of the improved globally adaptive PDD algorithm have clearly been shown. After considering the effect of both data communication and model regeneration costs, the adaptive PPD algorithm necessarily outperforms the static Domain Decomposition approach as expected. This new design also significantly improves the overall scalability of the proposed PDD algorithm as shown in Figure 110.46 and 110.47.

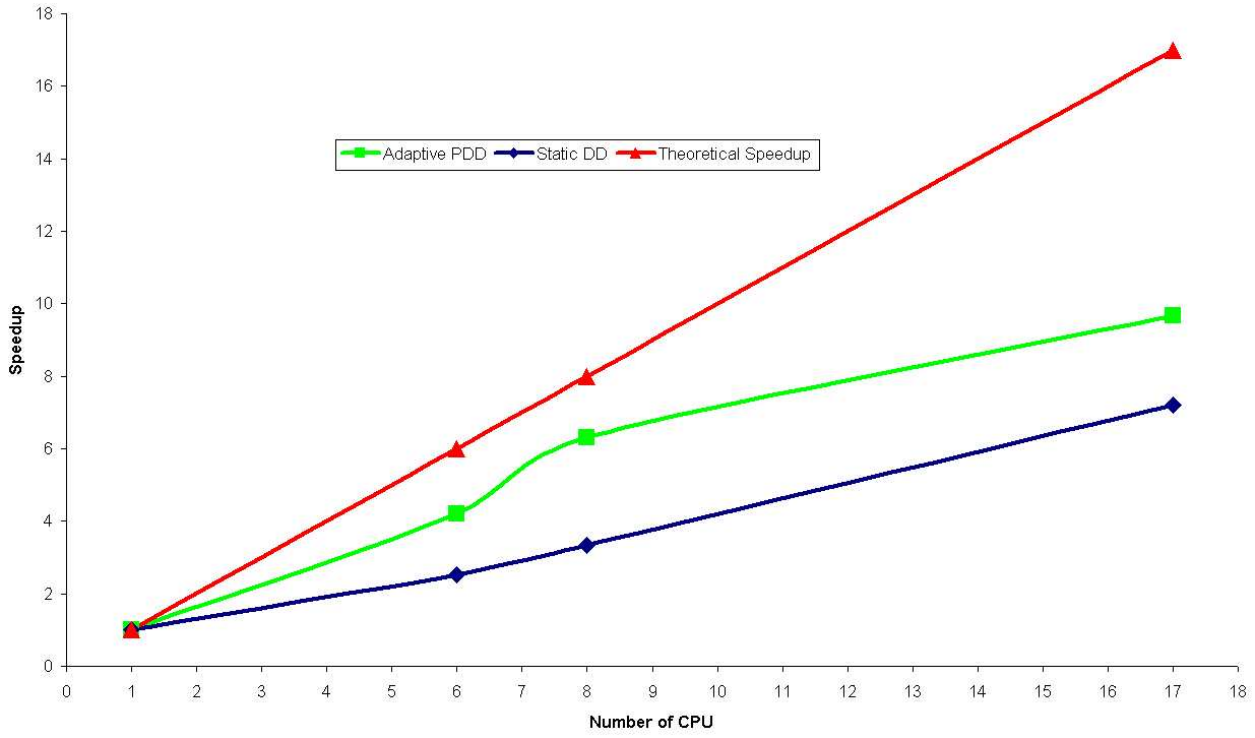



Figure 110.46: Scalability Study on 4,938 Elements, 17,604 DOFs Model, ITR=1e-3, Imbal Tol 5%

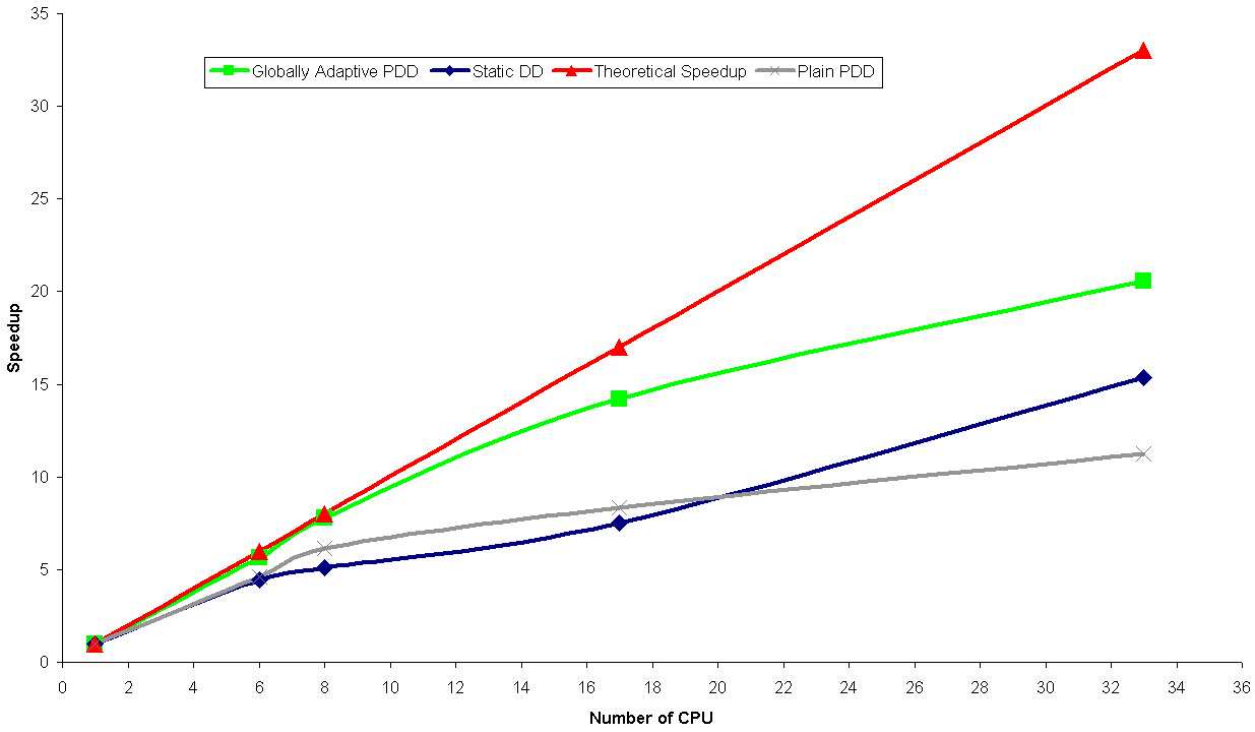



Figure 110.47: Scalability Study on 9,297 Elements, 32,091 DOFs Model, ITR=1e-3, Imbal Tol 5%

### 110.4.9 Scalability Study on Prototype Model

The ultimate purpose of this chapter is to develop an efficient parallel simulation tool for large scale earthquake analysis on prototype SFSI system. After in-depth development-refining process conducted in previous sections, real 3-bent production models have been set up to study the parallel performance of the proposed PDD algorithm using real world earthquake ground motions.

#### 110.4.9.1 3 Bent SFSI Finite Element Models

As described in later sections, various sizes of a 3 bent bridge SFSI system has been developed to study dynamic behaviors of the whole system in different frequency domain. These models provide perfect test cases for parallel scalability study of our proposed PDD algorithm.

Detailed model description will be presented in later chapters of this chapter and only model size and mesh pictures are shown here to indicate the range of model sizes we have covered.

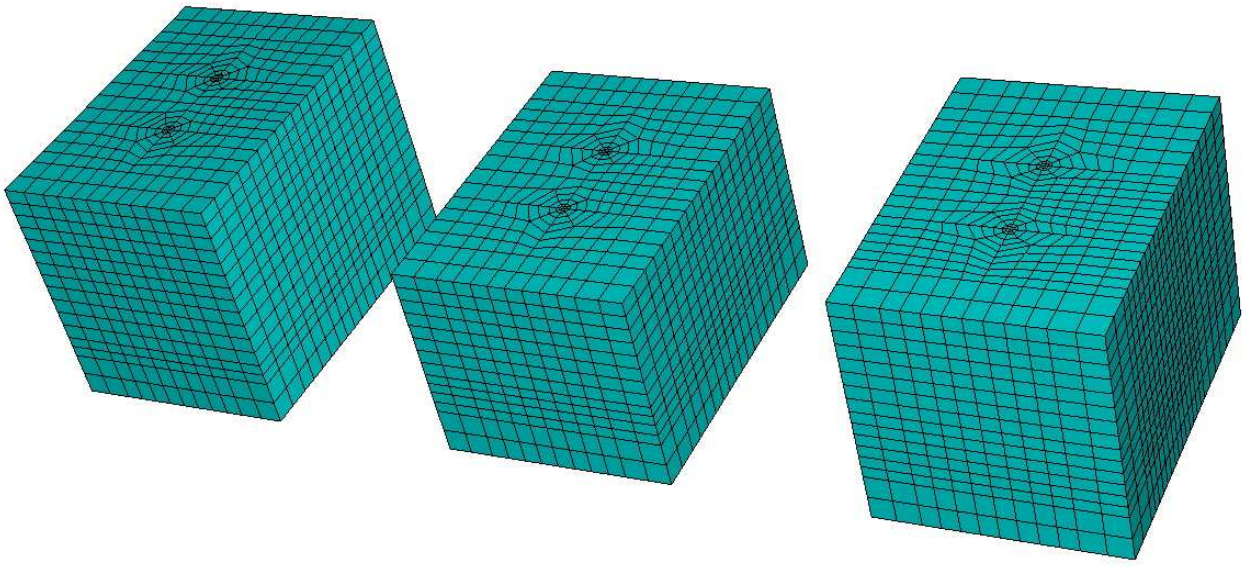


Figure 110.48: Finite Element Model - 3 Bent SFSI, 56,481 DOFs, 13,220 Elements, Frequency Cutoff $>$ 3Hz, Element Size 0.9m, Minimum $G/G_{max}$ 0.08, Maximum Shear Strain $\gamma$ 1%

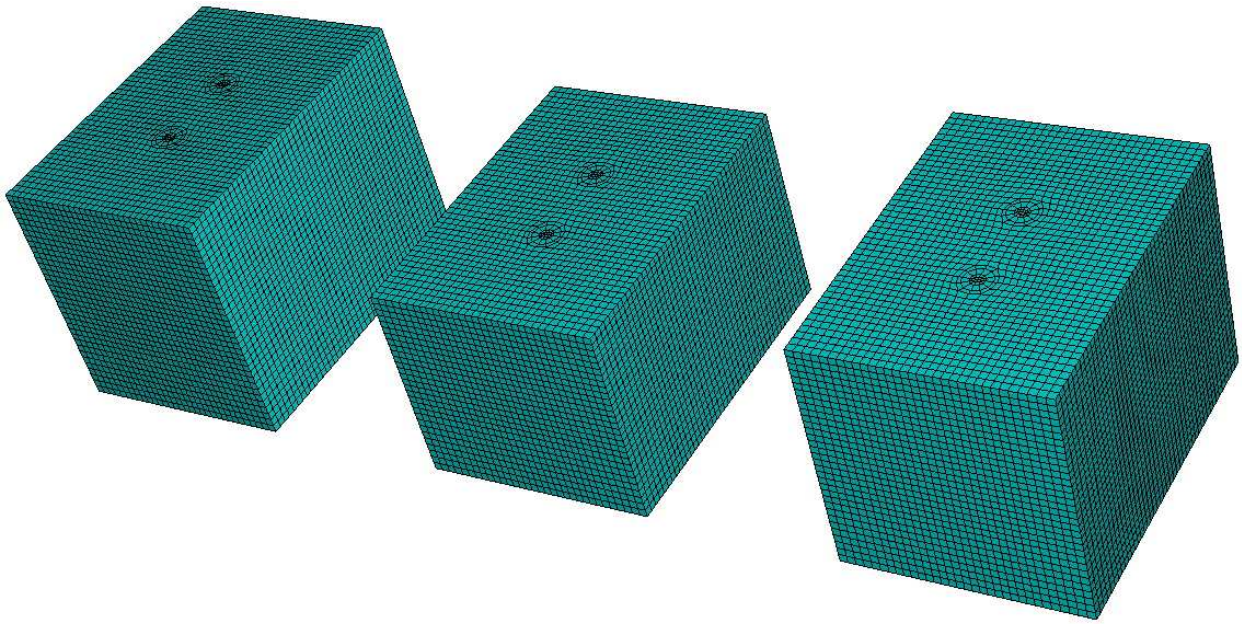


Figure 110.49: Finite Element Model - 3 Bent SFSI, 484,104 DOFs, 151,264 Elements, Frequency Cutoff 10Hz, Element Size 0.3m, Minimum $G/G_{max}$ 0.08, Maximum Shear Strain $\gamma$ 1%

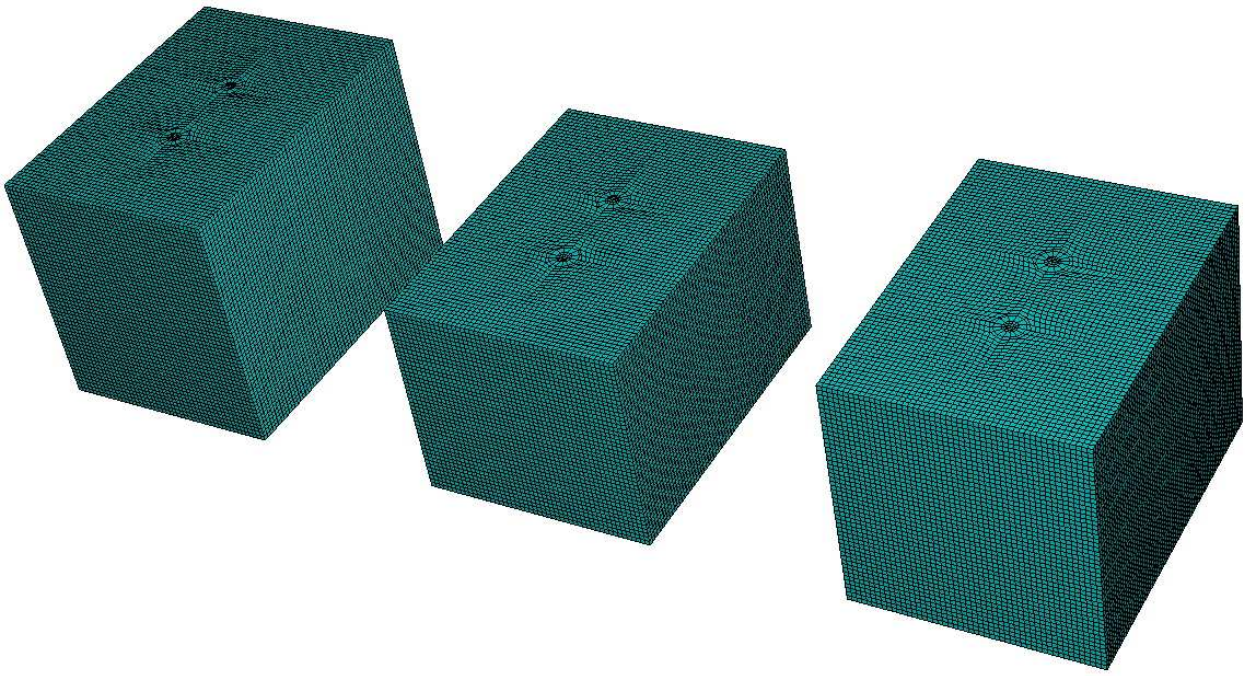


Figure 110.50: Finite Element Model - 3 Bent SFSI, 1,655,559 DOFs, 528,799 Elements, Frequency Cutoff 10Hz, Element Size 0.15m, Minimum $G/G_{max}$ 0.02, Maximum Shear Strain $\gamma$ 5%

### 110.4.9.2 Scalability Runs

The models with different detail levels have been subject to 1997 Northridge earthquake respectively for certain time steps and total wall clock time has been recorded to analyze the parallel scalability of our proposed PDD. The result is presented in Figure 110.51.

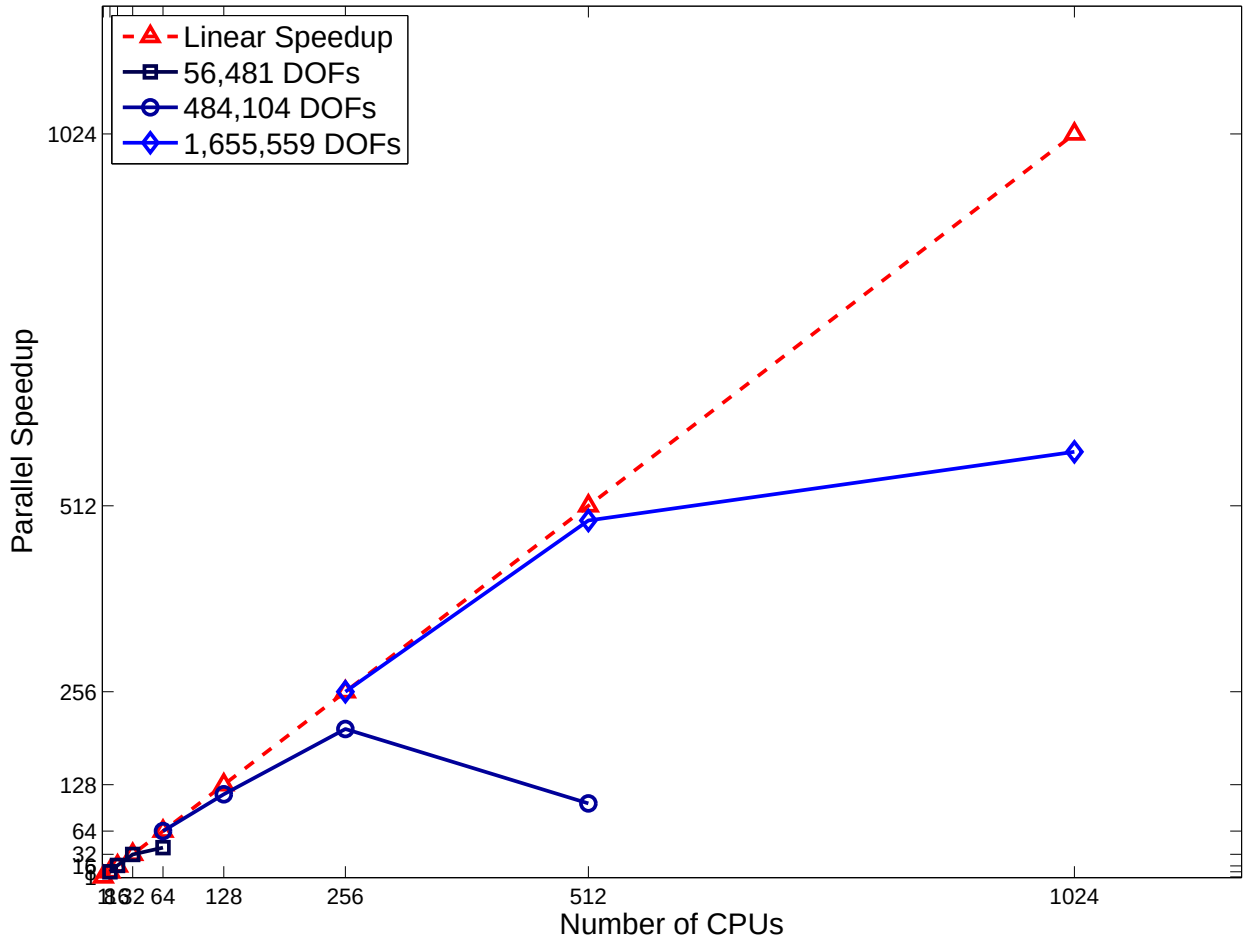



Figure 110.51: Scalability Study on 3 Bent SFSI Models, DRM Earthquake Loading, Transient Analysis, ITR=1e-3, Imbal Tol 5%, Performance Downgrade Due to Increasing Network Overhead

### 110.4.10 Conclusions

Through detailed performance studies as presented in previous sections, some conclusions can be drawn and future directions can be noted.

- Plastic Domain Decomposition (PDD) algorithm based on adaptive multilevel graph partitioning kernels has been shown to be effective for elastic-plastic parallel finite element calculations. PDD algorithm consistently outperforms classical Domain Decomposition method for models tested so far in this chapter as shown in Figures 110.52 and 110.54.

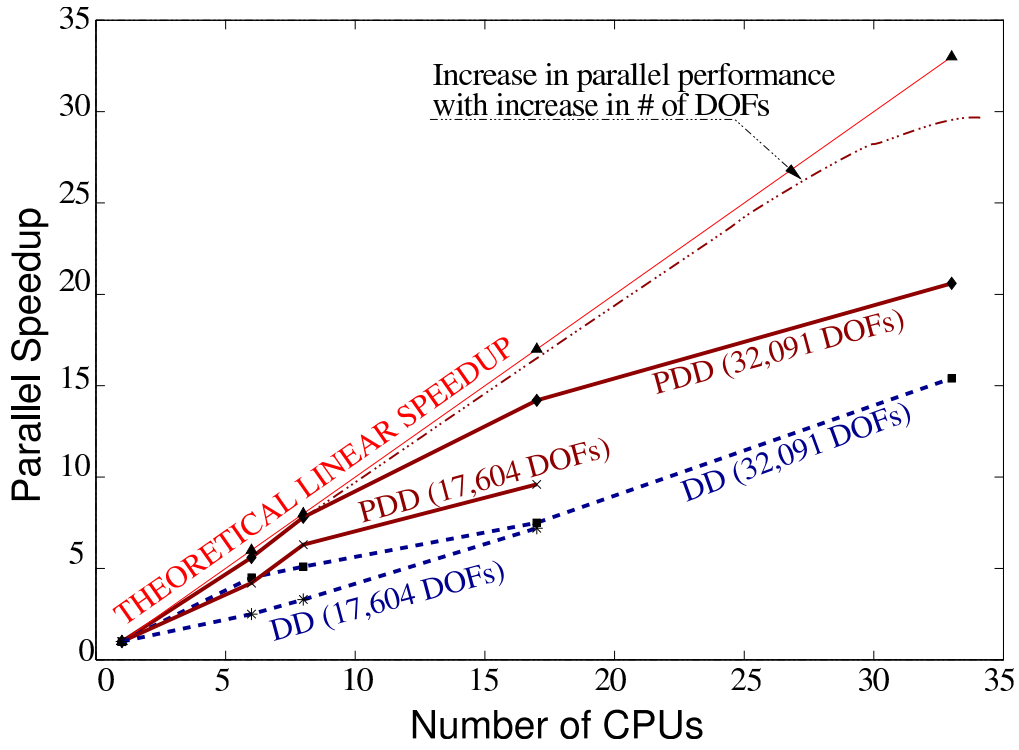



Figure 110.52: Relative Performance of PDD over DD, Shallow Foundation Model, Static Loading, ITR=1e-3, Imbal Tol 5%

- There are some parameters that can be calibrated in the current implementation. As indicated by results of thorough numerical tests, ITR=0.001 and load imbalance tolerance ubvec=1.05 (5%) should be adopted and studies on our application in this chapter have shown they are adequate and able to bring performance not worse than the commonly used domain decomposition method in parallel finite element analysis.

- For the parameters suggested in the chapter, we can see a general trend that the efficiency of PDD will drop as the number of processors increases. This can be explained. The implication of increasing processing units is that the subdomain problem size will decrease. It is naturally evident that the repartition load balancing won't be able to recover the overhead by balancing off small size local calculations. The improved design of globally adaptive PDD algorithm has been implemented in this chapter and both data communication and model regeneration costs associated with graph repartitioning have been integrated into the new globally adaptive strategy. With the new design, it has also been shown that the PDD algorithm consistently outperforms classic one step domain decomposition algorithm and better scalability can be obtained as shown in Figure 110.53. It has been shown that even for large number of processors, the current implementation can always guarantee that the performance of PDD is not worse than static DD method as shown in Figure 110.54. (the repartition routine has less than 5% overhead of the total wall clock time).

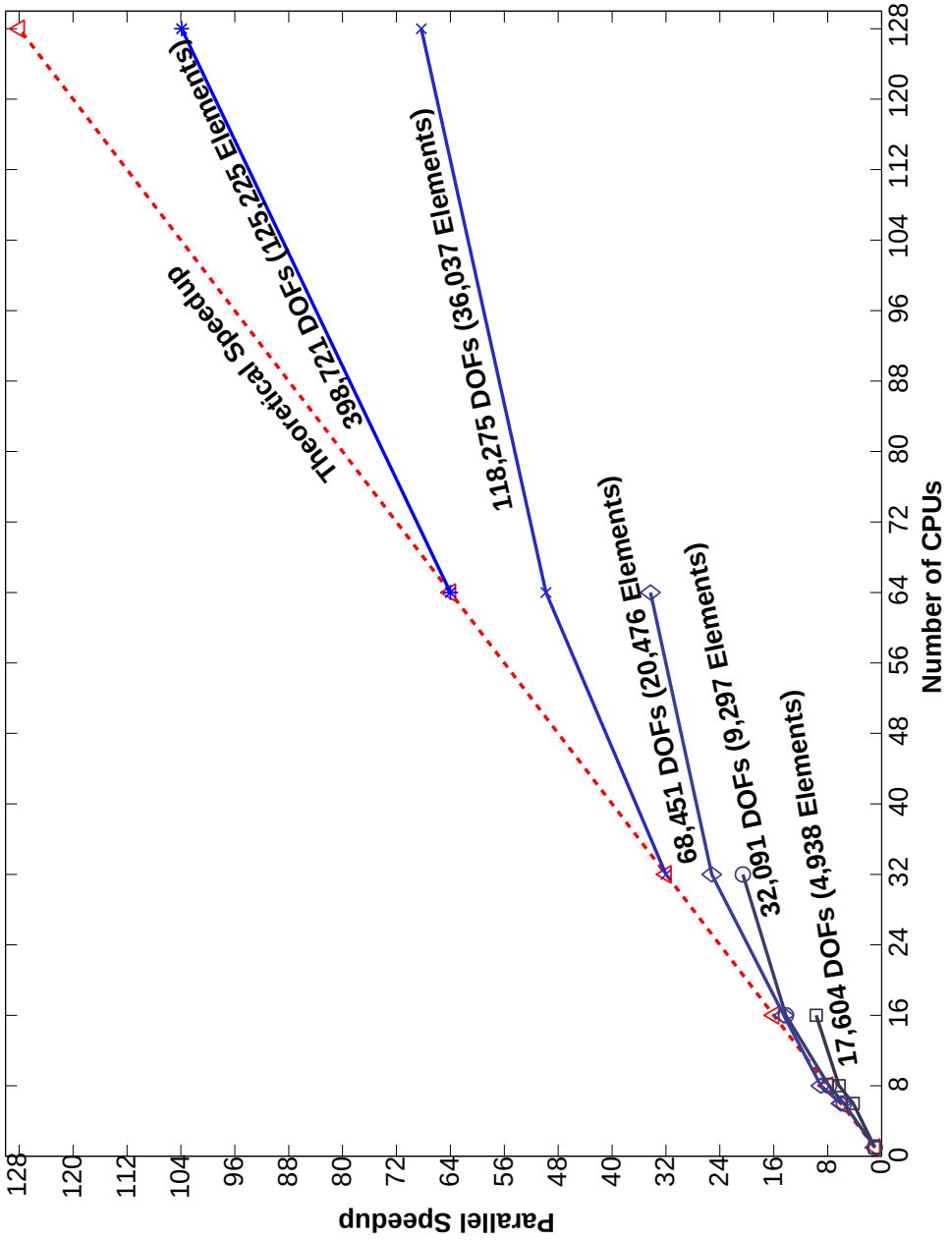



Figure 110.53: Scalability of PDD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5%

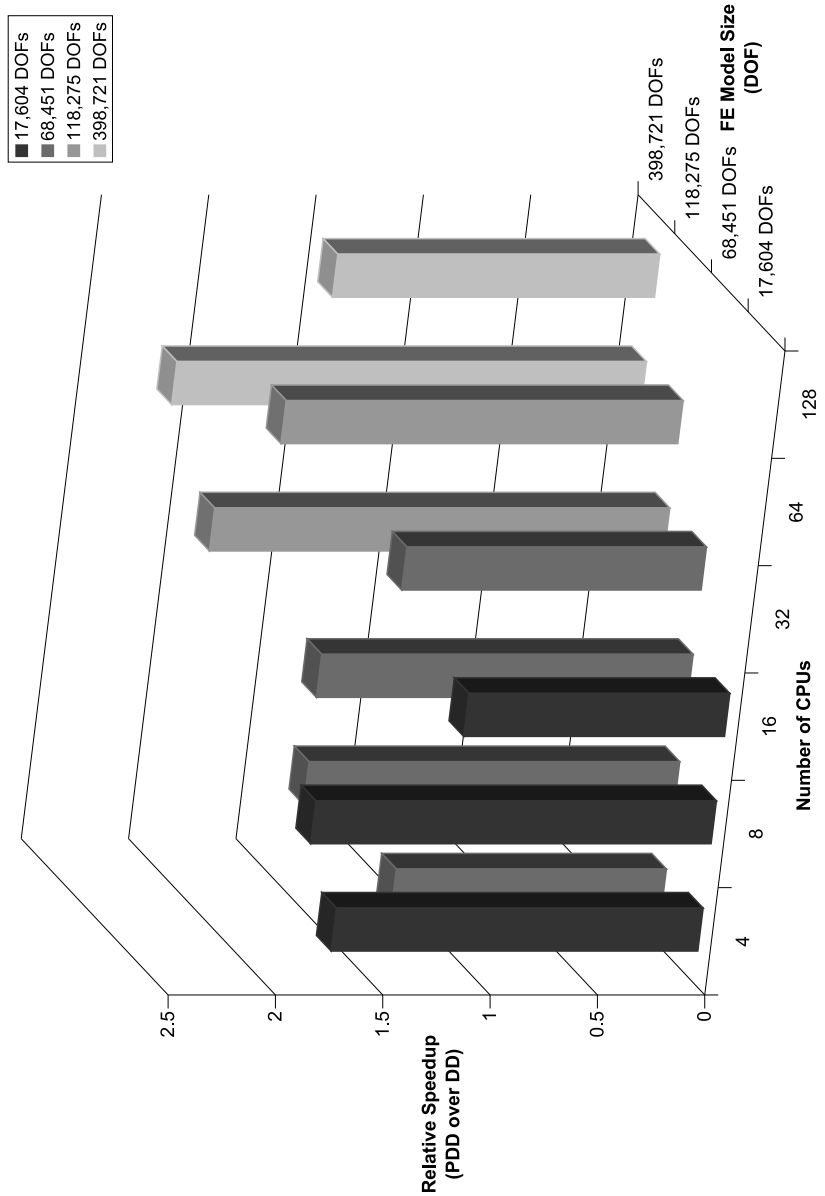



Figure 110.54: Relative Speedup of PDD over DD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5%

- If the problem size is fixed, there exists an optimum number of processors that can bring the best performance of the proposed load balancing algorithm. As the number of processing units increases after this number, the efficiency of proposed algorithm drops, which is understandable because the local load imbalance is so small overall that balancing gain won't offset the extra cost associated with repartitioning. But still the bottom line of proposed adaptive PDD algorithm is that it can run as fast as static one-step domain decomposition approach with less than 5% overhead of repartitioning routine calls. On the other hand, if the number of processing units is fixed, bigger finite element model will exhibit better performance. The conclusion is shown clearly in 3D in Figure 110.54.

- It is also worthwhile to point out that even without comparing with classical DD, PDD itself exhibits deteriorating performance as the number of processing units increases. Here the reproduction of Figure 110.53 is presented with some downside performance noted as shown in Figure 110.55.

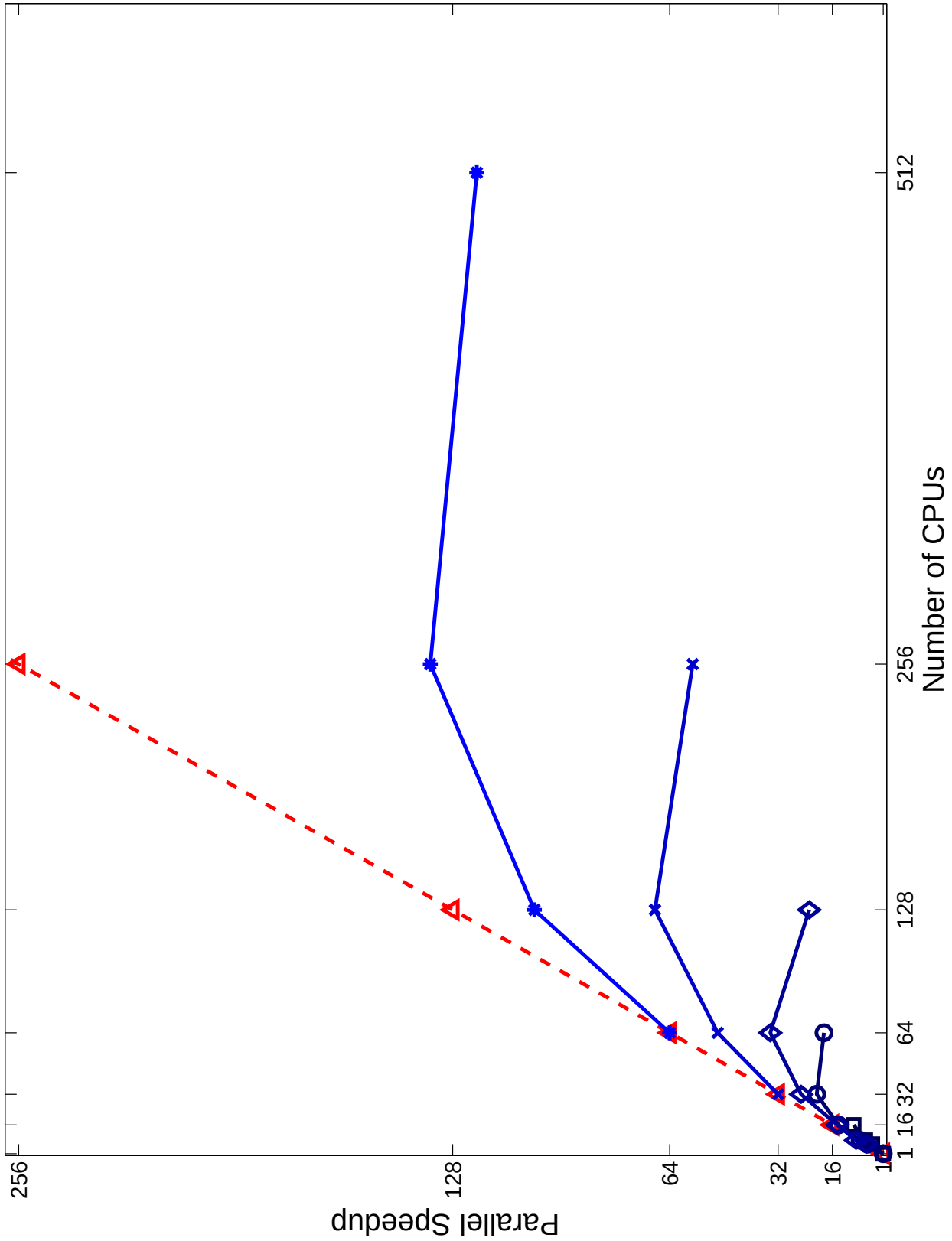



Figure 110.55: Full Range Scalability of PDD, Static Loading, Shallow Foundation Model, ITR=1e-3, Imbal Tol 5%, Performance Downgrade Due to Increasing Network Overhead

The implication is explained as follows:

- The performance drop partly is due to the communication overhead gets bigger and bigger so parallel processing will not be able to offset the communication loss.
- It is also noted that as the number of processing units increases, the elemental level calculation drops very scalably with the number of CPUs. This is inherently advantage of the proposed PDD algorithm. PDD through domain decomposition is very scalable for local level calculations because inherently local comp is element-based. when elements are distributed, loads are spread out evenly (during initial and redistribution). So as the number of CPU increases, the equation solving becomes more expensive.

  For the case of 56,481 DOFs prototype model with DRM earthquake loading, it has been observed that for sequential case (1 CPU), elemental computation takes 70% of time. As for parallel case (8 CPUs), we optimized parallel elemental computations through PDD, elemental computation only accounts for about 40%. As the number of CPU increases, parallel case (32 CPUs), the local level computation will only take less than 10% of total wall clock time. In other words, as the number of CPUs increases, PDD loses scalability because of the equation solving now dominates. As being discussed in Chapter 110.5, the parallel direct solver itself is not scalable up to large number of CPUs Demmel et al. (1999a). Parallel iterative solver is much more scalable but difficult to guarantee convergence. This is now also the most important topic in the whole scientific computing community.

For one set of fixed algorithm parameters, such as ITR and load imbalance tolerance, basic conclusion is there exists an optimal number of processors that can bring best performance and as finite element model size increases, this number increases as listed in Table 110.6.

Table 110.6: Best Performance Observed for ITR=0.001, Load Imbalance Tolerance %5

| # of DOFs | Speedup | # of CPUs |
|---|---|---|
| 4,035 | 1.553 | 4 |
| 17,604 | 1.992 | 7 |
| 32,091 | 1.334 | 7 |
| 68,451 | 1.068 | 16 |

The second point is related to the implementation of the multilevel graph partitioning algorithm. In current implementation of ParMETIS used in this chapter, vertex weight can only be specified as an int. That means in order to get timing data from local level calculation for each element, double data

returned by MPI timing routine has to be converted to int. Significant digit loss can happen depending on what accuracy the system clock can carry. We can also adjust the vertex weight by amplifying the timing by scale factors in order to save effective digits. 10 millisecond has been used in this chapter to represent the effective timing digits when converting from double to int.

## 110.5 Application of Project-Based Iterative Methods in SFSI Problems

### 110.5.1 Introduction

Finite element method has been the most extensively used numerical method in computational mechanics. Equation solver is the numerical kernel of any finite element package. Gauss elimination type direct solver has dominated due to its robustness and predictability in performance.

As modern computer becomes more and more powerful, more advanced and detailed models need to be analyzed by numerical simulation. Direct solver is not the favorite choice for large scale finite element calculations because of high memory requirements and the inherent lack of parallelism of the method itself.

The motivation for presented work on iterative solvers stems from the need to expand the toolset of parallel iterative solvers for large scale simulation problems related to Earthquake-Soil-Structure interaction problems

In this section, the effectiveness of Krylov iterative methods has been tested in solving soil-structure interaction problems. Preconditioning techniques have been introduced. Robustness of iterative solvers has been investigated on equation systems from real soil-structure interaction problems. Several popular parallel algorithms and tools have been collected and implemented on PETSc platform to solve the SFSI problems. Performance study has been carried out using IA64 super computers at San Diego Supercomputing Center. A complete implementation has been developed within our computational system, within MOSS libraries, with extensive use of ParMETIS, and other material and numerical libraries.

### 110.5.2 Projection-Based Iterative Methods

Projection techniques are defined as methods to find approximate solutions $\hat{x}$ for $Ax = b$ ($A \in \mathcal{R}^{n\times n}$) in a subspace $\mathcal{W}$ of dimension $m$. Then in order to determine $\hat{x}$, we need $m$ independent conditions. One way to obtain these is by requiring the residual $b - A\hat{x}$ is orthogonal to a subspace $\mathcal{V}$ of dimension $m$, i.e.,

$$\hat{x} \in \mathcal{W}, b - A\hat{x} \perp \mathcal{V} \tag{110.11}$$

The conditions shown in Equation 110.11 are known as Petrov-Galerkin conditions (Bai, 2007).

There are two key questions to answer if one wants to use projection techniques in solving large scale linear systems. Different answers lead to many variants of the projection method.

- Choice of Subspaces
  Krylov subspaces have been the favorite of most researchers and a large family of methods have been developed based on Krylov subspaces. Typically people choose either $\mathcal{V} = \mathcal{W}$ or $\mathcal{V} = A\mathcal{W}$ with $\mathcal{V}$ and $\mathcal{W}$ both Krylov subspaces.

- Enforcement of Petrov-Galerkin Conditions
  Arnoldi's procedure and Lanczos algorithm are two choices for building orthogonal or biorthogonal sequence to enforce the projection conditions.

The iterative methods discussed in this section are generally split into two categories, one based on Arnoldi's procedure and the other on Lanczos biorthoganalization. The most popular for the first family are Conjugate Gradient and General Minimum Residual methods, while Bi-Conjugate Gradient and Quasi-Minimum Residual methods represent the Lanczos family.

#### 110.5.2.1 Conjugate Gradient Algorithm

The conjugate gradient (CG) algorithm is one of the best known iterative techniques for solving sparse symmetric positive definite (SPD) linear systems. This method is a realization of an orthogonal projection technique onto the Krylov subspace $\mathcal{K}_m(A, r_0)$, where $r_0$ is the initial residual. Because $A$ is symmetry, some simplifications resulting from the three-term Lanczos recurrence will lead to more elegant algorithms (Demmel, 1997).

ALGORITHM CG (Saad, 2003)

1. *Compute* $r_0 := b - Ax_0$, $p_0 := r_0$
2. *For* $j = 0, 1, \cdots,$ *until convergence, Do*
3. $\alpha_j := (r_j, r_j)/(Ap_j, p_j)$
4. $x_{j+1} := x_j + \alpha_j p_j$
5. $r_{j+1} := r_j - \alpha_j A p_j$
6. $\beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7. $p_{j+1} := r_{j+1} + \beta_j p_j$
8. *EndDo*

- Applicability

  Matrix $A$ is SPD.

- Subspaces

  Choose $\mathcal{W} = \mathcal{V} = \mathcal{K}_m(A, r_0)$, in which initial residual $r_0 = b - Ax_0$.

- Symmetric Lanczos Procedure

  This procedure can be viewed as a simplification of the Arnoldi's procedure when $A$ is symmetric. Great three-term Lanczos recurrence is discovered when the symmetry of $A$ is considered (Demmel, 1997).

- Optimality

  If $A$ is SPD and one chooses $\mathcal{W} = \mathcal{V}$, enforcing Petrov-Galerkin conditions minimizes the $A$-norm of the error over all vectors $x \in \mathcal{W}$, i.e., $\hat{x}$ solves the problem,

  $$\min_{x \in \mathcal{W}} \|x - x^*\|_A, x^* = A^{-1}b \tag{110.12}$$

  From the lemma above, one can derive global minimization property of the Conjugate Gradient method. The vector $x_k$ in the Conjugate Gradient method solves the minimization problem

  $$\min_x \phi(x) = \frac{1}{2}\|x - x^*\|_A^2, x - x_0 \in \mathcal{K}_k(A, r_0) \tag{110.13}$$

- Convergence

  In exact arithmetic, the Conjugate Gradient method will produce the exact solution to the linear system $Ax = b$ in at most $n$ steps and it owns the superlinear convergence rate. The behavior of Conjugate Gradient algorithm in finite precision is much more complex. Due to rounding errors, orthogonality is lost quickly and finite termination does not hold anymore. What is more meaningful in application problems would be to use CG method for solving large, sparse, well-conditioned linear systems in far fewer than $n$ iterations.

#### 110.5.2.2 GMRES

The Generalized Minimum Residual method is able to deal with more general type of matrices.

ALGORITHM GMRES (Saad, 2003)

1. *Compute* $r_0 := b - Ax_0$, $\beta := \|r_0\|_2$, *and* $v_1 := r_0/\beta$
2. *For* $j = 1, 2, \cdots, m$, *Do*
3. *Compute* $\omega_j := Av_j$
4. *For* $i = 1, \cdots, j$, *Do*
5. $h_{ij} := (\omega_j, v_i)$
6. $\omega_j := \omega_j - h_{ij} v_i$
7. *EndDo*
8. $h_{j+1,j} = \|\omega_j\|_2$. *If* $h_{j+1,j} = 0$ *set* $m := j$ *and go to 11*
9. $v_{j+1} = \omega_j / h_{j+1,j}$
10. *EndDo*
11. *Define the* $(m+1) \times m$ *Hessenberg matrix* $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$
12. *Compute* $y_m$, *the minimizer of* $\|\beta_i e_1 - \bar{H}_m y\|_2$, *and* $x_m = x_0 + V_m y_m$

- Applicability

  Matrix $A$ is nonsingular.

- Subspaces

  Choose $\mathcal{W} = \mathcal{K}_m(A, r_0)$ and $\mathcal{V} = A\mathcal{W} = A\mathcal{K}_m(A, r_0)$, in which initial residual $r_0 = b - Ax_0$.

- Arnoldi's Procedure

  Classic Arnoldi's procedure (modified Gram-Schmidt) is followed in GMRES (Bai, 2007).

- Optimality

  If one chooses $\mathcal{V} = A\mathcal{W}$, enforcing Petrov-Galerkin conditions solves the least square problem

$$\|b - A\tilde{x}\|_2 = \min_{x \in \mathcal{W}} \|b - Ax\|_2 \tag{110.14}$$

- Convergence

  It has been shown that in exact arithmetic, GMRES can not breakdown and will give exact solutions in at most $n$ steps. In practice, the maximum steps GMRES can run depends on the memory due to the fact it needs to store all Arnoldi vectors. Restarting schemes have been proposed for a fixed $m$, which is denoted by GMRES($m$). Typical value for $m$ can be $m \in [5, 20]$. GMRES($m$) can not breakdown in exact arithmetic before the exact solution has been reached. But it may never converge for $m < n$ (Bai, 2007).

#### 110.5.2.3 BiCGStab and QMR

These two methods are based on nonsymmetric Lanczos procedure, which is quite different from Arnoldi's in the sense that it formulates biorthogonal instead of orthogonal sequence. They are counterparts of CG and GMRES method, which follows similar derivation procedure except the Lanczos biorthogonalization is used instead of Arnoldi's procedure (Bai, 2007).

### 110.5.3 Preconditioning Techniques

Lack of robustness is a widely recognized weakness of iterative solvers relative to direct solvers. Using preconditioning techniques can greatly improve the efficiency and robustness of iterative methods. Preconditioning is simply a means of transforming the original linear system into one with the same solution but easier to solve with an iterative solver. Generally speaking, the reliability of iterative techniques, when dealing with various applications, depends much more on the quality of the preconditioner than on the particular Krylov subspace accelerator used.

The first step in preconditioning is to find a preconditioning matrix $M$. The matrix $M$ can be defined in many different ways but there are a few minimal requirements the $M$ is supposed to satisfy (Benzi, 2002).

1. From practical point of view, the most important requirement of $M$ is that it should be inexpensive to solve linear system $Mx = b$. This is because the preconditioned algorithm will all require a linear system solution with the matrix $M$ at each step.
2. The matrix $M$ should be somehow close to $A$ and it should not be singular. We can see that actually most powerful preconditioners are constructed directly from $A$.
3. The preconditioned $M^{-1}A$ should be well-conditioned or has very few extreme eigenvalues thus $M$ can accelerate convergence dramatically.

Once a preconditioner $M$ is available, there are three ways to apply it.

1. Left Preconditioning

$$M^{-1}Ax = M^{-1}b \tag{110.15}$$

2. Right Preconditioning

$$AM^{-1}u = b, x \equiv M^{-1}u \tag{110.16}$$

3. Split Preconditioning

   It is a very common situation that $M$ is available in factored form $M = M_L M_R$, in which, typically, $M_L$ and $M_R$ are triangular matrices. Then the preconditioning can be split,

   $$M_L^{-1} A M_R^{-1} u = b, x \equiv M_R^{-1} u \tag{110.17}$$

   It is imperative to preserve symmetry when the original matrix $A$ is symmetric, so the split preconditioner seems mandatory in this case.

Consider that a matrix $A$ that is symmetric and positive definite and assume that a preconditioner $M$ is available. The preconditioner $M$ is a matrix that approximates $A$ in some yet-undefined sense. We normally require that the $M$ is also symmetric positive definite.

In order to preserve the nice SPD property, in the case when $M$ is available in the form of an incomplete Cholesky factorization, $M = LL^T$, people can simply just use the split preconditioning, which yields the SPD matrix

$$L^{-1} A L^{-T} u = L^{-1} b, x \equiv L^{-T} u \tag{110.18}$$

However, it is not necessary to split the preconditioner in this manner in order to preserve symmetry. Observe that $M^{-1}A$ is self-adjoint for the $M$ inner product

$$(x, y)_M \equiv (Mx, y) = (x, My) \tag{110.19}$$

since

$$(M^{-1}Ax, y)_M = (Ax, y) = (x, Ay) = (x, M(M^{-1}A)y) = (x, M^{-1}Ay)_M \tag{110.20}$$

Therefore, an alternative is to replace the usual Euclidean inner product in the CG algorithm with the $M$ inner product (Saad, 2003).

If the CG algorithm is rewritten for this new inner product, denoting by $r_j = b - Ax_j$ the original residual and by $z_j = M^{-1} r_j$ the residual for the preconditioned system, the following sequence of operations is obtained, ignoring the initial step:

1. $\alpha_j := (z_j, z_j)_M / (M^{-1}Ap_j, p_j)_M$,
2. $x_{j+1} := x_j + \alpha_j p_j$,
3. $r_{j+1} := r_j - \alpha_j A p_j$ and $z_{j+1} := M^{-1} r_{j+1}$,
4. $\beta_j := (z_{j+1}, z_{j+1})_M / (z_j, z_j)_M$,

5. $p_{j+1} := z_{j+1} + \beta_j p_j$.

Since $(z_j, z_j)_M = (r_j, z_j)$ and $(M^{-1}Ap_j, p_j)_M = (Ap_j, p_j)$, the $M$ inner products do not have to be computed explicitly. With this observation, the following algorithm is obtained.

ALGORITHM Preconditioned CG (Saad, 2003)

1. *Compute* $r_0 := b - Ax_0$, $z_0 := M^{-1}r_0$, $p_0 := z_0$
2. *For* $j = 0, 1, \cdots$, *until convergence, Do*
3. $\alpha_j := (r_j, z_j)/(Ap_j, p_j)$
4. $x_{j+1} := x_j + \alpha_j p_j$
5. $r_{j+1} := r_j - \alpha_j Ap_j$
6. $z_{j+1} := M^{-1}r_{j+1}$
7. $\beta_j := (r_{j+1}, z_{j+1})/(r_j, z_j)$
8. $p_{j+1} := z_{j+1} + \beta_j p_j$
9. *EndDo*

### 110.5.4 Preconditioners

Finding a good preconditioner to solve a given sparse linear system is often viewed as a combination of art and science. Theoretical results are rare and some methods work surprisingly well, often despite expectations. As it is mentioned before, the preconditioner $M$ is always close to $A$ in some undefined-yet sense. Some popular preconditioners will be introduced in this section.

#### 110.5.4.1 Jacobi Preconditioner

This might be the simplest preconditioner people can think of. If $A$ has widely varying diagonal entries, we may just use diagonal preconditioner $M = \text{diag}(a_{11}, \cdots, a_{nn})$. One can show that among all possible diagonal preconditoners, this choice reduces the condition number of $M^{-1}A$ to within a factor of $n$ of its minimum value.

#### 110.5.4.2 Incomplete Cholesky Preconditioner

Another simple way of defining a preconditioner that is close to $A$ is to perform an incomplete Cholesky factorization of $A$. Incomplete factorization formulates an approximation of $A \approx \hat{L}\hat{L}^T$, but with less or no fill-ins relative to the complete factorization $A = LL^T$ (Demmel, 1997).

ALGORITHM Incomplete Cholesky Factorization (Saad, 2003)

1. *For* $j = 1, 2, \cdots, n$, *Do*
3. $l_{jj} := \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2}$
4. *For* $i = j+1, \cdots, n$, *Do*
5. $l_{ij} = (a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}) / l_{jj}$
6. *Apply dropping rule to* $l_{ij}$
7. *EndDo*
8. *EndDo*

There are many ways to control the number of fill-ins in IC factorization. No fill-in version of incomplete Cholesky factorization IC(0) is rather easy and inexpensive to compute. On the other hand, it often leads to a very crude approximation of $A$, which may result in the Krylov subspace accelerator requiring too many iterations to converge. To remedy this, several alternative incomplete factorizations have been developed by researchers by allowing more fill-in in $L$, such as incomplete Cholesky factorization with dropping threshold IC($\epsilon$). In general, more accurate IC factorizations require fewer iterations to converge, but the preprocessing cost to compute the factors is higher.

#### 110.5.4.3 Robust Incomplete Factorization

Incomplete factorization preconditioners are quite effective for many application problems but special care must be taken in order to avoid breakdowns due to the occurrence of non-positive pivots during the incomplete factorization process.

The existence of an incomplete factorization $A \approx \hat{L}\hat{L}^T$ has been established for certain classes of matrices. For the class of $M$-matrices, the existence of incomplete Cholesky factorization was proved for arbitrary choices of the sparsity pattern (Meijerink and van der Vorst, 1977). The existence result was extended shortly thereafter to a somewhat larger class (that of $H$-matrices with positive diagonal entries) (Manteuffel, 1980; Varga et al., 1980; Robert, 1982). Benzi and Tůma (2003) presents reviews on the topic of searching for robust incomplete factorization algorithms and an robust algorithm based on $A$-Orthogonalization has been proposed.

In order to construct triangular factorization of $A$, the well-known is not the only choice. Benzi and Tůma (2003) shows how the factorization $A = LDL^T$ (root-free factorization) can be obtained by means of an $A$-orthogonalization process applied to the unit basis vectors $e_1, e_2, \cdots, e_n$. This is simply the Gram-Schmidt process with respect to the inner product generated by the SPD matrix $A$. This idea is not new and as a matter of fact, it was originally proposed at as early as 1940's in Fox et al. (1948). It has been observed in Hestenes and Stiefel (1952) that $A$-orthogonalization of the unit basis

vectors is closely related to Gaussian elimination but this algorithm costs twice as much as the Cholesky factorization in the dense case.

Factored Approximate Inverse Preconditioner In reference Benzi et al. (1996) $A$-orthogonalization has been exploited to construct factored sparse approximate inverse preconditioners noting the fact that $A$-orthogonalization also produces the inverse factorization $A^{-1} = ZD^{-1}Z^T$ (with $Z$ unit upper triangular and $D$ diagonal). Because the $A$-orthogonaliza-tion, even when performed incompletely, is not subject to pivot breakdowns, these preconditioners are reliable (Benzi et al., 2000). However, they are often less effective than incomplete Cholesky preconditioning at reducing the number of PCG iterations and their main interest stems from the fact that the preconditioning operation can be applied easily in parallel because triangular solve is not necessary in approximate inverse preconditioning.

Reference Benzi and Tůma (2003) investigates the use of $A$-orthogonalization as a way to compute an incomplete factorization of $A$ rather than $A^{-1}$ thus a reliable preconditioning algorithm can be developed. The basic $A$-orthogonalization procedure can be written as follows (Benzi et al., 2000).

ALGORITHM Incomplete Factored Approximate Inverse (Benzi et al., 1996)

1. *Let* $z_i^{(0)} = e_i$, *for* $i = 1, 2, \cdots, n$
2. *For* $i = 1, 2, \cdots, n$, *Do*
3. *For* $j = i, i+1, \cdots, n$, *Do*
4. $p_j^{(i-1)} := a_i^T z_j^{(i-1)}$
5. *EndDo*
6. *For* $j = i+1, \cdots, n$, *Do*
7. $z_j^{(i)} := z_j^{(i-1)} - (\frac{p_j^{(i-1)}}{p_i^{(i-1)}})$
8. *Apply dropping to* $z_j^{(i)}$
9. *EndDo*
10. *EndDo*
11. *Let* $z_i := z_i^{(i-1)}$ *and* $p_i := p_i^{(i-1)}$, *for* $i = 1, 2, \cdots, n$.
12. *Return* $Z = [z_1, z_2, \cdots, z_n]$ *and* $D = diag(p_1, p_2, \cdots, p_n)$.

The basic algorithm described above can suffer a breakdown when a negative or zero value of a pivot $p_i$. When no dropping is applied, $p_i = z_i^T A z_i > 0$. The incomplete procedure is well defined, i.e., no breakdown can occur, if $A$ is an $H$-matrix (in the absence of round-off). In the general case, breakdowns can occur. Breakdowns have a crippling effect on the quality of the preconditioner. A negative $p_i$ would result in an approximate inverse which is not positive definite; a zero pivot would force termination of the procedure, since step (7) cannot be carried out.

The way proposed to avoid non-positive pivots is simply to recall that in the exact $A$-orthogonalization process, the $p_i$'s are the diagonal entries of matrix $D$ which satisfies the matrix equation

$$Z^T A Z = D \tag{110.21}$$

hence for $1 < i < n$

$$p_i = z_i^T A z_i > 0 \tag{110.22}$$

since $A$ is SPD and $z_i \neq 0$. In the exact process, the following equality holds

$$p_i = z_i^T A z_i = a_i^T z_i \quad \text{and} \quad p_j = z_i^T A z_j = a_i^T z_j \tag{110.23}$$

Clearly it is more economical to compute the pivots using just inner product $a_i^T z_i$ rather than the middle expression involving matrix-vector multiply. However, because of dropping and the resulting loss of $A$-orthogonality in the approximate $\bar{z}$-vectors, such identities no longer hold in the inexact process and for some matrices one can have

$$a_i^T \bar{z}_i \ll \bar{z}_i^T A \bar{z}_i \tag{110.24}$$

The robust algorithm requires that the incomplete pivots $\bar{p}_i$'s be computed using the quadratic form $\bar{z}_i^T A \bar{z}_i$ throughout the AINV process, for $i = 1, 2, \cdots, n$.

ALGORITHM Stabilized Incomplete Approximate Inverse (Benzi et al., 2000)

1. *Let* $z_i^{(0)} = e_i$, *for* $i = 1, 2, \cdots, n$
2. *For* $i = 1, 2, \cdots, n$, *Do*
3. $\quad v_i := A z_i^{(i-1)}$
4. $\quad$ *For* $j = i, i+1, \cdots, n$, *Do*
5. $\quad\quad p_j^{(i-1)} := v_i^T z_j^{(i-1)}$
6. $\quad$ *EndDo*
7. $\quad$ *For* $j = i+1, \cdots, n$, *Do*
8. $\quad\quad z_j^{(i)} := z_j^{(i-1)} - (\frac{p_j^{(i-1)}}{p_i^{(i-1)}})$
9. $\quad\quad$ *Apply dropping to* $z_j^{(i)}$
10. $\quad$ *EndDo*
11. *EndDo*
12. *Let* $z_i := z_i^{(i-1)}$ *and* $p_i := p_i^{(i-1)}$, *for* $i = 1, 2, \cdots, n$.
13. *Return* $Z = [z_1, z_2, \cdots, z_n]$ *and* $D = diag(p_1, p_2, \cdots, p_n)$.

Obviously, the robust (referred to as SAINV) and plain algorithm are mathematically equivalent. However, the incomplete process obtained by dropping in the $z$-vectors in step (9) of the robust algorithm leads to a reliable approximate inverse. This algorithm, in exact arithmetic, is applicable to any SPD matrix without breakdowns. The computational cost of SAINV is higher than basic AINV and special care has to be taken to do sparse-sparse matrix-vector multiply.

Incomplete Factorization by SAINV Consider now the exact algorithm (with no dropping) and write $A = LDL^T$ with $L$ unit lower triangular and $D$ diagonal. Observe that $L$ in the $LDL^T$ factorization of $A$ and the inverse factor satisfy

$$AZ = LD \quad \text{or} \quad L = AZD^{-1} \tag{110.25}$$

where $D$ is the diagonal matrix containing the pivots. This easily follows from

$$Z^T AZ = D \quad \text{and} \quad Z^T = L^{-1} \tag{110.26}$$

If we recall that pivot $d_j = p_j = z_j^T A z_j = \langle Az_j, z_j \rangle$, then by equating corresponding entries of $AZD^{-1}$ and $L = [l_{ij}]$ we find that (Benzi and Tůma, 2003; Bollhöfer and Saad, 2001)

$$l_{ij} = \frac{\langle Az_j, z_i \rangle}{\langle Az_j, z_j \rangle} \quad i \geq j \tag{110.27}$$

Hence, the $L$ factor of $A$ can be obtained as a by-product of the $A$-orthogonal-ization, at no extra cost. In the implementation of SAINV, the quantities $l_{ij}$ in Equation 110.27 are the multipliers that are used in updating the columns of $Z$. Once the update is computed, they are no longer needed and are discarded. To obtain an incomplete factorization of $A$, we do just the opposite; we save the multipliers $l_{ij}$ , and discard the column vectors $z_j$ as soon as they have been computed and operated with. Hence, the incomplete $L$ factor is computed by columns; these columns can be stored in place of the $z_j$ vectors, with minimal modifications to the code. Here, we are assuming that the right-looking form of SAINV is being used. If the left-looking one is being used, then $L$ would be computed by rows. Please refer to Benzi and Tůma (2003) for more implementation details.

### 110.5.5 Numerical Experiments

Matrices from soil-structure interaction finite element analysis have been extracted from simulation system to study the performance of different preconditioning techniques on PCG method. The prototype of soil structure model has been shown in Figures 110.56 and 110.57. In order to introduce both of the nonlinear theories for soil and structures, we use continuum elements to model the soil and beam elements for the structures. Matrices from static pushover analysis and dynamic ground motion analysis have been collected for this research.

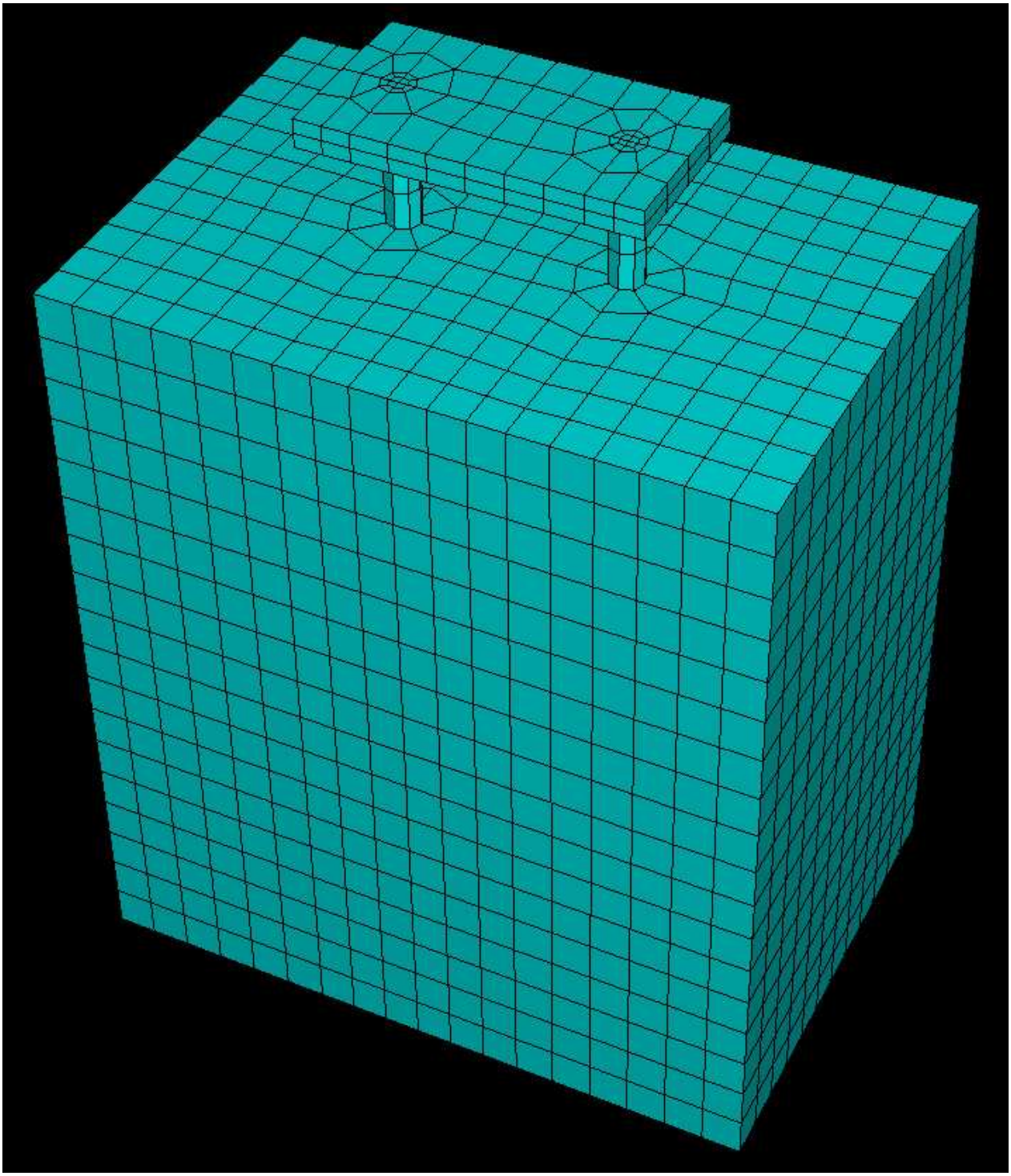


Figure 110.56: Finite Element Mesh of Soil-Structure Interaction Model

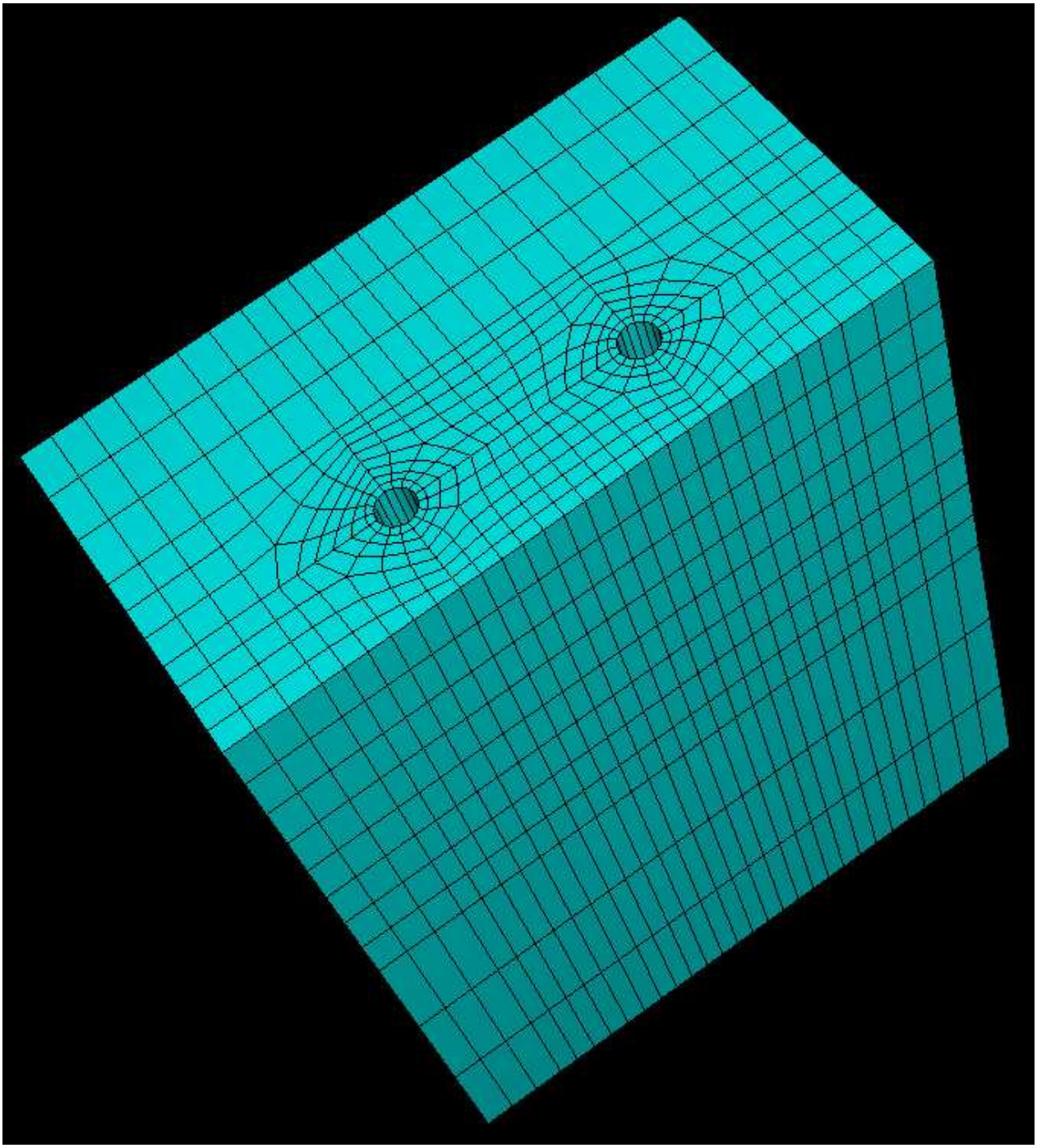


Figure 110.57: Finite Element Mesh of Soil-Structure Interaction Model

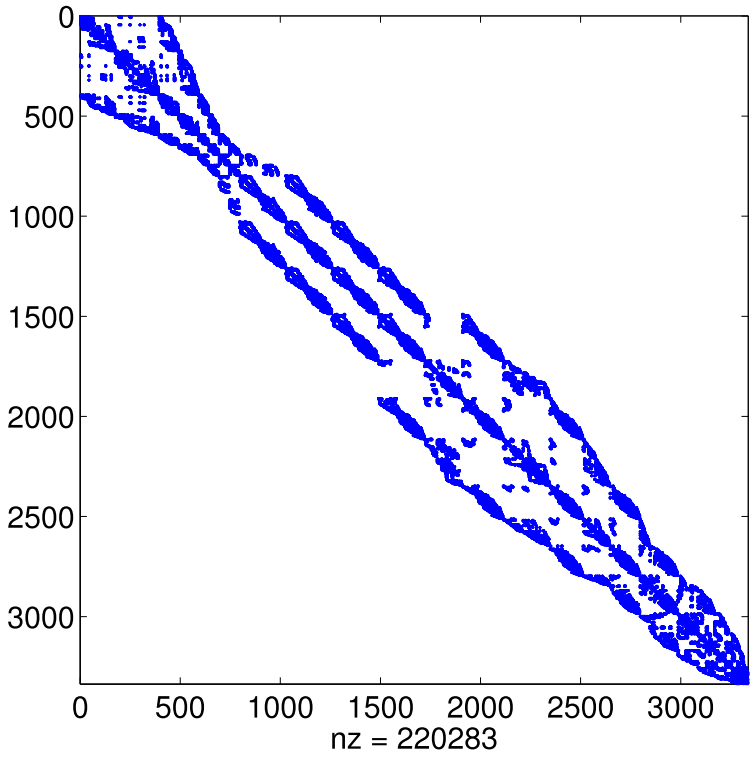



Figure 110.58: Matrices $N = 3336$ (Continuum FEM)

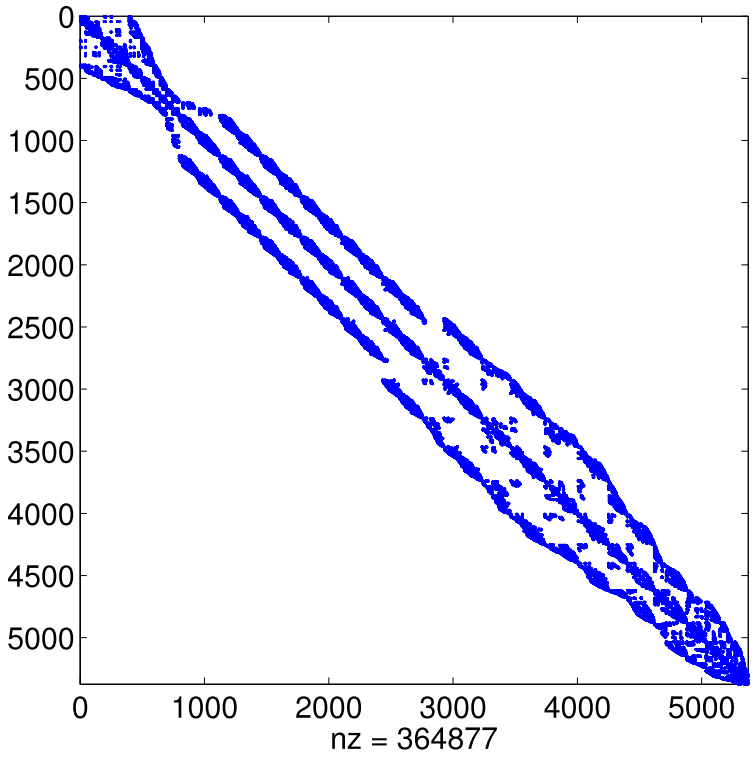



Figure 110.59: Matrices $N = 5373$ (Continuum FEM)

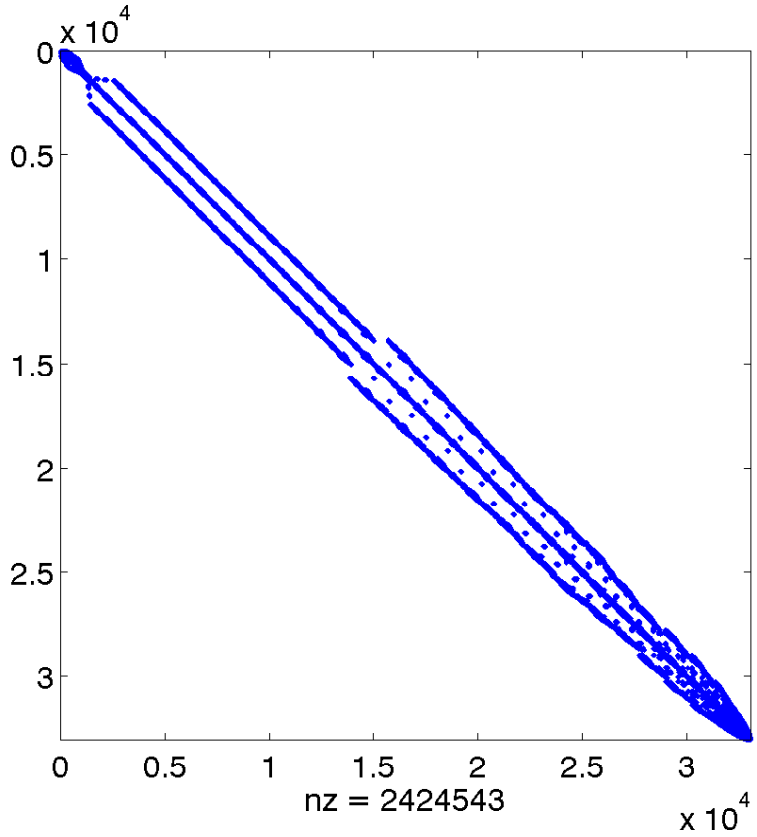



Figure 110.60: Matrices $N = 33081$ (Continuum FEM)

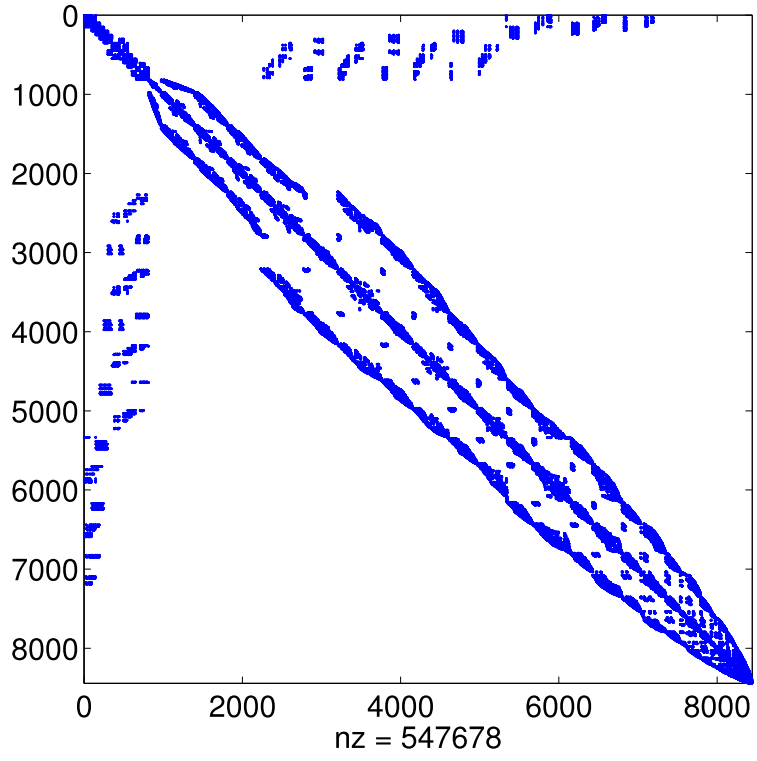



Figure 110.61: Matrices $N = 8842$ (Soil-Beam Static FEM)

Table 110.7: Matrices in FEM Models

| Continuum Model (Static) | | | |
|---|---|---|---|
| Matrix | Property | Dimension | # Nonzeros |
| m1188 | SPD | 3336 | 220283 |
| m1968 | SPD | 5373 | 364877 |
| m11952 | SPD | 33081 | 2424543 |
| **Soil-Beam Model (Static and Dynamic)** | | | |
| Matrix | Property | Dimension | # Nonzeros |
| SoilBeam | SPD | 8442 | 547678 |
| SoilBeamDyn | SPD | 8442 | 547671 |

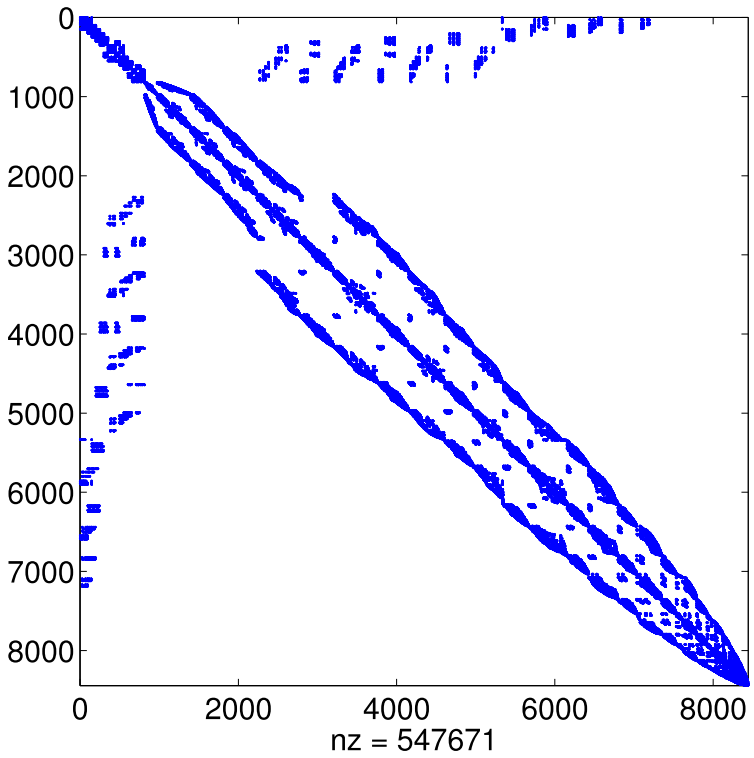



Figure 110.62: Matrices $N = 8842$ (Soil-Beam Dynamic FEM)

SPD matrices have been studied using Conjugate Gradient method with or without preconditioning. Performance has been summarized in Table 110.8.

Table 110.8: Performance of CG and PCG Method (Continuum FEM)

| 3336 DOFs FEM (Static) | | | | | |
|---|---|---|---|---|---|
| Preconditioner | # Iter | Pre Time(s) | Iter Time(s) | Total Time(s) | Density[2] |
| - | 4376 | - | 54.82 | 54.82 | - |
| Jacobi | 1612 | 0.01 | 20.18 | 20.19 | - |
| IC(0) | 413 | 2.19 | 11.07 | 13.26 | 1.00 |
| IC(1e-6) | 5 | 5.90 | 0.47 | 6.37 | 5.88 |
| RIF2(1e-2) | 571 | 9.37 | 14.94 | 24.31 | 0.94 |
| RIF3(1e-2) | 541 | 6.80 | 14.13 | 20.93 | 0.94 |
| 5373 DOFs FEM (Static) | | | | | |
| Preconditioner | # Iter | Pre Time(s) | Iter Time(s) | Total Time(s) | Density |
| - | 4941 | - | 103.78 | 103.78 | - |
| Jacobi | 1711 | 0.01 | 36.61 | 36.62 | - |
| IC(0) | 437 | 6.5 | 20.38 | 26.88 | 1.00 |
| IC(1e-6) | 6 | 19.81 | 1.3 | 21.11 | 8.10 |
| RIF2(1e-2) | 599 | 25.71 | 26.55 | 52.26 | 0.96 |
| RIF3(1e-2) | 566 | 21.31 | 25.23 | 46.54 | 0.96 |
| 33081 DOFs FEM (Static) | | | | | |
| Preconditioner | # Iter | Pre Time(s) | Iter Time(s) | Total Time(s) | Density |
| - | 6754 | - | 952.53 | 952.53 | - |
| Jacobi | 2109 | 0.03 | 308.46 | 308.49 | - |
| IC(0) | 565 | 273.83 | 173.05 | 446.88 | 1.00 |
| IC(1e-6)[3] | | | | | |
| RIF2(1e-2) | 694 | 1172.7 | 211.88 | 1384.58 | 0.99 |
| RIF3(1e-2) | 664 | 1245.4 | 202.67 | 1448.07 | 0.99 |

[2]Density is defined as the number of non-zeros of the incomplete factor divided by the number of non-zeros in the lower triangular part of $A$.

[3]Could not continue because memory requirement larger than 1.4GB.

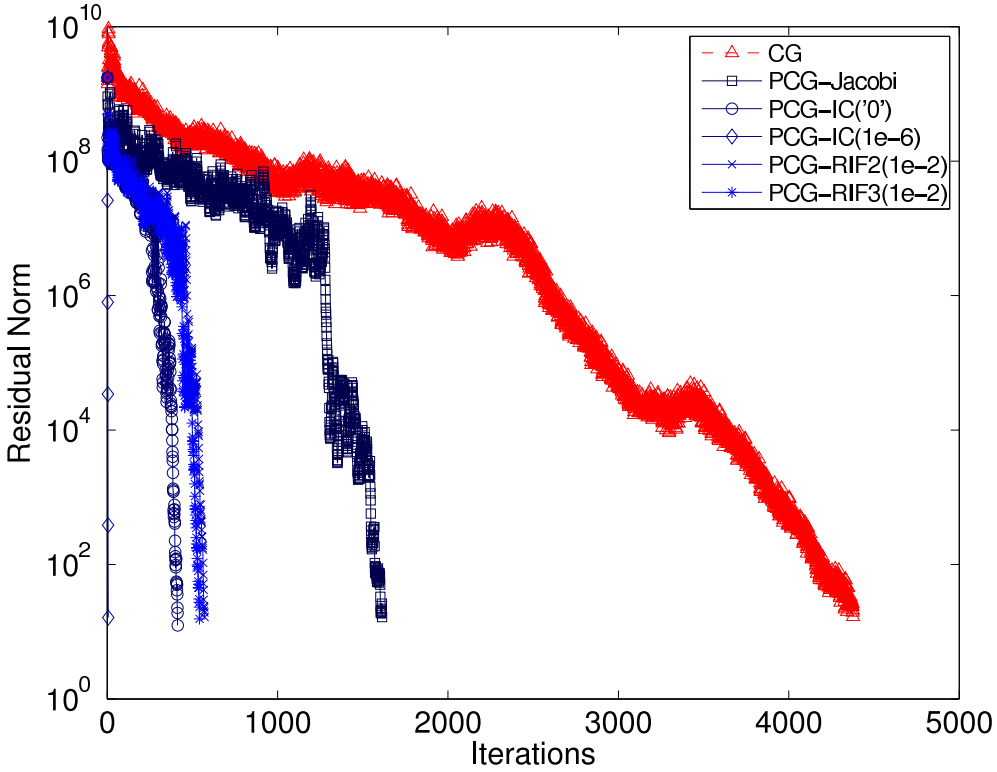



Figure 110.63: Convergence of CG and PCG Method (3336 DOFs Model)

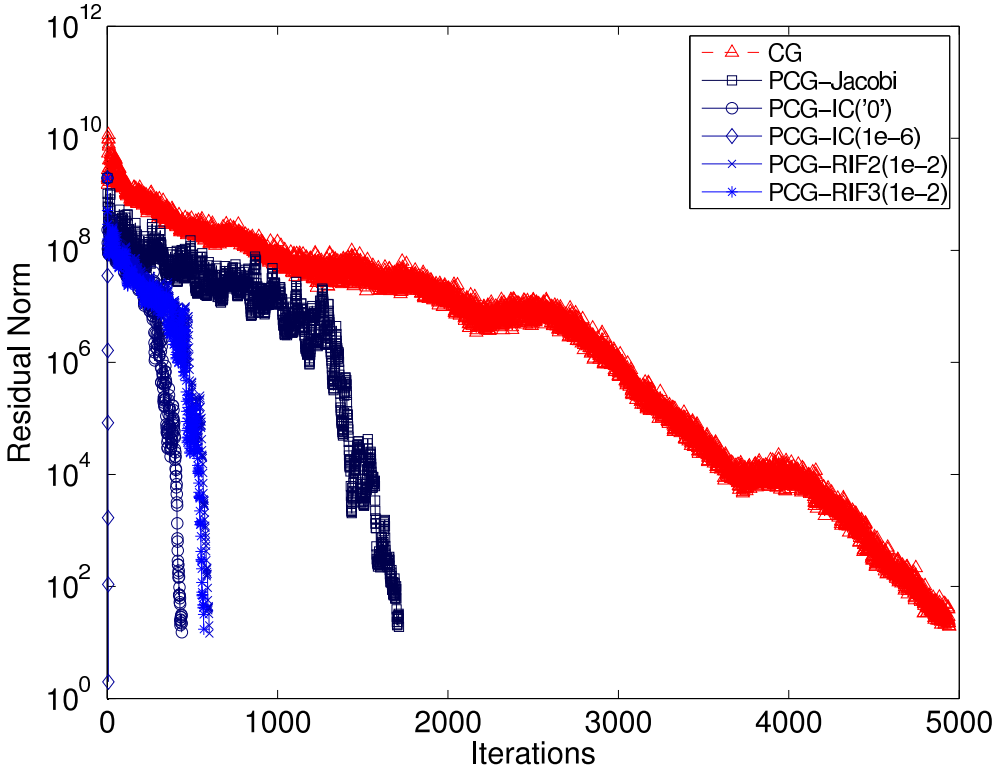



Figure 110.64: Convergence of CG and PCG Method (5373 DOFs Model)

Table 110.9: Performance of CG and PCG Method (Soil-Beam FEM)

| 8842 DOFs Soil-Beam FEM (Static) | | | | | |
|---|---|---|---|---|---|
| Preconditioner | # Iter | Pre Time(s) | Iter Time(s) | Total Time(s) | Density[4] |
| - | 3274 | - | 102.5 | 102.5 | - |
| Jacobi | 1687 | 0.01 | 54.56 | 54.57 | - |
| IC(0) | 26 | 15.77 | 1.95 | 17.72 | 1.00 |
| IC(1e-6) | 6 | 110.17 | 2.79 | 112.96 | 15.11 |
| RIF2(1e-6)[5] | 23 | 3364.8 | 3.44 | 3368.24 | 4.32 |
| RIF3(1e-6)[5] | 31 | 34541 | 9.26 | 34550.26 | 16.37 |
| 8842 DOFs Soil-Beam FEM (Dynamic) | | | | | |
| Preconditioner | # Iter | Pre Time(s) | Iter Time(s) | Total Time(s) | Density |
| - | 3276 | - | 136.7 | 136.7 | - |
| Jacobi | MaxIt | | | | |
| IC(0) | MaxIt | | | | |
| IC(1e-6) | MaxIt | | | | |
| RIF2(1e-2) | MaxIt | | | | |
| RIF3(1e-2) | MaxIt | | | | |

[4]Density is defined as the number of non-zeros of the incomplete factor divided by the number of non-zeros in the lower triangular part of $A$.

[5]Iteration with tolerance 1e-2 failed to converge.

### 110.5.6 Conclusion and Future Work

1. For the soil-structure interaction problems investigated in this section, Conjugate Gradient method works fine and the convergence is acceptable for most cases.

2. Incomplete Cholesky factorization preconditioner has been shown to be very powerful in static pushover problems.

3. Dynamic problems formulated by Newmark integration scheme have not been extensively tested. But according to the data available so far, neither IC nor RIF preconditioners performed well and further testing is necessary to reach a more persuasive conclusion. The difficulty in dynamic analysis results from the fact that consistent mass and damping matrices used in continuum finite element formulations significantly degrade the conditioning number of the final system. This situation deteriorates when penalty handler is used to apply multiple point constraints, which introduces huge off-diagonal numbers to stiffness, mass and damping matrices (Cook et al., 2002).

4. Robust incomplete factorization preconditioning based on A-orthogonalization has not been shown competitive with IC preconditioners in this research. It is also worth noting that all timings are

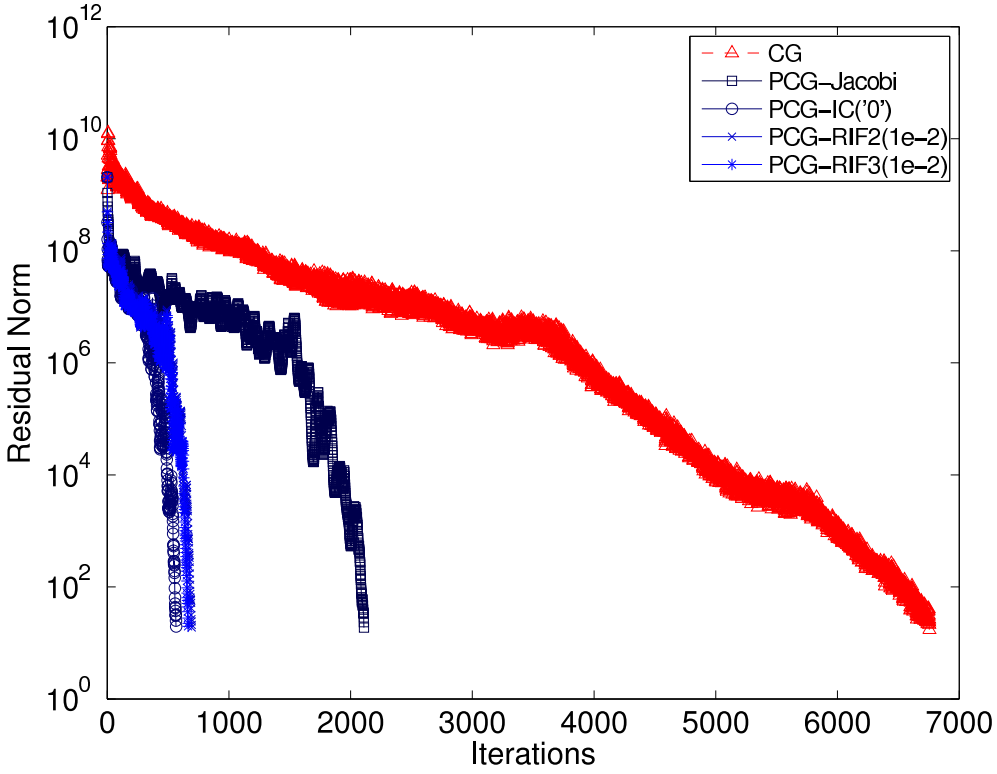



Figure 110.65: Convergence of CG and PCG Method (33081 DOFs Model)

taken in MATLAB. There are much more improvement can be achieved with a carefully coded FORTRAN program.

5. Static analysis has been extensively studied and it can be safely concluded that IC(0) and Jacobi preconditioners are good choices for the nonlinear soil-beam interaction simulations.

6. Dynamic analysis has also been studied but more work is needed to draw any detailed conclusion. Generally speaking, one should be alert if iterative solver is to be used for dynamic analysis. This partially comes from the fact that mass and damping matrices undoubtedly alter the structures of the coefficient matrix. This situation becomes more complicated if penalty handler is used to introduce off-diagonal numbers when handling multi-point constraints. So direct solver would be a more stable option for solving dynamic equations.

## 110.6 Performance Study on Parallel Direct/Iterative Solving in SFSI

The motivation of this section is to introduce a robust and efficient parallel equation solver into our parallel finite element analysis framework. Aside from sparsity, which has been well known as the

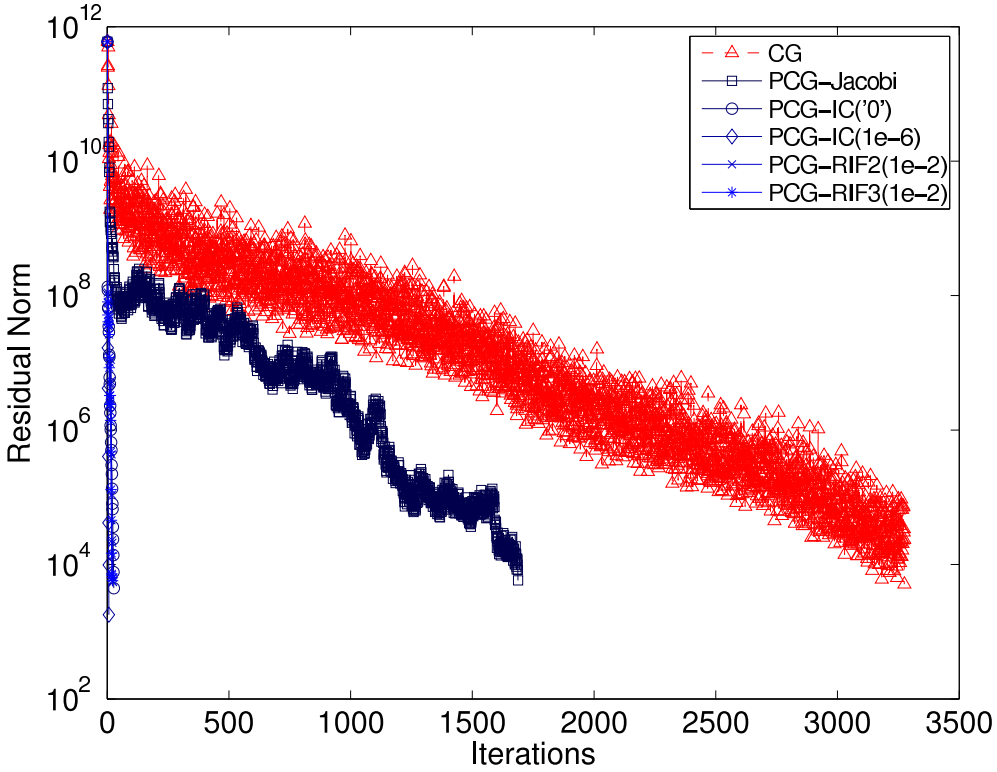



Figure 110.66: Convergence of CG and PCG Method (Soil-Beam Static Model)

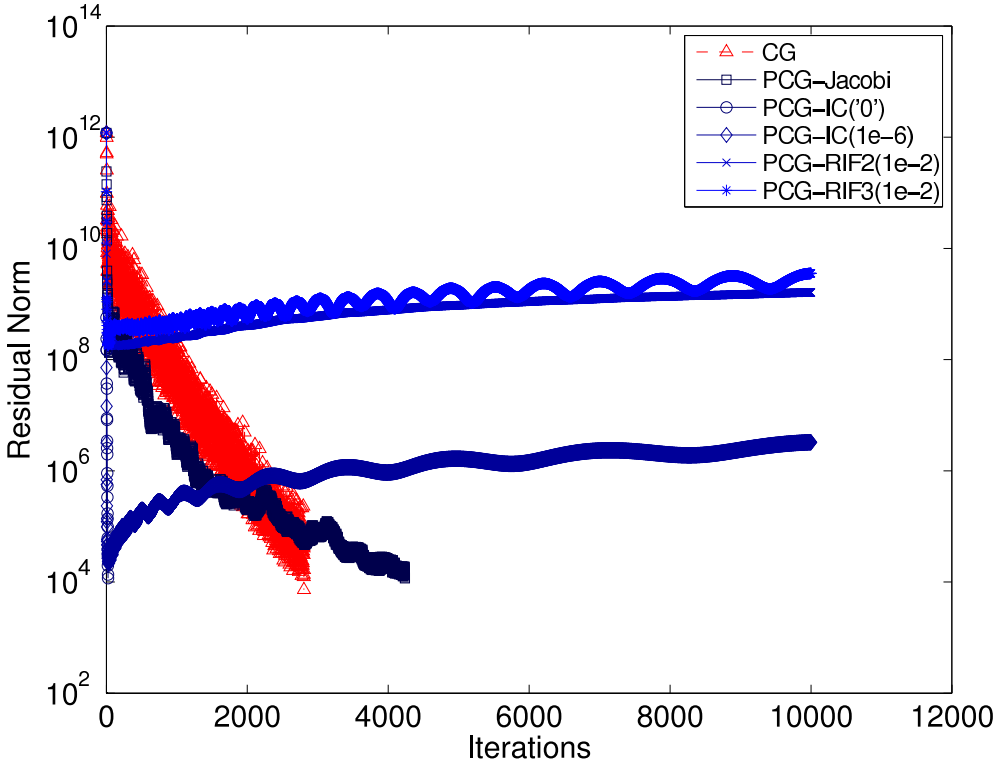



Figure 110.67: Convergence of CG and PCG Method (Soil-Beam Dynamic Model)

result of compact support that is inherent with finite element method, there exist some other special considerations that make the equation solving in finite element simulation a more involved problem.

In nonlinear finite element simulations, handling of constraints significantly affects the condition number of assembled equation systems. In SFSI simulations, multiple-point constraint is necessary to enforce the connection between soil and pile elements. In this research, penalty handler has been adopted to impose multiple point constraints on the assembled equation systems. Transformation and Lagrange multipliers are among those popular methods as well (Belytschko et al., 2000; Cook et al., 2002). The method of Lagrange multipliers adds extra constraints to the system and the resulted coefficient matrix will lose symmetric positive definiteness. Transformation is favorable especially in the sense that it reduces the order of the equation systems by condensing out slave/constrained DOFs. But the transformation is the most difficult to code and the situation of one single main/retained node with multiple follower/constrained nodes further complicates the problem.

Penalty method is chosen in this research due to the fact that it well preserves the symmetric positive definiteness of the system if the nice property is observed. Another consideration comes from the easiness with which the penalty methods can handle the single main/retained multiple follower/constrained situations. This is proven to be extremely valuable when data redistribution is required in adaptive parallel processing because the DOF_Graph object can be clearly tracked during partition and repartition phases.

The incapability of handling constraints accurately has been long known as the weakness of penalty method. The choice of the key penalty number seems arbitrary and largely depends on experience. The dilemma is with larger penalty number, the system can handle constraints more accurately while the coefficient matrix can become very ill-conditioned. This can lead to serious convergence problem for iterative solvers.

The majority of coefficient matrices resulted from finite element analysis are inherently symmetric positive definite, for which lots of numerical algorithms have been proposed and solving SPD, symmetric or closely symmetric systems has been relatively maturer than more common unsymmetric cases. Unfortunately, in geotechnical finite element simulations, unassociated constitutive models lead to unsymmetric stiffness matrices (Jeremić, 2004). More general parallel solvers must be coded to solve the problem.

In this section, both iterative and direct solvers are coded using the consistent PETSc interface (Balay et al., 2001, 2004, 1997). Popular direct solvers for general unsymmetric systems such as MUMPS, SPOOLES, SuperLU, PLAPACK have been introduced and performance study has been carried out to investigate the efficiency of different solvers on large scale SFSI simulations with penalty-handled unsymmetric equation systems. GMRES is always the first choice of iterative method when general unsymmetric systems are concerned. Preconditioning techniques have been thoroughly studied in this

research to explore possible advantage of preconditioned iterative solver over direct solving. Jacobi, incomplete LU decomposition and approximate inverse preconditioners represent the most popular choices for Krylov methods and they are chosen in this performance survey.

All numerical algorithms have been implemented through interface of PETSc, which provides a consistent platform on which implementation issues can be avoided to expose individual algorithmic performance.

### 110.6.1 Parallel Sparse Direct Equation Solvers

The methods that we consider for the solution of sparse linear equations can be grouped into four main categories: general techniques, frontal methods, multifrontal approaches and supernodal algorithms (Dongarra et al., 1996).

#### 110.6.1.1 General Techniques – SPOOLES

The so-called general approach can be viewed as parallel versions of sparse LU decomposition. Special cares must be taken to handle the sparse data structures. Sparsity ordering is crucial in parallel sparse equation solving in order to reduce fill-in and discover large-grain parallelism (Demmel et al., 1993).

Freely available package SPOOLES provides minimum degree (multiple external minimum degree (Liu, 1985)), generalized nested dissection and multisection ordering schemes for matrix sparsity ordering. Fundamental supernode tree built on top of vertex elimination tree is used to explore granularity in parallel (Ashcraft, 1999; Ashcraft et al., 1999).

#### 110.6.1.2 Frontal and Multifrontal Methods – MUMPS

Frontal methods have their origins in the solution of finite element problems from structural analysis. The usual way to describe the frontal method is to view its application to finite element problems where the matrix $A$ is expected as a sum of contributions from the elements of a finite element discretization (Dongarra et al., 1996). That is,

$$A = \sum_{l=1}^{m} A^{[l]}, \tag{110.28}$$

where $A^{[l]}$ is nonzero only in those rows and columns that correspond to variables in the $l$th element. If $a_{ij}$ and $a_{ij}^{[l]}$ denote the $(i,j)$th entry of $A$ and $A^{[l]}$, respectively, the basic assembly operation when forming $A$ is of the form

$$a_{ij} \Leftarrow a_{ij} + a_{ij}^{[l]}. \tag{110.29}$$

It is evident that the basic operation in Gaussian elimination

$$a_{ij} \Leftarrow a_{ij} + a_{ip}[a_{pp}]^{-1}a_{pj}. \tag{110.30}$$

may be performed as soon as all the the terms in the triple product 110.30 are fully summed (that is, are involved in no more sums of the form 110.29). The assembly and Gaussian elimination processes can therefore be interleaved and the matrix $A$ is never assembled explicitly. This allows all intermediate working to be performed in a dense matrix, termed frontal matrix, whose rows and columns correspond to variables that have not yet been eliminated but occur in at least one of the elements that have been assembled.

For general problems other than finite element, the rows of $A$ (equations) are added into the frontal matrix one at a time. A variable is regarded as fully summed whenever the equation in which it last appears is assembled. The frontal matrix will, in this case, be rectangular.

The idea of multifrontal method is to couple a sparsity ordering with the efficiency of a frontal matrix kernel so allowing good exploitation of high performance computers. The basic approach is to develop separate fronts simultaneously which can be chosen using a sparsity preserving ordering such as minimum degree.

Elimination tree, again is the most important notion in the factorization process and also utilized to discover the potential of parallelism. An elimination tree defines the a precedence order within the factorization. The factorization commences at the leaves of of the tree and data is passed towards the root along the edges in the tree. To complete the work associated with a node, all the data must have been obtained from the children of the node, otherwise work at different nodes is independent.

Freely available package MUMPS (MUltifrontal Massively Parallel sparse direct Solver) has been used in this research to investigate the performance of multifrontal methods (http://graal.ens-lyon.fr/MUMPS/, 2006).

MUMPS is a package for solving systems of linear equations of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a square sparse matrix that can be either unsymmetric, symmetric positive definite, or general symmetric. MUMPS uses a multifrontal technique which is a direct method based on either the $\mathbf{LU}$ or the $\mathbf{LDL^T}$ factorization of the matrix. MUMPS exploits both parallelism arising from sparsity in the matrix $\mathbf{A}$ and from dense factorizations kernels.

The main features of the MUMPS package include the solution of the transposed system, input of the matrix in assembled format (distributed or centralized) or elemental format, error analysis, iterative refinement, scaling of the original matrix, and return of a Schur complement matrix. MUMPS offers several built-in ordering algorithms, a tight interface to some external ordering packages such as METIS and PORD, and the possibility for the user to input a given ordering. Finally, MUMPS is available in

various arithmetics (real or complex, single or double precision).

The software is written in Fortran 90 although a C interface is available. The parallel version of MUMPS requires MPI for message passing and makes use of the BLAS, BLACS, and ScaLAPACK libraries. The sequential version only relies on BLAS.

MUMPS distributes the work tasks among the processors, but an identified processor (the host) is required to perform most of the analysis phase, to distribute the incoming matrix to the other processors (slaves) in the case where the matrix is centralized, and to collect the solution. The system $\mathbf{Ax} = \mathbf{b}$ is solved in three main steps:

1. Analysis. The host performs an ordering based on the symmetrized pattern $\mathbf{A} + \mathbf{A}^{\mathbf{T}}$, and carries out symbolic factorization. A mapping of the multifrontal computational graph is then computed, and symbolic information is transferred from the host to the other processors. Using this information, the processors estimate the memory necessary for factorization and solution.

2. Factorization. The original matrix is first distributed to processors that will participate in the numerical factorization. The numerical factorization on each frontal matrix is conducted by a main compute processor (determined by the analysis phase) and one or more slave processors (determined dynamically). Each processor allocates an array for contribution blocks and factors; the factors must be kept for the solution phase.

3. Solution. The right-hand side $\mathbf{b}$ is broadcast from the host to the other processors. These processors compute the solution $\mathbf{x}$ using the (distributed) factors computed during Step 2, and the solution is either assembled on the host or kept distributed on the processors.

Each of these phases can be called separately and several instances of MUMPS can be handled simultaneously. MUMPS allows the host processor to participate in computations during the factorization and solve phases, just like any other processor.

For both the symmetric and the unsymmetric algorithms used in the code, a fully asynchronous approach with dynamic scheduling of the computational tasks has been chosen. Asynchronous communication is used to enable overlapping between communication and computation. Dynamic scheduling was initially chosen to accommodate numerical pivoting in the factorization. The other important reason for this choice was that, with dynamic scheduling, the algorithm can adapt itself at execution time to remap work and data to more appropriate processors. In fact, the main features of static and dynamic approaches have been combined and the estimation obtained during the analysis to map some of the main computational tasks has been used; the other tasks are dynamically scheduled at execution time. The main data structures (the original matrix and the factors) are similarly partially mapped according to the analysis phase.

#### 110.6.1.3 Supernodal Algorithm – SuperLU

The left-looking or column Cholesky algorithm can be implemented for sparse system and can be blocked by using a supernodal formulation. The idea of a supernode is to group together columns with the same nonzero structure, so they can be treated as a dense matrix for storage and computation. Supernodes were originally used for (symmetric) sparse Cholesky factorization (Demmel et al., 1999a). In the factorization $\mathbf{A} = \mathbf{LL}^{\mathbf{T}}$ (or $\mathbf{A} = \mathbf{LDL}^{\mathbf{T}}$), a supernode is a range $(r : s)$ of columns of $\mathbf{L}$ with the same nonzero structure below the diagonal; that is, $\mathbf{L}(r : s; r : s)$ is full lower triangular and every row of $\mathbf{L}(r : s; r : s)$ is either full or zero.

Then in left-looking Cholesky algorithm, all the updates from columns of a supernode are summed into a dense vector before the sparse update is performed. This reduces indirect addressing and allows the inner loops to be unrolled. In effect, a sequence of col-col updates is replaced by a supernode-column (sup-col) update. The sup-col update can be implemented using a call to a standard dense Level 2 BLAS matrix-vector multiplication kernel. This idea can be further extended to supernode-supernode (sup-sup) updates, which can be implemented using a Level 3 BLAS dense matrix-matrix kernel. This can reduce memory traffic by an order of magnitude, because a supernode in the cache can participate in multiple column updates (Demmel et al., 1999a). It has been reported in (Ng and Peyton, 1993) that a sparse Cholesky algorithm based on sup-sup updates typically runs 2.5 to 4.5 times as fast as a col-col algorithm. Indeed, supernodes have become a standard tool in sparse Cholesky factorization.

To sum up, supernodes as the source of updates help because of the following (Demmel et al., 1999a):

1. The inner loop (over rows) has no indirect addressing. (Sparse Level 1 BLAS is replaced by dense Level 1 BLAS.)

2. The outer loop (over columns in the supernode) can be unrolled to save memory references. (Level 1 BLAS is replaced by Level 2 BLAS.)
   Supernodes as the destination of updates help because of the following:

3. Elements of the source supernode can be reused in multiple columns of the destination supernode to reduce cache misses. (Level 2 BLAS is replaced by Level 3 BLAS.)

Supernodes in sparse Cholesky can be determined during symbolic factorization, before the numeric factorization begins. However, in sparse $\mathbf{LU}$, the nonzero structure cannot be predicted before numeric factorization, so supernodes must be defined dynamically. Furthermore, since the factors $\mathbf{L}$ and $\mathbf{U}$ are no longer transposes of each other, the definition of a supernode must be generalized.

Freely available package SuperLU proposed a couple of ways to generalize the symmetric definition of supernodes to unsymmetric factorization (Demmel et al., 1999a). It is now not possible to use Level 3 BLAS efficiently for unsymmetric systems. The implementation in SuperLU performs a dense matrix multiplication of a block of vectors and, although these can not be written as another dense matrix, it has been shown that this Level 2.5 BLAS has most of the performance characteristics of Level 3 BLAS since the repeated use of the same dense matrix allows good use of cache and memory hierarchy.

There are three versions of libraries collectively referred as SuperLU (Demmel et al., 2003),

- Sequential SuperLU is designed for sequential processors with one or more layers of memory hierarchy (caches).
- Multithreaded SuperLU (SuperLU_MT) is designed for shared memory multiprocessors (SMPs), and can effectively use up to 16 or 32 parallel processors on sufficiently large matrices in order to speed up the computation (Demmel et al., 1999b).
- Distributed SuperLU SuperLU_DIST is designed for distributed memory parallel processors, using MPI for interprocess communication. It can effectively use hundreds of parallel processors on sufficiently large matrices (Li and Demmel, 2003).

Parallelizing sparse direct solver for unsymmetric systems is more complicated than parallel sparse Cholesky case. The advantage of sparse Cholesky over the unsymmetric case is that pivots can be chosen in any order from the main diagonal while guaranteeing stability. This lets us perform pivot choice before numerical factorization begins, in order to minimize fill-in, maximize parallelism. precompute the nonzero structure of the Cholesky factor, and optimize the (2D) distributed data structures and communication pattern (Li and Demmel, 2003).

In contrast, for unsymmetric or indefinite systems, distributed memory codes can be much more complicated for at least two reasons. First and foremost, some kind of numerical pivoting is necessary for stability. Classical partial pivoting or the sparse variant of threshold pivoting typically cause the fill-ins and workload to be generated dynamically during factorization. Therefore, we must either design dynamic data structures and algorithms to accommodate these fill-ins, or else use static data structures which can grossly overestimate the true fill-in. The second complication is the need to handle two factored matrices $\mathbf{L}$ and $\mathbf{U}$, which are structurally different yet closely related to each other in the filled pattern. Unlike the Cholesky factor whose minimum graph representation is a tree (elimination tree), the minimum graph representations of the $\mathbf{L}$ and $\mathbf{U}$ factors are directed acyclic graphs (elimination DAGs).

In SuperLU_DIST, a static pivoting approach, called GESP (Gaussian Elimination with Static Pivoting) (Li and Demmel, 1998) is used. In order to parallelize the GESP algorithm, a 2D block-cyclic

mapping of a sparse matrix to the processors is used. An efficient pipelined algorithm is also designed to perform parallel factorization. With GESP, the parallel algorithm and code are much simpler than dynamic pivoting.

The main algorithmic features of SuperLU_DIST solver are summarized as follows (Li and Demmel, 2003):

- supernodal fan-out (right-looking) based on elimination DAGs,
- static pivoting with possible half-precision perturbations on the diagonal,
- use of an iterative algorithm using the LU factors as a preconditioner, in order to guarantee stability,
- static 2D irregular block-cyclic mapping using supernodal structure, and
- loosely synchronous scheduling with pipelining.

In particular, static pivoting can be performed before numerical numerical factorization, allowing us to use all the techniques in good sparse Cholesky codes: choice of a (symmetric) permutation to minimize fill-in and maximize parallelism, precomputation of the fill pattern and optimization of 2D distributed data structures and communication patterns. Users are referred to Li and Demmel (2003) for algorithm details.

### 110.6.2 Performance Study on SFSI Systems

In this section, performance study on popular parallel direct and iterative solvers has been conducted. The purpose is to provide some guidelines on appropriate use of different solvers with the parallel finite simulation framework. Matrix systems from SFSI analysis are used as test cases. The performance investigation uses IA64 Intel-based cluster at SDSC.

#### 110.6.2.1 Equation System

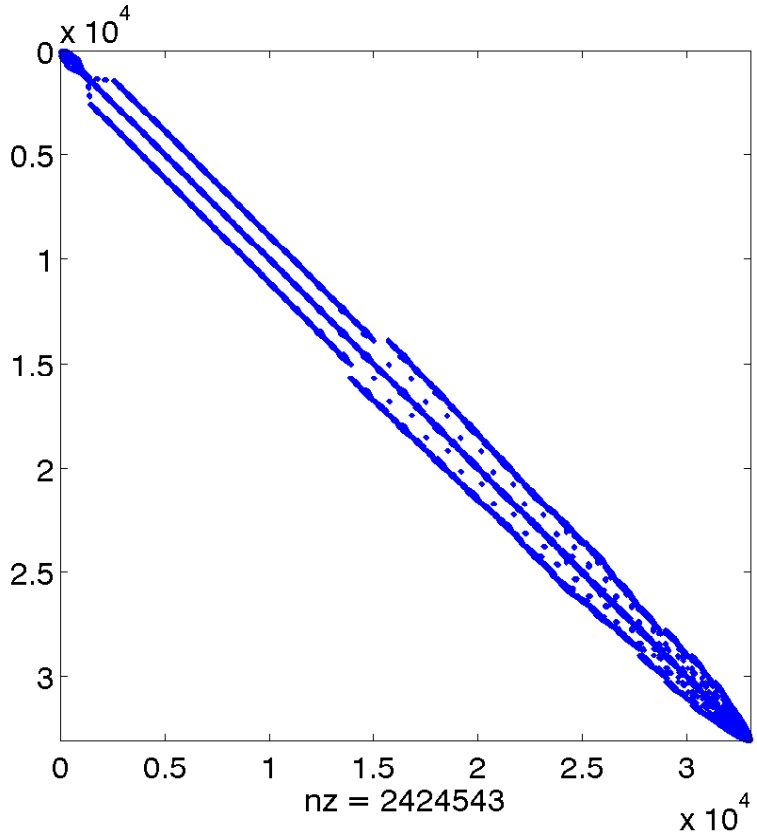



Figure 110.68: Matrices $N = 33081$ (Continuum FEM)

#### 110.6.2.2 Performance Results

Table 110.10: Performance Study on SFSI Systems ($N$=33081)

| Direct Solvers | | |
|---|---|---|
| Solvers | Num of CPUs | Time (s) |
| MUMPS | 4 | 6.0312 |
| | 8 | 7.0534 |
| | 16 | 5.3472 |
| SuperLU_DIST | 4 | 20.358 |
| | 8 | 13.803 |
| | 16 | 13.755 |
| SPOOLES | 4 | 10.696 |
| | 8 | 7.5338 |
| | 16 | 6.2448 |
| Iterative Solvers (GMRES) | | |
| Preconditioner | Num of CPUs | Time (s) |
| Jacobi | 16 | 96.441 |
| Parallel ILU(0) | 4 | 277.49 |
| | 8 | 276.07 |
| | 16 | 135.78 |

### 110.6.3 Conclusion

This section presents the parallel solvers implemented in parallel finite element framework. Table 110.10, draws several conclusions about appropriate use of solvers:

- Direct solvers outperforms the iterative solver significantly for general cases. It is worthwhile to note that nonsymmetric solvers are used here due to their generality. For special cases such as SPD system, preconditioned CG will show much better performance.
- The Conjugate Gradient method applies only to Symmetric Positive Definite (SPD) system. This puts restriction on the material models we can use in our simulations. Generally speaking, elastic material will yield a SPD stiffness matrix. Plastic material with associative flow rule also satisfies

this category. Plastic material with non-associative flow rule has non-symmetric element stiffness matrix and so will be the global coefficient matrix of the equation system.

- Another category of matrix that deserves attention is the stiffness matrix from softening materials, which possesses at least one negative eigenvalue so the SPD property will be broken. For advanced geo-materials subject to complicated loadings, as the material develops nonlinearity, the condition of stiffness matrix might vary greatly from SPD (elastic phase), to singular (elastic-perfectly-plastic), and non-symmetric non-positive-definite (elastic-non-associative-plastic-softening) cases. This poses another challenge when one tries to use iterative solver for production runs. The unpredictability of stiffness matrix will disable the application of powerful solvers such as Conjugate Gradient for iterative case and Cholesky for direct case.

- The reason why iterative solver exhibits poor efficiency is partly due to the problem size. We can also see from the Table 110.10 that parallel direct solver is not scalable in general. Iterative solver, on the other hand, is more scalable and it is safe to project that when the size of matrix increases, iterative solver has the advantage from the memory requirement point of view.

Parallel equation solving itself is a complicated topic in numerical computing community. In this section, the main purpose is to introduce a robust and generally efficient parallel solver for finite element simulations. So in this sense, parallel direct solvers such as MUMPS and SPOOLES are recommended.

# Chapter 111

# Solid, Structure – Fluid Interaction

(2017-2019-2021-)

(In collaboration with Dr. Hexiang Wang)

## 111.1 Chapter Summary and Highlights

## 111.2 Introduction

The analysis of problems several civil engineering fields often requires a study of fluid-structure systems that are excited by dynamic loads. For example, the evaluation of the structural integrity of nuclear reactor components involves the analysis of structures of complex shape and their interaction with the fluid in which they are embedded Donea et al. (1982). In these cases, both the fluid and the structure might undergo non-linear response.

These needs for safety evaluations have motivated the development of computational methods capable to treat transient, non-linear fluid-structure interaction problems. Donea et al. (1982) presented an arbitrary Lagrangian-Eulerian (ALE) finite element method with automatic and continuous rezoning technique of the fluid mesh. With this method, dynamic response of nuclear reactor under solid fluid interaction has been simulated. Recently Park et al. (2014) examined the modal characteristics of Reactor vessel internals (RVIs) based on scale-similarity analysis with fluid-structure interaction (FSI). It was observed that the added-mass (A-M) model for submerged structures is considerably dependent on mode shapes and natural frequencies. Sigrist et al. (2006) conducted comparative dynamic analysis with FSI modeling for pressure vessel and internals in a nuclear reactor. They proved that the coupling effect is significant, whereas the effect of added-stiffness on global behavior is negligible. Je et al. (2017) stressed that improvement of numerical analysis methods has been required to solve complicated phenomena that occur in nuclear facilities. Particularly, fluid-structure interaction (FSI) behavior should be resolved for accurate design and evaluation of complex reactor vessel internals (RVIs) submerged in coolant. They investigated the FSI effect on dynamic characteristics of RVIs in a typical 1,000 MWe nuclear power plant. Modal analyses of an integrated assembly were conducted by employing the fluid-structure (F-S) model as well as the traditional added-mass model.

Though numerous efforts have been made on simulation of SFI problem in nuclear reactors, to the author's best knowledge, high fidelity modeling of this dynamic nonlinear phenomena with complex geometry is still unavailable. Full sets of Navier-stokes (N-S) equation is rarely solved for fully solid fluid coupling. Instead, many simplified analysis procedure is adopted: like added-mass (A-M) model (as shown in equation 111.1) Park et al. (2014); Sigrist et al. (2006) and simplified acoustic wave equation Je et al. (2017) (equation 111.2). These simplified methods introduce great modeling uncertainty to the simulation system.

$$
\begin{aligned}
[M_s]\{\ddot{u}\} + [C_s]\{\dot{u}\} + [K_s]\{u\} &= \{f_e\} + \{f_f\} \\
f_f = \int_S \{N_p\}^T \{n\}\{P\} dS &= -[M_a]\{\ddot{u}\}
\end{aligned}
\tag{111.1}
$$

$$
[M_f]\{\ddot{P}\} + [C_f]\{\dot{P}\} + [K_f]\{P\} + \rho[R_{int}]^T\{\ddot{u}\} = 0 \tag{111.2}
$$

On the other hand, ALE method, which was originally put forward as a powerful tool for fluid dynamics with deforming boundary Hirt et al. (1974), has been applied to fully solve the coupled N-S equation and solid mechanic equation Le Tallec and Mouro (2001); Murea and Sy (2017). The freedom in moving the fluid mesh offered by the ALE formulation is very attractive. However, it can be overshadowed by the burden of specifying grid velocities, well suited to a particular problem. As a consequence, the practical implementation of the ALE description requires that an automatic mesh displacement prescription algorithm to be supplied. Many methods have been put forward to overcome this difficulty. For example, pseudo-solid method was adopted by Van Loon et al. (2007) and Jasak and Tukovic (2006) came up with a simplified procedure by solving a Laplacian equation 111.3 of grid velocity with finite element discretization.

$$
\begin{aligned}
\boldsymbol{\nabla} \cdot (\gamma \boldsymbol{\nabla} \boldsymbol{u}) &= \mathbf{0} \\
\boldsymbol{x}_{new} &= \boldsymbol{x}_{old} + \boldsymbol{u}\Delta t
\end{aligned}
\tag{111.3}
$$

Seemingly, introducing these additional equations to specify the movement of fluid mesh can well resolve the inherent problem of ALE method. Based on specified grid velocity, solutions to ALE-formed N-S equations 111.4 can give precise response of fluid flow.

$$
\begin{aligned}
\frac{\partial}{\partial t}(\rho J) &= J\boldsymbol{\nabla} \cdot (\boldsymbol{w} - \boldsymbol{v}) \\
\frac{\partial}{\partial t}(\rho \boldsymbol{v} J) &= J\boldsymbol{\nabla} \cdot \boldsymbol{v}(\boldsymbol{w} - \boldsymbol{v}) + J(\rho \boldsymbol{b} - \boldsymbol{\nabla} p)
\end{aligned}
\tag{111.4}
$$

However, mathematically the system equations (seen in section 111.3.1) to represent physical phenomena of SFI itself is sufficient and complete. Theoretically, no additional equations need to be added to the coupled system. The authors think that the ease gained here by introducing extra mesh movement equations to ALE method is sacrificed with the accuracy of the result of pressure field. Because in equation 111.4, the pressure is dependent on both absolute velocity $\boldsymbol{v}$ and relative velocity $(\boldsymbol{v} - \boldsymbol{w})$. Different configuration of mesh velocity can result in different relative velocity under the same absolute velocity, which in turn causes different pressure field.

This could be fine for pure fluid dynamics problem, where engineers care more about the fluid flow (i.e. velocity field). However, this may not be good enough for SFI, where hydrodynamic pressure at solid fluid interface is of great of importance for accurate Neumann boundary condition of solid domain. Therefore, precise pressure field is desired and indispensable in high-fidelity simulation of SFI.

The research presented here aims at realistic SFI modeling in nuclear reactors with solutions to fully coupled FSI system (i.e. N-S equations, solid mechanics equations and interface constraint equations). Geometric conformity is also achieved in this work. The great emphasis was put on accurate pressure field at solid fluid interface. A full sets of verification and validation tests are provided to guarantee the reliability of our modeling.

The limitation of current work is that relatively large displacement of solid fluid interface and accompanying Eulerian mesh distortion problem are not well resolved. Further development are needed for these topics.

## 111.3 Theoretical Formulation

### 111.3.1 Solid Fluid Interaction

The mathematical description of physical phenomenon of solid fluid interaction includes three parts Van Loon et al. (2007): The response of solid domain $\Omega_s$ is controlled by the theory of general continuum solid mechanics (equation 111.5). The governing equation in fluid domain is Navier-Stokes equation (N-S equations), which basically consists of mass conservation and momentum conservation equation (shown in equation 111.6). In the equations below, symbols $\boldsymbol{u}$, $\boldsymbol{\sigma}$, $\boldsymbol{f}$ , $p$, $G$, $\rho$ and $\eta$ denote velocity, Cauchy stress tensor, body force, pressure, solid shear modulus, density and fluid viscosity. $\boldsymbol{F}$ is deformation gradient tensor defined as $\boldsymbol{F} = \nabla\chi(X,t)$.

$$\rho^s \frac{d\boldsymbol{u^s}}{dt} = \boldsymbol{\nabla}\cdot\boldsymbol{\sigma^s} + \rho^s\boldsymbol{f^s} \quad in\ \Omega_s \tag{111.5a}$$

$$det(\mathrm{F}) = 1 \quad in\ \Omega_s \tag{111.5b}$$

$$\boldsymbol{\sigma^s} = G(\boldsymbol{F}\cdot\boldsymbol{F^T} - \boldsymbol{I}) - p^s\boldsymbol{I} \quad in\ \Omega_s \tag{111.5c}$$

$$\rho^f \frac{\partial\boldsymbol{u^f}}{\partial t} + = \boldsymbol{\nabla}\cdot\boldsymbol{\sigma^f} + \rho^f\boldsymbol{f^f} \quad in\ \Omega_f \tag{111.6a}$$

$$\boldsymbol{\nabla}\cdot\boldsymbol{u^f} = 0 \quad in\ \Omega_f \tag{111.6b}$$

$$\boldsymbol{\sigma^f} = 2\eta\boldsymbol{D}(\boldsymbol{u^f}) - p^f\boldsymbol{I} \quad in\ \Omega_f \tag{111.6c}$$

At the solid fluid interface $D = \partial\Omega_s \cap \partial\Omega_f$, kinematic and dynamic constraints should be met, as shown in equation 111.7.

$$\boldsymbol{u^s - u^f = 0} \quad in\ D \tag{111.7a}$$

$$\boldsymbol{\sigma^s \cdot n + \sigma^f \cdot n = 0} \quad in\ D \tag{111.7b}$$

### 111.3.2 Finite Volume Discretization

For general purpose, the standard form pf the transport equation for a scalar property $\phi$ is considered here in 111.8. It is a second order equation as the diffusion term includes the second derivative of $\phi$ in space. Finite volume discretization will be applied to the integral form regarding to control volume $V_p$ (equation 111.9) in both spatial and temporal sense Moukalled et al. (2016).

$$\underbrace{\frac{\partial \rho\phi}{\partial t}}_{\text{temporal derivative}} + \underbrace{\boldsymbol{\nabla} \cdot (\rho \boldsymbol{U} \phi)}_{\text{convection term}} - \underbrace{\boldsymbol{\nabla} \cdot (\rho \Gamma_\phi \boldsymbol{\nabla} \phi)}_{\text{diffusion term}} = \underbrace{S_\phi(\phi)}_{\text{source term}} \tag{111.8}$$

$$\underbrace{\int_{V_p} \frac{\partial \rho\phi}{\partial t} dV}_{\text{temporal derivative}} + \underbrace{\int_{V_p} \boldsymbol{\nabla} \cdot (\rho \boldsymbol{U} \phi) dV}_{\text{convection term}} - \underbrace{\int_{V_p} \boldsymbol{\nabla} \cdot (\rho \Gamma_\phi \boldsymbol{\nabla} \phi) dV}_{\text{diffusion term}} = \underbrace{\int_{V_p} S_\phi(\phi) dV}_{\text{source term}} \tag{111.9}$$

- Spatial discretization

  The spatial discretization of equation 111.9 includes three parts: discretization of convection term, diffusion term and source term, respectively. Using divergence theorem, the semi-discrete form of convection term at arbitrary control volume $V_p$ can be given in equation 111.10, where $F$ denotes the mass flux through the face defined in equation 111.11. In this semi-discrete form, we still need to calculate the face value $\phi_f$ and face mass flux $F$ in order to evaluate the whole volume integral of convection term.

$$\int_{V_p} \boldsymbol{\nabla} \cdot (\rho \boldsymbol{U} \phi) dV = \sum_f \boldsymbol{S} \cdot (\rho \boldsymbol{U} \phi)_f \tag{111.10a}$$

$$= \sum_f \boldsymbol{S} \cdot (\rho \boldsymbol{U})_f \phi_f \tag{111.10b}$$

$$= \sum_f F \phi_f \tag{111.10c}$$

$$F = \boldsymbol{S} \cdot (\rho \boldsymbol{U})_f \tag{111.11}$$

The face value $\phi_f$ at face center $f$ can be obtained from face interpolation scheme using $\phi$ value at control volume center $\phi_p$ and value at neighboring volume center $\phi_N$ (equation 111.12).

$$\phi_f = f_x \phi_p + (1 - f_x) \phi_N \tag{111.12}$$

Here $f_x$ is the interpolation factor defined as the ratio of distances $\overline{fN}$ and $\overline{PN}$:

$$f_x = \frac{\overline{fN}}{\overline{PN}} \tag{111.13}$$

Similarly, for diffusion term the semi-discrete form is presented in equation 111.14. With the semi-discrete form, $(\boldsymbol{\nabla} \phi)_f$ still needs to be evaluated to achieve full discretization. For orthogonal mesh, following equation 111.15 can be used to simplify our analysis, where $|d|$ is the magnitude of vector $\overline{PN}$.

$$\int_{V_p} \boldsymbol{\nabla} \cdot (\rho \Gamma_\phi \boldsymbol{\nabla} \phi) dV = \sum_f \boldsymbol{S} \cdot (\rho \Gamma_\phi \boldsymbol{\nabla} \phi)_f \tag{111.14a}$$

$$= \sum_f (\rho \Gamma_\phi)_f \boldsymbol{S} \cdot (\boldsymbol{\nabla} \boldsymbol{\phi})_f \tag{111.14b}$$

$$\boldsymbol{S} \cdot (\nabla \phi)_f = |\boldsymbol{S}| \frac{\phi_N - \phi_P}{|\boldsymbol{d}|} \tag{111.15}$$

The source term $S_\phi(\phi)$ can be a general function of $\phi$. Before the actual discretization, the source term is first linearized, where $S_u$ and $S_p$ also depend on $\phi$. Then the volume integral of source term can be evaluated with equation 111.17.

$$S_\phi(\phi) = Su + Sp\phi \tag{111.16}$$

$$\int_{V_p} S_\phi(\phi)dV = S_u V_p + S_p V_p \phi_p \tag{111.17}$$

- Temporal discretization

  Conducting time integration from $t$ to $t + \Delta t$ with both sides of equation 111.9 yields equation 111.18.

$$\int_t^{t+\Delta t} [\frac{\partial}{\partial t}\int_{V_p} \rho\phi dV + \int_{V_p} \boldsymbol{\nabla}\cdot(\rho \boldsymbol{U}\phi)dV - \int_{V_p} \boldsymbol{\nabla}\cdot(\rho\Gamma_\phi \boldsymbol{\nabla}\phi)]dt = \int_t^{t+\Delta t} (\int_{V_p} S_\phi(\phi)dV)dt \tag{111.18}$$

  Substituting the spatial semi-discretization shown above (equations 111.10, 111.14 and 111.17) into equation 111.18 and assuming that the control volumes do not change in time, equation 111.18 can be written as ;

$$\int_t^{t+\Delta t} [(\frac{\partial \rho\phi}{\partial t})_p V_p + \sum_f F\phi_f - \sum_f (\rho\Gamma_\phi)_f \boldsymbol{S}\cdot(\boldsymbol{\nabla}\phi)_f]dt = \int_t^{t+\Delta t} (S_u V_p + S_p V_p \phi_p)dt \tag{111.19}$$

  Here further temporal discretization are needed to evaluate equation 111.19:

$$(\frac{\partial \rho\phi}{\partial t})_p = \frac{\rho_P^n \phi_P^n - \rho_P^0 \phi_p^0}{\Delta t} \tag{111.20a}$$

$$\int_t^{t+\Delta t} \phi(t)dt = \frac{1}{2}(\phi^0 + \phi^n)\Delta t \tag{111.20b}$$

  where

$$\phi^n = \phi(t + \Delta t)$$

$$\phi^0 = \phi(t)$$

Assuming that the density and diffusivity do not change in time, the final semi-discrete form including both spatial and temporal discretization can be given in equation 111.21. Since in equation 111.21, $\phi_f^n$, $(\boldsymbol{\nabla}\phi)_f^n$ can be expressed with $\phi$ values at control cell $\phi_P$ and neighboring cells $\phi_N$ with equation 111.12 and 111.15. Therefore, equation 111.21 can be finalized into algebraic equation form (equation 111.22).

$$\begin{aligned}&\frac{\rho_P\phi_P^n-\rho_p\phi_P^0}{\Delta t}V_p+\frac{1}{2}\sum_f F\phi_f^n+\frac{1}{2}\sum_f F\phi_f^0\\&-\frac{1}{2}\sum_f(\rho\Gamma_\phi)_f\boldsymbol{S}\cdot(\boldsymbol{\nabla}\phi)_f^n-\frac{1}{2}\sum_f(\rho\Gamma_\phi)_f\boldsymbol{S}\cdot(\boldsymbol{\nabla}\boldsymbol{\phi})_f^0\\&=SuV_p+\frac{1}{2}S_pV_P\phi_P^n+\frac{1}{2}S_pV_P\phi_P^0\end{aligned}\tag{111.21}$$

$$a_P\phi_P^n+\sum_N a_N\phi_N^n=R_P\tag{111.22}$$

For every control volume, one equation of this form is assembled. The value of $\phi_P^n$ depends on the values in the neighboring cells, thus creating a system of algebraic equations 111.23, where [A] is a sparse matrix, with coefficients $a_P$ on the diagonal and $a_N$ off the diagonal, $[\phi]$ is the vector of $\phi$-s for all control volumes and $[R]$ is the source term vector.

$$[A][\phi]=[R]\tag{111.23}$$

### 111.3.3 Volume of Fluid Method

The physical phenomena of earthquake soil structure interaction in fluid domain is essentially free surface flow. There are three types of problems in the numerical treatment of free boundaries: (1) their discrete representation, (2) their evolution in time, and (3) the manner in which boundary conditions are imposed on them.

Volume of Fluid method is originally put forward by Hirt and Nichols (1981) to solve free surface flow problems (free boundary problems). Different from traditional one-phase method, VOF method considers more general two-phase flow problem. In special case where we take one phase as air and another phase as any liquid that needs to be simulated, then VOF becomes also applicable for free surface flow.

The idea of Volume of Fluid method is to introduce a new field variable - volume fraction value ($\alpha$) for each control volume, which is defined in equation ?? where $V_i$ is the total volume of cell $i$ and $V_i^w$ is the volume of water contained in the cell. From the definition, it can be seen that $\alpha$ values range

between 0 and 1. If the cell is completely filled with fluid then $\alpha = 1$ and if it is filled with void phase then its value should be 0. In VOF method, momentum equation and continuity equation are solved for one composite fluid phase characterized by volume fraction $\alpha$. The physical properties of this one fluid are calculated as weighted averages based on the volume fractions of two phases in one cell. E.g., the density of any point in the domain is calculated with equation 111.24.

$$\rho = \alpha\rho_w + (1-\alpha)\rho_a \tag{111.24}$$

The evolution of volume fraction $\alpha$ is controlled by an additional convection transport equation 111.26. By solving this equation, the distribution of volume fraction $\alpha$ can be obtained. Then the free fluid interface can be automatically identified as the zone where the volume fraction $\alpha$ ranges between 0 to 1. In this way, the fluid flow can be represented as the redistribution of $\alpha$ along with the time.

$$\alpha_i = \frac{V_i^w}{V_i} \tag{111.25}$$

$$\frac{\partial\alpha}{\partial t} + \boldsymbol{\nabla}\cdot(\alpha\boldsymbol{U}) = 0 \tag{111.26}$$

The free surface flow solver used here is interFoam implemented based on OpenFOAM. In interFoam, the necessary compression of the phase interface is achieved by introducing an extra artificial compression term into the transport convection equation of $\alpha$, as shown in equation 111.27. $U_r$ is the velocity field suitable to compress the interface. This artificial term is active only in the interface region due to the term $\alpha(1-\alpha)$.

$$\frac{\partial\alpha}{\partial t} + \boldsymbol{\nabla}\cdot(\alpha\boldsymbol{U}) + \boldsymbol{\nabla}\cdot(\alpha(1-\alpha)U_r) = 0 \tag{111.27}$$

### 111.3.4 Pressure-velocity coupling: PISO algorithm

For incompressible form of the fluid system (equation 111.28), two issues require special attention: non-linearity of the momentum equation and the pressure-velocity coupling Jasak (1996). The non-linear term in equation 111.29 is $\boldsymbol{\nabla}\cdot(\boldsymbol{UU})$, *i.e.* velocity is "being transported by itself". The discretized form of this expression would be quadratic in velocity and the resulting system of algebraic equations would therefore be non-linear. There are two possible solutions for this problem - either use a solver for non-linear systems, or linearize the convection term.

$$\boldsymbol{\nabla} \cdot \boldsymbol{U} = 0 \tag{111.28}$$

$$\frac{\partial \boldsymbol{U}}{\partial t} + \boldsymbol{\nabla} \cdot (\boldsymbol{U}\boldsymbol{U}) - \boldsymbol{\nabla} \cdot (\nu \boldsymbol{\nabla} \boldsymbol{U}) = -\boldsymbol{\nabla} p \tag{111.29}$$

According to section 111.3.2, convection term can be discretized with equation 111.30, where $F$, $a_P$ and $a_N$ are a function of $\boldsymbol{U}$. The important issue is that the fluxes F should satisfy the continuity equation 111.28. Linearization of the convection term implies than an existing velocity (flux) field that satisfies equation 111.28 will be used to calculate $a_P$ and $a_N$.

$$\begin{aligned} \boldsymbol{\nabla} \cdot (\boldsymbol{U}\boldsymbol{U}) &= \sum_f \boldsymbol{S} \cdot (\boldsymbol{U})_f (\boldsymbol{U})_f \\ &= \sum_f F(\boldsymbol{U})_f \\ &= a_P \boldsymbol{U}_P + \sum_N a_N \boldsymbol{U}_N \end{aligned} \tag{111.30}$$

PISO (Pressure-Implicit Splitting of Operators) procedure proposed by Venier et al. (2017) is used here for pressure-velocity coupling in transient calculations. In order to derive the pressure equation, a semi-discrete form (equation 111.31) of momentum equation is adopted, where pressure gradient term is not discretized at this stage. The $\boldsymbol{H}(\boldsymbol{U})$ term consists of two parts: the "transport part", including the matrix coefficients for all neighbors multiplied by corresponding velocities and the "source part" including the source part of the transient term.

$$\begin{aligned} a_P \boldsymbol{U}_P &= \boldsymbol{H}(\boldsymbol{U}) - \boldsymbol{\nabla} p \\ \boldsymbol{H}(\boldsymbol{U}) &= -\sum_N a_N \boldsymbol{U}_N + \frac{\boldsymbol{U}^0}{\Delta t} \end{aligned} \tag{111.31}$$

In addition, the discrete form of continuity equation is:

$$\boldsymbol{\nabla} \cdot \boldsymbol{U} = \sum_f \boldsymbol{S} \cdot \boldsymbol{U}_f = 0 \tag{111.32}$$

From equation 111.31, $\boldsymbol{U}$ can be explicitly expressed with equation 111.33. Then from the explicit solution $\boldsymbol{U}_P$ velocities at cell face $\boldsymbol{U}_f$ can be calculated through face interpolation (equation 111.34). Substituting equation 111.34 into equation 111.32 yields the equivalent form of continuity condition (equation 111.35).

$$\boldsymbol{U}_P = \frac{\boldsymbol{H}(\boldsymbol{U})}{a_P} - \frac{1}{a_P}\boldsymbol{\nabla} p \tag{111.33}$$

$$\boldsymbol{U}_f = (\frac{\boldsymbol{H}(\boldsymbol{U})}{a_P})_f - (\frac{1}{a_P})_f(\boldsymbol{\nabla} p)_f \tag{111.34}$$

$$\sum_f \boldsymbol{S} \cdot [(\frac{1}{a_P})_f(\boldsymbol{\nabla} p)_f] = \sum_f \boldsymbol{S} \cdot (\frac{\boldsymbol{H}(\boldsymbol{U})}{a_P})_f \tag{111.35}$$

The fully discrete form of momentum equation is given in equation 111.36. Equation 111.35 and equation 111.36 constitute the discrete form of incompressible Navier-Stokes system. The face flux $F$ can be calculated using equation 111.37. When equation 111.35 is satisfied, the face fluxes are guaranteed to be conservative.

$$a_P \boldsymbol{U}_P = \boldsymbol{H}(\boldsymbol{U}) - \sum_f \boldsymbol{S}(p)_f \tag{111.36}$$

$$F = \boldsymbol{S} \cdot \boldsymbol{U}_f = \boldsymbol{S} \cdot [(\frac{\boldsymbol{H}(\boldsymbol{U})}{a_P})_f - (\frac{1}{a_P})_f(\boldsymbol{\nabla} p)_f] \tag{111.37}$$

With all the discrete form of system equations prepared, the PISO algorithm can be described as follows:

- The momentum equation is solved first. The exact pressure gradient source term is not known at this stage – the pressure field from the previous time-step is used instead. This stage is called the momentum predictor. The solution of the momentum equation, Eqn. 111.36, gives an approximation of the new velocity field.
- Using the predicted velocities, the $\boldsymbol{H}(\boldsymbol{U})$ operator can be assembled and the pressure equation can be formulated. The solution of the pressure equation 111.35 gives the first estimate of the new pressure field. This step is called the pressure correction.
- Eqn. 111.37 provides a set of conservative fluxes consistent with the new pressure field. The velocity field should also be corrected as a consequence of the new pressure distribution. Velocity correction is done in an explicit manner, using Eqn. 111.36. This is the explicit velocity correction stage.

- Eqn. 111.36 reveals that velocity correction consists of two parts: a correction due to the change in the pressure gradient and the transported influence of corrections of neighboring velocities. Explicit velocity correction means that the latter part is neglected. The whole velocity error is assumed to come from the error in pressure term. This is not true. It is therefore necessary to correct the $\boldsymbol{H}(\boldsymbol{U})$ term and repeat pressure correction and explicit velocity correction stage. In other words, the PISO loop consists of an implicit momentum predictor followed by a series of pressure solutions and explicit velocity corrections. The loop is repeated until a pre-determined tolerance is reached.

### 111.3.5 Explicit transient algorithm

The algorithm for solid fluid coupling is an explicit segregated approach. The solving of system equation 111.5 in solid domain and system equation 111.6 in fluid domain are performed by RealESSI and InterFoam respectively. A new container called SSFI is implemented in RealESSI to control all the boundary information of solid fluid interface. The algorithm is illustrated in figure 111.1.

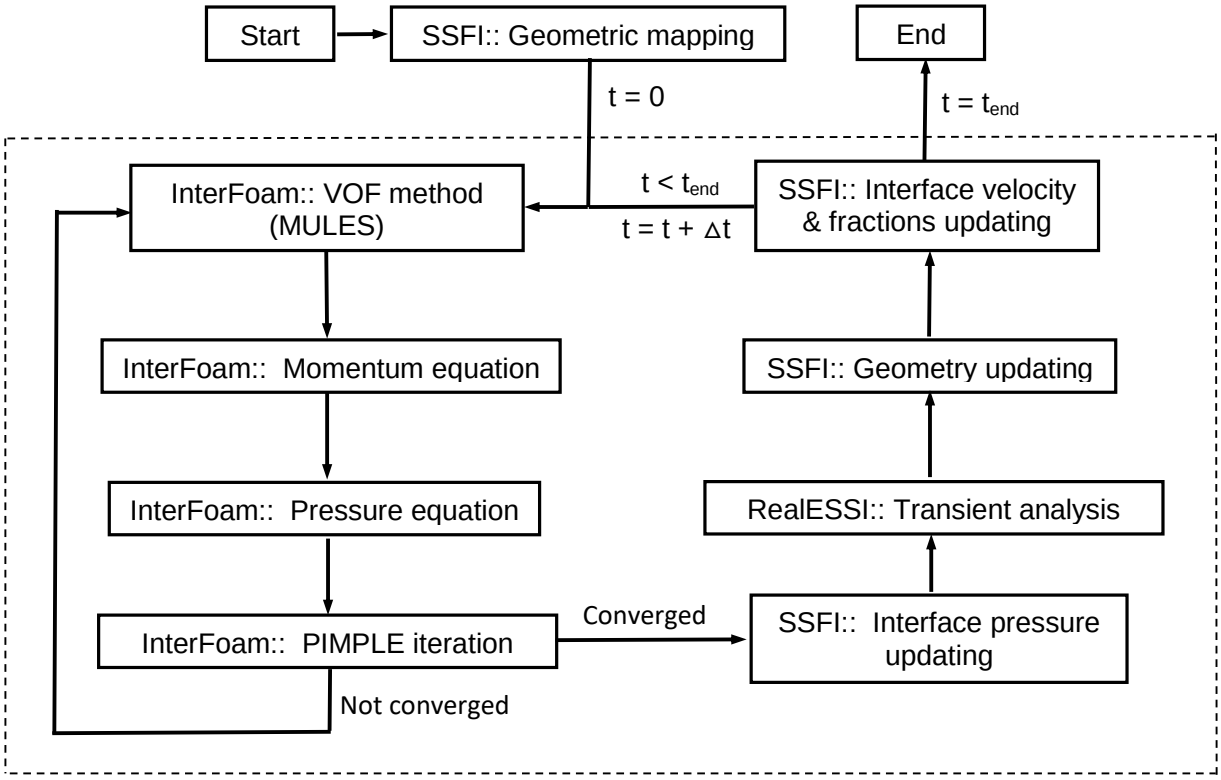



Figure 111.1: Flowchart of explicit transient algorithm

Initially all the geometric mapping information of solid fluid interaction is built by SSFI. There are 4

types of geometric mapping in SSFI: Foam node (interface node in fluid domain) maps to foam surface (interface surfaces in fluid domain), ESSI node (interface node in solid domain) maps to foam nodes, foam surface maps to ESSI nodes and foam node maps to ESSI nodes. The specific definitions of these geometric mappings and implementation details can be found in section 111.4.4. These geometric mapping information is indispensable part while conducting the interpolation for interface pressure, velocity and nodal displacement.

Then the equilibrium state of fluid domain is solved first by Interfoam based on the boundary conditions from the response of solid domain at last time step. Here the PIMPLE algorithm Chen et al. (2014) is implemented in InterFoam to couple pressure and velocity, which is a hybrid of the PISO and SIMPLE (Semi-Implicit Method for Pressure-Link Equations) algorithms. In the PIMPLE loop, the transport equation of volume fraction (equation 111.26) is firstly calculated based on existing velocities and surface fluxes. Following that, there is an implicit momentum predictor and several pressure-velocity correctors. After the fluid domain achieves equilibrium for the new time step, the pressures at fluid interface are transformed to equivalent nodal force and applied at the solid interface. This operation is called interface pressure updating.

With updated nodal force at solid interface and some other transient boundary conditions, the response of solid domain for the new step is obtained through transient analysis of RealESSI (Jeremić et al. (1988-2025)). Based on the latest location of solid interface, the geometry of fluid interface is updated correspondingly to make sure the geometric conformity of both domains.

No slip boundary condition is adopted here for the velocity at fluid interface. Therefore, the new boundary velocities are also calculated from the response of solid domain. In addition, it is crucially important to update volume fractions so that mass conservation is guaranteed. The detailed information about interpolation and updating of these physical fields (i.e. pressure, velocity and volume fractions) is presented in section 111.4.6 and 111.4.5.

For the above explicit transient algorithm, both solid domain and fluid domain can individually reach equilibrium states through iterations at both sides with updated interface boundary conditions. However, this two equilibrium states are not achieved at the same time. Current equilibrium state of solid domain matches with the equilibrium sate of fluid domain at the last time step. Therefore, the algorithm is only valid and accurate when time step is small enough. Different time step lengths in solid domain and fluid domain can also be handled through Shepherd method. SSFI can go through transient analysis in solid domain and fluid domain alternatively according to different time step length.

## 111.4 Implementation Details

### 111.4.1 Installation of OpenFoam

The official website of Openfoam: www.openfoam.com/down load/install-source.php gives detailed instruction about how to build Openfoam from source on different operating systems. Installation of customized version of OpenFOAM, that can interact, connect to Real-ESSI is described in some detail in chapter 209.6 on page 1335.

Note that after installation of Openfoam, you need to source the OpenFOAM environment by executing, e.g. for bash shells and OpenFoam version v1706:

```
source /OpenFOAM/OpenFOAM-v1706/etc/bashrc
```

### 111.4.2 Integrated Preprocessor-gmFoam

gmFoam is designed as an integrated preprocessor developed for analysis of solid fluid interaction. It enables user to build an integrated geometric model (including both solid part and fluid part) in Gmsh and generate input files for both RealESSI and interFoam in a very easy way. Some simple examples can be seen in figure 111.2. Its main functions are listed below:

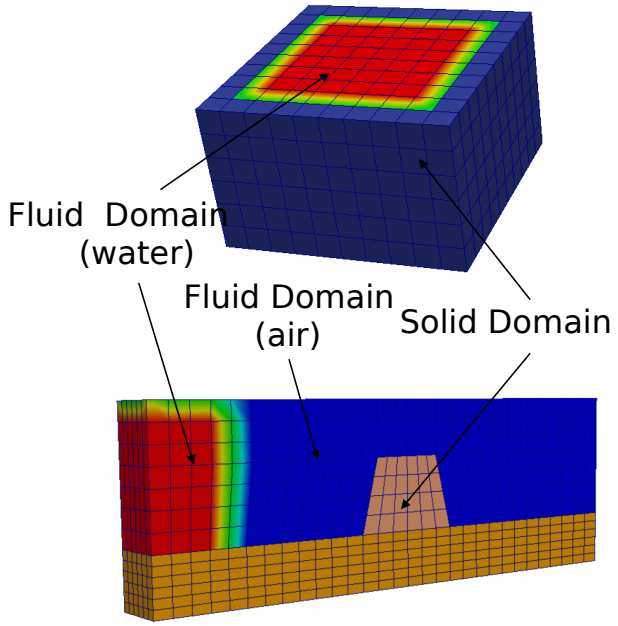



Figure 111.2: Numerical models built with gmFoam

- Mesh separation

Note that FVM mesh supported by Openfoam is very flexible. It can be any types of convex polyhedron as shown in figure 111.3). And the geometric description of FVM mesh required by Openfoam is totally different from that of FEM mesh. The description of FVM mesh is face-based. Faces (including both boundary surfaces and exterior surfaces) of control volumes are defined by a list a point IDs that consist of the face. Also the owner cell ID and neighbor cell ID of each face have to be specified. In contrast, the description of FEM mesh is element-based. After defining all the nodes in the model, the element is described by a list of node IDs. gmFoam supports both types of mesh description. User can build an integrated geometric model in Gmsh and define solid part and fluid part as different physical volume groups. After meshing it, gmFoam can separate the mesh information about solid domain and fluid domain and transfer them to FEM mesh and FVM mesh, respectively.

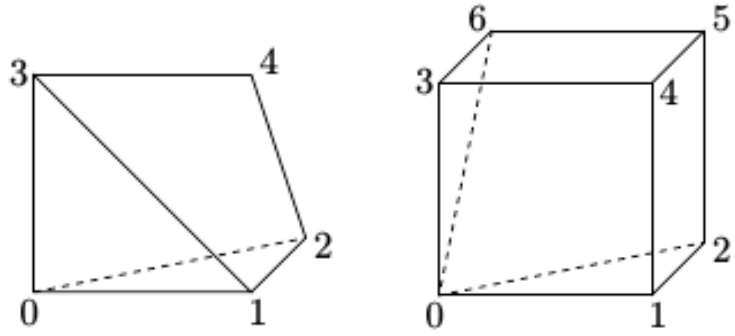



Figure 111.3: Mesh for FVM

- GmESSI incorporation

gmFoam perfectly incorporates current RealESSI preprocessor GmESSI so that it can quickly generate ESSI input files for the simulation of solid part. Also it has the capability to quickly generate input files for Openfoam. Currently the input file organization of Openfoam is very complicated. Several folders and files are needed to prepare in order to complete a very simple simulation. But with gmFoam all these basic information can be written in one single file with suffix as .gmfoam and gmFoam will automatically parse the content inside and produce all the input folders and files. In addition, with the help of physical group, gmFoam enables user to set different boundary conditions in a very convenient way. This is extremely helpful when we conduct solid fluid interaction

analysis for big models with complicated boundary conditions.

- Interface geometry extraction

  gmFoam can extract the geometric information of solid-fluid interface and write down corresponding information as input files. The information is used to initialize SSFI object and build important geometric interface mapping.

- Support discontinuous mesh

  Discontinuous mesh is supported. This is very important, especially for large-scale simulation for solid fluid interaction. Usually refined mesh is needed for fluid part to get accurate enough result using VOF. Discontinuous mesh enables us to arbitrarily refine fluid mesh without changing solid mesh. Therefore, increase of computational efforts in solid part can be avoided. Figure 111.4 shows a model with discontinuous mesh (fluid mesh size : solid mesh size=1:3).

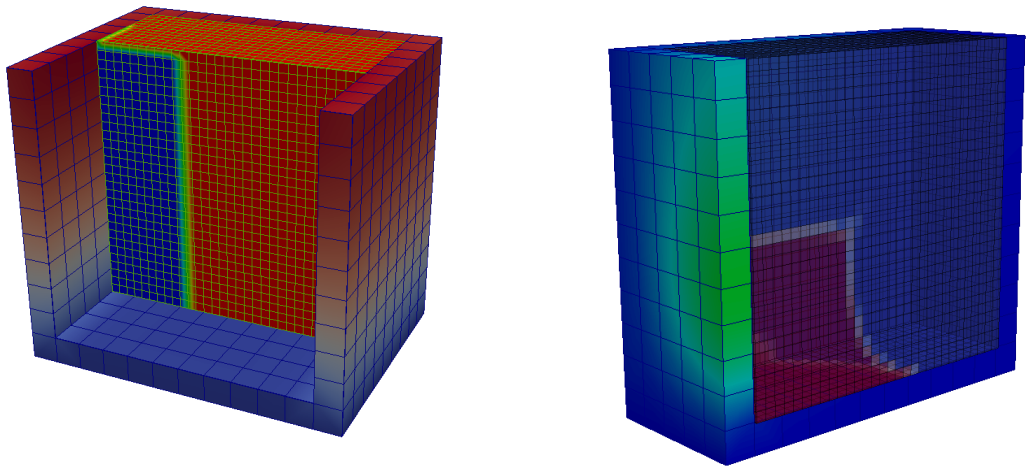


Figure 111.4: Numerical model with discontinuous mesh

### 111.4.3 Interface Domain-SSFI

An interface class SSFI was implemented in RealESSI to couple computations between solid domain and fluid domain. SSFI behaves like a container (called interface domain) to control geometric mapping, sequence of computation and boundary data interpolation and transmission. SSFI class contains and

operates the objects of four base classes: ESSINode, ESSISurface, FoamNode, FoamSurface. Two core member functions: SSFI::FoamToESSIUpdate(double t) and SSFI::ESSIToFoamUpdate(Domain* theDomain, double t) are designed to perform all necessary updates on solid domain and fluid domain, respectively.

### 111.4.4 Geometric Mapping

Following geometrical mappings are built and maintained in SSFI.

- Foam node mapping to Foam surfaces

  For each foam node in solid fluid interface, this mapping returns all the IDs of its surrounding foam surfaces. For example, as shown in figure 111.5(a), foam node 5 is mapped to foam surface 1,2,3 and 4.

  Foam node → Foam Surfaces

  5 (1,2,3,4)

- ESSI node mapping to Foam nodes

  For each ESSI node in solid fluid interface, this mapping returns all the IDs of its surrounding foam nodes within certain search radius (by default the radius is set as 0.1 meters). If the mesh size is very refined, reducing the searching radius helps to improve accuracy.

- Foam face mapping to ESSI nodes

  For each foam face in solid fluid interface, this mapping returns 4 vertex node IDs of an ESSI surface that contains the center of this foam face. Like in figure 111.5(b), foam face 1 (consists of foam node 1,2,3 and 4) is mapped to ESSI nodes 1,2,3 and 4.

  Foam face → ESSI nodes

  1 (1,2,3,4)

- Foam node mapping to ESSI nodes

  For each foam node in solid fluid interface, this mapping returns 4 vertex node IDs of an ESSI surface that contains the Foam node. In figure 111.5(b), foam node 1 is mapped to ESSI nodes 1,3,8 and 7.

  Foam node → ESSI nodes

  1 (1,3,8,7)

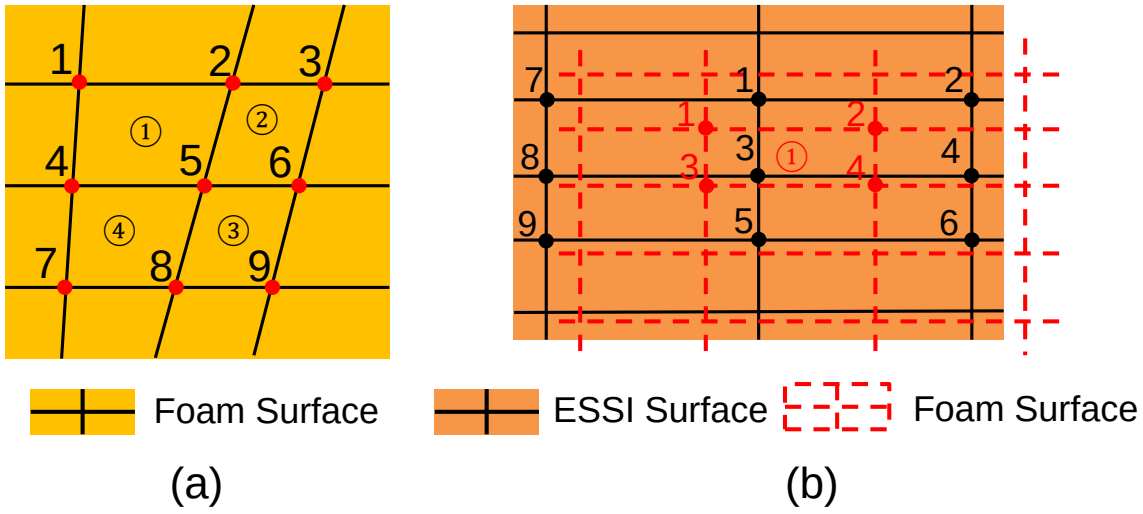



Figure 111.5: Geometric mapping in SSFI

### 111.4.5 SFI Interpolation

After building all of these geometric mapping, interpolation scheme is also needed to fully determine the values of interface variables (pressure, velocity and displacement) and update these values during the interaction process. There are three types of interpolation and updating involved here:

- Pressure interpolation

  The interpolation and updating of interface pressure happen during the process of FOAM to ESSI updating. After the fluid domain achieves its equilibrium, the new pressure values at interface foam faces need to be interpolated and transfered to corresponding ESSI nodes. The pressure interpolation scheme is following:

  Firstly, using the mapping from foam node to foam surfaces, pressure at each interface foam node is calculated by taking the average pressure values of its surrounding foam surfaces. Like in figure 111.5(a), the pressure at interface foam node 5 is the average value of pressure at interface foam surface 1,2,3 and 4. Then according to the mapping from ESSI node to foam nodes, the updated pressure at interface ESSI node is calculated as the average pressure value of corresponding foam nodes.

- Velocity interpolation

  Velocity interpolation takes place during the process of ESSI to FOAM updating. After RealESSI

conducts transient analysis for solid domain, the velocities at interface ESSI nodes need to be fed back to corresponding interface foam surface. The velocity interpolation scheme is following:

The mapping from Foam face to ESSI nodes can give four vertex interface ESSI node ID. For example, in figure 111.5(b), foam face 1 is mapped to four ESSI nodes (1,2,3,4). With coordinates of foam face center and four ESSI nodes already known, inverse isoparametric mapping Hua (1990) is used here to determine the local coordinates $(\zeta, \eta)$ of foam face center. The formula of inverse isoparametric mapping is shown below:

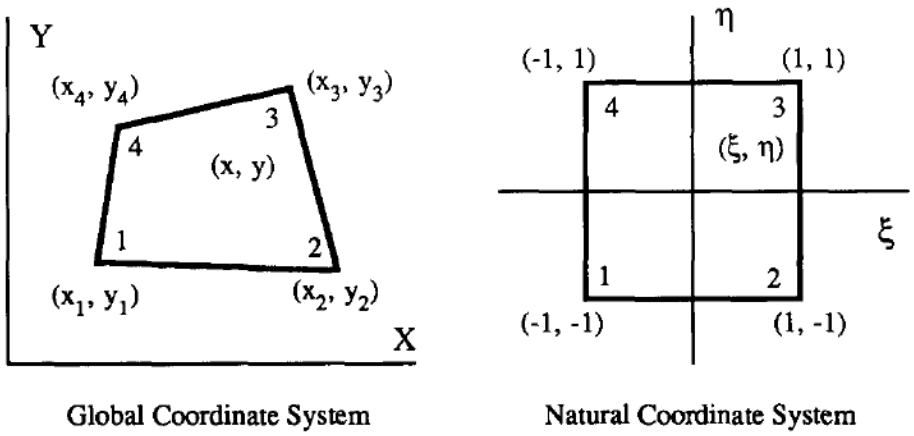



Figure 111.6: Illustration of isoparametric mapping Hua (1990)

$$\begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \\ c_1 & c_2 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \tag{111.38}$$

$$\begin{aligned} d_1 &= 4x - (x_1 + x_2 + x_3 + x_4) \\ d_2 &= 4y - (y_1 + y_2 + y_3 + y_4) \end{aligned} \tag{111.39}$$

A compact notation to represent the determinant of a 2 matrix is introduced as

$$r_s = \begin{vmatrix} r_1 & s_1 \\ r_2 & s_2 \end{vmatrix} = r_1 s_2 - r_2 s_1 \tag{111.40}$$

where $r, s = a, b, c, d$. Notice that $r_s = -s_r$.

With all the notations well defined, the solutions to local coordinate given by Hua (1990) are shown below:

- $a_1 a_2 a_b a_c \neq 0$

$$\begin{cases} a_b\zeta^2 + (c_b + d_a)\zeta + d_c = 0 \\ \eta = (a_d + b_a\zeta)/a_c \end{cases} \quad \text{where } \zeta \in [-1, 1]$$

- $a_1 = 0$ *and* $a_2 c_1 \neq 0$

$$\begin{cases} a_b\zeta^2 + (c_b + d_a)\zeta + d_c = 0 \\ \eta = (a_d + b_a\zeta)/a_c \end{cases} \quad \text{where } \zeta \in [-1, 1]$$

- $a_2 = 0$ *and* $a_1 b_2 \neq 0$

$$\begin{cases} a_b\zeta^2 + (c_b + d_a)\zeta + d_c = 0 \\ \eta = (a_d + b_a\zeta)/a_c \end{cases} \quad \text{where } \zeta \in [-1, 1]$$

- $a_1 a_2 \neq 0$ *and* $a_b = 0$

$$\begin{cases} \zeta = (a_1 d_c)/(b_1 a_c + a_1 a_d) \\ \eta = a_d/a_c \end{cases}$$

- $a_1 a_2 \neq 0$ *and* $a_c = 0$

$$\begin{cases} \zeta = a_d/a_b \\ \eta = (a_1 d_b)/(c_1 a_b + a_1 a_d) \end{cases}$$

- *All other conditions*

$$\begin{cases} \zeta = d_c/(a_1 d_2 + b_c) \\ \eta = b_d/(a_2 d_1 + b_c) \end{cases}$$

After obtaining local coordinates $(\zeta, \eta)$ corresponding to foam surface center, isoparametric mapping can be conducted through the shape function of 4-node quadrilateral element. The interpo-

lated velocity can be calculated with equation 111.41.

$$
\begin{aligned}
N_1 &= \frac{1}{4}(1-\zeta)(1-\eta) \\
N_2 &= \frac{1}{4}(1+\zeta)(1-\eta) \\
N_3 &= \frac{1}{4}(1+\zeta)(1+\eta) \\
N_4 &= \frac{1}{4}(1-\zeta)(1+\eta) \\
\boldsymbol{v} &= N_1\boldsymbol{v_1} + N_2\boldsymbol{v_2} + N_3\boldsymbol{v_3} + N_4\boldsymbol{v_4}
\end{aligned}
\tag{111.41}
$$

- Displacement interpolation

  Displacement interpolation also takes place during ESSI to FOAM updating. In order to meet the geometric conformity, the Eulerian mesh of fluid domain should dynamically move along with the real-time response of solid domain. Therefore, it is necessary to interpolate the displacement of interface ESSI nodes to interface foam nodes. The displacement interpolation scheme is similar to velocity interpolation scheme and shown below:

  The mapping from Foam node to ESSI nodes can give four vertex interface ESSI node ID. Inverse isoparametric mapping is first performed to compute local coordinate of Foam node. Then displacement of foam node is interpolated from displacement of four vertex ESSI nodes through isoparametric mapping.

### 111.4.6 Mass Conservation

There are two levels of mass conservation conditions needed to be satisfied during the SFI.

One is local level: Regarding each control volume, the amount of fluid flows in should equal to the amount of fluid that flows out for incompressible fluid. This local mass conservation condition is mathematically represented by continuity equation in equation 111.6 and can be approximately met through finite volume discretization. Another level of mass conservation is global level: For a closed fluid domain (no fluid transfer with other external fluid system), the total amount of fluid should keep constant during SFI. The global level of mass conservation is trivial for pure fluid flow system with static Eulerian boundary mesh. Since for this kind of flow system, global level of mass conservation is free and automatically holds based on local level of mass conservation.

However, for flow system with deforming boundary, especially when Lagrangian movement of Eulerian mesh is involved (like Arbitrary Lagrangian method Souli and Zolesio (2001)), the local level of mass

conservation does not guarantee the mass conservation of global level. Demirdžić and Perić (1988) pointed out that for moving mesh, one more conservation equation so called space conservation needed to be solved simultaneously with the mass, momentum and energy conservation equations. Otherwise artificial mass addition or reduction is generated which may cause the solution to be greatly in error.

The space conservation law is expressed by equation 111.42, where $J$ is the determinant of the metric tensor and $\boldsymbol{v_g}$ is the grid velocity of the mesh. Correspondingly, the finite volume discrete evaluation of volume integral of equation 111.42 can be given in equation 111.43, where $\delta V = V^n - V^0$ is the change of cell volume during $\Delta t$, $\boldsymbol{v_{g(f)}}$ is the mesh velocity of cell face and $\boldsymbol{S}$ is cell face vector.

$$\frac{\partial J}{\partial t} - J\boldsymbol{\nabla} \cdot \boldsymbol{v_g} = 0 \tag{111.42}$$

$$\frac{V^n - V^0}{\Delta t} = \sum_f \boldsymbol{v_{g(f)}} \cdot \boldsymbol{S} \tag{111.43}$$

In our implementation, consider the coupling with VOF method and our SFI simulation is generally for small deformation of fluid boundary, a simplified procedure is adopted here to guarantee the global level of mass conservation: Only the location of foam nodes at solid fluid interface is updated with Lagrangian motion of corresponding ESSI nodes while interior foam nodes remain static. After geometry updating of the mesh, the volume of foam cells at interface is re-evaluated and volume fraction values are also updated according to equation 111.44 to ensure the mass conservation. Then the transport equation 111.27 is solved based on new $\alpha$ values.

$$\alpha = \frac{\int \alpha^0 dV^0}{\int dV} = \frac{\alpha^0 V^0}{V} \tag{111.44}$$

Part 200

# Software and Hardware Platform: Design, Development, Procurement and Use

Chapter 201

# The Real ESSI Simulator System

(1986-1989-1993-1994-1996-1999-2003-2007-2019-2020-2021-)

## 201.1 Chapter Summary and Highlights

## 201.2 Introduction to the Real-ESSI Simulator System

The Real-ESSI Simulator ( Realistic Modeling and Simulation of Earthquakes, and/or Soils, and/or Structures and their Interaction) is a software, hardware and documentation system for high performance, sequential or parallel, time domain, linear or nonlinear, elastic and inelastic, deterministic or probabilistic, finite element modeling and simulation of

- statics and dynamics of soil,
- statics and dynamics of rock,
- statics and dynamics of structures,
- statics of and dynamics of soil-structure systems,
- dynamics of earthquakes, and
- dynamic earthquake-soil-structure interaction.

The Real-ESSI Simulator systems is used for design and for assessment of static and dynamic behavior of infrastructure objects, including buildings, bridges, dams, nuclear installations, tunnels, etc.

Design: Multiple linear elastic load cases can be combined and design quantities, sectional forces exported for design.

Assessment: Practical, realistic, inelastic, nonlinear load staged analysis, with accurate modeling of elastic and inelastic, nonlinear components, and with all the simulation, algorithmic features available, as listed below, is performed to assess design, current safety margins and economy of objects.

The Real-ESSI Simulator is developed at the University of California, Davis, in collaboration and with partial financial support from the USDOE, USNRC, USNSF, USBR, USFEMA, CalTrans, CNSC-CCSN, UN-IAEA, Shimizu, Private Donors, etc. The Real-ESSI Simulator develops methods and models that inform and predict rather than (force) fit.

The Real-ESSI Simulator systems consists of the Real-ESSI Program, Real-ESSI Pre-Processing and Post-Processing tools, Real-ESSI Computer and Real-ESSI Notes.

### 201.2.1 Real-ESSI Program

The Real-ESSI program is a general purposes finite element program that features models and methods for analyzing static and dynamic behavior of Civil Engineering objects, such as buildings, bridges, dams, nuclear installations, tunnels, etc.

### 201.2.2 Real-ESSI Pre-Processing tools

The Real-ESSI Pre-Processing tools are a set of programs, scripts and modules that are used to develop Real-ESSI models. Mesh generation relies on Gmsh (Geuzaine and Remacle, 2009) and our own plugins Gm-ESSI, while there are also mesh translators from other input formats to Real-ESSI input format/language.

### 201.2.3 Real-ESSI Post-Processing tools

The Real-ESSI Post-Processing tools rely on Paraview (Ayachit, 2015) visualization platform, with our own plugins, as well as on a number of programs and scripts to visualize output using matlab, python, etc.

### 201.2.4 Real-ESSI Computer

The Real-ESSI Computer can be any single CPU, multiple CPU, and/or cluster of single/multiple CPUs computers using Linux operating system. The reason for using Linux is that state of the art development tools are available, and that Linux is used on virtually all large supercompters, so that the very same sources can be compiled and executables developed for small desktop computers, large server computers, local clusters of computers and large supercomputers. There is a possibility to build Real-ESSI and create executables on Macintosh and Windows platforms, as long as standard software tools and compiles, C++ and Fortran, are available. However, such work is not currently pursued.

### 201.2.5 Real-ESSI Notes

The Real-ESSI notes, documentation for the Real-ESSI Simulator System, is available through Lecture Notes by Jeremić et al. (1989-2025).

### 201.2.6 Real-ESSI Name

The Real-ESSI Simulator System name is based on an acronym: Realistic Modeling and Simulation of Earthquakes, and/or Soils, and/or Structures and their Interaction. Pronunciation of Real-ESSI is similar to "real easy", as in "as easy as pie". Translation of name Real ESSI to languages of developers and users is:

- Врло просто,
- Стварно просто,
- Стварно лако,
- Просто к'о пасуљ,
- Vrlo prosto,
- Stvarno prosto,
- 真简单,
- Muy fácil,
- অতি সহজ,
- آسان واقعی,
- Molto facile,
- 本当に簡単,
- Πραγματικά εύκολο,
- बहुत ही आसान,
- Très facile,
- Вистински лесно,
- Wirklich einfach,
- سهلٌ جداً
- Zelo enostavno
- Zares enostavno
- Muito Fácil
- Res lahko
- Dziecinna igraszka
- Çocuk oyuncağı

Chapter 202

# Object Oriented Software Platform Design

(1992-1993-1994-1996-1999-2003-2005-2007-2008-2009-2010-2011-2015-2016-2019-)

(In collaboration with Dr. Guanzhou Jie)

## 202.1 Chapter Summary and Highlights

## 202.2 Object-Oriented Design Basics

Booch (1994); Gamma et al. (1995); Coplien (1992); Koenig (1989 - 1993); Stroustrup (1986); Stroustrup (1994); Ellis and Stroustrup (1990); Johnson (1994); Felippa (1992a); Dubois-Pelerin (1992); Dubois-Pèlerein and Zimmermann (1993); Raphael and Krishnamoorthy (1993); Menéntrey and Zimmermann (1993); Zimmermann et al. (1992a); Donescu and Laursen (1996); Zimmermann and Eyheramendy (1996); Eyheramendy and Zimmermann (1996); Jeremić and Sture (1998); Forde et al. (1990); Miller (1991); Scholz (1992); Fenves (1990); Eyheramendy (1997); Dubois-Pélerin and Zimmermann (1992); Zimmermann et al. (1992b); Dubois-Pélerin and Pegon (1998); Eyheramendy and Zimmermann (2001);

Veldhuizen (1995a); Veldhuizen (1995b); Veldhuizen (1996); Veldhuizen and Jernigan (1997); Archer (1996); Archer et al. (1999); McKenna (1997);

## 202.3 Object-Oriented Design of the Plastic Domain Decomposition (PDD)

### 202.3.1 Introduction

This section describes the object oriented design of the proposed PDD algorithm and its implementation into MOSS library framework. At the beginning of this section, the Object-Oriented approach to programming the Finite Element Method is reviewed based on the existing (as of 2005) implementation of OpenSees. Object-Oriented parallel design is then extended from the existing framework. Parallel algorithm adopts Main-Follower paradigm and the new design of data structures have strictly followed the Object-Oriented principle using C++ language. External utility libraries such as ParMETIS and PETSc have been incorporated to provide seamless parallel numerical manipulations including partitioning/repartitioning and equation solving.

In this chapter, the algorithm overview will be presented first. Then the implementation details in C++ will follow. The challenges of achieving load balancing in parallel Finite Element simulation have been divided into two parts, global level equation solving and constitutive level iterations. This research presents the PDD algorithm to demonstrate how to balance each stage systematically in applications.

### 202.3.2 Object-Oriented Parallel Finite Element Algorithm

Parts of OpenSees software framework (McKenna, 1997) have been used in this chapter. Object-Oriented design of OpenSees enables software reuse that greatly shortens the development life cycle of application

codes.

OpenSees is comprised of a set of classes and objects that represent models perform computations for solving the governing equations, and provide access to processing results. There are four types of class objects in OpenSees McKenna (1997).

- Modeling Classes are used to create the Finite Element Model Classes for a given problem.
- Finite Element Model Classes are used to describe the finite element model and to store the results of the analysis performed on the model. Main class abstractions used in OpenSees are Node, Element, Constraint, Load and Domain. The relationship amongst these classes can be shown using the class diagram Figure 202.2 using the Rumbaugh notation as shown in Figure 202.1 Rumbaugh et al. (1991).
- Analysis Classes are used to perform the finite element analysis, i.e., to form and solve the global system of equations
- Numerical Classes are used to handle numerical operations in the solution procedure. Also included in this category are data structure classes such as Vector, Matrix and Tensor.

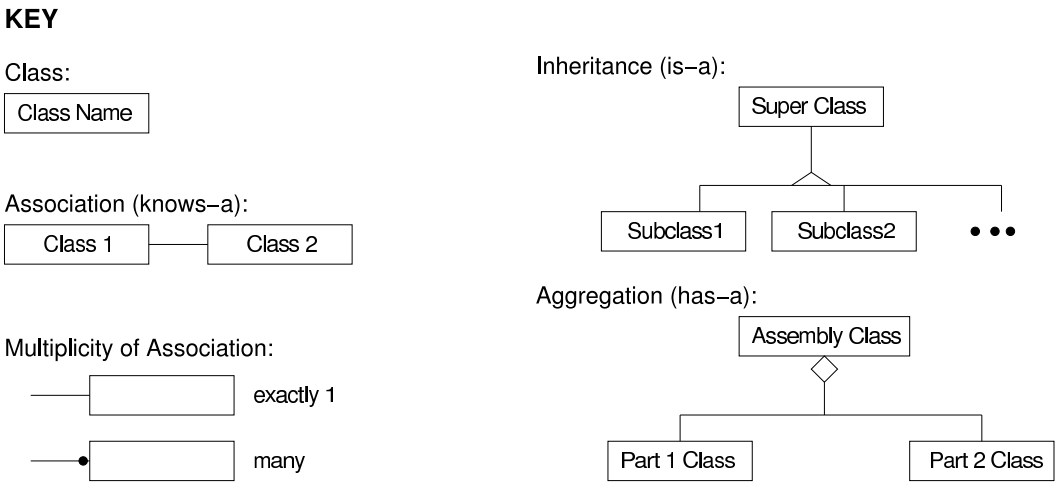



Figure 202.1: Rumbaugh Notation of-Object Oriented Design

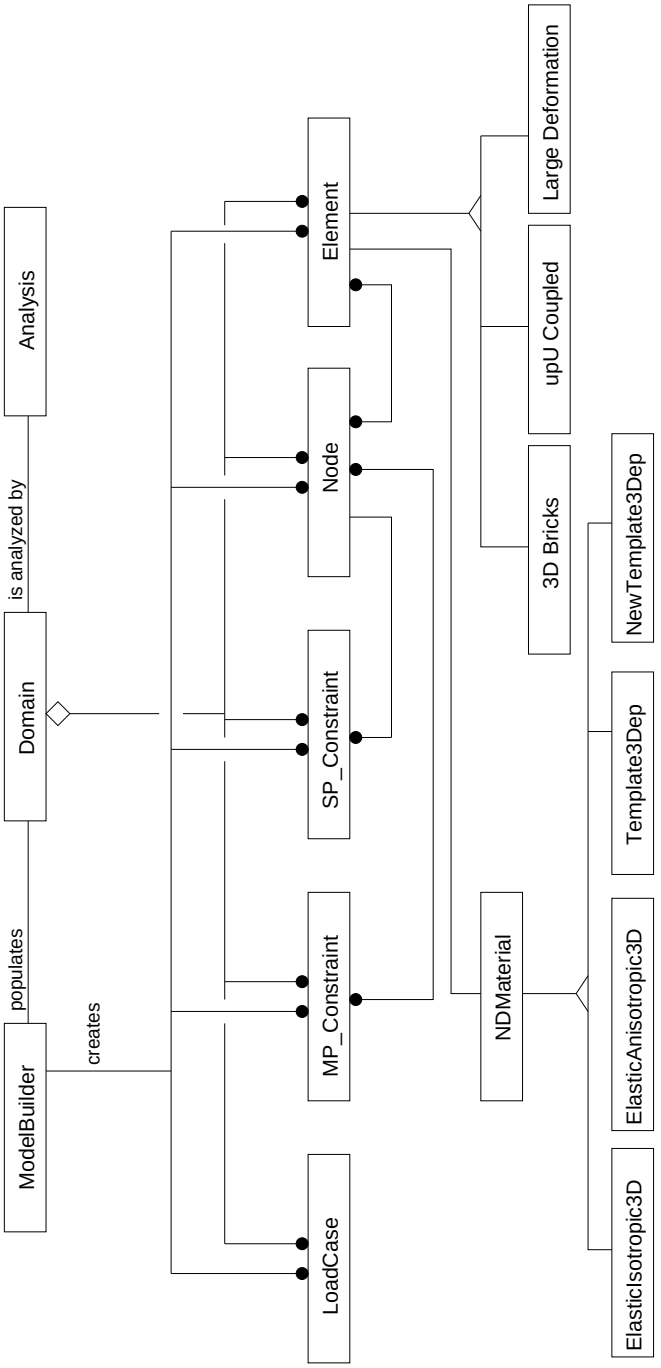



Figure 202.2: Class Diagram of Finite Element Model Classes

#### 202.3.2.1 Modeling Classes

The modeling classes are responsible of creating the necessary components of the finite element model, such as nodes, elements, loads and constraints. There are a number of approaches proposed by various researchers. In some works, the user has the responsibility to create the finite element model in a single driver-type file Ross et al. (1992); Zeglinski et al. (1994); Cardona et al. (1994). In other works, an input file containing the model data is used to be read by the main program to create the model Forde et al. (1990); Dubois-Pelerin et al. (1992); Dubois-Pelerin and Zimmermann (1993); Menéntrey and Zimmermann (1993). Graphical interface for building models visually has also been proposed Ostermann et al. (1995); Mackie (1995).

In this research, the existing ModelBuilder interface class is reused to facilitate the finite element model construction. As shown in Figure 202.2, the ModelBuilder is associated with a single finite element Domain object. The interface (pure virtual function) buildFE_Model() must be redefined depending on the specific type of finite element model users want to build.

In parallel processing, PartitionedModelBuilder is used instead, in which the building process includes higher level control of building Subdomains from the PartitionedDomain. For each Subdomain, the PlaneFrameModelBuilder-type is invoked to build the finite element model on each Subdomain.

The Object-Oriented interface design of ModelBuilder through pure virtual function provides a consistent framework from which all kinds of engineering model can be readily extended.

#### 202.3.2.2 Finite Element Model Class

In most of the works presented, main class abstractions used to describe a finite element model are: Node, Element, Constraint, Load and Domain Forde et al. (1990); Zimmermann et al. (1992a); Dubois-Pelerin et al. (1992); Dubois-Pelerin and Zimmermann (1993); Menéntrey and Zimmermann (1993); Pidaparti and Hudli (1993); Cardona et al. (1994); Chudoba and Bittnar (1995); Zahlten et al. (1995); Rucki and Miller (1996).

Node The most important feature of the Node class is the associativity with DOF class which contains the degree of freedoms of any specific instance of the Node class. The response quantities such as displacements of each DOF object will be stored in the Node class. Routines are available to set/get those solution quantities.

Element The functionality of an Element object is to provide the tangent stiffness, mass and the residual force corresponding to current loadings. Element class contains reference to its associated Node objects.

Element class is one of the most fundamental abstractions in Object-Oriented finite element software design. In this research, Element also acts as a container for material models, which is critical for simulations with nonlinear materials. Chudoba and Bittnar (1995) proposed a MaterialPoint object which is associated with GaussPoint object. In Zahlten et al. (1995), class abstractions such as cross section, material point, material law, yield surface, hardening rule and flow rule are introduced to model complicated materials within the Element class in an Object-Oriented flavor.

Jeremić and Yang (2002) present the complete formulation of Template3Dep material class, which is wrapped inside the Element class to enable a consistent interface for complex elastic-plastic material modeling.

Constraint There are two types of constraints in finite element simulations,

1. Single-Point constraints, which are applied to a specific DOF object;
2. Multi-Point constraints, which describe the relationship between more than one DOF objects.

In current implementation of OpenSees (version from 2005), the two classes SP_Constraint and MP_Contraint are designed but they do not handle the constraints. These two classes are responsible of setting up relations between Nodes and the constrained DOF_Groups. This will be covered shortly in Analysis class design.

Load There are also two types of loads that are commonly seen in finite element analysis:

1. node loads that act on specific Nodes;
2. element loads that act on specific Elements, which can be due to body forces, surface tractions, initial stresses and temperature gradients.

In the current (version version 2005) implementation of OpenSees, three extra classes are introduced to handle loading conditions, LoadPattern, NodalLoad and ElementLoad. The LoadPattern is a container class that provides methods in its interface to allow NodalLoad and ElementalLoad objects to be created, traversed and removed. As shown in Figure 202.2, each NodalLoad or ElementalLoad object is associated with a Node or Element object and is responsible of applying nodal or elemental loads to that object.

Domain The Domain class is the most important container class that is responsible of holding all components of the finite element model, i.e. all the Nodes, Elements, Constraints and Loads. Domain class acts as the interface between Analysis class and all the individual components of the finite element model. The interface of Domain enables component creation, information access and component removal.

#### 202.3.2.3 Analysis

The Analysis class (McKenna, 1997) is responsible for forming and solving the governing equations for the finite element model. As for nonlinear problems, incremental solution techniques are required and iterative schemes such as Newton-Raphson needed to solver the nonlinear system of equations.

For incremental solution algorithm, the computational tasks are more involved for the finite element analysis.

- Assign equation numbers and map these to the nodal DOFs. This step can be of significant influence on the bandwidth of the coefficient matrix, which is inherently sparse due to the compact support of finite element formulation.
- Form the matrix equations using contributions from elements and nodes.
- Apply the constraints, which may involve transforming the element and nodal contributions or adding additional terms and unknowns to the matrix equations depending the method employed to handle constraints.
- Solve the matrix equations for the incremental nodal displacements.
- Determine the internal state and stresses in the elements.

The Object-Oriented design of the Analysis class is done by firstly breaking down the main tasks performed in a finite element analysis, abstracting them into separate classes, and then specifying the interface for these classes. The Analysis class is an aggregation of all the sub-functionality classes of following types:

1. SolutionAlgorithm class describes the complete computation procedure (steps) in the analysis.
2. AnalysisModel is a container class that stores and provides access to the following types of classes:
   (a) DOF_Group class represents the DOF at the Nodes or new DOF introduced into the analysis to enforce the constraints;
   (b) FE_Element class represents the real Elements in the Domain or they are introduced to add stiffness and/or load to the system of equations in order to enforce the constraints.
   It is worthwhile to mention that the FE_Elements and DOF_Groups have very important design implications although they might seem redundant at the first sight. The significance comes from the facts that
      i. they record the mapping between DOFs and equation numbers in the global system which greatly simplifies the interfaces of Node and Element class;

ii. they also provide the interfaces for forming tangent and residual vectors which are used to form the global system of equations;

iii. they are major utility classes of handling constraints.

3. Integrator defines how the FE_Elements and DOF_Groups contribute to the system of equations and how the response quantities should be updated given the solution to the global system of equations.

4. ConstraintHandler handles the constraint by creating adequate FE_Elements and DOF_Groups.

5. DOF_Numberer maps the equation number to the DOFs in the DOF_Groups.

6. SystemOfEqn encapsulates the global system of equations.

The aggregation of the Analysis object is shown in Figure 202.3.

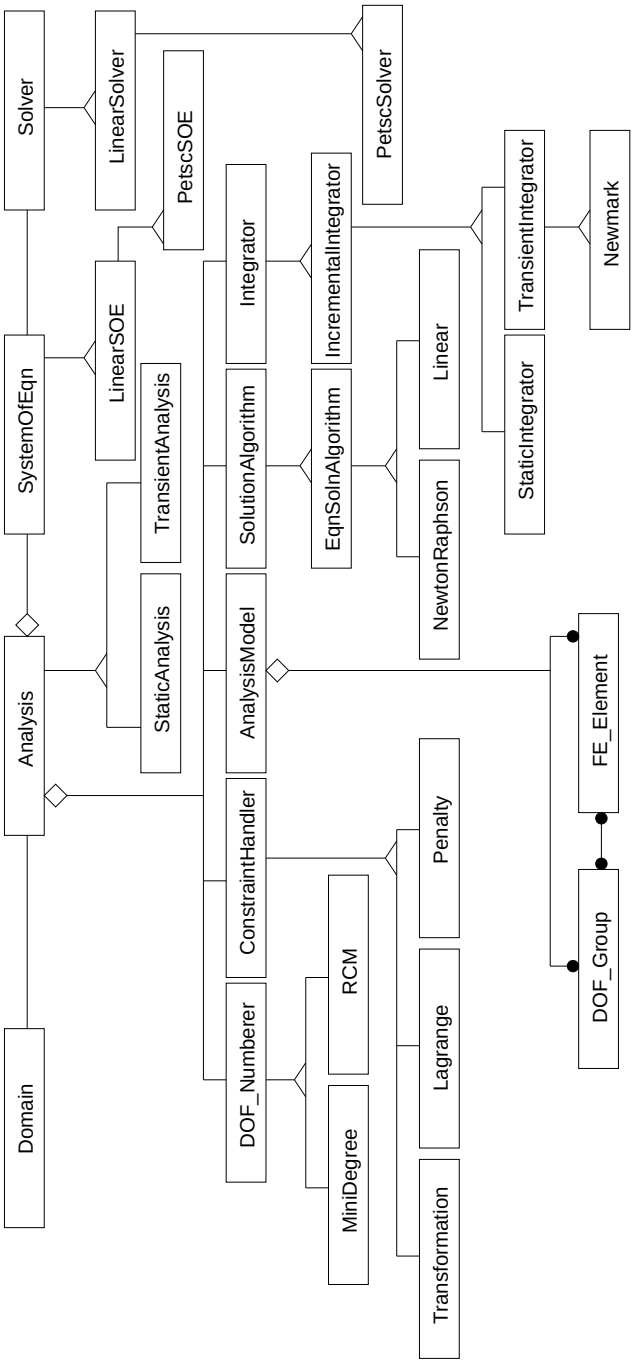



Figure 202.3: Class Diagram of Analysis Aggregation

Traditional program flow diagrams are used to describe how is the nonlinear finite element algorithm control flow implemented in OpenSees. These flow charts are organized as following:

Figure 202.4 shows the overall analysis algorithm flow for nonlinear finite elements.
Then this overall analysis flow is broken down into detailed subroutines, such as theIntegrator::newStep() and theAlgorithm::solveCurrentStep().

- Figure 202.5 explains in detail the function flow of theIntegrator::newStep(), which illustrates the (fairly standard) incremental finite element solution techniques implemented in OpenSees.
- Figure 202.6 shows the function flow of forming the tangent stiffness matrix, which is a loop assembling the global equation system involved in function theIntegrator::newStep().
- Figure 202.7 further describes the Newton-Raphson type iterative solution schemes involved in function theAlgorithm::solveCurrentStep().

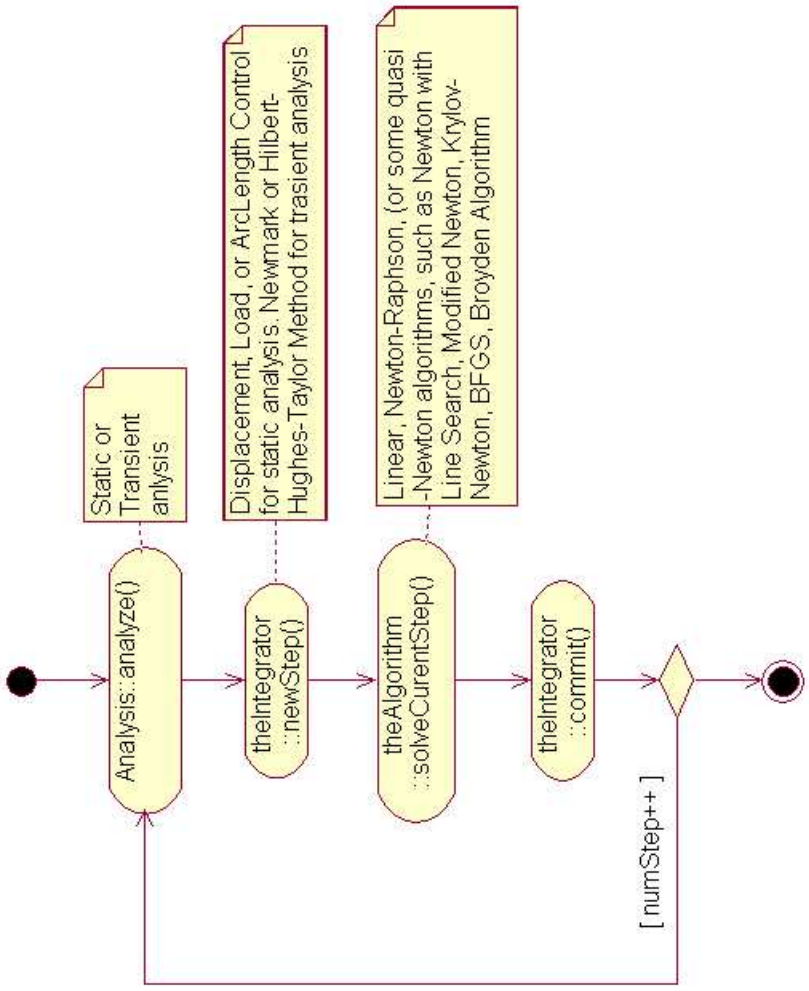



Figure 202.4: Overall Algorithm Flow Chart for Nonlinear Finite Element Analysis

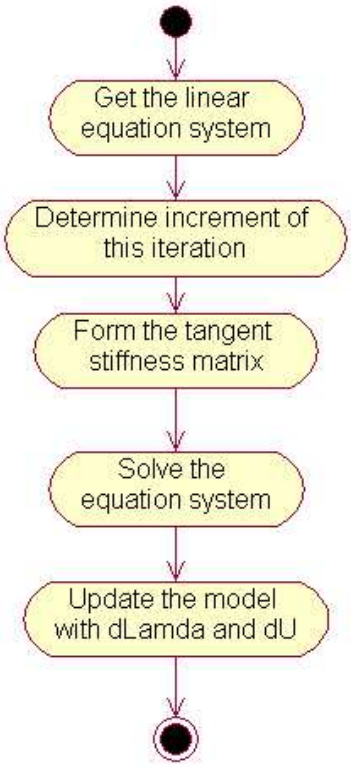



Figure 202.5: Detailed View: theIntegrator::newStep() - Incremental Solution Techniques for Nonlinear Finite Element Analysis

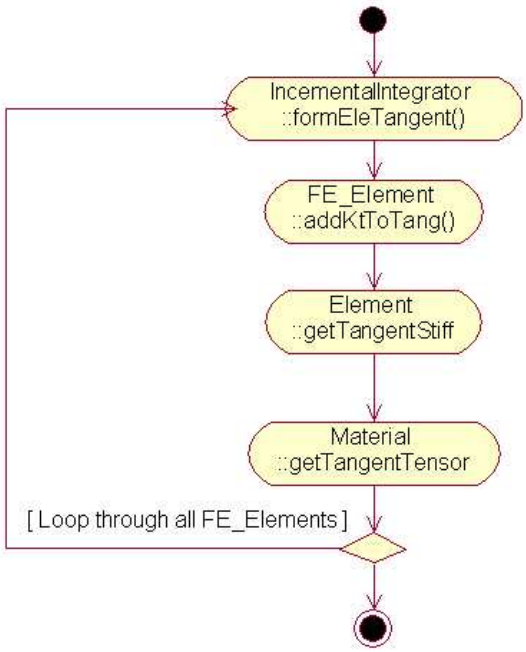



Figure 202.6: Detailed View: Assembly of Global Equation System in theIntegrator::newStep()

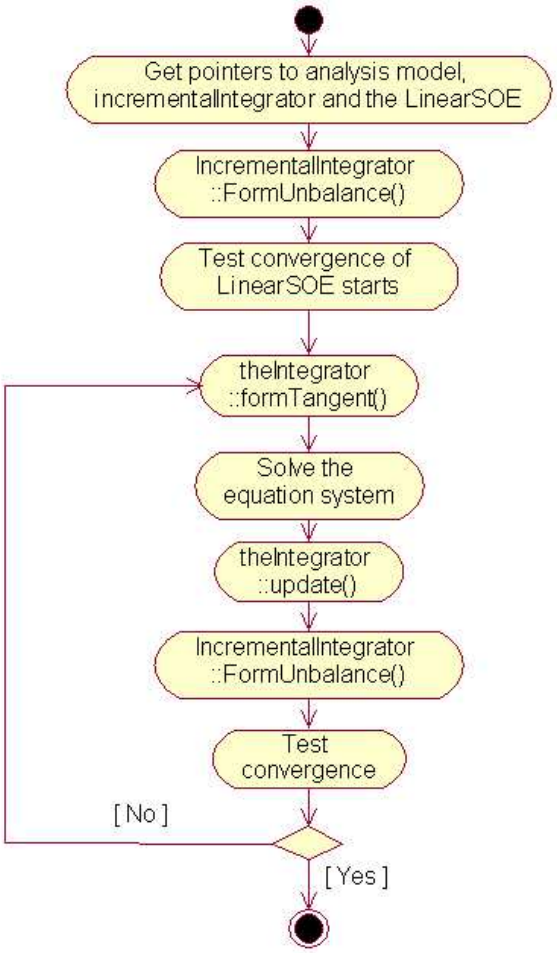



Figure 202.7: Detailed View: theAlgorithm::solveCurrentStep() - Newton-Raphson Iterative Schemes for Nonlinear Finite Element Analysis

Finite element simulations inherently are element-based operations, so little modification is needed to parallelize the algorithms described above, although special attention has to be paid to synchronize the computation among different processors. Figure 202.8 shows the activity flow for parallel nonlinear finite element simulations.

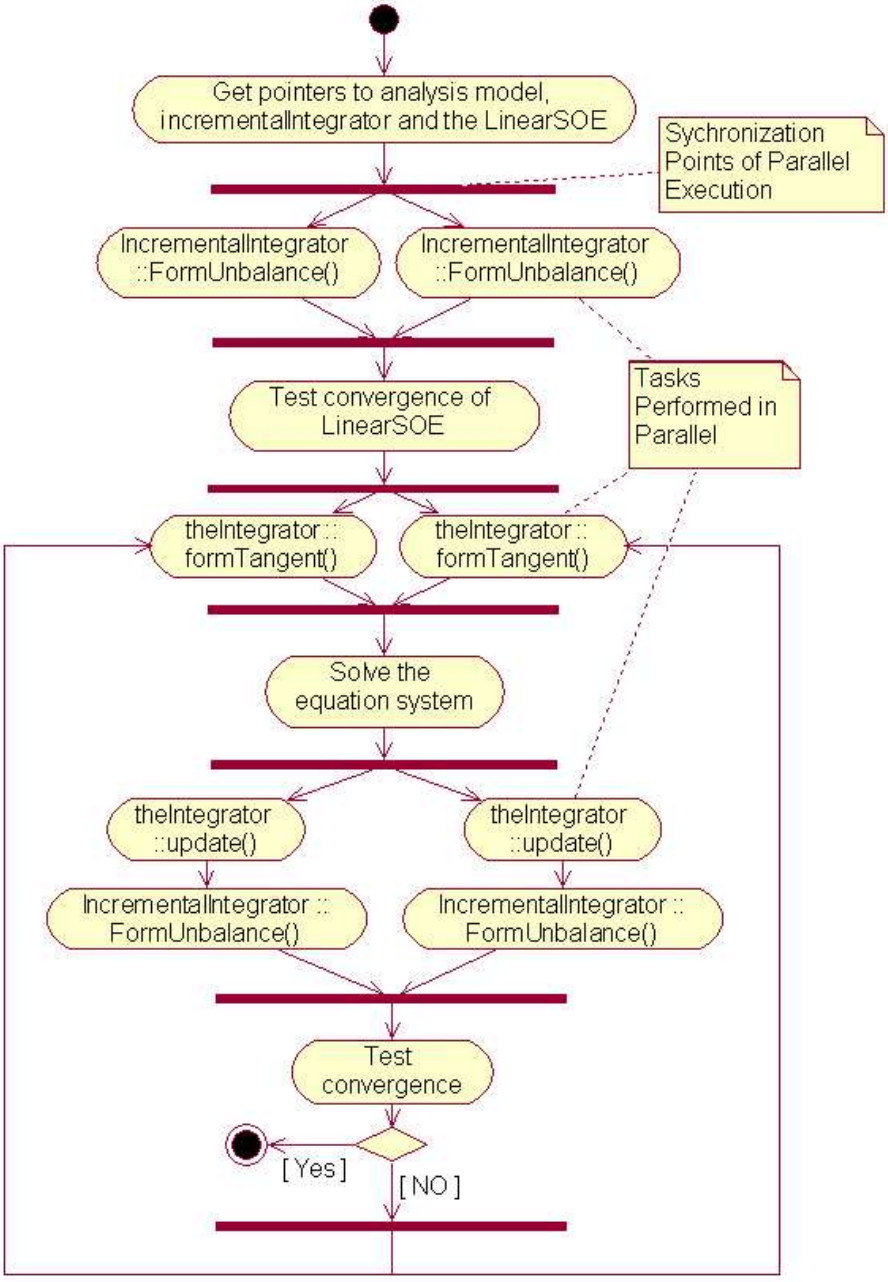



Figure 202.8: Parallel Activity Flow Diagram of Nonlinear Finite Element Analysis

#### 202.3.2.4 Object-Oriented Domain Decomposition

There are three most notable designs of Domain Decomposition method in literature McKenna (1997).

1. Sause and Song (1994) presents an Object-Oriented design for linear static analysis using substructuring. The interface is restricted to substructuring or FETI Farhat and Roux (1991b) only, and repeated geometry limits the applicability of this design to large problems.

2. Archer (1996) proposes a SuperElement class that is a subclass of Element and has a Domain class aggregated. This design is conceptually inappropriate and it results excessive method calls as methods that are for the SuperElement must be called by the SuperElement on the associated Domain McKenna (1997).

3. Miller and Rucki (1993) introduces the Partition class which is associated with an Algorithm class. The Algorithm class is responsible for updating the state of a Partition so that it will be in equilibrium. Again, this design is good for substructuring type Domain Decomposition analysis. If we want to solve a problem before the interface solution can be determined, the design fails.

The current design of OpenSees McKenna (1997) proposes many new classes to facilitate flexible Object-Oriented Domain Decomposition. The main abstractions include PartitionedDomain, DomainDecompAnalysis, DomainDecompSolver Subdomain, DomainPartitioner and GraphPartitioner. The class diagram is shown in Figure 202.9.

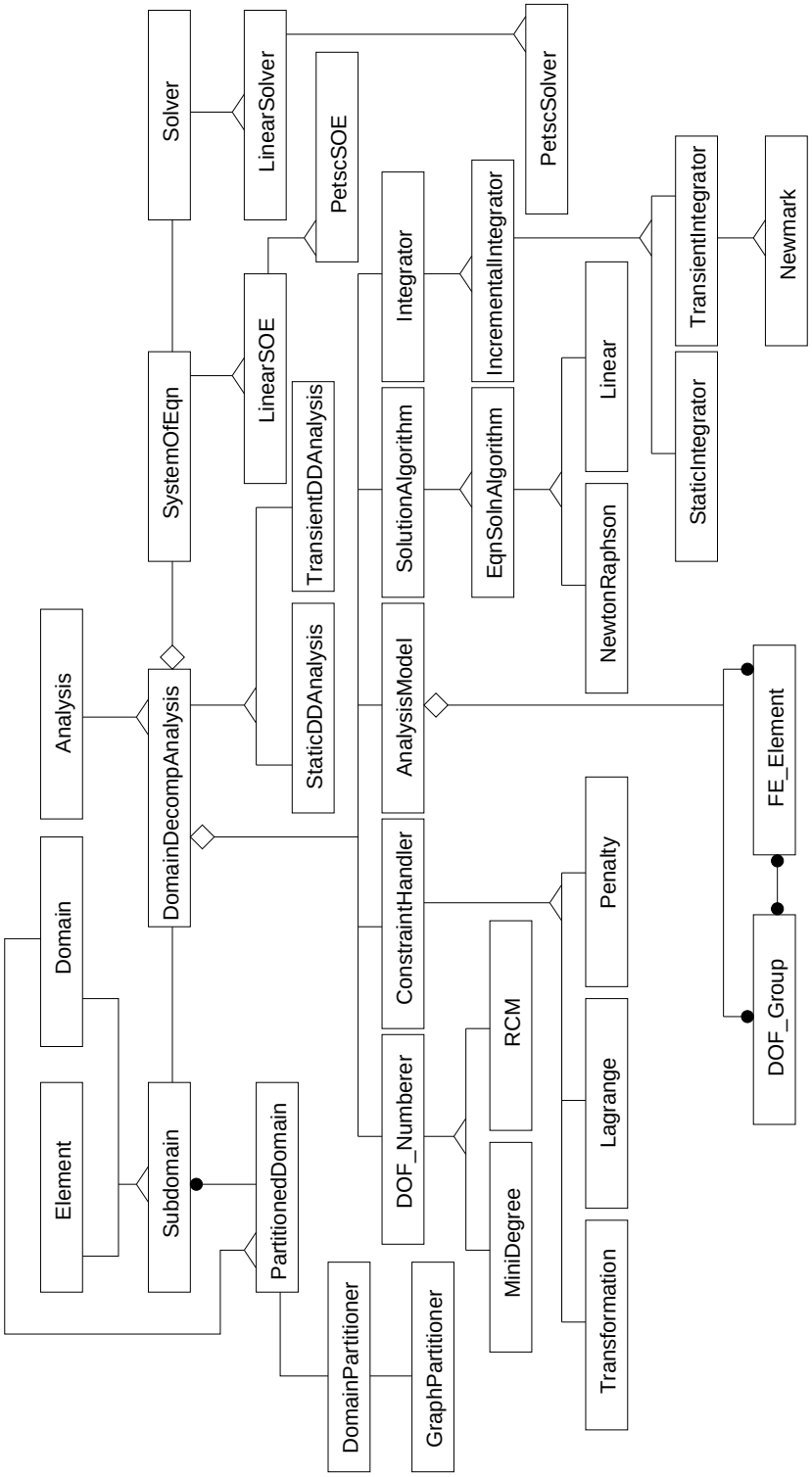



Figure 202.9: Class Diagram of Domain Decomposition Analysis

PartitionedDomain The PartitionedDomain class is a subclass of Domain whose objects can be partitioned into Subdomain objects. Aside from common functionality inherited from Domain, PartitionedDomain class provides methods for partitioning the Domain and retrieving information from Subdomains. PartitionedDomain the aggregation of Subdomains and is the major containing class in main compute process.

DomainPartitioner The DomainPartitioner class is responsible for performing the actual operation to split the PartitionedDomain. The DomainPartitioner will call its associated GraphPartitioner to partition the PartitionedDomain. It also provides the methods to migrate Elements, Nodes, Constraints, Loads amongst Subdomains.

DomainPartitioner is one of the most important utility class in OpenSees in the sense that all partitioning routine and data migration operations will be rooted from this class.

GraphPartitioner This class utilizes external graph partitioner to color the finite element connectivity graph, which will be constructed from the PartitionedDomain. The result will be fed back to DomainPartitioner to facilitate subsequent data distribution.

GraphPartitioner introduces graph partitioning into OpenSees and the main functionality of this class is to call API and provide necessary data structures from the specific application.

Subdomain The Subdomain class inherits from both Element and Domain. This has a dual-level design implication:

1. for the top PartitionedDomain, superclass Element is a proxy class of subclass Subdomain, in the sense that all the relevant operations on Elements invoked by PartitionedDomain will be redirected to the specific Subdomain;

2. for any specific Subdomain, it inherits all the interfaces of Domain to do all the computations required by PartitionedDomain.

#### 202.3.2.5 Parallel Object-Oriented Finite Element Design

There has been much effort by researchers on parallel implementation of finite element computations, which can be categorized into either domain decomposition methods or parallel equation solving.

Domain decomposition is favored by many researchers due to its nice "divide and conquer" approach. The subdomains in the domain decomposition method are each assigned to a processing node, which will perform all the computations on that subdomain.

Of the domain decomposition methods, the substructuring method has been the most popular choice although other methods such as iterative substructuring Carter et al. (1989) and FETI (Finite Element Tearing and Interconnecting) Farhat and Roux (1991b); Farhat and Crivelli (1994) have also been used. In the substructuring method presented, static condensation is typically performed on the assembled system of equations.

Earlier works on parallel processing for inelastic mechanics focused on structural problems. We mention work by Noor et al. (1978); Utku et al. (1982); Storaasli and Bergan (1987) in which they used substructuring to achieve partitions. Fulton and Su (1992) developed techniques to account for different types of elements but used substructures of same element types (non–balanced computations). Hajjar and Abel (1988) developed techniques for dynamic analysis of framed structures with the objective of minimizing communications. Klaas et al. (1994) developed parallel computational techniques for elastic–plastic problems but tied the algorithm to the specific multiprocessor computers used (and specific network connectivity architecture). Farhat (1987) developed the so–called Greedy domain partitioning algorithm but stayed short of using redistribution of domains as a function of developed nonlinearities.

The major parallel programming model in OpenSees (McKenna, 1997) is the so-called Actor model, which is a mathematical model of concurrent computation that has its origins in Hewitt et al. (1973) . Actors Agha (1984) are autonomous and concurrently executing objects which execute asynchronously. Actors can create new actors and can send messages to other actors. The Actor model is an Object-Oriented version of message passing in which the Actors represent processes and the methods sent between Actors represent communications.

The Actor model adopts the philosophy that everything is an Actor. This is similar to the everything is an Object philosophy used by object-oriented programming languages, but differs in that object-oriented software is typically executed sequentially, while the Actor model is inherently concurrent, http://en.wikipedia.org/wiki/Actor_model.

An Actor is a computational entity with a behavior such that in response to each message received it can concurrently:

- send a finite number of messages to (other) Actors;
- create a finite number of new Actors;
- designate the behavior to be used for the next message received.

Note that there is no assumed sequence to above actions and that they could in fact be carried out in parallel.

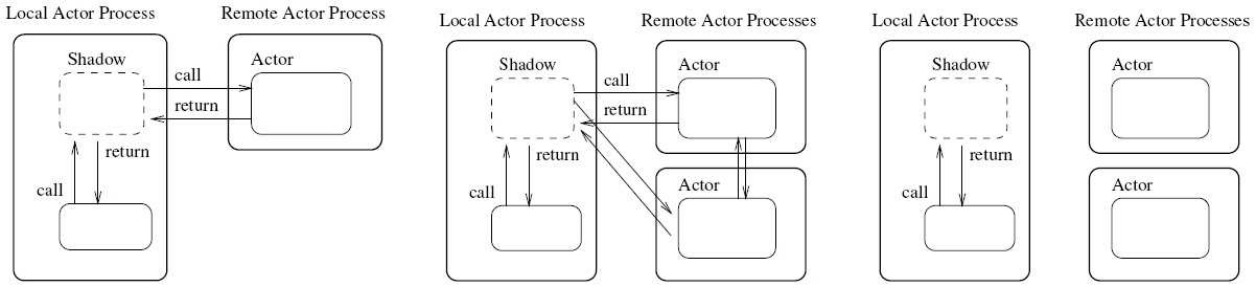
Communications with other Actors occur asynchronously (i.e. the sending Actor does not wait until the message has been received before proceeding with computation), which is the unblocking behavior.

Messages are sent to specific Actors, identified by address (sometimes referred to as the Actor's "mailing address"). As a result, an Actor can only communicate with Actors for which it has an address which it might obtain in the following ways:

- The address is in the message received;
- The address is one that the Actor already had, i.e. it was already an "acquaintance";
- The address is for a just created Actor.

The Actor model is characterized by inherent concurrency of computation within and among Actors, dynamic creation of Actors, inclusion of Actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

In order to minimize the changes to the sequential Domain Decomposition design presented in previous sections, McKenna (1997) introduces the Shadow class. A Shadow object is an object in an Actor's local address space. Each Shadow is associated with one Actor or multiple Actors in the case of an aggregation. The Shadow object represents the remote object to the objects in the local Actor's space. The Shadow object is responsible for sending an appropriate message to the remote Actor or Actors if broadcasting. The remote Actor(s) will then, if required, return the result to the local Shadow object, which in turn replies to the local object. The communication process is shown in Figure 202.10.




(a) Actor - Communication Required

(b) Aggregate - Communication Required

(c) No Communication Required

Figure 202.10: Communication Pattern of Actor-Shadow Models McKenna (1997)

Some other new classes of parallel finite element programming are:

- Channel is the bridge through which the Actors and Shadows can communicate.
- Address represents the location of a Channel object in the machine space. Channel objects send/receive information to/from other Channel objects, whose locations are given by the Address objects.

- MovableObject is an object which can send its state from one actor process to another.
- ObjectBroker is an object in a local actor process for creating new objects.
- MachineBroker is an object in a local actor process that is responsible for creating remote actor processes at the request of Shadow objects in the same local process.

The relation between these classes is shown in Figure 202.11.

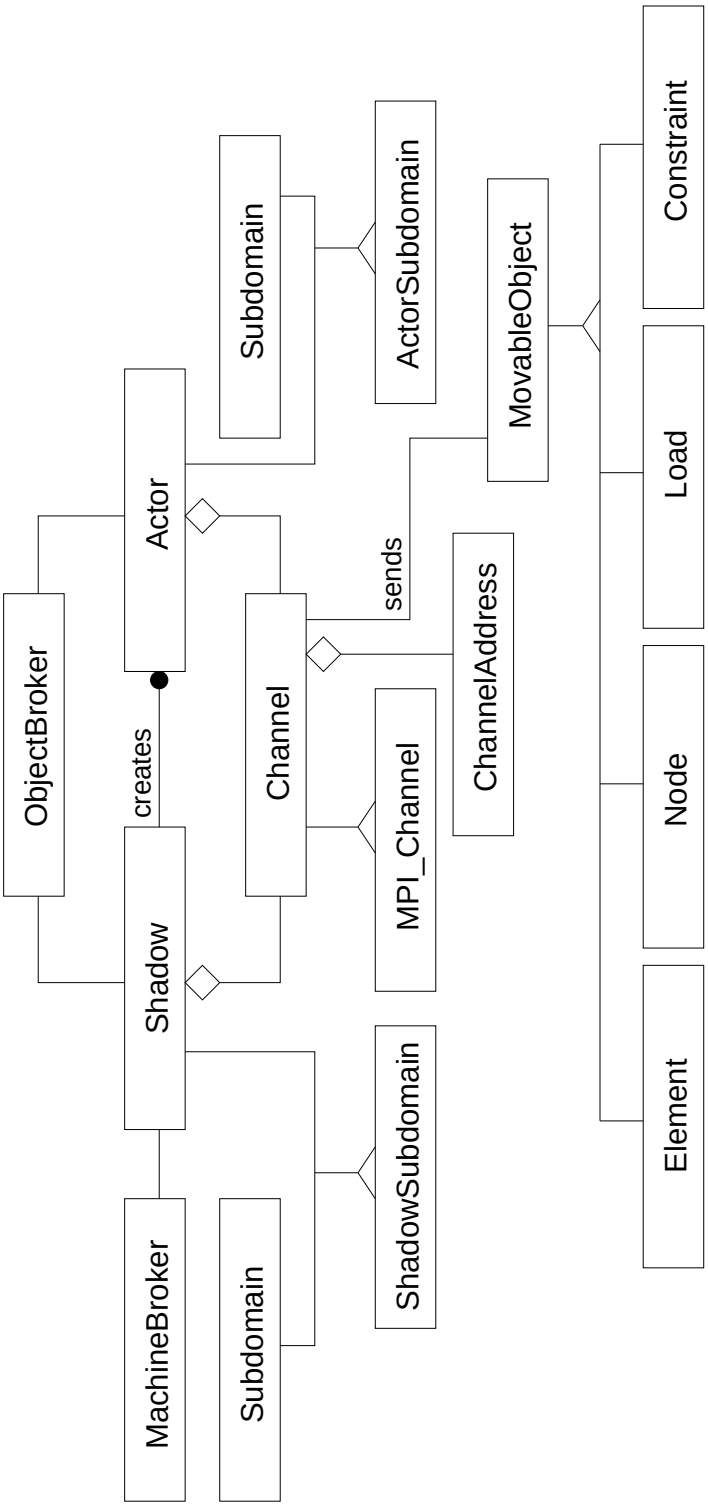



Figure 202.11: Class Diagram for Parallel Finite Element Analysis

### 202.3.3 Dual-Phase Adaptive Load Balancing

From the Figure 202.7, one can easily identify two computational phases that are fundamental to nonlinear elastic-plastic finite element simulations. One is well known as global level equation solving and the other is local level elemental calculations during which the elemental update happens for each element. In nonlinear elastic-plastic finite element simulations, the local computational phase can be much more expensive than the global equation solving phase due to the presence of complex material models and nonlinearity.

In this chapter, the implementation of proposed PDD algorithm has considered load balancing issues on both elemental level elastic-plastic computations and global level equation solving.

#### 202.3.3.1 Elemental Level Load Balancing

The load balancing operation on constitutive level is built on the foundation of adaptive multilevel graph partitioning algorithm available through ParMETIS.

In this chapter, element-based graph is constructed from the Finite Element mesh on which the graph partitioning algorithm acts on to obtain partitions and/or repartitions. Each element will be assigned a vertex tag for identification.

When two elements at least share a single node, we assign an edge to both vertices because the element graph is deemed to be undirected, which means the edge is equally identified by two vertices without ordering required.

We creatively specify vertex weight to represent elemental level computational load for each vertex (element). In the implementation of this chapter, the vertex weight will be automatically updated as simulation progresses to reflect element computation cost. Performance timing has been added for constitutive update routines and the graph data structure will be refreshed every single iteration.

The last metric used is the vertex size of each vertex which basically contains the information that how much memory each vertex (element) requires in order to reproduce itself to other processes during data distribution. Adaptive load balancing is a multi-objective operation in the sense that both edge cut and data migration cost must be minimized simultaneously. The vertex size exactly describes the size of data that need to be shipped via communication. This metric must be correctly obtained for all available element types in order for the multi-objective load balancing algorithm to ensure the best performance.

#### 202.3.3.2 Equation Solving Load Balancing

Parallel equation solving algorithm falls into two major different categories, direct solver and iterative solver.

Direct solver stems from Gaussian-type elimination and effective elimination tree is determined by the sparsity pattern of the stiffness matrix. Load balancing issue is addressed inherently when forming the elimination tree. Various packages such as SPOOLES and SuperLU provide scalable direct solutions to parallel equation systems. Chapter 110.5 discusses in further details about parallel direct solvers that are available as part of the release of this chapter.

Iterative solver has been the focus of this chapter in the sense that special care has been paid to achieve dynamic load balancing for each partition/repartition. The kernel of project-based iterative solvers is matrix-vector multiply. The issues of how to evenly distribute the stiffness matrix in parallel among different processors and how to reorder the sparse matrix to reduce data communications have been the focus of this chapter.

In order to achieve load balancing for parallel iterative solvers, parallel matrix/vector storage scheme and sparse matrix ordering are key factors. In the implementation of this chapter, even row-distribution of stiffness matrix among processing units is assumed. As shown in Figure 202.12, each processing unit has equal number of rows stored locally. The right hand side of the system is the force vector, which will be replicated for each processing unit. In this way, one can expect fastest matrix-vector multiply with the least amount of data needed to be communicated through network. As matrix-vector multiply is performed in parallel, load balancing issue is related to the number of nonzero numbers of the sparse stiffness matrix, which directly determines how many floating point multiplications are needed. In finite element computations, this nonzero pattern is determined by DOF numbering. Bandwidth reducing numbering scheme, or matrix ordering scheme, such as RCM Dongarra et al. (2003), can effectively lead to a sparse pattern that has similar number of nonzero elements on majority of rows as shown in Figure 202.12.

Finite element method inherently possesses compact support. Off-diagonal data of the stiffness matrix need to be synchronized among different processors. In order to reduce the extra overhead involved, in this chapter, several implementation solutions have been considered.

- Graph Partitioning Phase. As stated in previous chapters, minimizing edge-cut is one of the main objectives of the partitioning operation on the element graph. One extra benefit is that the bandwidth of the stiffness matrix will be greatly reduced. The number of nodes that need to be synchronized will be greatly reduced.

- DOF Numbering Phase. This phase is to renumber the DOFs of the finite element model after data redistribution in order to make sure contributions from local elements will sit on rows that are stored locally. This is done every time when the data migration is triggered. The idea is to start numbering the DOFs from local elements in Processor 1 to local elements in Processor N.

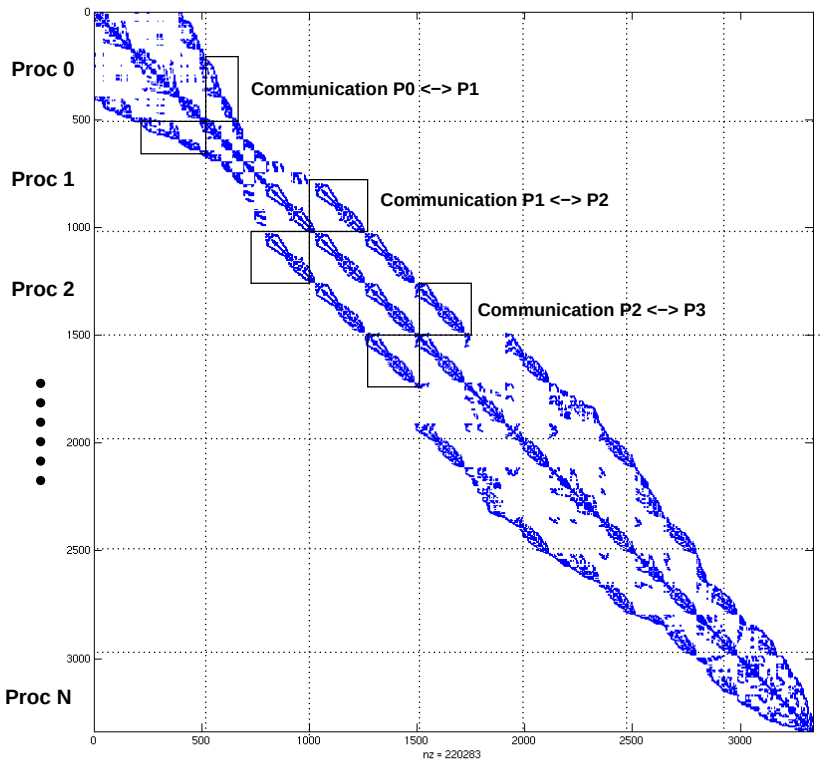



Figure 202.12: Parallel Data Organization of SFSI Equation System

In this way, when the global matrix is formed, local element stiffness matrix will always become clustered along the diagonal.

### 202.3.4 Object-Oriented Design of PDD

The parallel design of PDD basically follows Main-Follower algorithm structure as shown in Figure 202.13 and MPI has been adopted to facilitate inter-processor communications. The Actor/Shadow model described in previous sections is the used in PDD implementation and does nicely interact with parts of OpenSees framework, which uses Actor and Shadow classes to facilitate the inter-process communication between the main compute process and tied/follower compute processes.

- Main Compute Process

  Main compute process assumes the role to orchestrate the whole computation process. OpenSees uses tcl as an interpreter (or any other interpreted language that can be embedded into c or C++) to read input scripts from user. In parallel implementation of described here, main compute process

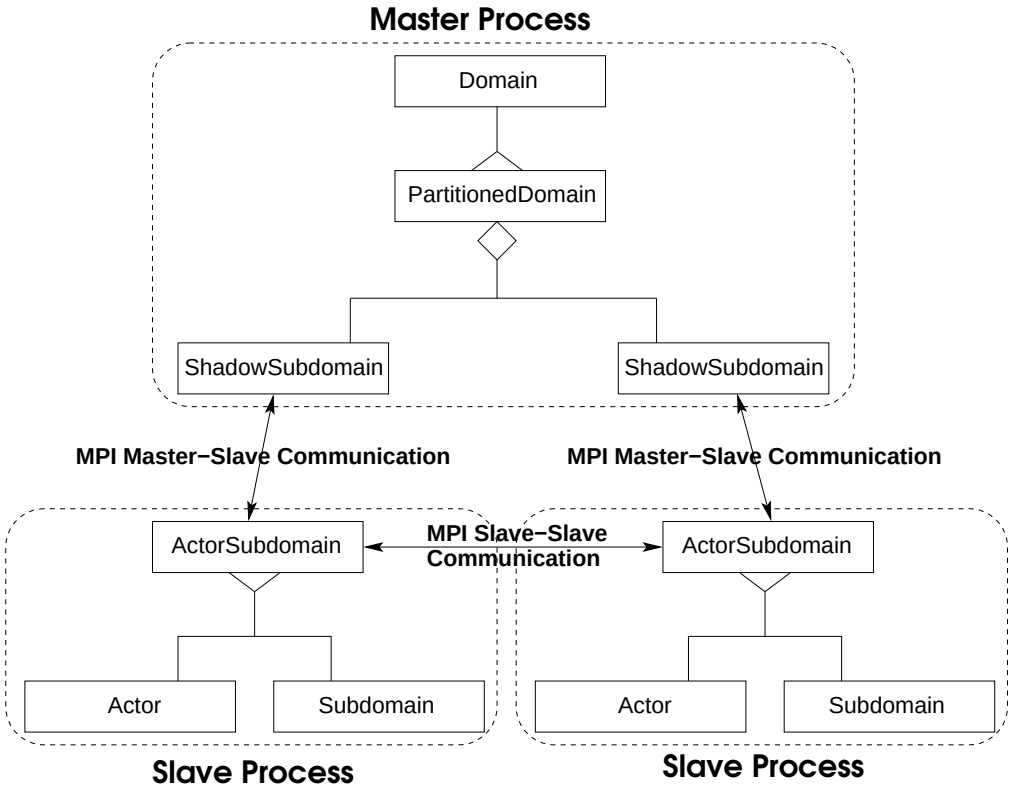



Figure 202.13: Main-Follower design used for PDD development.

is responsible for establishing the whole model for analysis and then distributing data among sub-processors. An important improvement in this chapter is that the main compute process does not actually create all finite element objects, whose memory space will only be allocated after they are sent to subdomains. This design helps avoid the high memory requirement on the main compute process side. Initial partitioning is done solely by main compute process process or in parallel by all working processes. Data movement is coordinated by the main compute process process, in which a complete element graph is kept intact.

As for repartitioning, the main compute process is still responsible for issuing commands to migrate data from this subprocessor to another even though the data is not in main compute process.

- Follower Compute Process

  The actor model has been used and modifications have been added to avoid unnecessary data communications. Basically speaking, actors in follower compute processes will be waiting for orders untill main compute process issues one and then do corresponding work on their own copy of data. The original design in OpenSees framework has disabled follower compute process to initiate communication, which means in order for a sub-processor to communicate with another sub-processor,

it has to send all the data back to the main compute process first. This is highly inefficient and needed to be redeveloped, improved. In this research actor model has been implemented to enable direct communications between sub-processes and this improvement greatly reduced unnecessary communications.

All of the class designs for sequential version of OpenSees can be reused in parallel version following the Object-Oriented paradigm. There are some very important additions however in order to facilitate main–follower parallel processing. In this section, these classes will be revisited and updated/changes/improvements originally developed during this research will be explained thereafter.

- PartitionedDomain

  The PartitionedDomain class basically inherits all functionality from the Domain class in sequential version. This class acts as a container class in the main compute process. It differs from Domain class in the respect that all actions performed on the domain will be propagated to all subdomains when doing parallel processing.

- Subdomain

  The Subdomain is a child class of Domain. This class will be instanced by each follower compute process and it covers all functionality of the Domain class in sequential version. It can be called as an instance of Domain taking care of components only for the local follower compute process.

- ActorSubdomain & ShadowSubdomain

  The Actor/Shadow Subdomain classes are the most important classes for parallel version OpenSees. They are assuming the roles to initiate and facilitate all communications between main and follower compute processes. Both Actor/Shadow Subdomain will be instanced automatically when user creates follower compute process.

  ShadowSubdomain sits on main compute process. The function of this class is to represent a specific follower compute process in main compute process. Main compute process does not directly interact with follower compute process. Whatever action that needs to be performed by the follower compute process will be issued to ShadowSubdomain. This extra layer smooths the communication between main and follower nodes.

  On the other hand, ActorSubdomain sits on follower compute process and it hides main compute process from follower compute processes/nodes. All commands from main compute node will be received by ActorSubdomain and ActorSubdomain will match the command with some actions performed by Subdomain.

  Actor/Shadow Subdomain are extremely important classes in the parallel implementation of this

chapter. They carry all communication functionality required to finish the partition and adaptive repartition.

- Channel
  Channel is the class that really does the job of sending/receiving data between processors. Only MPI channel has been used in this chapter. Specific data structure, such as ID (integer array), vector (double array) or matrix needs to provide its own implementation for send/receive functionality.

- FEMObjectBroker
  This class is instanced only at follower compute processors, which is in charge of creating new model data for subdomains. This design isolates model creation from communication classes.

- Address
  Address class identifies parallel processes. With MPI channel used, the address corresponds to global process ID.

- DomainPartitioner
  DomainPartitioner assumes the responsibilities of invoking the GraphPartitioner and feeding necessary data to finish the partition/repartition. This class will also be in charge of data migration after partition/repartition is done.

- SendSelf & RecvSelf
  These two should be called functions rather than classes. SendSelf & RecvSelf are functions implemented to provide copy of model data to finish sending/receiving operations.

The old parallel design of OpenSees is not capable of performing elastic–plastic computations since it was designed and implemented for a single stage loading only. This single stage loading works fine for elastic analysis, but since elastic–plastic materials do have memory, staged loading is essential for any realistic computations with elastic–plastic material. This is particularly true for geotechnical and structural models, where simulations support for staged loading (self weight of soil medium for initial stress, construction process and subsequent static or dynamic loading) is essential if any modeling accuracy is to be achieved. One of new developments in this chapter was the addition of multi-stage elastic-plastic analysis. This improvement included modification of 3D solid and beam elements, Template3Dep/NewTemplate3Dep material models and DRM loading pattern for seismic analysis. Some of the old utility commands, such as "wipeAnalysis", were improved/redeveloped to enable parallel multi-stage analysis.

The most significant significant improvement developed during research over the old parallel design of OpenSees is the introduction of load balancing technique by adaptive graph partitioning algorithm through ParMETIS. Major improvements/updates have been introduced in PartitionedDomain, Actor/ShadowSubdomain, DomainPartitioner, FEM_ObjectBroker and Subdomain. Modifications done in this chapter also focus very much on performance issue. In order to reduce unnecessary data communication during partitioning/repartitioning, some functions have been rewritten. The functionality of Actor and ShadowSubdomain have been expanded so that any ActorSubdomain can initiate communication to another ActorSubdomain. The old design of OpenSees had to use main compute process as intermediate layer if subdomains want to exchange information.

For example, if Subdomain No. 1 needs to migrate an Element to Subdomain No. 2, the old design would issue a "remove Element" command from main compute process PartitionedDomain to Subdomain No. 1, then Subdomain No. 1 would remove the Element and send the Element back to main compute process, finally the Element would be migrated to Subdomain No. 2. We can clearly recognize the communication to main compute process is not necessary here. In order to develop adaptive load balancing while minimizing data redistribution cost, the improvement in this chapter is to allow ActorSubdomain at source Subdomain initiates communication with ActorSubdomain at target Subdomains and they can exchange information without recourse to main compute process. So the new communication pattern will be, again for the "migrate element" case, the main compute process will issue an "export element" command to Subdomain No. 1 and a "receive element from Subdomain No. 1" command to Subdomain No. 2, and then the element information will be directly sent from Subdomain No. 1 to No. 2.

Details of implementation are given in following sections.

#### 202.3.4.1 MPI_Channel

- Functions sendnDarray and recvnDarray have been added to facilitate the data communication of Template3D material classes, which are based on nDarray tensor data structures.

```
int MPI_Channel::sendnDarray(int,int, const nDarray&, ChannelAddress*)
int MPI_Channel::recvnDarray(int,int, const nDarray&, ChannelAddress*)
```

#### 202.3.4.2 MPI_ChannelAddress

- Function getOtherTag has been added to get MPI global ID for the specific MPI_Channel. This function is mainly used for data migration. It provides the MPI global communicator ID of the target process which the next communication will be directed to.

int MPI_ChannelAddress::getOtherTag(void)

#### 202.3.4.3 FEM_ObjectBroker

- New functionality to instance 3D continuum brick elements has been added to getNewElement function.

  Element* FEM_ObjectBroker::getNewElement(EightNodeBrickTag)

- New functionality to instance Template3D/NewTemplate3D material models for continuum brick elements has been added to getNewNDMaterial function.

  NDMaterial* FEM_ObjectBroker::getNewNDMaterial(int)

- Template3D material is a stand-alone material library designed for general elastic-plastic materials. User can define separately YieldSurface, PotentialSurface, Scalar Evolution Law and Tensorial Evolution Law. Various material models have been implemented in OpenSees Jeremić and Yang (2002), such as Cam Clay, Drucker Prager and von Mises yield/potential surfaces, Armstrong Frederick nonlinear kinematic hardening law and bounding surface plasticity. All the material models have to be instanced by FEM_ObjectBroker during parallel processing.

  YieldSurface* FEM_ObjectBroker::getYieldSurfacePtr(int)

  PotentialSurface* FEM_ObjectBroker::getPotentialSurfacePtr(int)

  EvolutionLaw_S* FEM_ObjectBroker::getEL_S(int)

  EvolutionLaw_T* FEM_ObjectBroker::getEL_T(int)

- NewTemplate3D material is a newly designed material library which includes more advanced elastic-plastic constitutive models for geomaterials, such as Dafalias and Manzari 2004 model. The design of NewTemplate3D extends the principle of Template3D, in which key parameters describing plasticity model are abstracted as different class objects, such as YieldFunction, PlasticFlow, etc. In order to reduce unnecessary data allocation, new MaterialParameter class has been developed to carry all material parameters. New ElasticState has been used to store all intermediate and/or committed stress/strain data. All these material classes have to be instanced by FEM_ObjectBroker during parallel processing and new functions have been implemented in this chapter.

  MaterialParameter* FEM_ObjectBroker::getNewMaterialParameterPtr(void)

  ElasticState* FEM_ObjectBroker::getNewElasticStatePtr(int)

  YieldFunction* FEM_ObjectBroker::getNewYieldFunctionPtr(int)

  PlasticFlow* FEM_ObjectBroker::getNewPlasticFlowPtr(int)

  ScalarEvolution* FEM_ObjectBroker::getNewScalarEvolutionPtr(int)

  TensorEvolution* FEM_ObjectBroker::getNewTensorEvolutionPtr(int)

#### 202.3.4.4 Domain

- Timing routines have been added to update function to measure computation time of constitutive level iterations for each element during every single loading increment. This metric will be assigned to the corresponding vertex of the element graph as the vertex weight. This metric represents element-level computational load against which subsequent load balancing techniques will be applied.

#### 202.3.4.5 PartitionedDomain

- addElementalLoad function has been added to add ElementalLoad into LoadPattern, which was not supported in the old design.
  `bool PartitionedDomain::addElementalLoad(ElementalLoad*, int)`
- repartition function has been implemented to initiate adaptive repartitioning on the element graph of the Domain after every loading increment.
  `int PartitionedDomain::repartition(int)`

#### 202.3.4.6 Node & DOF_Group

- sendSelf and recvSelf functions for Node class have been changed mainly to deal with the DOF_Group object associated with the Node. In the old design of parallel version of OpenSees, only one-step static domain partitioning would be invoked so that there is no need to pass the DOF_Group. But in this chapter, adaptive load balancing is developed to achieve better performance. The Node class should keep the information of its own DOF_Group, which guarantees the consistency of the DOF_Graph of the whole Domain. This point is extremely important when user tries to invoke Transformation constraint handler on the DOF_Graph. The addition of this feature in Node improved the robustness of the whole program.
- DOF_Group is a class carries information about the DOF_Graph of the analysis model, which will be used to finish assembling the stiff/mass/damping matrices. Each Node has its own DOF_Group to record the IDs of degree of freedoms in the global analysis model. Function unSetMyNode has been introduced to avoid segmentation fault. The reason is that after each round of repartitioning, if data movement is required, the AnalysisModel will be wiped off but Nodes are still in existence. Introduction of unSetMyNode function separates Node from its DOF_Group so the DOF_Group can be wiped and regenerated for the new model. `void DOF_Group::unSetMyNode(void)`

#### 202.3.4.7 DomainPartitioner

DomainPartitioner is one of the most extensively changed classes in this chapter. This class acts as the entry point for PartitionedDomain to do domain decomposition and it basically has been rewritten to introduce new partition/repartition functionality and new data structures.

- Function repartition is implemented to do repartitioning after each loading increment. Partition and repartition are both implemented in parallel through ParMETIS library in this chapter. This function will collect ElementGraph from each Subdomain and pass them to GraphPartitioner. The global ElementGraph will be kept intact from which connectivity/adjacency information will be gathered to assemble child ElementGraphs and provide initial graph distribution data for repartition routines. After repartitioning by ParMETIS finishes, the function will verify the new partition against the original one to see if data redistribution is required to achieve load balancing. This repartition function also acts as a commander to control the data migration for adaptive load balancing. It issues commands to ShadowSubdomain to export/import Nodes, Elements, Constraintss, Loads, etc.

- The old design of OpenSees used multiplication of prime numbers as index number to record which partitions a specific node belongs to. This is a very good idea because with this approach, we only need one integer for each node to keep track of node partitions, which can be called as an index number for the node. The idea was to name each Subdomain with one specific prime number, if a node belongs to this Subdomain, we would multiply the index number of the node with the prime number of this Subdomain. In order to determine if a node belongs to on specific Subdomain, all we need is to divide the index number of the node with the prime number the Subdomain represents to see if we can get zero residual.

- The drawback of the old data structure based on prime numbers is that it only works when the number of processing units is small, say less than 16. In 3D continuum models, a single node might belong to up to 8 partitions simultaneously, which happens when a corner node sits on intersections of different Subdomains. As we know, prime numbers grow up very fast, multiplication with 8 prime numbers can easily overflow the index number of the node. A new data structure inspired by the Compressed Sparse Row (CSR) storage format popular in sparse matrix calculations has been introduced into in this chapter to solve the problem. One integer array has been used to store the partition data of all nodes, i.e. which partitions this node belongs to. Another integer array has been employed to record the count of partitions for each node. With these two arrays, we can load as many partitions as we want in our parallel processing.

#### 202.3.4.8 Shadow/ActorSubdomain

As mentioned in previous sections of this chapter, Shadow/ActorSubdomain are the most important classes in parallel design of OpenSees McKenna (1997). ShadowSubdomains represent Subdomains in the main compute process PartitionedDomain. If PartitionedDomain requires one specific Subdomain to carry out some operations, it will send out orders to the ShadowSubdomain associated with the target Subdomain. Then the ShadowSubdomain sets up communication channel to communicate with the Subdomain through ActorSubdomain. ActorSubdomain, on the other hand, sits on each child process as an agent receiving and processing incoming operation requests. The major improvements in this chapter include new functionality for adaptive repartitioning and data migration, and several other minor changes to reduce unnecessary data communications, such as when the Subdomain is required to removeElement, the new design won't send the element information out, etc. New features will be introduced in this section.

- ShadowActorSubdomain_Partition
  New design used ParMETIS to do parallel graph partitioning instead of sequential partitioning by METIS in old design. This improvement helps to reduce partition/repartition overhead and enable the parallel adaptive repartitioning for PDD algorithms proposed in this chapter.

- ShadowActorSubdomain_BuildElementGraph
  In order to provide input graphs for adaptive load balancing, all Subdomains have to construct their own subElementGraph, which will be fed into ParMETIS routines for repartitioning.

- ShadowActorSubdomain_Repartition
  The repartitioning is implemented in parallel in this chapter so this entry point is set in the ActorSubdomain for each Subdomain.

- ShadowActorSubdomain_reDistributeData
  If data migration is needed to achieve load balancing, the main compute process will orchestrate the data redistribution process and the functionality here helps to facilitate the data communications between processes. This is one of the major additions to the existing design. Starting from this point, the ActorSubdomain is able to handle all required data movement on its own and ActorSubdomains representing other Subdomains will connect to the current working ActorSubdomain to receive/send data. Logically only one ActorSubdomain will be doing ShadowActorSubdomain_reDistributeData while others including the main compute process will be listening to separate MPI port for data migration requests.

- ShadowActorSubdomain_recvChangedNodeList
  This function is used to simplify the data migration routine. With this function, only Nodes/Elements and their associated Constraints, Loads etc. need to be moved between processors.

- ShadowActorSubdomain_ChangeMPIChannel
  This function prepares the current ActorSubdomain for messages from some specific processes. It changes the destination/source for subsequent outgoing/incoming communications, which helps redistributing data after load balancing.

- ShadowActorSubdomain_restoreChannel
  The default communication pattern in the old design of OpenSees was one to one, main to follower computer processes. This function helps restoring communication patterns of the whole model after data redistribution finishes.

- ShadowActorSubdomain_swapNodeFromInternalToExternal
  Nodes that only belongs to one single Subdomain is called internal nodes whose information will be stored only in that specific Subdomain. While for those nodes that belong to more than one Subdomains, their information should be accessible from all Subdomains with which the nodes are associated. Those nodes are called external nodes instead. It is possible that former internal nodes to one Subdomain become external after the adaptive repartitioning. What the old design would do is to remove the internal nodes from that Subdomain, gather the information back to the main compute process and then distribute it externally among those Subdomains as indicated by the newly obtained partitions. The improvement in this chapter avoids unnecessary data communication between current working Subdomain and the main compute process Domain. We can just swap the node in working Subdomain from internal status to external status and then export them to other specified Subdomains. This new design can improve performance if the data migration is extensive by avoiding unnecessary communications.

- ShadowActorSubdomain_swapNodeFromExternalToInternal
  This function is introduced due to the same reason as described previously although now the swapping direction is in reverse. It is noted that along with the swapping, removing operations must be invoked for those Subdomains that does not contain the node anymore.

- ShadowActorSubdomain_exportInternalNode
  This function handles the situation when a Node does not belong to the current Subdomain after adaptive repartitioning. The node will be removed from current Subdomain and exported to

other Subdomains specified by the graph repartitioning. This again avoids the unnecessary data communication to/from main compute process by directly sending data to other Subdomains.

- ShadowActorSubdomain_resetRecorders
  The Recorders have to be reset after data migration to reflect component changes in each Subdomain.

#### 202.3.4.9 Send/RecvSelf

As stated in previous sections, Send/RecvSelf must be provided by all domain components to finish data communication operations, such as Nodes, Elements, Loads, Constraints, Materials etc. In this chapter, new communication functions have been developed for EightNodeBrick element, ElasticIsotropic3D material, Template3Dep/NewTemplate3Dep material. The basic requirement to implement Send/RecvSelf is to replicate the source object instance in target process. For the old design, only one-step initial partitioning is performed and thus greatly simplifies the Send/RecvSelf routines because all the analysis-related information is null or void and only geometry-related data need to be transferred. But in this chapter, data migration is needed periodically to achieve load balance so the Send/RecvSelf has to be redesigned to carry analysis-related information besides the geometry model data. This is extremely important for Element and Material classes because they contain intermediate iteration/solution data of nonlinear finite element simulations. Figure 202.15 shows the class diagrams of brick Element and the associated Template3Dep material model. Send/RecvSelf operations have been implemented also for all classes associated with Template3Dep which are necessary to define a complete material model, such as Cam Clay, Drucker Prager and von Mises PotentialSurfaces, Cam Clay, Drucker Prager and von Mises YieldSurfaces, linear and nonlinear isotropic and kinematic hardening rules, etc.

### 202.3.5 Graph Partitioning

Graph partitioning approach has been extensively used in implementing domain decomposition type parallel finite element method. The element-based graph naturally becomes the favorite due to the fact that elemental operation forms the fundamental calculation unit in finite element analysis.

In this chapter, element graph has been constructed upon which graph partitioning algorithm acts to get domain decomposition for parallel finite element analysis. In the current implementation of this chapter, vertices of the element graph represent elements of the analysis model. Vertex weight is then specified as the computational load of each element. In elastic-plastic finite element simulations, the most expensive part has shown to be the elemental level calculations, which include constitutive-level stress update (strain-driven constitutive driver assumed) and formulation of elastic-plastic modulus

(or so-called tangent stiffness tensor/matrix). In this research, the wall clock time used by elemental calculations has been dynamically collected and specified as the corresponding vertex weight for each element. The elemental calculation time clearly tells whether the element is elastic or plastified. With this timing metric, the graph can effectively reflect load distribution among elements thus load balancing repartition can be triggered on the graph to redistribute element between processors to achieve more balanced elastic-plastic calculation.

On the other hand, vertex size has to be defined for repartitioning problem as mentioned in previous sections. In this research, vertex size has been specified to be redistribution cost associated with each element. This information depends on the parallel implementation of the software and is discussed in the section immediately following.

#### 202.3.5.1 Construction of Element Graph

Each element is considered as one vertex in the element graph. An edge is formed when two elements share a common node. In this chapter, the graph structure is assumed to be undirected, which means the same edge will be added to both vertex ends. The edge is weightless in our application considering the fact that the purpose of minimizing edge-cut is to reduce the data migration when assembling global stiffness matrix. In that sense, the edge of element graph should carry the same weight or, more directly no weight at all.

#### 202.3.5.2 Interface to ParMETIS/METIS

Interfaces to both ParMETIS and METIS have been implemented in this chapter. ParMETIS is the parallel implementation of METIS and new adaptive repartitioning functionality is only available through ParMETIS.

All of the graph routines in ParMETIS/METIS take as input the adjacency structure of the graph, the weights of the vertices and edges (if any), and an array describing how the graph is distributed among the processors Karypis et al. (2003). The structure of the graph is represented by the compressed storage format (CSR), extended for the context of parallel distributed-memory computing. We will first describe the CSR format for serial graphs and then describe how it has been extended for storing graphs that are distributed among processors.

- Serial CSR Format The CSR format is a widely-used scheme for storing sparse graphs. Here, the adjacency structure of a graph is represented by two arrays, $xadj$ and $adjncy$. Weights on the vertices and edges (if any) are represented by using two additional arrays, $vwgt$ and $adjwgt$. For example, consider a graph with $n$ vertices and $m$ edges. In the CSR format, this graph can be

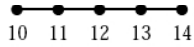
described using arrays of the following sizes:

$$xadj[n+1], \quad vwgt[n], \quad adjncy[2m], \quad \text{and} \quad adjwgt[2m] \tag{202.1}$$

Note that the reason both *adjncy* and *adjwgt* are of size $2m$ is because every edge is listed twice (i.e., as $(v,u)$ and $(u,v)$). Also note that in the case in which the graph is unweighted (i.e., all vertices and/or edges have the same weight), then either or both of the arrays *vwgt* and *adjwgt* can be set to *NULL*. ParMETIS_V3_AdaptiveRepart additionally requires a *vsize* array. This array is similar to the *vwgt* array, except that instead of describing the amount of work that is associated with each vertex, it describes the amount of memory that is associated with each vertex.

The adjacency structure of the graph is stored as follows. Assuming that vertex numbering starts from 0 (C style), the adjacency list of vertex $i$ is stored in array *adjncy* starting at index $xadj[i]$ and ending at (but not including) index $xadj[i+1]$ (in other words, $adjncy[xadj[i]]$ up through and including $adjncy[xadj[i+1]-1]$). Hence, the adjacency lists for each vertex are stored consecutively in the array *adjncy*. The array *xadj* is used to point to where the list for each specific vertex begins and ends. Figure 202.14(a) illustrates the CSR format for the 15-vertex graph shown in Figure 202.14(b). If the graph has weights on the vertices, then $vwgt[i]$ is used to store the weight of vertex $i$ . Similarly, if the graph has weights on the edges, then the weight of edge $adjncy[j]$ is stored in $adjwgt[j]$. This is the format that is used by (serial) METIS library routines.

- Distributed CSR Format ParMETIS uses an extension of the CSR format that allows the vertices of the graph and their adjacency lists to be distributed among the processors. In particular, PARMETIS assumes that each processor $P_i$ stores $n_i$ consecutive vertices of the graph and the corresponding $m_i$ edges, so that $n = \sum_i n_i$ , and $2m = \sum_i m_i$. Here, each processor stores its local part of the graph in the four arrays $xadj[n_i+1]$, $vwgt[n_i]$, $adjncy[m_i]$, and $adjwgt[m_i]$, using the CSR storage scheme. Again, if the graph is unweighted, the arrays *vwgt* and *adjwgt* can be set to *NULL*. The straightforward way to distribute the graph for PARMETIS is to take $n/p$ consecutive adjacency lists from *adjncy* and store them on consecutive processors (where $p$ is the number of processors). In addition, each processor needs its local *xadj* array to point to where each of its local vertices' adjacency lists begin and end. Thus, if we take all the local *adjncy* arrays and concatenate them, we will get exactly the same *adjncy* array that is used in the serial CSR. However, concatenating the local *xadj* arrays will not give us the serial *xadj* array. This is because the entries in each local *xadj* must point to their local *adjncy* array, and so, $xadj[0]$ is zero for all processors. In addition to these four arrays, each processor also requires the array $vtxdist[p+1]$ that indicates the range of vertices that are local to each processor. In particular, processor $P_i$ stores the vertices from $vtxdist[i]$ up to (but not including) vertex $vtxdist[i+1]$.




(a) A sample graph

Description of the graph on a serial computer (serial MeTiS)

| | |
|---|---|
| xadj | 0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44 |
| adjncy | 1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 |

(b) Serial CSR format

Description of the graph on a parallel computer w

| | | |
|---|---|---|
| Processor 0: | xadj | 0 2 5 8 11 13 |
| | adjncy | 1 5 0 2 6 1 3 7 2 |
| | vtxdist | 0 5 10 15 |
| Processor 1: | xadj | 0 3 7 11 15 18 |
| | adjncy | 0 6 10 1 5 7 11 2 |
| | vtxdist | 0 5 10 15 |
| Processor 2: | xadj | 0 2 5 8 11 13 |
| | adjncy | 5 11 6 10 12 7 11 |
| | vtxdist | 0 5 10 15 |

(c) Distributed CSR form

Figure 202.14: An example of the parameters passed to PARMETIS in a three processor case Karypis et al. (2003).

Figure 202.14(c) illustrates the distributed CSR format by an example on a three-processor system. The 15-vertex graph in Figure 202.14(a) is distributed among the processors so that each processor gets 5 vertices and their corresponding adjacency lists. That is, Processor Zero gets vertices 0 through 4, Processor One gets vertices 5 through 9, and Processor Two gets vertices 10 through 14. This figure shows the *xadj*, *adjncy*, and *vtxdist* arrays for each processor. Note that the *vtxdist* array will always be identical for every processor. All five arrays that describe the distributed CSR format are defined in PARMETIS to be of type *idxtype*. By default *idxtype* is set to be equivalent to type *int* (i.e., integers). However, *idxtype* can be made to be equivalent to a *short int* for certain architectures that use 64-bit integers by default. (Note that doing so will cut the memory usage and communication time required approximately in half.) The conversion of *idxtype* from *int* to

*short* can be done by modifying the file parmetis.h. (Instructions are included there.) The same *idxtype* is used for the arrays that store the computed partitioning and permutation vectors.

When multiple vertex weights are used for multi-constraint partitioning, the $c$ vertex weights for each vertex are stored contiguously in the *vwgt* array. In this case, the *vwgt* array is of size $nc$, where $n$ is the number of locally stored vertices and $c$ is the number of vertex weights (and also the number of balance constraints).

New GraphPartitioner class ParMETIS has been developed in this chapter to provide seamless interface to adaptive partitioning/repartitioning routines.

### 202.3.6 Data Redistribution

Data redistribution after repartitioning has been a challenging problem which needs careful study to guarantee correctness of subsequent analysis. In this research, Object-Oriented philosophy has been followed to abstract container classes to facilitate analysis and model data redistribution after repartition. As for the initial partitioning, only model data, such as geometry parameters, has to be exported to sub-processors, while in adaptive repartitioning finite element simulation, analysis data has to be moved as well. It is extremely important to have well-designed container classes to carry data around. Basic units of finite element analysis, such as nodes and elements naturally become our first choices. Not to give up generality, the design in OpenSees adopts basic iterative approach for nonlinear finite element analysis Crisfield (1997b), important intermediate analysis data include trial data, commit data, incremental data, element residual, element tangent stiffness, etc. Vertex size of each element has been defined as the total number of bytes that have to be transferred between sub-processors.

1. Node

   Other than geometric data such as node coordinates and number of degree of freedoms, the Node class contains nodal displacement data which should be sent together with the node to preserve continuity of the analysis model.

2. Element

   Element class is the basic construction unit in finite element model. In the design of this research, Element class keeps internal links to Template3D material class Jeremić and Yang (2002). In order to facilitate elastic-plastic simulation, EPState class is constructed to hold all the intermediate response data. This object-oriented abstraction greatly systematize the communication pattern. The information on class design is shown in the class diagram Figure 202.15 by Rational Rose Boggs and Boggs (2002).

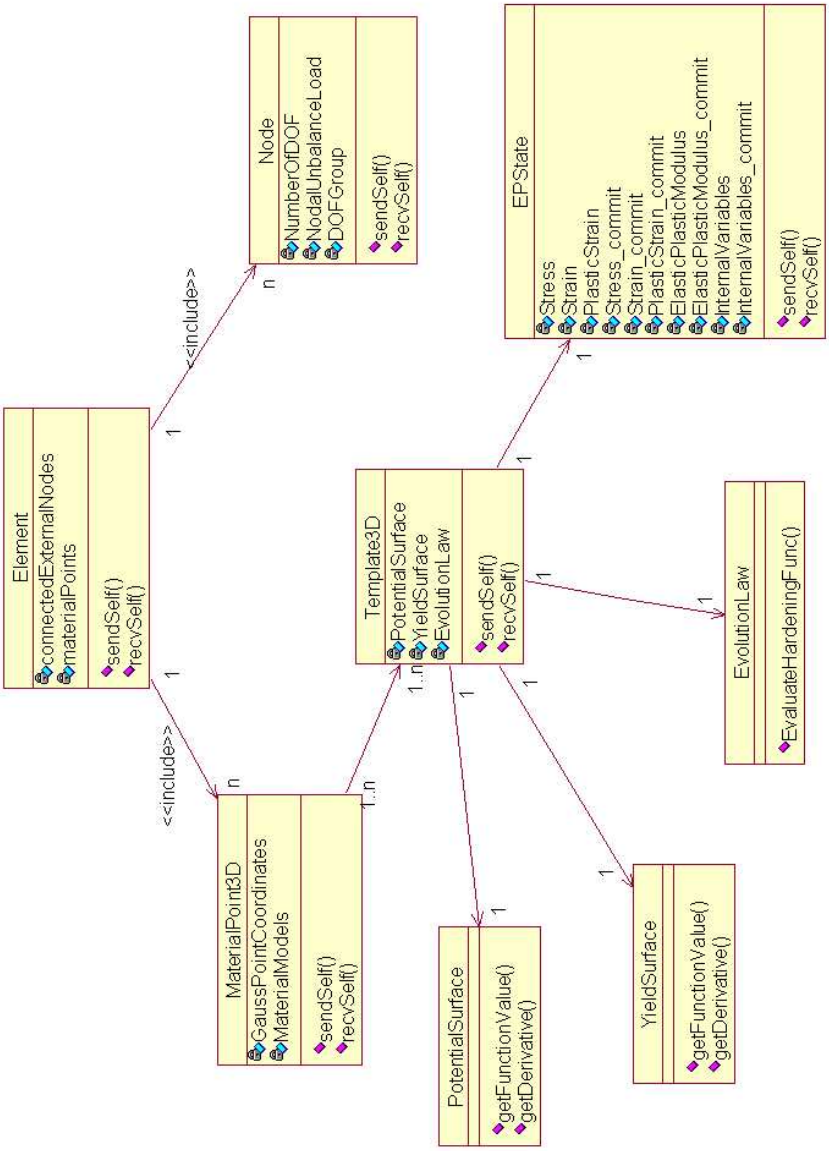



Figure 202.15: Class Diagram: Major Container Classes for Data Redistribution

All data communication operations have been implemented through the standard Send/RecvSelf interface, which forms a complete set of consistent point-to-point communication patterns and is convenient for future additions of new element/materials.

Chapter 203

# Library Centric Software Platform Design

(1993-1994-1996-2005-2009-2010-2011-2019-)

## 203.1 Chapter Summary and Highlights

Veldhuizen (2005); Stroustrup (2005); Ramey (2005); Veldhuizen (2005);

### 203.1.1 Finite Elements

#### 203.1.1.1 Single Phase Solid Elements

8 Node Brick

20 Node Brick

27 Node Brick

8-20 Node Brick

#### 203.1.1.2 Fully Coupled, Two-Phase (Porous Solid – Pore Fluid) Solid Elements

u-p-U

u-p

#### 203.1.1.3 Structural Elements

Truss

Beam

#### 203.1.1.4 Special Elements

Contact Element

Seismic Isolator Element

### 203.1.2 Constitutive Integration and Material Models

#### 203.1.2.1 Explicit Integration

#### 203.1.2.2 Implicit Integration

#### 203.1.2.3 Material Models

### 203.1.3 Modified OpenSees Services Library

PDD...

# Chapter 204

# Application Programming Interface

(2005-2009-2010-2011-2017-2019-)

(In collaboration with Dr. Nima Tafazzoli and Dr. Yuan Feng

## 204.1 Chapter Summary and Highlights

## 204.2 Introduction

Bloch (2005);

Dmitriev (2004); Stroustrup (2005); Niebler (2005); Mernik et al. (2005); Ward (2003);

## 204.3 Application Programming Interface for Domain Specific Language (DSL)

### 204.3.1 Modeling

---

Start new loading stage:

```
int start_new_stage(string CurrentStageName);
```

---

```
define_model_name(string theModelName)
```

---

```
obtain_pseudo_time()
```

---

```
wipe_model()
```

---

```
check_mesh(string outputfilename)
```

---

---

---

#### 204.3.1.1 Modeling: Material Models

---

```
int add_constitutive_model_NDMaterial_linear_elastic_isotropic_3d(int MaterialNumber,
                double ElasticModulus,
                double nu,
                double rho)
```

MaterialNumber: Number of the predefined ND material to be used;
ElasticModulus: elastic modulus;
nu: Poisson's ratio;
rho: density;

---

```
add_constitutive_model_NDMaterial_linear_elastic_crossanisotropic(int MaterialNumber,
                double Ehp,
                double Evp,
                double nuhvp,
                double nuhhp,
                double Ghvp,
                double rhop)
```

`MaterialNumber`: Number of the ND material to be used ; `Ehp`: Elastic modulus on "horizontal" direction ; `Evp`: Elastic modulus on "vertical" direction ; `nuhvp`: Poisson ratio for "horizontal" - "vertical" direction ; `nuhhp`: Poisson ratio for "horizontal" - "horizontal" direction ; `Ghvp`: Shear moduls for "horizontal" - "vertical" direction ; `rhop`: density ;

---

```
add_constitutive_model_NDMaterial_vonmises_perfectly_plastic(int MaterialNumber,
                int Algorithm,
                double rho,
                double E,
                double v,
                double k,
```

```
                double initialconfiningstress,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

MaterialNumber: Number of the ND material to be used ; Algorithm: Explicit (=0) or Implicit (=1) ; rho: density ; E: Elastic modulus ; v: Poisson's ratio ; k: initial radius of von Mises cylinder ; initialconfiningstress: initial confining pressure (possitive for compression) ;

---

```
add_constitutive_model_NDMaterial_vonmises_isotropic_hardening(int MaterialNumber,
                int Algorithm,
                double rho,
                double E,
                double v,
                double k,
                double H,
                double initialconfiningstress,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

MaterialNumber: Number of the ND material to be used ; Algorithm: Explicit (=0) or Implicit (=1) ; rho: density ; E: Elastic modulus ; v: Poisson's ratio ; k: initial radius of von Mises cylinder ; H: rate of isotropic hardening ; initialconfiningstress: initial confining pressure (possitive for compression) ;

---

```
add_constitutive_model_NDMaterial_vonmises_kinematic_hardening(int MaterialNumber,
                int Algorithm,
                double rho,
                double E,
                double v,
```

```
                double k,
                double ha,
                double Cr,
                double initialconfiningstress,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

MaterialNumber: Number of the ND material to be used ; Algorithm: Explicit (=0) or Implicit (=1) ; rho: density ; E: Elastic modulus ; v: Poisson's ratio ; k: initial radius of von Mises cylinder ; ha: Armstrong-Frederick nonlinear kinematic hardening constant, initial slope ; Cr: Armstrong-Frederick nonlinear kinematic hardening constant, asymptote ; initialconfiningstress: initial confining pressure (possitive for compression) ;

---

```
add_constitutive_model_NDMaterial_vonmises_linear_kinematic_hardening(int MaterialNumber,
                int Algorithm,
                double rho,
                double E,
                double v,
                double k,
                double H,
                double initialconfiningstress,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

MaterialNumber: Number of the ND material to be used ; Algorithm: Explicit (=0) or Implicit (=1) ; rho: density ; E: Elastic modulus ; v: Poisson's ratio ; k: initial radius of von Mises cylinder ; H: Kinematic hardening rate; initialconfiningstress: initial confining pressure (possitive for compression) ;

---

```
add_constitutive_model_NDMaterial_druckerprager_perfectly_plastic(int MaterialNumber,
                int Algorithm,
                double rho,
                double E,
                double v,
                double k,
                double initialconfiningstress,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

`MaterialNumber`: numer/Number of the nD material to be used ; `AlgorithmType`: Explicit (=0) or Implicit (=1) ; `rho`: density ; `E`: Elastic modulus ; `v`: Poisson's ratio ; `k`: initial equivalent friction angle ; `initialconfiningstress`: initial confining pressure (positive for compression) ;

---

```
add_constitutive_model_NDMaterial_druckerprager_isotropic_hardening(int MaterialNumber,
                int Algorithm,
                double rho,
                double E,
                double v,
                double k,
                double H,
                double initialconfiningstress,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

`MaterialNumber`: number/Number of the nD material to be used ; `AlgorithType`: Explicit (=0) or Implicit (=1) ; `rho`: density ; `E`: Elastic modulus ; `v`: Poisson's ratio ; `k`: initial equivalent friction angle ; `H`: rate of isotropic hardening ; `initialconfiningstress`: initial confining pressure (positive for compression) ;

```
add_constitutive_model_NDMaterial_druckerprager_kinematic_hardening(int MaterialNumber,
                int Algorithm,
                double rho,
                double E,
                double v,
                double k,
                double ha,
                double Cr,
                double initialconfiningstress,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

`MaterialNumber`: number/Number of the ND material to be used ; `Algorithm`: Explicit (=0) or Implicit (=1) ; `rho`: density ; `E`: Elastic modulus ; `v`: Poisson's ratio ; `k`: initial equivalent friction angle ; `ha`: Armstrong-Frederick nonlinear kinematic hardening constant, initial slope ; `Cr`: Armstrong-Frederick nonlinear kinematic hardening constant, asymptote ; `initialconfiningstress`: initial confining pressure (positive for compression) ;

---

```
add_constitutive_model_NDMaterial_camclay(int MaterialNumber,
                int Algorithm,
                double rho,
                double e0,
                double M,
                double lambda,
                double kappa,
                double v,
                double Kc,
                double P0,
                double initialconfiningstress,
```

```
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

MaterialNumber: number/Number of the material to be used ; AlgorithmType: Explicit (=0) or Implicit (=1) ; rho: density ; e0: initial void ratio ; M: slope of the critical state line ; lambda: slope of the Normal Consolidation Line (NCL) ; kappa: slope of the Unloading-Reloading Line (URL) ; v: Poisson ratio ; Kc: Bulk modulus ; initialconfiningstress: initial confining stress/pressure (positive for compression) ;

---

```
add_constitutive_model_NDMaterial_sanisand_2004(int MaterialNumber,
                int Algorithm,
                double rho,
                double e0,
                double G0,
                double nu,
                double Pat,
                double p_cut,
                double Mc,
                double c,
                double lambda_c,
                double xi,
                double ec_ref,
                double m,
                double h0,
                double ch,
                double nb,
                double A0,
                double nd,
                double z_max,
                double cz,
                double initialconfiningstress,
```

```
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

MaterialNumber: number/Number of the nD material to be used ; AlgorithmType: Explicit (=0) or Implicit (=1) ; rho: density ; e0: initial viod ratio ; G0: elastic shear modulus (same unit as stress) ; nu: Poisson's ratio ; Pat: atmospheric pressure ; p_cut: pressure cut-off ratio ; Mc: ; c: tension-compression strength ratio ; lambda_c: parameter for critical state line ; xi: parameter for critical state line ; ec_ref: reference void ratio for critical state line, ; $e_c$ = $e_r lambda(p_c/P_{at})^x i$ m: opening of the yield surface ; h0: bounding parameter ; ch: bounding parameter ; nb: bounding parameter ; A0: dilatancy parameter ; nd: dilatancy parameter ; z_max: fabric parameter ; cz: fabric parameter ; initialconfiningstress initial confining pressure (positive for compression) ;

---

```
add_constitutive_model_NDMaterial_sanisand_2008(int MaterialNumber,
                int Algorithm,
                double rho,
                double e0,
                double G0,
                double K0,
                double Pat,
                double k_c,
                double alpha_cc,
                double c,
                double lambda,
                double xi,
                double ec_ref,
                double m,
                double h0,
                double ch,
                double nb,
                double A0,
                double nd,
```

```
                double p_r,
                double rho_c,
                double theta_c,
                double X,
                double z_max,
                double cz,
                double p0,
                double p_in,
                int number_of_subincrements,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

MaterialNumber: Number of the ND material to be used ; Algorithm: Explicit (=0) or Implicit (=1) ; rho: density ; e0: initial void ratio at zero strain ; G0: Reference elastic shear modulus (same unit as stress) ; K0: Reference elastic bulk modulus (same unit as stress) ; Pat: atmospherics pressure for critical state line ; k_c: cut-off factor; for $p < k_c P_{at}$, use $p = k_c P_{at}$ for calculation of $G$; (a default value of $k_c = 0.01$ should work fine) ; alpha_cc: critical state stress ratio ; c: tension-compression strength ratio ; lambda: parameter for critical state line ; xi: parameter for critical state line ; ec_ref: reference void for critical state line, ; $e_c = e_r lambda(p_c/P_{at})^x i$ ; m: opening of the yield surface ; h0: bounding parameter ; ch: bounding parameter ; nb: bounding parameter ; A0: dilatancy parameter ; nd: dilatancy parameter ; p_r: LCC parameter ; rho_c: LCC parameter ; theta_c: LCC parameter ; X: LCC parameter ; z_max: fabric parameter ; cz: fabric parameter ; p0: yield surface size ; p_in ;

---

```
add_constitutive_model_NDMaterial_pisano(int tag,
                        double E_in,
                        double v_in,
                        double M_in,
                        double kd_in,
                        double xi_in,
                        double h_in,
                        double m_in,
                        double rho_in,
```

```
                    double initialconfiningstress_in,
                    double beta_min_in)
```

---

```
add_constitutive_model_NDMaterial_accelerated_vonmises_perfectly_plastic(int MaterialNumber,
                double rho,
                double E,
                double v,
                double k,
                double initialconfiningstress,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

---

```
add_constitutive_model_NDMaterial_accelerated_vonmises_isotropic_hardening(int MaterialNumber,
                double rho,
                double E,
                double v,
                double k,
                double H,
                double initialconfiningstress,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

---

```
add_constitutive_model_NDMaterial_accelerated_vonmises_kinematic_hardening(int MaterialNumber,
                double rho,
                double E,
                double v,
                double k,
                double ha,
                double Cr,
                double initialconfiningstress,
```

```
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

---

```
add_constitutive_model_NDMaterial_accelerated_vonmises_linear_kinematic_hardening(int MaterialN
                double rho,
                double E,
                double v,
                double k,
                double H,
                double initialconfiningstress,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

---

---

```
add_constitutive_model_NDMaterial_accelerated_druckerprager_perfectly_plastic(int MaterialNumbe
                double rho,
                double E,
                double v,
                double k,
                double initialconfiningstress,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

---

```
add_constitutive_model_NDMaterial_accelerated_druckerprager_isotropic_hardening(int MaterialNum
                double rho,
                double E,
                double v,
```

```
                double k,
                double H,
                double initialconfiningstress,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

---

```
add_constitutive_model_NDMaterial_accelerated_druckerprager_kinematic_hardening(int MaterialNum
                double rho,
                double E,
                double v,
                double k,
                double ha,
                double Cr,
                double initialconfiningstress,
                int maximum_number_of_iterations,
                double tolerance_1,
                double tolerance_2)
```

---

```
add_constitutive_model_NDMaterial_accelerated_camclay(int MaterialNumber,
            double rho,
            double e0,
            double M,
            double lambda,
            double kappa,
            double v,
            double Kc,
            double p0,
            double initialconfiningstress,
            int maximum_number_of_iterations,
            double tolerance_1,
            double tolerance_2)
```

```
add_constitutive_model_uniaxial_elastic(int MaterialNumber,
                double elasticmodulus,
                double eta)
```

MaterialNumber: unique material Number ; elasticmodulus: elastic modulus of the material ; eta: damping tangent ;

```
add_constitutive_model_uniaxial_concrete02(int MaterialNumber,
  double fpc, double epsc0, double fpcu,
  double epscu, double rat, double ft,
  double Ets)
```

MaterialNumber: unique material Number ; fpc: compressive strength ; epsc0: strain at compressive strength ; fpcu: crushing strength ; epsU: strain at crushing strength ; lambda: ratio between unloading slope at epscu and initial slope ; ft: tensile strength ; Ets: tension softening stiffness (absolute value) (slope of the linear tension softening branch) ;

```
int add_constitutive_model_uniaxial_steel01(int MaterialNumber,
                double fy,
                double Ep,
                double Hd,
                int a1,
                int a2,
                int a3,
                int a4)
```

MaterialNumber: unique material Number ; fy: yield strength ; Ep: initial elastic tangent ; Hd: strain-hardening ratio (ratio between post-yield tangent and initial elastic tangent) ; a1, a2, a3, a4: isotropic hardening parameters ; a1: isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic strain of a2*(fy/Ep). ; a2: isotropic hardening parameter (see explanation under a1) ; a3: isotropic hardening parameter, increase of tension yield

envelope as proportion of yield strength after a plastic strain of a4*(fy/Ep) ; a4: isotropic hardening parameter (see explanation under a3) ;

---

```
int add_constitutive_model_uniaxial_steel02(int MaterialNumber,
  double fy, double E0, double b,
  double R0, double cR1, double cR2,
  double a1, double a2, double a3, double a4)
```

MaterialNumber: unique material object Number ; fy: yield strength ; E0: initial elastic tangent ; b: strain-hardening ratio (ratio between post-yield tangent and initial elastic tangent) ; R0, cR1, cR2: control the transition from elastic to plastic branches. Recommended values: R0=between 10 and 20, cR1=0.925, cR2=0.15 ; a1, a2, a3, a4: isotropic hardening parameters ; a1: isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic strain of a2*(Fy/E). ; a2: isotropic hardening parameter (see explanation under a1) ; a3: isotropic hardening parameter, increase of tension yield envelope as proportion of yield strength after a plastic strain of a4*(Fy/E) ; a4: isotropic hardening parameter (see explanation under a3) ;

#### 204.3.1.2 Modeling: Nodes

---

```
int add_node(int NodeNumber,
             int number_of_DOFs,
             double coordinate_x,
             double coordinate_y,
             double coordinate_z)
```

NodeNumber: integer Number identifying node ; number_of_DOFs: number of degrees of freedom for node ; coordinate_x: x coordinate of the node ; coordinate_y: y coordinate of the node ; coordinate_z: z coordinate of the node ;

---

```
int remove_node(int NodeNumber)
```

NodeNumber: integer Number identifying the node to be removed ;

---

```
int add_mass_to_node(int NodeNumber,
                     double massvalue1,
                     double massvalue2,
                     double massvalue3)

int add_mass_to_node(int NodeNumber,
                   double massvalue1,
                   double massvalue2,
                   double massvalue3,
                   double massvalue4,
                   double massvalue5,
                   double massvalue6)

int add_mass_to_node(int NodeNumber,
                   double massvalue1,
                   double massvalue2,
                   double massvalue3,
                   double massvalue4,
                   double massvalue5,
                   double massvalue6,
                   double massvalue7)
```

NodeNumber: integer Number of the node that mass would be applied to ; massvalue(#): the amount of mass assigned to each degree of freedom ;

#### 204.3.1.3 Modeling: Finite Elements

---

```
add_element_truss(int ElementNumber,
                int iNode,
                int jNode,
                int MaterialNumber,
                double sectionarea,
```

```
                double rho)
```

ElementNumber: unique element object Number ; dimension: number of dimensions of the beam ; iNode , jNode: end nodes ; MaterialNumber: Number of the uniaxial material to be used ; sectionarea: section area of the truss element ; rho: density ;

---

```
add_element_beam_elastic(int ElementNumber,
                double A,
                double E,
                double G,
                double Jx,
                double Iy,
                double Iz,
                int iNode,
                int jNode,
                double rho,
                double vecxzPlane_X, double vecxzPlane_Y, double vecxzPlane_Z,
                double jntOffsetI_X, double jntOffsetI_Y, double jntOffsetI_Z,
                double jntOffsetJ_X, double jntOffsetJ_Y, double jntOffsetJ_Z)
```

ElementNumber: unique element object Number ; A: section area ; E: Young's modulus ; G: Shear Modulus ; Jx: polar moment of inertia ; Iy: moment of inertia around y ; Iz: moment of inertia around z ; iNode , jNode: end nodes ; TransformationNumber: identifier for previously-defined coordinate-transformation (CrdTransf) object ; rho: density ; sectionTag: identifier for previously-defined section object ;

---

```
add_element_beam_elastic_lumped_mass(int ElementNumber,
                double A,
                double E,
                double G,
                double Jx,
                double Iy,
```

```
                double Iz,
                int iNode,
                int jNode,
                double rho,
                double vecxzPlane_X, double vecxzPlane_Y, double vecxzPlane_Z,
                double jntOffsetI_X, double jntOffsetI_Y, double jntOffsetI_Z,
                double jntOffsetJ_X, double jntOffsetJ_Y, double jntOffsetJ_Z)
```

ElementNumber: unique element object Number ; A: section area ; E: Young's modulus ; G: Shear Modulus ; Jx: polar moment of inertia ; Iy: moment of inertia around y ; Iz: moment of inertia around z ; iNode , jNode: end nodes ; TransformationNumber: identifier for previously-defined coordinate-transformation (CrdTransf) object ; rho: density ; sectionTag: identifier for previously-defined section object ;

---

```
add_element_brick_8node(int ElementNumber,
                int node_numb_1,
                int node_numb_2,
                int node_numb_3,
                int node_numb_4,
                int node_numb_5,
                int node_numb_6,
                int node_numb_7,
                int node_numb_8,
                int MaterialNumber)
```

elementTag: unique element object Number ; node_numb_#: eight node numbers specified in appropriate order ; materialTag: material Number associated with previsouly-defined NDMaterial object ;

---

```
add_element_brick_8node_elastic(int ElementNumber,
                int node_numb_1,
                int node_numb_2,
```

```
                int node_numb_3,
                int node_numb_4,
                int node_numb_5,
                int node_numb_6,
                int node_numb_7,
                int node_numb_8,
                int MaterialNumber)
```

elementTag: unique element object Number ; node_numb_#: eight node numbers specified in appropriate order ; materialTag: material Number associated with previsouly-defined NDMaterial object ;

---

```
add_element_brick_20node(int ElementNumber,
                             int node_numb_1,
                             int node_numb_2,
                             int node_numb_3,
                             int node_numb_4,
                             int node_numb_5,
                             int node_numb_6,
                             int node_numb_7,
                             int node_numb_8,
                             int node_numb_9,
                             int node_numb_10,
                             int node_numb_11,
                             int node_numb_12,
                             int node_numb_13,
                             int node_numb_14,
                             int node_numb_15,
                             int node_numb_16,
                             int node_numb_17,
                             int node_numb_18,
                             int node_numb_19,
                             int node_numb_20,
```

```
                                int MaterialNumber)
```

ElementNumber: unique element object Number ; node_numb_#: eight node numbers specified in appropriate order ; MaterialNumber: material Number associated with previsouly-difined NDMaterial object ;

---

```
add_element_brick_20node_elastic(int ElementNumber,
                                int node_numb_1,
                                int node_numb_2,
                                int node_numb_3,
                                int node_numb_4,
                                int node_numb_5,
                                int node_numb_6,
                                int node_numb_7,
                                int node_numb_8,
                                int node_numb_9,
                                int node_numb_10,
                                int node_numb_11,
                                int node_numb_12,
                                int node_numb_13,
                                int node_numb_14,
                                int node_numb_15,
                                int node_numb_16,
                                int node_numb_17,
                                int node_numb_18,
                                int node_numb_19,
                                int node_numb_20,
                                int MaterialNumber)
```

ElementNumber: unique element object Number ; node_numb_#: eight node numbers specified in appropriate order ; MaterialNumber: material Number associated with previsouly-difined NDMaterial object ;

---

```
int add_element_brick_27node(int ElementNumber,
                             int node_numb_1,
                             int node_numb_2,
                             int node_numb_3,
                             int node_numb_4,
                             int node_numb_5,
                             int node_numb_6,
                             int node_numb_7,
                             int node_numb_8,
                             int node_numb_9,
                             int node_numb_10,
                             int node_numb_11,
                             int node_numb_12,
                             int node_numb_13,
                             int node_numb_14,
                             int node_numb_15,
                             int node_numb_16,
                             int node_numb_17,
                             int node_numb_18,
                             int node_numb_19,
                             int node_numb_20,
                             int node_numb_21,
                             int node_numb_22,
                             int node_numb_23,
                             int node_numb_24,
                             int node_numb_25,
                             int node_numb_26,
                             int node_numb_27,
                             int MaterialNumber)
```

`ElementNumber`: unique element object Number ; `node_numb_#`: eight node numbers specified in appropriate order ; `MaterialNumber`: material Number associated with previsouly-difined NDMaterial object ;

```
int add_element_brick_27node_elastic(int ElementNumber,
                                     int node_numb_1,
                                     int node_numb_2,
                                     int node_numb_3,
                                     int node_numb_4,
                                     int node_numb_5,
                                     int node_numb_6,
                                     int node_numb_7,
                                     int node_numb_8,
                                     int node_numb_9,
                                     int node_numb_10,
                                     int node_numb_11,
                                     int node_numb_12,
                                     int node_numb_13,
                                     int node_numb_14,
                                     int node_numb_15,
                                     int node_numb_16,
                                     int node_numb_17,
                                     int node_numb_18,
                                     int node_numb_19,
                                     int node_numb_20,
                                     int node_numb_21,
                                     int node_numb_22,
                                     int node_numb_23,
                                     int node_numb_24,
                                     int node_numb_25,
                                     int node_numb_26,
                                     int node_numb_27,
                                     int MaterialNumber)
```

`ElementNumber`: unique element object Number ; `node_numb_#`: eight node numbers specified in appropriate order ; `MaterialNumber`: material Number associated with previsouly-difined NDMaterial object ;

```
add_element_brick_8node_up(int ElementNumber,
                  int node_numb_1,
                  int node_numb_2,
                  int node_numb_3,
                  int node_numb_4,
                  int node_numb_5,
                  int node_numb_6,
                  int node_numb_7,
                  int node_numb_8,
                  int MaterialNumber,
                  double porosity,
                  double alpha,
                  double rho_s,
                  double rho_f,
                  double k_x,
                  double k_y,
                  double k_z,
                  double K_s,
                  double K_f)
```

---

```
add_element_brick_8node_upU(int ElementNumber,
                  int node_numb_1,
                  int node_numb_2,
                  int node_numb_3,
                  int node_numb_4,
                  int node_numb_5,
                  int node_numb_6,
                  int node_numb_7,
                  int node_numb_8,
                  int MaterialNumber,
                  double porosity,
                  double alpha,
                  double rho_s,
```

```
                double rho_f,
                double k_x,
                double k_y,
                double k_z,
                double K_s,
                double K_f)
```

---

```
add_element_brick_20node_upU(int ElementNumber,
                                int node_numb_1,
                                int node_numb_2,
                                int node_numb_3,
                                int node_numb_4,
                                int node_numb_5,
                                int node_numb_6,
                                int node_numb_7,
                                int node_numb_8,
                                int node_numb_9,
                                int node_numb_10,
                                int node_numb_11,
                                int node_numb_12,
                                int node_numb_13,
                                int node_numb_14,
                                int node_numb_15,
                                int node_numb_16,
                                int node_numb_17,
                                int node_numb_18,
                                int node_numb_19,
                                int node_numb_20,
                                int MaterialNumber,
                                double porosity,
                                double alpha,
                                double rho_s,
                                double rho_f,
```

```
                                    double k_x,
                                    double k_y,
                                    double k_z,
                                    double K_s,
                                    double K_f)
```

---

```
add_element_brick_8node_variable_number_of_gauss_points(int ElementNumber,
                  int number_of_gauss_points,
                  int node_numb_1,
                  int node_numb_2,
                  int node_numb_3,
                  int node_numb_4,
                  int node_numb_5,
                  int node_numb_6,
                  int node_numb_7,
                  int node_numb_8,
                  int MaterialNumber)
```

---

```
add_element_brick_20node_variable_number_of_gauss_points(int ElementNumber,
                               int number_of_gauss_points,
                               int node_numb_1,
                               int node_numb_2,
                               int node_numb_3,
                               int node_numb_4,
                               int node_numb_5,
                               int node_numb_6,
                               int node_numb_7,
                               int node_numb_8,
                               int node_numb_9,
                               int node_numb_10,
                               int node_numb_11,
                               int node_numb_12,
```

```
                                int node_numb_13,
                                int node_numb_14,
                                int node_numb_15,
                                int node_numb_16,
                                int node_numb_17,
                                int node_numb_18,
                                int node_numb_19,
                                int node_numb_20,
                                int MaterialNumber)
```

---

```
add_element_brick_27node_variable_number_of_gauss_points(int ElementNumber,
                                int number_of_gauss_points,
                                int node_numb_1,
                                int node_numb_2,
                                int node_numb_3,
                                int node_numb_4,
                                int node_numb_5,
                                int node_numb_6,
                                int node_numb_7,
                                int node_numb_8,
                                int node_numb_9,
                                int node_numb_10,
                                int node_numb_11,
                                int node_numb_12,
                                int node_numb_13,
                                int node_numb_14,
                                int node_numb_15,
                                int node_numb_16,
                                int node_numb_17,
                                int node_numb_18,
                                int node_numb_19,
                                int node_numb_20,
                                int node_numb_21,
```

```
                                int node_numb_22,
                                int node_numb_23,
                                int node_numb_24,
                                int node_numb_25,
                                int node_numb_26,
                                int node_numb_27,
                                int MaterialNumber)
```

---

```
add_element_brick_variable_node_8_to_27(int ElementNumber, int number_of_gauss_points,
                                        int node_numb_1,  int node_numb_2,  int node_numb_3
                                        int node_numb_4,  int node_numb_5,  int node_numb_6
                                        int node_numb_7,  int node_numb_8,  int node_numb_9
                                        int node_numb_10, int node_numb_11, int node_numb_1
                                        int node_numb_13, int node_numb_14, int node_numb_1
                                        int node_numb_16, int node_numb_17, int node_numb_1
                                        int node_numb_19, int node_numb_20, int node_numb_2
                                        int node_numb_22, int node_numb_23, int node_numb_2
                                        int node_numb_25, int node_numb_26, int node_numb_2
                                        int MaterialNumber)
```

---

```
add_element_contact_3dof_to_3dof(int ElementNumber,
                 int iNode,
                 int jNode,
                 double Knormal,
                 double Ktangent,
                 double frictionRatio,
                 double x_local_1,
                 double x_local_2,
                 double x_local_3)
```

---

```
add_element_contact_nonlinear_3dof_to_3dof(int ElementNumber,
                 int iNode,
```

```
                int jNode,
                double Knormal,
                double Ktangent,
                double frictionRatio,
   double maxmimum_gap,
   double maximum_normal_stiffness,
                double x_local_1,
                double x_local_2,
                double x_local_3)
```

---

```
add_element_contact_nonlinear_3dof_to_7dof(int ElementNumber,
                int iNode,
                int jNode,
                double Knormal,
                double Ktangent,
                double frictionRatio,
                double maximum_gap,
                double maximum_normal_stiffness,
                double x_local_1,
                double x_local_2,
                double x_local_3)
```

---

```
add_element_shell_andes_3node(int ElementNumber,
                           int node_numb_1,
                           int node_numb_2,
                           int node_numb_3,
                           double thickness,
                           int MaterialNumber)
```

---

```
add_element_shell_andes_4node(int ElementNumber,
                           int node_numb_1,
                           int node_numb_2,
```

```
                                int node_numb_3,
                                int node_numb_4,
                                double thickness,
                                int MaterialNumber)
```

---

```
add_element_shell_MITC4(int ElementNumber,
                                    int node_numb_1,
                                    int node_numb_2,
                                    int node_numb_3,
                                    int node_numb_4,
                                    double thickness,
                                    int MaterialNumber)
```

---

```
add_element_shell_NewMITC4(int ElementNumber,
                                    int node_numb_1,
                                    int node_numb_2,
                                    int node_numb_3,
                                    int node_numb_4,
                                    double thickness,
                                    int MaterialNumber)
```

---

```
add_element_penalty(int ElementNumber,
int node1,
int node2,
double penalty_stiffness,
int dof)
```

---

```
add_element_penalty_for_applying_generalized_displacement(int ElementNumber,
                          int Exist_Node,
                          double penalty_stiffness,
                          int direction)
```

```
add_element_rank_one_deficient_elastic_pinned_fixed_beam(int ElementNumber,
                 double A,
                 double E,
                 double G,
                 double Jx,
                 double Iy,
                 double Iz,
                 int iNode,
                 int jNode,
                 double rho,
                 double vecxzPlane_X, double vecxzPlane_Y, double vecxzPlane_Z,
                 double jntOffsetI_X, double jntOffsetI_Y, double jntOffsetI_Z,
                 double jntOffsetJ_X, double jntOffsetJ_Y, double jntOffsetJ_Z)
```

```
add_element_beam_displacement_based(int ElementNumber,
                 int iNode,
                 int jNode,
                 int numberofintegrationpoints,
                 int SectionNumber,
                 double rho,
                 string integrationrule,
                 double vecInLocXZPlane_x, double vecInLocXZPlane_y, double vecInLocXZPlane_z,
                 double rigJntOffset1_x, double rigJntOffset1_y, double rigJntOffset1_z,
                 double rigJntOffset2_x, double rigJntOffset2_y, double rigJntOffset2_z)
```

```
int remove_element(int ElementNumber)
```

ElementNumber: number identifying the element to be removed ;

#### 204.3.1.4 Modeling: Damping

```
int add_damping_rayleigh(int dampingNumber,
                         double a0,
                         double a1,
                         string which_stiffness_to_use)
```

dampingNumber: damping Number number to be used in element definition ; a0, a1: Rayleigh order damping coefficients ; which_stiffness_to_use: Initial_Stiffness/Current_Stiffness/Last_Commited_Stif

---

```
add_damping_caughey3rd(int dampingNumber, double a0, double a1, double a2, string which_stiffne
```

---

```
add_damping_caughey4th(int dampingNumber, double a0, double a1, double a2, double a3, string wh
```

---

```
int add_damping_to_element(int elementNumber,
                           int dampingNumber)
```

dampingNumber: damping number to be assigned to element ; elementNumber: element number which damping is going to be assigned to ;

---

```
int add_damping_to_node(int nodeNumber,
                int dampingNumber)
```

dampingNumber: damping number to be assigned to node (note that only the mass proportional coefficient will be used for node) ; nodeNumber: node number which damping is going to be assigned to ;

#### 204.3.1.5 Modeling: Constraints, Supports, Tied Nodes Connections, etc.

---

```
int add_support_to_all_dofs_of_node(int NodeNumber)
```

NodeNumber: integer Number of the node to be fixed ;

---

```
int add_support_to_node(int NodeNumber,
                        int dof_number)
```

NodeNumber: integer Number identifying the node to be constrained ; dof_number: dof to be fixed ;

---

```
int add_equaldof_to_two_nodes(int nodeRetain,
                              int nodeConstr,
                              int dofID1,
                              int dofID2,
                              ...,
                              int dofID7)
```

nodeRetain: integer Number identifying the retained, or master node (rNode) ; nodeConstr: integer Number identifying the constrained, or slave node (cNode) ; dofID: nodal degrees-of-freedom that are constrained at the nodeConstr to be the same as those at the nodeRetain Valid range is from 1 to 7. ;

---

```
int remove_support_from_node_by_fixity_number(int FixityNumber)
```

FixityNumber: integer Number identifying the fixity to be removed ;

---

```
remove_support_from_node(int NodeNumber, int dofNumber)
```

NodeNumber: integer Number identifying the node number ; dofNumber: integer Number identifying the dof number ;

---

```
int remove_equaldof_from_node(int NodeNumber)
```

FixityNumber: integer Number identifying the fixity to be removed ;

#### 204.3.1.6 Modeling: Static Loads

---

```
add_force_time_history_linear(int PatternNumber,
                                                    int NodeNumber,
                                                    int dof_to_be_shaken,
                                                    double final_load_value)
```

---

```
add_force_time_history_path_series(int PatternNumber,
                                                    int NodeNumber,
                                                    int dof_to_be_shaken,
                                                    double TimeIncrement,
                                                    double LoadFactor,
                                                    string Forceinputfilename)
```

---

```
add_force_time_history_path_time_series(int PatternNumber,
                                                    int NodeNumber,
                                                    int dof_to_be_shaken,
                                                    double LoadFactor,
                                                    string Forceinputfilename)
```

---

---

```
add_load_selfweight_to_element(int SelfWeightNumber,
                     int ElementNumber, int AccelerationFieldNumber)
```

---

```
add_acceleration_field(int GravityFieldNumber,
                 double accelerationfield_x,
                 double accelerationfield_y,
                 double accelerationfield_z)
```

---

```
add_load_constant_normal_pressure_to_8node_brick_surface(int SurfaceLoadNumber,
                            int ElementNumber,
                            int Node_1,
                            int Node_2,
                            int Node_3,
                            int Node_4,
                            double SurfaceLoadMagnitude)
```

```
add_load_different_normal_pressure_to_8node_brick_surface(int SurfaceLoadNumber,
                            int ElementNumber,
                            int Node_1,
                            int Node_2,
                            int Node_3,
                            int Node_4,
                            double SurfaceLoadMagnitude1,
                            double SurfaceLoadMagnitude2,
                            double SurfaceLoadMagnitude3,
                            double SurfaceLoadMagnitude4)
```

```
add_load_constant_normal_pressure_to_20node_brick_surface(int SurfaceLoadNumber,
                            int ElementNumber,
                            int Node_1,
                            int Node_2,
                            int Node_3,
                            int Node_4,
                            int Node_5,
                            int Node_6,
                            int Node_7,
                            int Node_8,
                            double SurfaceLoadMagnitude)
```

```
add_load_different_normal_pressure_to_20node_brick_surface(int SurfaceLoadNumber,
                            int ElementNumber,
                            int Node_1,
                            int Node_2,
                            int Node_3,
                            int Node_4,
                            int Node_5,
                            int Node_6,
                            int Node_7,
                            int Node_8,
                            double SurfaceLoadMagnitude1,
                            double SurfaceLoadMagnitude2,
                            double SurfaceLoadMagnitude3,
                            double SurfaceLoadMagnitude4,
                            double SurfaceLoadMagnitude5,
                            double SurfaceLoadMagnitude6,
                            double SurfaceLoadMagnitude7,
                            double SurfaceLoadMagnitude8)
```

---

```
add_load_constant_normal_pressure_to_27node_brick_surface(int SurfaceLoadNumber,
                            int ElementNumber,
                            int Node_1,
                            int Node_2,
                            int Node_3,
                            int Node_4,
                            int Node_5,
                            int Node_6,
                            int Node_7,
                            int Node_8,
                            int Node_9,
                            double SurfaceLoadMagnitude)
```

---

```
add_load_different_normal_pressure_to_27node_brick_surface(int SurfaceLoadNumber,
```

```
                                int ElementNumber,
                                int Node_1,
                                int Node_2,
                                int Node_3,
                                int Node_4,
                                int Node_5,
                                int Node_6,
                                int Node_7,
                                int Node_8,
                                int Node_9,
                                double SurfaceLoadMagnitude1,
                                double SurfaceLoadMagnitude2,
                                double SurfaceLoadMagnitude3,
                                double SurfaceLoadMagnitude4,
                                double SurfaceLoadMagnitude5,
                                double SurfaceLoadMagnitude6,
                                double SurfaceLoadMagnitude7,
                                double SurfaceLoadMagnitude8,
                                double SurfaceLoadMagnitude9)
```

---

```
add_penalty_displacement_time_history_linear(int PatternNumber,
                                             int PenaltyElementNumber,
                                             int dof_to_be_shaken,
                                             double Final_Displacement_Value)
```

---

```
add_penalty_displacement_time_history_path_series(int PatternNumber,
                                             int PenaltyElementNumber,
                                             int dof_to_be_shaken,
                                             double TimeIncrement,
                                             double LoadFactor,
                                             string Displacementinputfilename)
```

---

```
add_single_point_constraint_to_node(int NodeNumber,
                                    int dof_number,
                                    double DOFvalue)
```

---

#### 204.3.1.7 Modeling: Dynamic Loads

---

`add_load_pattern_domain_reduction_method`

```
add_load_pattern_domain_reduction_method(int PatternNumber,
                                         double dt,
                                         double factor,
                                         int numberofsteps,
                                         int numberofdrmnodes,
                                         int numberofdrmelements,
                                         double xpositive,
                                         double xminus,
                                         double ypositive,
                                         double yminus,
                                         double zpositive,
                                         double zminus,
                                         string ElementNumbersFilename,
                                         string NodeNumbersFilename,
                                         string DisplacementTimeHistoryFilename,
                                         string AccelerationTimeHistoryFilename)
```

Inputs: `PatternNumber`: number assigned to DRM load pettern ; `dt`: time interval of input files for time histories ; `factor`: factor to multiply to the input time history ; `numberofsteps`: Number of the time steps in acceleration/displacement time history ; `numberofdrmnodes`: Number of the nodes in DRM layer ; `numberofdrmelements`: Number of the elements in DRM layer ; `xpositive, xminus`: boundary layer range in x direction ; `ypositive, yminus`: boundary layer range in y direction ; `zpositive, zminus`: boundary layer range in z direction ; `ElementNumbersFilename`:

File including element numbers inside the platic bowl (1 element number per line in the input file) ; NodeNumbersFilename: File including node numbers inside the platic bowl (1 node number per line in the input file) ; DisplacementTimeHistoryFilename: File including displacement time history (in each line write the values of displacement in time for first degree of freedom of the first node defined in NodeNumbersFilename, next line should have the values for second dof of the first node and continue for all degrees of freedom. Then move to the second node defined in NodeNumbersFilename and ... ; AccelerationTimeHistoryFilename: File including acceleration time history (in each line write the values of displacement in time for first degree of freedom of the first node defined in NodeNumbersFilename, next line should have the values for second dof of the first node and continue for all degrees of freedom. Then move to the second node defined in NodeNumbersFilename and ... ;

---

```
remove_load(int LoadPatternNumber)
```

---

```
add_load_pattern_domain_reduction_method_save_forces(int PatternNumber,
                                                     double dt,
                                                     double factor,
                                                     int numberofsteps,
                                                     int numberofdrmnodes,
                                                     int numberofdrmelements,
                                                     double xpositive,
                                                     double xminus,
                                                     double ypositive,
                                                     double yminus,
                                                     double zpositive,
                                                     double zminus,
                                                     string ElementNumbersFilename,
                                                     string NodeNumbersFilename,
                                                     string DisplacementTimeHistoryFilename,
                                                     string AccelerationTimeHistoryFilename,
                                                     string ForceTimeHistoryFilename)
```

---

```
add_load_pattern_domain_reduction_method_restore_forces(int PatternNumber,
```

```
                                        double dt,
                                        double factor,
                                        int numberofsteps,
                                        int numberofdrmnodes,
                                        int numberofdrmelements,
                                        double xpositive,
                                        double xminus,
                                        double ypositive,
                                        double yminus,
                                        double zpositive,
                                        double zminus,
                                        string ElementNumbersFilename,
                                        string NodeNumbersFilename,
                                        string ForceTimeHistoryFilename)
```

#### 204.3.1.8 Modeling: Prescribed Displacements

---

```
add_imposed_motion(int GroundMotionNumber,
                   int NodeNumber,
                   int degree_of_freedom,
                   double timestep,
                   double displacement_scale,
                   string displacementfilename,
                   double velocity_scale,
                   string velocityfilename,
                   double acceleration_scale,
                   string accelerationfilename)
```

#### 204.3.1.9 Solid-Fluid Interface

Two new APIs and corresponding DSL commands have been added to RealESSI.

- define solid fluid interface "interface_name"

  This API is used to define solid fluid interface. Passing parameters into RealESSI to initialize our interface class SSFI.

- simulate No. steps using solid fluid interaction transient algorithm time_step = < time >
  This API aims to launch solid fluid transient interaction analysis. The time step defined here refers to the time step for the transient analysis in solid domain. The time step for fluid domain can be different and defined in the input files for OpenFOAM.

#### 204.3.1.10 Outputs to mySQL database

---

```
restore_response_of_model_mysql_format(int Node_Number, int DOF_Number, int Step_Number,
                                       string databaseName, string host,
                                       string username, string password, unsigned int port,
                                       string socket)
```

---

```
restore_state_of_model_mysql_format(string databaseName, string host,
                                    string username, string password, unsigned int port,
                                    string socket)
```

---

```
save_response_of_model_mysql_format(string databaseName, string host,
                                    string username, string password, unsigned int port,
                                    string socket)
```

---

```
save_state_of_model_mysql_format(string databaseName, string host,
                                 string username, string password, unsigned int port,
                                 string socket)
```

### 204.3.2 Simulation

#### 204.3.2.1 Simulation: Solvers

Definition of system of equation (linear) solvers to be used.

---

```
int define_solver_profilespd_for_analysis()
```

```
int define_solver_umfpack_for_analysis()
```

```
int define_solver_petsc_for_analysis()
```

#### 204.3.2.2 Simulation: Static Solution Advancement

Definition of static solution advancement algorithms (see more in section 107.6 on page 526).

```
int define_static_solution_advancement_integrator_displacement_control(int node_number,
        int doftomove,
        double dispincrement)
```

`dispincrement`: increment of displacement in each step of analysis ; `node_number`: node whose response controls the solution ; `doftomove`: degree-of-freedom whose response controls the solution. Valid range is from 1 through the number of nodal degrees-of-freedom. ;

```
int define_static_solution_advancement_integrator_load_control(double loadstep)
```

`loadstep`: load step size ;

#### 204.3.2.3 Simulation: Dynamic Solution Advancement

Definition of dynamic, time integration/advancement algorithms (see more in section 108.3 on page 538).

```
int define_dynamic_solution_advancement_integrator_hilber_hughes_taylor(double HHT_Alpha)
```

`HHT_Alpha`: HHT $\alpha$ parameter ;

```
int define_dynamic_solution_advancement_integrator_newmark(double gamma, double beta)
```

`newmark_gamma, newmark_beta`: Newmark $\gamma$ and $\beta$ parameters ;

#### 204.3.2.4 Simulation: Solution Algorithms

Definition of solution algorithms to be used:

---

```
int define_algorithm_with_no_convergence_check_for_analysis()
```

---

```
int define_algorithm_newton_for_analysis()
```

---

```
int define_algorithm_modifiednewton_for_analysis()
```

#### 204.3.2.5 Simulation: Convergence Criteria

---

```
int define_convergence_test_energyincrement_for_analysis(double theTol,
                      int maxIter,
                      int PrintFlag)
```

theTol: convergence tolerance ; maxIter: maximum number of iterations that will be performed before "failure to converge" is returned ; PrintFlag: flag used to print information on convergence (optional) : 0: no print output ; 1: print information on each step ; 2: print information when convergence has been achieved ; 4: print norm, dU and dR vectors ; 5: at convergence failure, carry on, print error message, but do not stop analysis ;

---

```
int define_convergence_test_normdisplacementincrement_for_analysis(double theTol,
               int maxIter,
               int PrintFlag)
```

theTol: convergence tolerance ; maxIter: maximum number of iterations that will be performed before "failure to converge" is returned ; PrintFlag: flag used to print information on convergence (optional) ; 0: no print output ; 1: print information on each step ; 2: print information when convergence has been achieved ; 4: print norm, dU and dR vectors ; 5: at convergence failure, carry on, print error message, but do not stop analysis ;

---

```
int define_convergence_test_normunbalance_for_analysis(double theTol,
             int maxIter,
             int PrintFlag)
```

theTol: convergence tolerance ; maxIter: maximum number of iterations that will be performed before "failure to converge" is returned ; PrintFlag: flag used to print information on convergence (optional) ; 0: no print output ; 1: print information on each step ; 2: print information when convergence has been achieved ; 4: print norm, dU and dR vectors ; 5: at convergence failure, carry on, print error message, but do not stop analysis ;

#### 204.3.2.6 Simulating Response

---

```
int simulate_using_static_multistep(int numSteps)
```

numSteps: number of static analysis steps which will advance the solution, ;

---

```
int simulate_using_static_onestep()
```

---

```
int simulate_using_transient_multistep(double dT,
                        int numSteps)
```

dT: time-step increment ; numSteps: number of time steps ;

---

```
int simulate_using_transient_onestep()
```

---

```
int simulate_using_transient_variable_multistep(double dT,
                                int numSteps,
                                double dtMin,
                                double dtMax,
                                int Jd)
```

dT: time-step increment ; numSteps: number of time steps ; dtMin, dtMax: minimum and maximum time steps ; Jd: ideal number of iterations performed at each step ;

---

```
int simulate_using_transient_variable_onestep()
```

---

```
int simulate_using_eigen_analysis(int number_of_eigen_values)
```

## 204.4 Application Programming Interface for Constitutive Simulations

## 204.5 Application Programming Interface for Finite Elements

## 204.6 Adding a New Command into Real-ESSI Simulator

This section describes how to add a new command, needed for the generation of random fields, into Real-ESSI Simulator by modifying an existing command.

### 204.6.1 Introduction

So far, random fields have been generated using the discrete form of auto-covariance. The size of the auto-covariance matrix is $n \times n$ with $n$ equal to the number of Gauss points (GPs). In the case of a fine FE mesh, i.e., many GPs, this may result in a large computation time. Here, "shear beam" element is used. It has only one GP, in the middle of an element. Hence the number of GPs is here equal to the number of FE elements. It is easier for a user to input the number of FE elements than to input the number of GPs.

Generation of a random field in Real-ESSI is here extended with the possibility of the use of much less discrete points in the solution of the eigenproblem of auto-covariance (now, with the continuous form of auto-covariance using the PCE of auto-covariance function with Legendre polynomials, i.e., the eigenvalues and eigenfunctions are obtained instead of the eigenvalues and eigenvectors) than the GPs in the FEM.

### 204.6.2 Parser

Parser reads the DSL from input file `*.fei` and interpretes it. `flex` and `bison` are used.

`HERMITE POLYNOMIAL CHAOS DIMENSION` is the chosen number of the first eigenvalues and eigenfunctions (or eigenvectors) of the auto-covariance function (or matrix) that will be calculated.

Copy existing commands that take

```
1 HERMITE POLYNOMIAL CHAOS DIMENSION
```

and modify them by adding `number_of_fe_elements`.

For example, copy:

```
1 Hermite polynomial chaos Karhunen Loeve expansion to random field # <.> with
2 Hermite polynomial chaos dimension <.> order <.> ↩
    correlation_kernel_inverse_order = <.>;
```

and modify it into:

```
1 Hermite polynomial chaos Karhunen Loeve expansion to random field # <.> with
2 Hermite polynomial chaos dimension <.> order <.> ↩
    correlation_kernel_inverse_order = <.> number_of_fe_elements = <.>;
```

In .../GLOBAL_RELEASE/Real-ESSI/DSL/, modify 2 files:

- feiparser.yy = grammar part handled by bison
- feiparser.l = lexical part handled by flex.

### 204.6.3 feiparser.yy

Change:

```
%token KARHUNEN LOEVE DIMENSION correlation_kernel_inverse_order AS GLOBAL dimension_file TRIPLE PRODUCT shape_parameter scale_parameter WEIBULL TRIANGULAR
```

into:

```
%token KARHUNEN LOEVE DIMENSION correlation_kernel_inverse_order AS GLOBAL dimension_file TRIPLE PRODUCT shape_parameter scale_parameter WEIBULL TRIANGULAR number_of_fe_elements
```

Then, copy the existing commands, like:

```
| HERMITE POLYNOMIAL CHAOS KARHUNEN LOEVE EXPANSION TO RANDOM FIELD TEXTNUMBER exp WITH HERMITE POLYNOMIAL CHAOS DIMENSION exp ORDER exp correlation_kernel_inverse_order '=' exp
    {
        args.clear(); signature.clear();
        args.push_back($11); signature.push_back(this_signature("RandomField_tag", &isAdimensional));
        args.push_back($17); signature.push_back(this_signature("Dimension_num", &isAdimensional));
        args.push_back($19); signature.push_back(this_signature("Order_num", &isAdimensional));
        args.push_back($22); signature.push_back(this_signature("correlation_kernel_inverse_order", &isAdimensional));
        $$ = new FeiDslCaller4<int, int, int, int>(&Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order, args, signature,
        "Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order");
        for(int i = 1; i <= 4; i++) nodes.pop();
        nodes.push($$);
    }
```

and modify it into:

```
| HERMITE POLYNOMIAL CHAOS KARHUNEN LOEVE EXPANSION TO RANDOM FIELD TEXTNUMBER exp WITH HERMITE POLYNOMIAL CHAOS DIMENSION exp ORDER exp correlation_kernel_inverse_order '=' number_of_fe_elements '=' exp
```

```
2     {
3         args.clear(); signature.clear();
4         args.push_back($11); ↩
              signature.push_back(this_signature("RandomField_tag", ↩
              &isAdimensional));
5         args.push_back($17); ↩
              signature.push_back(this_signature("Dimension_num", &isAdimensional));
6         args.push_back($19); signature.push_back(this_signature("Order_num", ↩
              &isAdimensional));
7         args.push_back($22); ↩
              signature.push_back(this_signature("correlation_kernel_inverse_order", ↩
              &isAdimensional));
8         args.push_back($25); ↩
              signature.push_back(this_signature("number_of_fe_elements", ↩
              &isAdimensional));
9         $$ = new FeiDslCaller5<int, int, int, int, ↩
              int>(&Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order_FE_elements, ←
              args, signature,
10        "Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order_FE_elements");
11        for(int i = 1; i <= 5; i++) nodes.pop();
12        nodes.push($$);
13    }
```

### 204.6.4 feiparser.l

Just add:

```
1 "number_of_fe_elements" {return token::number_of_fe_elements;}
```

### 204.6.5 create_parallel.sh

create_parallel.sh takes feiparser.l and feiparser.yy, and creates files needed for compilation of essi.parallel. For example, it creates feiparser.lex.cpp. In feiparser.lex.cpp: *yytext and yyleng will be defined twice and the compiler will complain during linking for essi.parallel. The second definitions of *yytext and yyleng need to be found in feiparser.lex.cpp and deleted manually. For this purpose, in create_parallel.sh, there are 2 lines:

```
sed -i '319d' feiparser.lex.cpp
sed -i '5642d' feiparser.lex.cpp
```

and lines, here 319 and 5643 (mind the preceding removal of 319), in feiparser.lex.cpp will be deleted.

### 204.6.6 create_sequential.sh and create_parallel.sh

In .../GLOBAL_RELEASE/Real-ESSI/DSL/, one can find create_sequential.sh and create_parallel.sh. Run both.

### 204.6.7 Application Programming Interface (API)

In .../GLOBAL_RELEASE/Real-ESSI/API/MODELING/, copy the following files:
Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order.h
Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order_hdf5_input.h
and change their names to:
Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order_FE_elements.h
Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order_hdf5_input_FE_elements.h
and modify them to account for the number of GPs (= the number of FE elements here).

In .../GLOBAL_RELEASE/Real-ESSI/API/, there is file api.h. In api.h, add the following lines:

```
#include "MODELING/Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order_FE_elements.h"
#include "MODELING/Hermite_polynomial_chaos_Karhunen_Loeve_expansion_inverse_order_hdf5_input_FE_elements.h
```

In this case, existing files RandomField.cpp and RandomField.h will be modified and no new files *.cpp and *.h have to be created. If new files, for example, RandomFieldGaussianCorrelation.cpp and RandomFieldGaussianCorrelation.h, are created, then additionally in ESSI_API.h, one needs to add:

```
#include <../CompGeoMechUCD_StochasticFEM/PolynomialChaos/RandomFieldGaussianCorrelation.h>
```

## 204.7 Adding New Finite Element into Real-ESSI Simulator

This section illustrates how to add a new element in Real-ESSI simulator. A detailed description of each steps involved is given. The developer is expected to understand these steps and replicate it for their new element. Also, it is quite useful to look at some previous elements already implemented in Real-ESSI.

### 204.7.1 Introduction

This documents provides detailed description of steps for adding a new element into the Real-ESSI Simulator. New Element Template source (.ccp) and header (.h) files can be located inside source code in *CompGeoMechUCD_FiniteElements* directory, and are also shown below in subsection 204.7.3 and subsection 204.7.4.

The list of all the steps to be followed are listed below

1. SubSec 204.7.2:: Creating New Element Directory and Linking to Real-ESSI
2. SubSec 204.7.3:: Writing the New Element Header File
3. SubSec 204.7.4:: Writing the New Element Source File
4. SubSec 204.7.5:: Setting the *ELE_TAG_NewElement* class tag and its description
5. SubSec 204.7.6:: Integrating new element with parser.
6. SubSec 204.7.7:: Compiling Real-ESSI
7. SubSec 204.7.8:: Verification of Implementation

These steps are shown in each sub-section. The first step starts creating a directory for the new element. After step [3], the new element would be linked with Real-ESSI source code. So, its good to start compiling (step [6]) and fixing bugs rather than going to step [4] or further ahead.

### 204.7.2 Getting Started:: Creating New Element Directory

The new element can be added in *CompGeoMechUCD_FiniteElements* directory of Real-ESSI source. A directory for new element lets say NewElement must be created. The next step is to add CMakeLists.txt and place into that directory. The contents of the cmake file is

```
# Builds all the CompGeoMechUCD_FiniteElements/NewElement module
# message("scanning newelements module...")

BUILD_LIB("newelements" ESSI_LIBS)
```

Also, in that directory new element header (NewElement.h) and source files (NewElement.cpp) must be placed. The contents of the cpp files are shown in subsection 204.7.3 and subsection 204.7.4.

The new element then must be included to the header file of Real-ESSI Elements i.e. *CompGeoMechUCD_FiniteElements.h* . The element header file is loaded in *CompGeoMechUCD_FiniteElements* directory. So, just add the new element header as

```
// ////////////////////////////////////////////////
// New Elements [XYZ, Month, Year]
// ////////////////////////////////////////////////

#include "./NewElement/NewElement.h"
```

This would link the new element source code to Real-ESSI. Next is to write the source code and header files of the new element.

### 204.7.3 Element Header File

The header file is self documented (read fully and carefully).

```
// Rename the header guard
#ifndef NewElementTemplate_h
#define NewElementTemplate_h

#include <Element.h>
#include <Matrix.h>
#include <Vector.h>

class Node;
class Channel;

class NewElementTemplate: public Element
{

public:
  // Constructor
  NewElementTemplate(int tag, int node1, .....); //You must implement this
  // Empty constructor
  NewElementTemplate();
  //Destructor
  ~NewElementTemplate();

  /*****************************************************************************/
  /********* Functions to obtain information about dof & connectivity *****/
  /*****************************************************************************/
  // returns the number of external nodes of the element
  int getNumExternalNodes(void) const;
  // returns the ID of external nodes of the element
  const ID &getExternalNodes(void);
  // returns the node pointers array to the nodes of the elements
  Node **getNodePtrs(void);
  // rerturn the total number of degrees of freedom for the element
  int getNumDOF(void);
  // returns the DofList containing number of degrees of freedom for each node
  const ID &getDofList();
  // update all the necessary variables before simulation starts
  void setDomain(Domain *theDomain);
  /*****************************************************************************/
  /*************** Functions to set the state of the element *************/
  /*****************************************************************************/
  // Functions to update the state of the element on obtaining convergence
  int commitState(void);
  // Function to revert to the last commited (converged) state
  int revertToLastCommit(void);
  // Function to revert to the start of the state of the element at the ↩
     beginning of simulation
  int revertToStart(void);
  // Update the element variables for each iteration
```

```
48  int update(void);
49  // Remove the load from element
50  void zeroLoad(void);
51  // Add Element load
52  int addLoad(ElementalLoad *theLoad, double loadFactor);
53  // Send Current Intertial Load of the element
54  int addInertiaLoadToUnbalance(const Vector &accel);
55  /*****************************************************************************/
56  /********** Functions to obtain stiffness, mass and residual ***********/
57  /*****************************************************************************/
58  // return the current tangent stiffness of the element
59  const Matrix &getTangentStiff(void);
60  // return the initial tangent stiffness of the element
61  const Matrix &getInitialStiff(void);
62  // return the damping stiffness of the element
63  const Matrix &getDamp(void);
64  // return the mass of the element
65  const Matrix &getMass(void);
66  // return the resisting force of the element (static case)
67  const Vector &getResistingForce(void);
68  // return the resisting force of the element (dynamic case)
69  const Vector &getResistingForceIncInertia(void);
70  /*****************************************************************************/
71  /********** Functions to implement parallel processing ***********/
72  /*****************************************************************************/
73  // Send the variables to the other CPU in a unique order
74  int sendSelf(int commitTag, Channel &theChannel);
75  // For the send order to recieve the information
76  int receiveSelf(int commitTag, Channel &theChannel, FEM_ObjectBroker ↩
        &theBroker);
77  /*****************************************************************************/
78  /********** Function to Print information about elemnt ***********/
79  /*****************************************************************************/
80  // Print out element info
81  void Print(ostream &s, int flag = 0);
82  // Check element correctness
83  int CheckMesh(ofstream &);
84  // Give the element a name
85  std::string getElementName() const
86  {
87      return "NewElementTemplate";
88  }
89  /*****************************************************************************/
90  /****************** Generate Output of the Element ********************/
91  /*****************************************************************************/
92  /*****************************************************************************/
93  /* No. of Element Outputs should be as per the Element_Class_tag_Desc
94  /* See the classtags.h for more description on encoding of Class_tag ↩
        Descriptions.
95  /* For Optimization all the information about elements are encoded
96  /* in the Element_Class_tag Description
```

```
97    /* NOTE:- Element_Class_Description [see classTags.h] must be obeyed
98    /*********************************************************************/
99    // Declare if there is element output except at gauss points
100   const vector<float> &getElementOutput() ;
101   // Declare only if there is any gauss point and there is 18 outputs
102   // per gauss points i.e stress, strain and plastic strain
103   const vector<float> &getGaussOutput();
104   // Send the Gauss Coordinates of the Elements
105   Matrix &getGaussCoordinates(void);
106
107 protected:
108   /*********************************************************************/
109   //Implementation-specific member functions...
110   // Should be protected, because they're not going to be called
111   // from outside the class, but you might want to inherit them
112   /*********************************************************************/
113
114 private:
115   /*********************************************************************/
116   // All data must be private. Provide setter and getter methods if
117   // this class interacts with other classes.
118   /*********************************************************************/
119   // Declare if there is element output except at gauss points
120   static vector<float> Element_Output_Vector() ;
121   // Declare only if there is any gauss point and there are 18
122   // outputs per gauss points i.e stress, strain and plastic strain
123   static vector<float> Gauss_Output_Vector();
124   // Contains information about Number of Dof for each node of the element
125   static ID DofList;
126 };
127 #endif
```

### 204.7.4 Element Source File

The source file is self documented (read fully and carefully).

```
#include <NewElement.h>
// Must define the class tag for
// the new element in this file.
#include <classTags.h>

// NOTE!! Follow the Element_Class_Desc Encoding
// See classTags.h for more details about encoding
// Declare if there is element output except at gauss points
vector<float> ↩
    NewElementTemplate::Element_Output_Vector(number_of_Element_outputs) ;
// Declare only if there is any gauss point and there is
// 18 outputs per gauss points i.e stress, strain and plastic strain
vector<float> NewElementTemplate::Gauss_Output_Vector(number_of_gauss_points*18);
// Contains information about Number of Dof for each node of the element
ID NewElementTemplate::DofList(number_of_element_nodes);

//*********************************************************************
// Constructor. Receive all input parameters. Should not allocate resources!
// * Input: Defined by user. At least should receive an integer tag, so that base
// class can be initialized.
// * Output: void
NewElementTemplate::NewElementTemplate(int tag, int node1, .....):
    Element(tag, ELE_TAG_NewElement),
    // add more initializers
{
    //ATTENTION!
    // ELE_TAG_NewElement !! Define the class tag in classTags.h
    // with provided encoding formula
    // for setting the material id to the element
    this->setMaterialTag(material->getTag());
    // fill DofList container with number of dofs for each node
    you must implement
}

//*********************************************************************
// Empty constructor. Create an empty element (with possibly a bad state)
// * Input: Defined by user. At least should receive an integer tag,
// so that base class can be initialized.
// * Output: void
NewElementTemplate::NewElementTemplate():
    Element(0, ELE_TAG_NewElement),
    // add more initializers setting internal variables to null values
{
    //ATTENTION!
    // ELE_TAG_NewElement !! Define the class tag in classTags.h
    // with provided encoding formula
    // fill DofList container with number of dofs for each node
    you must implement
```

```
48 }
49
50 //***********************************************************************
51 // Destructor. Deallocate resources used by element.
52 // * Input: void
53 // * Output: void
54 NewElementTemplate::~NewElementTemplate()
55 {
56     you must implement
57 }
58
59 //***********************************************************************
60 // returns the number of nodes of the element.
61 // * Input: void
62 // * Output: number of nodes
63 int NewElementTemplate::getNumExternalNodes(void) const
64 {
65     you must implement
66     return number_of_nodes;
67 }
68
69 //***********************************************************************
70 // Return an ID (integer vector) with the external nodes
71 // * Input: void
72 // * Output: ID with tags of external nodes
73 const ID &NewElementTemplate::getExternalNodes(void)
74 {
75     you must implement
76     return external_nodes;
77 }
78
79 //***********************************************************************
80 // Return pointer array to the nodes
81 // * Input: void
82 // * Output: node pointer array.
83 Node **NewElementTemplate::getNodePtrs(void)
84 {
85     you must implement
86     return nodes;
87 }
88
89 //***********************************************************************
90 //Return the number of dofs in the element.
91 // * Input: void
92 // * Output: number of dofs (sum of dofs over all of element's nodes)
93 int NewElementTemplate::getNumDOF(void)
94 {
95     you must implement
96 }
97
98 //***********************************************************************
```

```
99  //Return the number of dofs in the element.
100 // * Input: void
101 // * Output: DofList containing number of degrees of freedom for each node
102 const ID &getDofList(){
103
104     you must implement
105     return this->DofList;
106 }
107
108 //*************************************************************
109 // Receives a domain pointer, and sets the local domain pointer through
110 // calling the base class setDomain.
111 // At this point the domain is defined and set, one can allocate resources
112 // (get nodal pointers, compute some internal variables like lengths, volumes, ↩
        etc. ).
113 // Usually we'll set the node pointers here (will be needed for getNodePtrs ↩
        function).
114 // Also, we'll check that the given nodes are defined (you get a valid pointer ↩
        to them) and
115 // that they have the right number of DOFS (implementation specific)
116 // * Input: domain pointer (see Domain.h)
117 // * Output: void
118 void NewElementTemplate::setDomain(Domain *theDomain)
119 {
120     // check Domain is not null - invoked when object removed from a domain
121     if (theDomain == 0)
122     {
123         //set node pointers to null
124     }
125     else
126     {
127         //Use the domain to set the node pointers...
128         //nodePointers[0] = theDomain->getNode(Nd1);
129         //nodePointers[1] = theDomain->getNode(Nd2);
130         // Check the pointers...
131         // if (nodePointers == 0)
132         // {
133         // bad error, usually means node was never
134         // return;
135         // }
136         // Check the number of DOFs
137         // if(nodePointers[0]->getNumberDOF() != MY_NUMBER_OF_DOFS)
138         // {
139         // print a tantrum
140         // return;
141         // }
142         //
143         // More checks maybe
144         //
145         // Set the base class domain pointer
146
```

```
147         this->DomainComponent::setDomain(theDomain);
148     }
149
150 //Additionally one can allocate resources at this point.
151    you must implement
152
153 }
154
155 //***********************************************************
156 // Accept current state of the element and save it. (If applicable)
157 // I this is a gauss-point based element, one calls commitState on
158 // the material pointers (Gauss points) owned by this element.
159 // return 0 if success.
160 // * Input: void
161 // * Output: error flag, 0 if success
162 int NewElementTemplate::commitState(void)
163 {
164     you must implement
165     return 0;
166 }
167
168 //***********************************************************
169 // Revert the state of the element to the last committed state.
170 // Must call for gausspoints if needed.
171 // * Input: void
172 // * Output: error flag, 0 if success
173 int NewElementTemplate::revertToLastCommit(void)
174 {
175     you must implement
176     return 0;
177 }
178
179 //***********************************************************
180 // Revert the state of the element to the initial state.
181 // Must call for gausspoints if needed.
182 // * Input: void
183 // * Output: error flag, 0 if success
184 int NewElementTemplate::revertToStart(void)
185 {
186     you must implement
187     return 0;
188 }
189
190 //***********************************************************
191 // Update the state of the element. I.E. compute new tangent stiffness,
192 // compute new resisting force, advance state variables.
193 // These changes should not be permanent until commit function is called.
194 // * Input: void
195 // * Output: error flag, 0 if success
196 int NewElementTemplate::update(void)
197 {
```

```
198     you must implement
199     return 0;
200 }
201
202 //***************************************************************
203 // (optionl) Set the elemental load to zero.
204 // * Input: void
205 // * Output: void
206 void NewElementTemplate::zeroLoad(void)
207 {
208     // optional to implement
209     return 0;
210 }
211
212 //***************************************************************
213 // (optionl) Add a new elemental load. This will modify the
214 // resisting force vector.
215 // * Input: ElementalLoad pointer and a loadFactor.
216 // * Output: error flag, 0 if success
217 // Notes:
218 // * ElementalLoads have a type interger (a tag defined elsewhere) and a Vector ↩
        (array
219 // of doubles) with data. Use these to generate the elemental load scaled by the
220 // load factor (which is also the time-step of the analysis).
221 int NewElementTemplate::addLoad(ElementalLoad *theLoad, double loadFactor)
222 {
223     // optional to implement
224     //
225     // Some code to get the type and data. Example is for self_weight.
226     // int type;
227     // const Vector &data = theLoad->getData(type, loadFactor);
228     //
229     // if (type == LOAD_TAG_ElementSelfWeight) // Load tags are defined in ↩
            classTags.h
230     // {
231     // do something, like add a the forces to the resisting_force vector.
232     // }
233     return 0;
234 }
235
236 //***************************************************************
237 // Add intertial terms to resisting force vector.
238 // * Input: A vector with nodal accelerations???
239 // * Output: error flag, 0 if success
240 // Notes: use node pointers to get accelerations from nodes,
241 // form an acceleration vector and multiply this with the mass matrix, then
242 // add this into the load unbalance (with negative sign, cause it is inertia)
243 int NewElementTemplate::addInertiaLoadToUnbalance(const Vector &accel)
244 {
245     you must implement
246     return 0;
```

```
247 }
248
249 //*********************************************************************
250 // Functions to obtain stiffness, mass, damping and residual information
251 // * Input: void
252 // * Output: reference to tangent stiffness matrix (of size nDOF x nDOF,
253 // where nDOF = NewElementTemplate::getNumDOF();
254 // Pro tip. If this matrix computes the tangent stiffness, then
255 // it can be stored as a static member variable so that all elements share
256 // the same memory space (each element overwrites the tangent). This saves
257 // memory.
258 const Matrix &NewElementTemplate::getTangentStiff(void)
259 {
260     you must implement
261     return K;
262 }
263
264 //*********************************************************************
265 // Functions to obtain initial stiffness
266 // * Input: void
267 // * Output: reference to initial tangent stiffness matrix (of size nDOF x nDOF,
268 // where nDOF = NewElementTemplate::getNumDOF();
269 const Matrix &NewElementTemplate::getInitialStiff(void)
270 {
271     you must implement
272     return *K;
273     // --suggested variable name for stiffness :).
274     // Will be a pointer, so that it can be after construction.
275 }
276
277 //*********************************************************************
278 // (optional) If element provides its own damping matrix, then this
279 // function returns it
280 // * Input: void
281 // * Output: reference to damping stiffness matrix (of size nDOF x nDOF,
282 // where nDOF = NewElementTemplate::getNumDOF();
283 const Matrix &NewElementTemplate::getDamp(void)
284 {
285     // optional to you must implement
286 }
287
288 //*********************************************************************
289 // (optional) If element provides its own damping matrix, then this
290 // function returns it
291 // * Input: void
292 // * Output: reference to damping stiffness matrix (of size nDOF x nDOF,
293 // where nDOF = NewElementTemplate::getNumDOF();
294 const Matrix &NewElementTemplate::getMass(void)
295 {
296     // optional to implement
297 }
```

```
298
299 //**********************************************************************
300 // (optional) If element provides its own damping matrix,
301 // then this function returns it
302 // * Input: void
303 // * Output: reference to damping stiffness matrix (of size nDOF x nDOF,
304 // where nDOF = NewElementTemplate::getNumDOF();
305 const Vector &NewElementTemplate::getResistingForce(void)
306 {
307     you must implement
308 }
309
310 //**********************************************************************
311 // (optional) Add inertial terms to resisting force.
312 // * Input: void
313 // * Output: Vector of doubles with new resisting force.
314 // Note: Regularly, this function calls getResistingForce() and then
315 // adds inertial terms.
316 const Vector &NewElementTemplate::getResistingForceIncInertia(void)
317 {
318     // (optional to implement)
319 }
320
321 //**********************************************************************
322 // (optional, a must if you want to do parallel processing)
323 // Send all state data of the element through a channel (usually an MPI_Channel)
324 // * Input: a reference to the Channel to use.
325 // * Output: error flag, 0 if success
326 // Note: This function is usually very involved, and should do a lot of checking
327 // for pointers and for success of the send.
328 // Note2: setDomain(...) *might* not be called before using this function.
329 int NewElementTemplate::sendSelf(int commitTag, Channel &theChannel)
330 {
331     // Useful constructs
332     // int error_flag;
333     // error_flag = theChannel.sendVector(0, 0, double_data); // the first two ↩
        parameters are deprecated
334     //
335     // Check that error_flag is not < 0
336     //
337     // theChannel.sendID(0, 0, integer_data); // the first two parameters are ↩
        deprecated
338     //
339     // Check that error_flag is not < 0
340     you must implement
341     return 0;
342 }
343
344 //**********************************************************************
345 // (optional, a must if you want to do parallel processing)
346 // Receive all state data of the element through a channel
```

```
347 // (usually an MPI_Channel). This data comes from an element
348 // that is calling sendSelf in some other process.
349 // * Input: a reference to the Channel to use.
350 // * Output: error flag, 0 if success
351 // Note: This function is called after setDomain() so all resources should be ↩
        made available.
352 int NewElementTemplate::receiveSelf(int commitTag, Channel &theChannel, ↩
        FEM_ObjectBroker &theBroker)
353 {
354     // Useful constructs
355     // int error_flag;
356     // theChannel.receiveVector(0, 0, double_data); // the first two parameters ↩
            are deprecated
357     // Check that error_flag is not < 0
358     // theChannel.receiveID(0, 0, integer_data); // the first two parameters ↩
            are deprecated
359     // Check that error_flag is not < 0
360
361     you must implement
362     return 0;
363 }
364
365 //***************************************************************
366 // Print out element info
367 // * Input: an ostream to print stuff into, and a flag
368 // * Output: void
369 // Print stuff into the ostream by using the "<<" operator.
370 // The flag can be used to request different levels of printing, ie.
371 // a flag of 0 might be very basic information, while flag > 0 might
372 // give increasing ammount of info.
373 void NewElementTemplate::Print(ostream &s, int flag = 0)
374 {
375     you must implement
376 }
377
378 //***************************************************************
379 // Check element correctness
380 // * Input: an ostream to print stuff into (print details of what is being ↩
        checked here.)
381 // * Output: an error flag (<0 if element is not right in some way)
382 // Note: be verbose print element tag, etc. Print out only if an error is ↩
        encountered.
383 int NewElementTemplate::CheckMesh(ofstream &)
384 {
385     you must implement
386 }
387
388 //***************************************************************
389 // Output interface functions
390 // * Input: void
391 // * Output: Vector (array of doubles) with the element output.
```

```
392 const vector<float> &NewElementTemplate::getElementOutput()
393 {
394
395     Fill the Element_Output_Vector
396
397     return Element_Output_Vector;
398 }
399
400 //************************************************************
401 // Output interface functions
402 // * Input: void
403 // * Output: Vector (array of doubles) with the element output.
404 const vector<float> &NewElementTemplate::getGaussOutput()
405 {
406     Fill the Gauss_Output_Vector
407
408     NOTE!!! - Exactly 18 outputs should be there per gauss point
409
410     return Gauss_Output_Vector;
411 }
412
413 //************************************************************
414 // Return a Matrix with the coordinates of Gauss points (or points
415 // where outputs are generated, could be the endpoints of a beam)
416 // * Input: void
417 // * Output: Matrix (array of doubles) with the gauss coordinates
418 // Note: Format is
419 // gauss_coordinates[0,:] = [x_0,y_0,z_0] -- Coordinates of first Gauss point
420 // gauss_coordinates[1,:] = [x_1,y_1,z_1] -- Coordinates of second Gauss point
421 // ...
422 // gauss_coordinates[Ngauss,:] = [x_Ngauss,y_Ngauss,z_Ngauss]
423 // -- Coordinates of Ngauss-th Gauss point
424 Matrix &NewElementTemplate::getGaussCoordinates(void)
425 {
426     you must implement
427 }
428
429 //************************************************************
430 // Add you own member functions at the end!
```

### 204.7.5 Element Class Tag Description

This subsection describes how to set up the *ELE_TAG_NewElement* for the new element. All the tags for element, material, load etc, must be included in *ClassTags.h* located in *ModifiedOpenSeesServices* directory. Each element has unique identifiers:

- Element Tag :: It is a unique tag given to each new element type. The element tags are in sequential order. So the new tag must be the next available tag in sequence.
- Element Tag Description :: It is an encoding containing information about the elements such as, type of element, number of nodes, number of gauss points , number of outputs etc. The element tag has 9 digits and follows a strict encoding a shown below

```
// All elements class tags would be in serial
// numbers from 1-N for optimization
/*********************************************************************
* Desc is [Dimension][N. Nodes][Dof per nodes][N. Gauss][No.of Outputs]
* <1-digit> <2-digit> <1-digit> <3-digit> <2-digit>
* - - - - - - - - - - - - - - -- - - - - - - - - -- - - - -
*
* [ElementCategory] = <num_of_digits = 1> Category of the element
* 1-> Structural Elements
* 2-> Contact Elements
* 3-> Brick Elements
* 4-> Special elements
*
* [N. Nodes] = <num_of_digits = 2> Number of nodes in elements
* xx-> number of nodes
*
* [Dof per nodes] = <num_of_digits = 1> Degree of freedom per node
* x-> DOFS per node
*
* [N. Gauss] = <num_of_digits = 3> Number of gauss points in elements
* xxx-> number of gauss points
*
* [No.of Outputs] = <num_of_digits = 2> no. of outputs other than at gauss ↩
    points.
* xx-> no. of outputs other than at gauss points.
*
* Default Features
* - - - - - -- - -
* 1) Per gauss point there are in total 18 outputs
* of stress, plastic strain and total strain
* 2) No.of Outputs -> here means the extra outputs by an element
* except gauss points. For example:
* for eight node brick there is 000 No. of Outputs.
*********************************************************************/

//###---------------------------------------
```

```
//### NOTE!! :- Every Element have a responsibility to set
//### their tag_description Array. Based on the above encoding
//### NOTE!! :- Also increase the ELE_TAG_DESC_ARRAY_SIZE to the
//### number of element tags
//### ----------------------------------------
```

For example: Eight node brick element has element tag description as 308300800. Simple truss element has 102300002 as element tag description.

In order to set up the element tag, look for the next available element tag. Usually, the next available element tag would be equal to the *ELE_TAG_DESC_ARRAY_SIZE* which can be found inside *ClassTag.h* file below the initial element tags. The available number should be added as new element tag and the *ELE_TAG_DESC_ARRAY_SIZE* should be increased by 1.

This should be followed by appending the element class tag description in *ELE_TAG_DESC_ARRAY*. All the steps are shown below,

`ClassTags.h` Before:

```
........
#define ELE_TAG_DispBeamColumn3d 94 // 102600012
#define ELE_TAG_Cosserat_8node_brick 95 // 408600800

#define ELE_TAG_DESC_ARRAY_SIZE 96

.......................
.......................


#define ELE_TAG_DESC_ARRAY int ele_tag_desc_array[] = \
{ELE_TAG_DESC_ENCODING, \
100000000, \
308300800, \
308400800, \
308700800, \
308300100, \
308400100, \
........
........
202300009, \
102300002, \
102600024, \
103600006, \
104600000, \
302300100, \
102600012, \
102600012, \
408600800, \
}
```

ClassTags.h After:

```
.........
#define ELE_TAG_DispBeamColumn3d 94 // 102600012
#define ELE_TAG_Cosserat_8node_brick 95 // 408600800
#define ELE_TAG_NewElement 96 // XXXXXXXXX

#define ELE_TAG_DESC_ARRAY_SIZE 97

.........................
.........................


#define ELE_TAG_DESC_ARRAY int ele_tag_desc_array[] = \
{ELE_TAG_DESC_ENCODING, \
100000000, \
308300800, \
308400800, \
308700800, \
308300100, \
308400100, \
........
........
202300009, \
102300002, \
102600024, \
103600006, \
104600000, \
302300100, \
102600012, \
102600012, \
408600800, \
XXXXXXXXX, \
}
```

### 204.7.6 Integrating New Finite Element into Parser

Next is to integrate the new element with the parser using *feiparser.yy,feiparser.l* files located in *DSL* directory located in Real-ESSI source. This step requires some knowledge of yacc and lex. The NewElement DSL should added along with other defined element DSL's in *feiparser.yy* file. A typical parser for an element looks like as shown below

```
| TEXTNUMBER exp TYPE NewElement WITH NODES
 '(' exp ',' exp ',' exp ',' exp ')'
  USE MATERIAL TEXTNUMBER exp
  parameter1 '=' exp
  parameter2 '=' exp
{
 args.clear(); signature.clear();
 args.push_back($2); signature.push_back(this_signature("number", &isAdimensional));
 args.push_back($8); signature.push_back(this_signature("node1", &isAdimensional));
 args.push_back($10); signature.push_back(this_signature("node2", &isAdimensional));
 args.push_back($12); signature.push_back(this_signature("node3", &isAdimensional));
 args.push_back($14); signature.push_back(this_signature("node4", &isAdimensional));
 args.push_back($19); signature.push_back(this_signature("material", &isAdimensional));

 args.push_back($22); signature.push_back(this_signature("parameter1", &isThisUnit<-1,3,1>));
 //L^3*T/M
 args.push_back($25); signature.push_back(this_signature("parameter2", &isThisUnit<-1,3,1>));

 $$ = new FeiDslCaller8<int,int,int,int,int,int,
       double,double>(&add_element_new_element, args, signature, "add_element_new_element");

 for(int ii = 1;ii <=8; ii++) nodes.pop();
 nodes.push($$);
}
```

where, the code for DSL of the element corresponds as

```
TEXTNUMBER exp TYPE NewElement WITH NODES
 '(' exp ',' exp ',' exp ',' exp ')'
  USE MATERIAL TEXTNUMBER exp
  parameter1 '=' exp
  parameter2 '=' exp
```

which would look like the following in DSL language

```
add element # <.> type NewElement with nodes
 (<.>, <.>, <.>, <.>) use material # 1
 parameter1 = <.>
 parameter2 = <.>;
```

In the above DSL each word represents a token which must be included in *feiparser.yy* and defined in *feiparser.l*. There are some already defined tokens, which needs not to be defined. One can find if the token exits by searching it in *feiparser.l* or *feiparser.yy*.

In the above DSL, the tokens are TEXTNUMBER, TYPE, NewElement, WITH, NODES, USE, MATERIAL, TEXTNUMBER, parameter1 and parameter2. Among them, some of them like TEXTNUMBER, TYPE, NODES, .. etc are already defined and could be searched. But the tokens NewElement, parameter1 and parameter2 needs to be defined. First, these all undefined tokens needs to be included in *feiparser.yy* and then defined in *feiparser.l*.

#### 204.7.6.1 feiparser.yy

The new tokens must be added in the beginning of the feiparser, where other tokens are defined.

The new tokens can be included as described below.

```
// Tokens for elements
%token EightNodeBrick TwentyNodeBrick TwentySevenNodeBrick
%token NewElement

// Element options tokens
%token porosity alpha rho_s rho_f k_x k_y k_z K_s K_f pressure cross_section ↩
    shear_modulus
%token friction_ratio maximum_gap
%token parameter1 parameter2
```

#### 204.7.6.2 feiparser.l

The tokens needs to be defined as the following.

```
"HardContact" {return token::HardContact;}
"SoftContact" {return token::SoftContact;}
"NewElement" {return token::NewElement;}
```

#### 204.7.6.3 Argument Stack, Signature and Units

Returning back to the code in parser:

```
| TEXTNUMBER exp TYPE NewElement WITH NODES
 '(' exp ',' exp ',' exp ',' exp ')'
  USE MATERIAL TEXTNUMBER exp
  parameter1 '=' exp
  parameter2 '=' exp
{
 args.clear(); signature.clear();
 args.push_back($2); signature.push_back(this_signature("number", &isAdimensional));
 args.push_back($8); signature.push_back(this_signature("node1", &isAdimensional));
 args.push_back($10); signature.push_back(this_signature("node2", &isAdimensional));
 args.push_back($12); signature.push_back(this_signature("node3", &isAdimensional));
 args.push_back($14); signature.push_back(this_signature("node4", &isAdimensional));
 args.push_back($19); signature.push_back(this_signature("material", &isAdimensional));

 args.push_back($22); signature.push_back(this_signature("parameter1", &isThisUnit<-1,3,1>));
 //L^3*T/M
 args.push_back($25); signature.push_back(this_signature("parameter2", &isThisUnit<-1,3,1>));

 $$ = new FeiDslCaller8<int,int,int,int,int,int,
       double,double>(&add_element_new_element, args, signature, "add_element_new_element");

 for(int ii = 1;ii <=8; ii++) nodes.pop();
 nodes.push($$);
}
```

Each variable or parameter in Real-ESSI has units. So, for each of the variables UNIT must be specified. In the above code. *args* is a stack that should be filled with the tokens that are the parameters of element. The first step is to clear the *args* and the *signature* stacks. Pushing the element parameter tokens is done as described below

```
args.push_back($2); signature.push_back(this_signature("number", &isAdimensional));
args.push_back($22); signature.push_back(this_signature("parameter1", &isThisUnit<-1,3,1>));
 //L^3*T/M
```

Here, $2 responds to the the second token i.e exp after *TEXTNUMBER*. Similarly, $5^{th}$ token in TYPE. In signature, the string can be anything, but should usually be the parameter name. The last

is enforcing the units for each of the parameters by defining the required units. There are many units, such as *isAdimensional, isAdimensional, isMass, isLength, isTime, isFrequency, isArea, isVolume, isForce, isEnergy, isTorque, isPressure, isBodyForce, isDensity, isVelocity, isAcceleration, isAreaMomentOfInertia, isMassMomentOfInertia*. If the parameter has some other units then the other units can be defined as

$$\text{isThisUnit} < m, l, t >$$

which refers to the unit $M^m L^l T^t$ in standard units. Here, $M, L, T$ are mass, length and time respectively.

#### 204.7.6.4 FeiDslCaller

The next step is to send all the parameters to DSL Header File. The header file contains the code to create a new element inside simulation domain. The code that does this is

```
$$ = new FeiDslCaller8<int,int,int,int,int,int,
     double,double>
     (&add_element_new_element, args,
     signature, "add_element_new_element");

for(int ii = 1;ii <=8; ii++) nodes.pop();
nodes.push($$);
```

*FeiDslCaller* takes all the arguments and passes to the *add_element_new_element.h* header file. In the above code, the number 8 in function *FeiDslCaller* corresponds to the total number of arguments to the element and following that, the type of the arguments is defined. Here, the type of 8 arguments are *int, int, int,int, int, int, double, double*. The last step is the remove everything from the node stack by popping it equal to the number of times of argument and then finally pushing the *FeiDslCaller* to the nodes stack.

#### 204.7.6.5 New DSL Header File

The header file of the new DSL must be created in */API/MODELING* directory. The header file must also be included in */API/api.h* header file. Here, the header file *add_element_new_element.h* has been created, which is called by *FeiDslCaller*.

```
int add_element_new_element(int ElementNumber, int Node1, int Node2, int Node3,
       int Node4, int MaterialId, double Parameter1, double Parameter2)
{
    Element* theElement = 0;
    theElement = new NewElement(ElementNumber, Node1, Node2, Node3, Node4,
    MaterialId, Parameter1, Parameter2);

    if (theElement == NULL){
```

```
9         cerr << "Error: (add_element_new_element)
10        memory allocation problem for theElement!" << endl;
11        return -1;
12    }
13
14    if (theDomain.addElement(theElement) == false){
15        cerr << "WARNING (add_element_new_element)
16        could not add element to the domain\n";
17        cerr << "Element Number: " << ElementNumber << endln;
18        return -1;
19    }
20
21    return 0;
22 };
```

Header file in */API/api.h* file needs to be included:

```
1 // ##########################
2 // New Element [ABCD Month, Year]
3 // ##########################
4
5 #include "MODELING/add_element_new_element.h"
```

### 204.7.7 Compiling Real-ESSI

It is assumed that the person reading this document is developer and thus should already have the Real-ESSI dependencies. To compile Real-ESSI with the new element, it should be build from the beginning. Assuming that the build directory in *build* inside Real-ESSI source code, the steps to recompile are

```
1 cd build
2 rm -r *
3 cmake ..
4 make -j 20
```

### 204.7.8 Verification of Implementation

Once, the element is fully integrated with Real-ESSI Simulator, the developer should fully verify the implementation by carrying out verification runs. In addition, the developer should be able to run their examples in sequential and parallel and verify the implementation.

# Chapter 205

# Input, Domain Specific Language

(1991-2005-2010-2011-2012-2015-2016-2017-2018-2019-2020-2021-)

(In collaboration with Prof. José Abell, Dr. Yuan Feng, and Dr. Hexiang Wang)

## 205.1 Chapter Summary and Highlights

## 205.2 Introduction

This chapter presents the domain specific language developed for the Real-ESSI. The language was designed with a primary goal of developing FEA models and interfacing them with various Real-ESSI functionalities. In addition to that, syntax is used to self-document models, provide physical-unit safety, provide common flow control structures, provide modularity to scripting via user functions and "include" files, and provide an interactive environment within which models can be created, validated and verified.

The development of Real-ESSI Domain specific language (DSL) (the Finite Element Interpreter, FEI) is based on LEX (Lesk and Schmidt, 1975) and YACC (Johnson, 1975).

Self-documenting ensures that the resulting model script is readable and understandable with little or no reference to the users manual. This is accomplished by providing a command grammar structure and wording similar to what would be used in a natural language description of the problem.

FEA analysis is unitless, that is, all calculations are carried out without referencing a particular unit system. This leaves the task of unit correctness up to the user of FEA analysis. This represents a recurring source of error in FEA analysis. Physical unit safety is enforced in Real-ESSI by implementing all base variables as physical quantities, that is, all variables have a unit associated with it. The adimensional unit is the base unit for those variables which have no relevant unit (like node numbers). Command calls are sensitive to units. For example, the node creation command call expects the node coordinates to be input with the corresponding units (length in this case). Additionally, the programming/command language naturally supports operation with units like arithmetic operations (quantities with different unit types will not add or subtract but may be multiplied). This approach to FEA with unit awareness provides an additional layer of security to FEA calculations, and forces the user to carefully think about units. This can help catch some common mistakes.

The Real-ESSI language provides modularity through the `include` directive/command, and user functions. This allows complex analysis cases to be parameterized into modules and functions which can be reused in other models.

Finally, an emphasis is placed on model verification and validation. To this end, Real-ESSI provides an interactive programming environment with all the ESSI syntax available. By using this environment. the user can develop tests to detect errors in the model that are not programming errors. For example, the user can query nodes and elements to see if they are set to appropriate states. Also, several standard tools are provided to check element validity (Jacobian, etc.).

The ESSI language provides reduced model development time by providing the aforementioned fea-

tures along with meaningful error reporting (of syntax and grammatical errors), a help system, command completion and highlighting for several open source and commercial text editors.

Some additional ideas are given by Dmitriev (2004), Stroustrup (2005), Niebler (2005), Mernik et al. (2005), Ward (2003), etc.

## 205.3 Domain Specific Language (DSL), English Language Binding

Overview of the language syntax.

- Each command line has to end with a semicolon "`;`"
- Comment on a line begins with either "`//`" or "`!`" and last until the end of current line.
- Units are required (see more below) for all quantities and variables.
- Include statements allow splitting source into several files
- All variables are double precision (i.e. floats) with a unit attached.
- All standard arithmetic operations are implemented, and are unit sensitive.
- Internally, all units are represented in the base SI units ($m$ - $s$ - $kg$).
- The syntax ignores extra white spaces, tabulations and newlines. Wherever they appear, they are there for code readability only. (This is why all commands need to end with a semicolon).
- The user should be familiar with the list of the reserved keywords from Section 205.7 on page 1163.

### 205.3.1 Running Real-ESSI

At the command line type "`essi`", to get to the ESSI prompt and start Real-ESSI in interactive mode.

Command line output

```
  The Finite Element Interpreter Endeavor

  The Real-ESSI Simulator
  Modeling and Simulation of Earthquakes, and Soils, and ↩
     Structures and their Interaction

  Sequential processing mode.

Version Name       : Real-ESSI Global Release, June2018. Release ↩
   date: Jun 13 2018 at 11:02:19. Tag: adc085ae70
```

```
Version Branch    : GLOBAL_RELEASE
Compile Date      : Jun 13 2018 at 14:36:56
Compile User      : jeremic
Compile Sysinfo   : sokocalo 4.13.0-43-generic x86_64 GNU/Linux
Runtime User      : jeremic
Runtime Sysinfo   : sokocalo 4.13.0-43-generic x86_64 GNU/Linux
Time Now          : Jun 13 2018 at 15:32:52
Days From Release : 0
PostProcessing Compatible Version: ParaView 5.1.2
PostProcessing Compatible Version: ESSI-pvESSI Date: Feb 15 2018 ↩
   at 11:00:28. Tag: 58fe430a19

Static startup tips:
 * Remember: Every command ends with a semicolon ';'.
 * Type 'quit;' or 'exit;' to finish.
 * Run 'essi -h' to see available command line options.

ESSI>
```

A number of useful information about Real-ESSI is printed on the screen. From here, commands can be input manually or a file may be included via the `include` command which is as follows.

```
1 include "foobar.fei";
```

to include the file `foobar.fei`.

A more efficient way to start Real-ESSI and analyze an example is to pass input file name to the command line. Real-ESSI command to execute an input file immediately is done by issuing the following command: `essi -f foobar.fei`. This will execute essi directly on input file `foobar.fei`. After executing the file, the essi interpreter will continue in interactive mode unless the command line flag `-n` or `--no-interactive` is set. A list of command line options is available by calling essi from the command line as `essi -h`.

Command line output

```


  The Finite Element Interpreter Endeavor

  The Real-ESSI Simulator
  Modeling and Simulation of Earthquakes, and Soils, and ↩
     Structures and their Interaction

  Sequential processing mode.

Version Name      : Real-ESSI Global Release, June2018. Release ↩
   date: Jun 13 2018 at 11:02:19. Tag: adc085ae70
Version Branch    : GLOBAL_RELEASE
Compile Date      : Jun 13 2018 at 14:36:56
```

```
Compile User      : jeremic
Compile Sysinfo   : sokocalo 4.13.0-43-generic x86_64 GNU/Linux
Runtime User      : jeremic
Runtime Sysinfo   : sokocalo 4.13.0-43-generic x86_64 GNU/Linux
Time Now          : Jun 13 2018 at 16:22:08
Days From Release : 0
PostProcessing Compatible Version: ParaView 5.1.2
PostProcessing Compatible Version: ESSI-pvESSI Date: Feb 15 2018 ↩
   at 11:00:28. Tag: 58fe430a19

Static startup tips:
 * Remember: Every command ends with a semicolon ';'.
 * Type 'quit;' or 'exit;' to finish.
 * Run 'essi -h' to see available command line options.

The Real-ESSI Simulator
Modeling and Simulation of Earthquakes, and Soils, and Structures ↩
   and their Interaction

Usage: essi [-cfhnsmbe FILENAME]
  -c --cpp-output                : Output cpp version of the model.
  -f --filename [FILENAME]       : run ESSI on a FILENAME.
  -h --help                      : Print this message.
  -n --no-interactive            : Disable interactive mode.
  -s --set-variable              : Set a variable from the ↩
     command line.
  -d --dry-run                   : Do not execute ESSI API calls. ↩
     Just parse.
  -m --model-name [NAME]         : Set the model name from the ↩
     command line.
  -p --profile-report [FILENAME] : Set the filename for the ↩
     profiler report (and activate lightweight profiling)


Example to set a variable name from command line:
    essi -s a=10,b=20,c=30
Runs ESSI with variables a, b, and c set to 10, 20 and 30 ↩
   respectively.
At this time, only ESSIunits::unitless variables can be set.
```

### 205.3.2 Finishing Real-ESSI Program Run

To properly finish Real-ESSI program run, and save and close all the output files, user has to use final, closure command:

```
1 bye;
```

Command bye; has to be included at the end of input file script, or at the end of each interactive/interpretative session. Command bye; ensures that Real-ESSI program gracefully exits simulation, and

that all the output files are properly saved and closed. Proper finishing of simulation using Real-ESSI Simulator is very much necessary, while the choice of command bye; is done as an homage to Professor Knuth and his Literate Programming endeavor (Knuth, 1984), that is driving much of the Real-ESSI DSL development.

There are a number of alternative final commands, for example:

```
exit;
quit;
zdravo;
vozdra;
dvojka;
voljno;
zaijian;
tschuess;
geia-sou;
tchau;
sair;
khoda-hafez;
doei;
nasvidenje;
ajde-bok;
izhod;
konec;
czesc;
ciao;
hoscakal;
```

These additional, alternative final commands can all be written using original scripts:

zdravo ↔ здраво

vozdra ↔ воздра

dvojka ↔ двојка

voljno ↔ вољно

zaijian ↔ 再见

tschuess ↔ tschüss

geia-sou ↔ γεια σου

khoda-hafez ↔ خداحافظ

hoscakal ↔ hoşçakal

### 205.3.3 Real-ESSI Variables, Basic Units and Flow Control

Variables are defined using the assignment (=) operator. For example,

```
var_x = 7; //Results in the variable x be set to 7 (unitless)
var_y = 3.972e+2; //Scientific notation is available.
```

The language contains a list of reserved keywords. Throughout this documentation, reserved keywords are highlighted in blue or red.

All standard arithmetic operations are available between variables. These operations can be combined arbitrarily and grouped together with parentheses.

```
var_a = var_x + var_y; // Addition
var_b = var_x - var_y; // Subtraction
var_c = var_x * var_y; // Product
var_d = var_x / var_y; // Quotient
var_e = var_y % var_x; // Modulus (how many times x fits in y)
```

The 'print' command can be used to display the current value of a variable.

```
print var_x;
print var_y;
print var_a;
print var_b;
print var_c;
print var_d;
print var_e;
```

Command line output

```
var_x = 7 []
var_y = 397.2 []
var_a = 404.2 []
var_b = -390.2 []
var_c = 2780.4 []
var_d = 0.0176234 []
var_e = 5.2 []
```

Here the "unit" (sign) [] means that the quantities are unitless.

The command 'whos' is used to see all the currently defined variables and their values. After a fresh start of essi, needed to clear up all the previously defined variables, command whos;' produces a list of predefined variables:

Command line output

```
ESSI> whos;

Declared variables:
  *       Day =            86400 [s]
```

```
  *       GPa =                1 [GPa]
  *      Hour =             3600 [s]
  *        Hz =                1 [Hz]
  *       MPa =                1 [MPa]
  *    Minute =               60 [s]
  *         N =                1 [N]
  *        Pa =                1 [Pa]
  *      Week =           604800 [s]
  *        cm =                1 [cm]
  *      feet =           0.3048 [m]
  *        ft =           0.3048 [m]
  *         g =             9.81 [m*s^-2]
  *      inch =           0.0254 [m]
  *        kN =                1 [kN]
  *       kPa =                1 [kPa]
  *        kg =                1 [kg]
  *       kip =          4448.22 [N]
  *        km =                1 [km]
  *       ksi =     6.89476e+06 [Pa]
  *       lbf =          4.44822 [N]
  *       lbm =         0.453592 [kg]
  *         m =                1 [m]
  *      mile =          1609.35 [m]
  *        mm =                1 [mm]
  *        pi =          3.14159 []
  *       psi =          6894.76 [Pa]
  *         s =                1 [s]
  *      yard =           0.9144 [m]

    * = locked variable
ESSI>
```

Predefined variables shown above have a preceding asterisk to show they are locked variables which cannot be modified. The purpose of these locked variables are to provide names for units. Imperial units are also supported as shown above.

The units for variable are shown between the brackets. Note that unit variables have the same name as their unit, which is not the case for user defined variables. Variables preceded by a star (*) are locked variables which can't be modified.

For example, the variable 'm' defines 'meter'. So to define a new variable L1 which has meter units we do:

```
1 L1 = 1*m; // Defines L1 to 1 m.
2 L2 = 40*mm; // Defines L2 to be 40 millimeters.
```

Even though L2 was created with millimeter units, it is stored in base units.

print L2; displays

Command line output

```
L2 = 0.04 [m]
```

As additional examples, let us define few forces:

```
F1 = 10*kN;
F2 = 300*N;
F3 = 4*kg*g;
```

Here g is the predefined acceleration due to gravity.

Arithmetic operations do check (and enforce) for unit consistency. For example, `foo = L1 + F1;` produces an error because units are not compatible. However, `bar = L1 + L2;` is acceptable. On the other hand, multiplication, division and modulus, always work because the result produces a quantity with new units (except when the adimensional quantity is involved).

```
A = L1*L2;
Stress_n = F1 / A;
```

Units for all variables are internally converted to SI units ($kg-m-s$) and stored in that unit system. Variables can be displayed using different units by using the `[]` operator. This does not change the variable, it just displays the value of variable with required unit. For example,

```
print Stress_n; //Print in base SI units.
print Stress_n in Pa; //Print in Pascal
print Stress_n in kPa; //Print in kilo Pascal
```

Command line output

```
Stress_n = 250000 [kg*m^-1*s^-2]

Stress_n = 250000 [Pa]

Stress_n = 250 [kPa]
```

The DSL provides functions to test the physical units of variables. For example,

`print isForce(F1);`

Will print an adimensional, Boolean `1` because `F1` has units of force. While,

`print isPressure(F);`

will print an adimensional, Boolean `0`. The language also provides comparison of quantities with same units (remember all values are compared in SI Units).

`print F1 > F2;`

will print an adimensional, Boolean `1` since `F1` is greater than `F2`.

The program flow can be controlled with `if` and `while` statements, i.e.:

```
if (isForce(F1))
{
 print F1; // This will be executed
};

if (isForce(L1))
{
 print L1; // This will not.
};
```

Note the necessary semicolon (;) at the closing brace. Unlike C/C++, the braces are always necessary. Closing colon is also always necessary.

The "else" statement is also available:

```
if (isForce(L1))
{
 print L1; // This will not execute
}
else
{
 print L2; // This will execute instead
};
```

While loops are also available:

```
i = 0;
while( i < 10)
{
 print i;
 i = i +1;
};
```

### 205.3.4 Modeling

This section details ESSI modeling commands. Angle brackets <> are used for quantity or variable placeholder, that is, they indicate where user input goes. Within the angle brackets, the expected unit type is given as well, i.e.. <L> means the command expects an input with a value and a length unit. The symbol <.> represents the adimensional quantity.

In addition to that, the vertical bar | ("OR" sign)) is used to separate two or more keyword options, i.e. [a|b|c] is used indicate keyword options a or b or c. The symbol |...| is used to denote where several long options exist and are explained elsewhere (an example of this is available below in a material model definitions).

All commands require unit consistency. Base units, SI or other can be used as indicated below:

- length, symbol $L$, units [m, inch, ft]
- mass, symbol $M$, units [kg, lbm],
- time, symbol $T$, units [s]

Derived units can also be used:

- angle, symbol rad (radian), unit $[dimensionless, L/L]$
- force, symbol N (Newton), units $[N, kN, MN, M * L/T^2]$,
- stress, symbol Pa (Pascal), units $[Pa, kPa, MPa, N/L^2, M/L/T^2]$
- strain, symbol (no symbol), units $[L/L]$
- mass density, symbol (no symbol), units $[M/L^3]$
- force density, symbol (no symbol), units $[M/L^2/T^2]$

All models have to be named: `model name "model_name_string";` This is important as output files are named based on model name.

Each loading stage has to be named as well. A new loading stage[1] is defined like this:

```
new loading stage "loading stage name string";
```

In addition to model name, loading stage name is used for output file name for given loading stage.



[1] See more in section 101.4.5 on page 97 in Jeremić et al. (1989-2025).

#### 205.3.4.1 Modeling, Material Model: Adding a Material Model to the Finite Element Model

Adding constitutive material model to the finite element model/domain is done using command:

```
add material # <.> type |...|
               mass_density = <M/L^3>
               (more model dependent parameters) ;
```

- Material number # (or alternatively `No`) is a distinct integer number used to uniquely identify this material.
- Mass density should be defined for each material (even if only static analysis is performed, for example if self weight is to be used as a loading stage).
- Depending on material model, there will be additional material parameters that are defined for each material model/type below:

Starting with version `03-NOV-2015` all elastic-plastic material models require explicit specification of the constitutive integration algorithm. More information on this can be found in 205.3.5.15. Only the material `linear_elastic_isotropic_3d_LT` ignores this option.

Choices for `material_type` are listed below.

#### 205.3.4.2 Modeling, Material Model: Linear Elastic Isotropic Material Model

The command is:

```
add material # <.> type linear_elastic_isotropic_3d
    mass_density = <M/L^3>
    elastic_modulus = <F/L^2>
    poisson_ratio = <.>;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is an isotropic modulus of elasticity of a material (units: stress)
- `poisson_ratio` is a Poisson's ratio [dimensionless]

More on this material model can be found in Section 104.6.1 on Page 220 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.3 Modeling, Material Model: Cross Anisotropic Linear Elastic Material Model

The command is:

```
add material # <.> <material_number>
            type linear_elastic_crossanisotropic
            mass_density = <M/L^3>
            elastic_modulus_horizontal = <F/L^2>
            elastic_modulus_vertical = <F/L^2>
            poisson_ratio_h_v = <.>
            poisson_ratio_h_h = <.>
            shear_modulus_h_v = <F/L^2>;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus_horizontal` is an anisotropic modulus of elasticity for horizontal plane of a material $[F/L^2]$
- `elastic_modulus_vertical` is an anisotropic modulus of elasticity for vertical direction of a material $[F/L^2]$
- `poisson_ratio_h_v` is a Poisson's ratio for horizontal-vertical directions [dimensionless]
- `poisson_ratio_h_h` is a Poisson's ratio for horizontal-horizontal directions [dimensionless]
- `shear_modulus_h_v` is a shear modulus for horizontal-vertical directions $[F/L^2]$

It is assumed that vertical axes is global $Z$ axes.

More on this material model can be found in Section 104.6.1 on Page 220 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.4 Modeling, Material Model: von Mises Associated Material Model with Linear Isotropic and/or Kinematic Hardening

Implements von Mises family of constitutive models, with linear kinematic and/or isotropic hardening.

The command is:

```
add material # <.> type vonMises
  mass_density = <M/L^3>
  elastic_modulus = <F/L^2>
  poisson_ratio = <.>
  von_mises_radius = <F/L^2>
  kinematic_hardening_rate = <F/L^2>
  isotropic_hardening_rate = <F/L^2> ;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material [ ]
- `von_mises_radius` is the radius of the deviatoric section of the von Mises yield surface $[F/L^2]$
- `kinematic_hardening_rate` is the rate of the kinematic hardening $[F/L^2]$
- `isotropic_hardening_rate` is the rate of the kinematic hardening $[F/L^2]$

More on this material model can be found in Section 104.6.6 on Page 226 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.5 Modeling, Material Model: von Mises Associated Material Model with Isotropic Hardening and/or Armstrong-Frederic Nonlinear Kinematic Hardening

This command is for von Mises family of constitutive models, with Armstrong-Frederick kinematic and/or isotropic hardening.

The command is:

```
add material # <.> type vonMisesArmstrongFrederick
    mass_density = <M/L^3>
    elastic_modulus = <F/L^2>
    poisson_ratio = <.>
    von_mises_radius = <.>
    armstrong_frederick_ha = <F/L^2>
    armstrong_frederick_cr = <.>
    isotropic_hardening_rate = <F/L^2> ;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material [ ]
- `von_mises_radius` is the radius of the deviatoric section of the von Mises yield surface $[F/L^2]$
- `armstrong_frederick_ha` controls rate of the kinematic hardening $[F/L^2]$
- `armstrong_frederick_cr` controls the saturation limit for kinematic hardening [Dimensionless]
- `isotropic_hardening_rate` is the rate of the kinematic hardening $[F/L^2]$

More on this material model can be found in Section 104.6.6 on Page 226 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.6 Modeling, Material Model: Drucker-Prager Associated Material Model with Linear Isotropic and/or Kinematic Hardening

This command is for Drucker-Prager family of constitutive models, with linear kinematic and/or isotropic hardening. This material uses associate plastic flow rule.

The command is:

```
add material # <.> type DruckerPrager
  mass_density = <M/L^3>
  elastic_modulus = <F/L^2>
  poisson_ratio = <.>
  druckerprager_k = <.>
  kinematic_hardening_rate = <F/L^2>
  isotropic_hardening_rate = <F/L^2>
  initial_confining_stress = <F/L^2> ;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material [ ]
- `druckerprager_k` slope of the Drucker-Prager yield surface in $p$-$q$m space (equivalent to M parameter) [dimensionless]
- `kinematic_hardening_rate` is the rate of the kinematic hardening $[F/L^2]$
- `isotropic_hardening_rate` is the rate of the isotropic hardening $[F/L^2]$

More on this material model can be found in Section 104.6.7 on Page 232 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.7 Modeling, Material Model: Drucker-Prager Associated Material Model with Isotropic Hardening and/or Armstrong-Frederick Nonlinear Kinematic Hardening

A Drucker-Prager constitutive model with associative plastic-flow rule, Armstrong-Frederick kinematic hardening, and linear isotropic hardening and linear elastic isotropic elasticity law.

The command is:

```
add material # <.> type DruckerPragerArmstrongFrederickLE
    mass_density = <M/L^3>
    elastic_modulus = <F/L^2>
    poisson_ratio = <.>
    druckerprager_k = <.>
    armstrong_frederick_ha = <F/L^2>
    armstrong_frederick_cr = <.>
    isotropic_hardening_rate = <F/L^2>
    initial_confining_stress = <F/L^2>;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material [ ]
- `druckerprager_k` slope of the Drucker-Prager yield surface in $p$-$q$m space (equivalent to M parameter) [Dimensionless]
- `armstrong_frederick_ha` controls rate of the kinematic hardening $[F/L^2]$
- `armstrong_frederick_cr` controls the saturation limit for kinematic hardening [Dimensionless]
- `isotropic_hardening_rate` is the rate of the isotropic hardening $[F/L^2]$
- `initial_confining_stress` initial confining (mean) pressure $[F/L^2]$

More on this material model can be found in Section 104.6.7 on Page 232 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.8 Modeling, Material Model: Drucker-Prager Associated Material Model with Isotropic Hardening and/or Armstrong-Frederick Nonlinear Kinematic Hardening and Nonlinear Duncan-Chang Elasticity

A Drucker-Prager constitutive model with associative plastic-flow rule, Armstrong-Frederick kinematic hardening, and Duncan-Chang non-linear isotropic elasticity law.

The command is:

```
add material # <.> type DruckerPragerArmstrongFrederickNE
  mass_density = <M/L^3>
  DuncanChang_K = <.>
  DuncanChang_pa = <F/L^2>
  DuncanChang_n = <.>
  DuncanChang_sigma3_max = <F/L^2>
  DuncanChang_nu = <.>
  druckerprager_k = <.>
  armstrong_frederick_ha = <F/L^2>
  armstrong_frederick_cr = <.>
  isotropic_hardening_rate = <F/L^2>
  initial_confining_stress = <F/L^2>;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `DuncanChang_K` parameter controlling Young's modulus $[< . >]$
- `DuncanChang_pa` reference pressure $[F/L^2]$
- `DuncanChang_n` exponent $[< . >]$
- `DuncanChang_sigma3_max` maximum value for $\sigma_3$ ($\sigma_3 < 0$) elastic properties are constant for greater values of $\sigma_3$ $[F/L^2]$
- `DuncanChang_nu` Poisson's ratio $[F/L^2]$
- `druckerprager_k` slope of the Drucker-Prager yield surface in $p$-$q$m space (equivalent to M parameter) [Dimensionless]
- `armstrong_frederick_ha` controls rate of the kinematic hardening $[F/L^2]$
- `armstrong_frederick_cr` controls the saturation limit for kinematic hardening [Dimensionless]
- `isotropic_hardening_rate` is the rate of the isotropic hardening $[F/L^2]$
- `initial_confining_stress` initial confining (mean) pressure $[F/L^2]$

More on this material model can be found in Section 104.6.7 on Page 232 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.9 Modeling, Material Model: Drucker-Prager Nonassociated Material Model with Linear Isotropic and/or Kinematic Hardening

This command defines Drucker-Prager family of constitutive models, with linear kinematic and/or isotropic hardening. This material uses non-associate plastic flow rule.

The command is:

```
add material # <.> type DruckerPragerNonAssociateLinearHardening
    mass_density = <M/L^3>
    elastic_modulus = <F/L^2>
    poisson_ratio = <.>
    druckerprager_k = <.>
    kinematic_hardening_rate = <F/L^2>
    isotropic_hardening_rate = <F/L^2>
    initial_confining_stress = <F/L^2>
    plastic_flow_xi = <.>
    plastic_flow_kd = <.>;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material [ ]
- `druckerprager_k` slope of the Drucker-Prager yield surface in $p$-$q$m space (equivalent to M parameter) [Dimensionless]
- `kinematic_hardening_rate` is the linear rate of the kinematic hardening $[F/L^2]$
- `isotropic_hardening_rate` is the linear rate of the isotropic hardening $[F/L^2]$
- `initial_confining_stress` initial confining (mean) pressure $[F/L^2]$
- `plastic_flow_xi` governs the amplitude of plastic volume changes. The higher $\xi$, the higher the dilatancy. If $\xi = 0$, the material model will only produce deviatoric plastic strains. [.]
- `plastic_flow_kd` governs the size of the dilatancy surface, a cone in the stress space on which no plastic volume changes occur. $k_d$ governs the size of this cone: if $k_d$ is equal to zero, the dilatancy surface shrinks to a line (the hydrostatic axis), so that only dilative soil deformation is possible. [.]

More on this material model can be found in Section 104.6.7 on Page 232 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.10 Modeling, Material Model: Drucker-Prager Nonassociated Material Model with Linear Isotropic and/or Armstrong-Frederick Nonlinear Kinematic Hardening

This command defines Drucker-Prager family of constitutive models, with nonlinear kinematic and/or linear isotropic hardening. This material uses non-associated plastic flow rule.

The command is:

```
add material # <.> type DruckerPragerNonAssociateArmstrongFrederick
   mass_density = <M/L^3>
   elastic_modulus = <F/L^2>
   poisson_ratio = <.>
   druckerprager_k = <.>
   armstrong_frederick_ha = <F/L^2>
   armstrong_frederick_cr = <.>
   isotropic_hardening_rate = <F/L^2>
   initial_confining_stress = <F/L^2>
   plastic_flow_xi = <.>
   plastic_flow_kd = <.>;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material [ ]
- `druckerprager_k` slope of the Drucker-Prager yield surface in $p$-$q$m space (equivalent to M parameter) [Dimensionless]
- `armstrong_frederick_ha` a kinematic hardening parameter, which governs the initial stiffness after the yield $[F/L^2]$
- `armstrong_frederick_cr` a kinematic hardening parameter. $\frac{h_a}{c_r}$ governs the limit of the back-stress [Dimensionless]
- `isotropic_hardening_rate` is the rate of the kinematic hardening $[F/L^2]$
- `initial_confining_stress` initial confining (mean) pressure $[F/L^2]$
- `plastic_flow_xi` governs the amplitude of plastic volume changes - the higher $\xi$, the higher the dilatancy. If $\xi = 0$, the material model will only produce deviatoric plastic strains. [.]
- `plastic_flow_kd` governs the size of the dilatancy surface, a cone in the stress space on which no plastic volume changes occur. $k_d$ governs the size of this cone: if $k_d$ is equal to zero, the

dilatancy surface shrinks to a line (the hydrostatic axis), so that only dilative soil deformation is possible. [.]

More on this material model can be found in Section 104.6.7 on Page 232 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

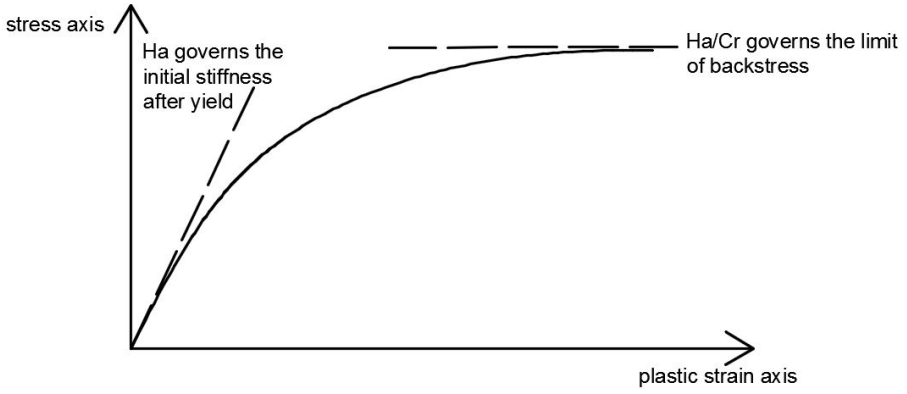



Figure 205.1: The physical meanings of $h_a$ and $c_r$

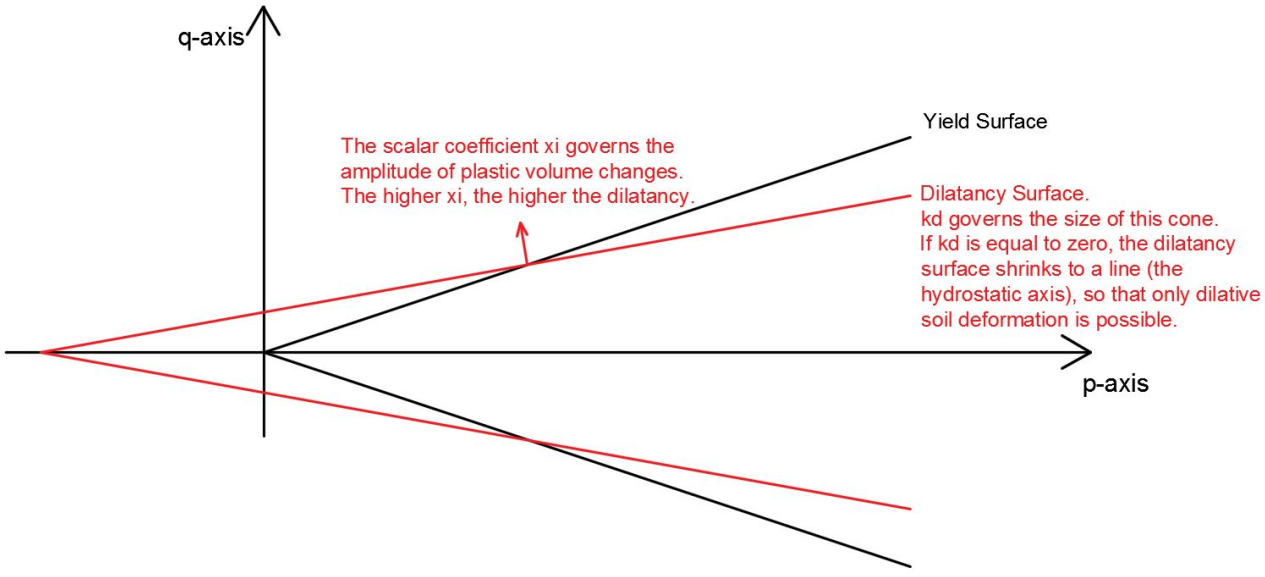



Figure 205.2: The physical meanings of $\xi$ and $k_d$

The physical meanings of $h_a$, $c_r$, $\xi$, and $k_d$ are shown in Figure (205.1) and Figure (205.2).

#### 205.3.4.11 Modeling, Material Model: Hyperbolic Drucker-Prager Nonassociated Material Model with Linear Isotropic and/or Armstrong-Frederick Nonlinear Kinematic Hardening

This command defines a hyperbolic Drucker-Prager constitutive model, with nonlinear kinematic and/or linear isotropic hardening. This material uses non-associated plastic flow rule.

The command is:

```
add material # <.> type HyperbolicDruckerPragerNonAssociateArmstrongFrederick
   mass_density = <M/L^3>
   elastic_modulus = <F/L^2>
   poisson_ratio = <.>
   friction_angle = <.>
   cohesion = <F/L^2>
   rounded_distance = <F/L^2>
   armstrong_frederick_ha = <F/L^2>
   armstrong_frederick_cr = <.>
   isotropic_hardening_rate = <F/L^2>
   initial_confining_stress = <F/L^2>
   plastic_flow_xi = <.>
   plastic_flow_kd = <.>;
```

where:

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio for material [ ]
- `friction_angle` is the initial friction angle of the material. If isotropic hardening is present, friction angle will evolve. [rad]
- `cohesion` is a material constant that defines the cohesion of the material $[F/L^2]$
- `rounded_distance` is the parameter that controls the shape of the rounded apex of yield surface $[F/L^2]$
- `armstrong_frederick_ha` a kinematic hardening parameter, that governs the initial stiffness after the yield $[F/L^2]$
- `armstrong_frederick_cr` a kinematic hardening parameter. It is noted that ratio $\frac{h_a}{c_r}$ controls the asymptote the back-stress, that can be related to the ultimate shear strength [.]
- `isotropic_hardening_rate` is the rate of the isotropic hardening $[F/L^2]$

- `initial_confining_stress` initial confining (mean) pressure, a small value just get initial stress out of cone zone $[F/L^2]$
- `plastic_flow_xi` governs the amplitude of plastic volume changes - the higher $\xi$, the higher the dilatancy. If $\xi = 0$, the material model will only produce deviatoric plastic strains. [.]
- `plastic_flow_kd` governs the size of the dilatancy surface, a cone in the stress space on which no plastic volume changes occur. $k_d$ governs the size of this cone: if $k_d$ is equal to zero, the dilatancy surface shrinks to a line (the hydrostatic axis), so that only dilative soil deformation is possible. [.]

More on this material model can be found in Section 104.6.8.5 on Page 247 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.12 Modeling, Material Model: Rounded Mohr-Coulomb Associated Linear Isotropic Hardening Material Model

The command is:

```
add material # <.> type roundedMohrCoulomb
    mass_density = <M/L^3>
    elastic_modulus = <F/L^2>
    poisson_ratio = <.>
    RMC_m = <.>
    RMC_qa = <F/L^2>
    RMC_pc = <F/L^2>
    RMC_e = <.>
    RMC_eta0 = <.>
    RMC_Heta = <F/L^2>
    initial_confining_stress = <F/L^2>
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material [ ]
- `RMC_m` $0 < m < 1$ parameter of the RMC yield function. Controls roundness of apex in $p$-$q$ space. [ ]
- `RMC_qa` $q_a$ parameter of the RMC yield function. Controls roundness of apex in $p$-$q$ space. $[F/L^2]$
- `RMC_pc` $p$ pressure offset $[F/L^2]$
- `RMC_e` $e$ parameter controls roundness of the deviatoric cross-section of the yield surface. $0.5 < e <= 1$, $e = 0.5$ results in a triangular deviatoric section while $e = 1$ is round. [ ]
- `RMC_eta0` controls the opening of the yield surface [ ]
- `RMC_Heta` isotropic (linear) hardening of the yield surface $[F/L^2]$
- `initial_confining_stress` initial confining (mean) pressure $[F/L^2]$

More on this material model can be found in Section 104.6.9 on Page 250 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.13 Modeling, Material Model: Cam Clay Material Model

The command is:

```
add material # <.> type CamClay
  mass_density = <M/L^3>
  M = <.>
  lambda = <.>
  kappa = <.>
  e0 = <.>
  p0 = <F/L^2>
  poisson_ratio = <.>
  initial_confining_stress = <F/L^2>
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `e0` void ratio ($e_0$) at the reference pressure, [dimensionless]
- `M` Cam-Clay slope of the critical state line in stress space, [dimensionless]
- `lambda` Cam-Clay normal consolidation line slope, (unit: dimensionless)
- `kappa` Cam-Clay unload-reload line slope, (unit: dimensionless)
- `poisson_ratio` Constant Poisson-ratio
- `p0` Cam-Clay parameter ($p_0$). Tip of the yield surface in $q$-$p$ space. $[F/L^2]$
- `initial_confining_stress` initial confining (mean) pressure $[F/L^2]$

More on this material model can be found in Section 104.6.10 on Page 251 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.14 Modeling, Material Model: von Mises Associated Multiple Yield Surface Material Model

The command is:

```
add material # <.> type vonMisesMultipleYieldSurface
  mass_density = <M/L^3>
  elastic_modulus = <F/L^2>
  poisson_ratio = <.>
  total_number_of_yield_surface = <.>
  radiuses_of_yield_surface = <string>
  radiuses_scale_unit = <F/L^2>
  hardening_parameters_of_yield_surfaces = <string>
  hardening_parameters_scale_unit = <F/L^2> ;
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of the material $[F/L^2]$
- `poisson_ratio` is the constant Poisson-ratio [dimensionless]
- `total_number_of_yield_surface` is the total number of yield surfaces. [dimensionless]
- `radiuses_of_yield_surface` is the radius list of multiple yield surfaces. This parameter gives the radiuses of each yield surface from the smallest to the biggest. This parameter should be a string which contains the dimensionless radiuses. The radiuses should be separated by a blank space or a comma. [string]
- `radiuses_scale_unit` is the unit of the each yield surface. This parameter also provides a method to scale up or scale down the radiuses of each yield surfaces. $[F/L^2]$
- `hardening_parameters_of_yield_surfaces` is the hardening parameters corresponding to each yield surface. This parameter should be a string which contains the dimensionless hardening parameters. The hardening parameters should be separated by a blank space or a comma. [string]
- `hardening_parameters_scale_unit` The unit of the each hardening parameter. This parameter also provides a method to scale up or scale down the hardening parameter of each yield surfaces. $[F/L^2]$

#### 205.3.4.15 Modeling, Material Model: von Mises Associated Multiple Yield Surface Material Model that Matches $G/G_{max}$ Curves

The command is:

```
add material # <.> type vonMisesMultipleYieldSurfaceGoverGmax
  mass_density = <M/L^3>
  initial_shear_modulus = <F/L^2>
  poisson_ratio = <.>
  total_number_of_shear_modulus = <.>
  GoverGmax = <string>
  ShearStrainGamma = <string> ;
```

Command Example is

```
add material # 1 type vonMisesMultipleYieldSurfaceGoverGmax
   mass_density = 0.0*kg/m^3
   initial_shear_modulus = 3E8 * Pa
   poisson_ratio = 0.0
   total_number_of_shear_modulus = 9
   GoverGmax =
   "1,0.995,0.966,0.873,0.787,0.467,0.320,0.109,0.063"
   ShearStrainGamma =
   "0,1E-6,1E-5,5E-5,1E-4, 0.0005, 0.001, 0.005, 0.01";
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `initial_shear_modulus` is the initial maximum shear modulus, namely, the Gmax. $[F/L^2]$
- `poisson_ratio` is the constant Poisson-ratio. [dimensionless]
- `total_number_of_shear_modulus` is the total number of shear modulus, including the initial maximum shear modulus. The total number of yield surface is one less than the total number of shear modulus. Namely, (N+1) areas are divided by N surfaces. [dimensionless]
- `GoverGmax` is the G/Gmax from experiments, including the initial shear modulus. Namely, the first element should be 1.0. Each element is dimensionless. The input should be separated by a blank space or a comma. [string]
- `ShearStrainGamma` is the shear strain $\gamma$ corresponding to the GoverGmax. Note that $\gamma = 2\varepsilon$ when the input is prepared. The first element should be 0.0 corresponding to the initial shear modulus. Each element is dimensionless. The input should be separated by a blank space or a comma. [string]

#### 205.3.4.16 Modeling, Material Model: Drucker-Prager Nonassociated Multi-Yield Surface Material Model

The command is:

```
add material # <.> type DruckerPragerMultipleYieldSurface
  mass_density = <M/L^3>
  elastic_modulus = <F/L^2>
  poisson_ratio = <.>
  initial_confining_stress = <F/L^2>
  reference_pressure = <F/L^2>
  pressure_exponential_n = <.>
  cohesion = <F/L^2>
  dilation_angle_eta = <.>
  dilation_scale = <.>
  total_number_of_yield_surface = <.>
  sizes_of_yield_surfaces = <string>
  yield_surface_scale_unit = <F/L^2>
  hardening_parameters_of_yield_surfaces = <string>
  hardening_parameters_scale_unit = <F/L^2>;
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of the material $[F/L^2]$
- `poisson_ratio` is the constant Poisson-ratio [dimensionless]
- `initial_confining_stress` is the initial confining (mean) pressure $[F/L^2]$
- `reference_pressure` is the reference pressure for the initial modulus. This parameter is usually 101kPa. $[F/L^2]$
- `pressure_exponential_n` is the exponential number of the pressure dependent modulus. [dimensionless]
- `cohesion` is the attraction force is the soil. $[F/L^2]$
- `dilation_angle_eta` controls the dilation and compaction of the material. When the stress ratio is smaller than this parameter, plastic compaction takes place. When the stress ratio is greater than this parameter, the plastic dilation takes place. [dimensionless]
- `dilation_scale` controls the rate of the dilation or compaction in the plastic flow. [dimensionless]
- `total_number_of_yield_surface` is the total number of yield surfaces. [dimensionless]

- `radiuses_of_yield_surface` is the radius list of multiple yield surfaces. This parameter gives the radiuses of each yield surface from the smallest to the biggest. This parameter should be a string which contains the dimensionless radiuses. The radiuses should be separated by a blank space or a comma. [string]

- `radiuses_scale_unit` is the unit of the each yield surface. This parameter also provides a method to scale up or scale down the radiuses of each yield surfaces. $[F/L^2]$

- `hardening_parameters_of_yield_surfaces` is the hardening parameters corresponding to each yield surface. This parameter should be a string which contains the dimensionless hardening parameters. The hardening parameters should be separated by a blank space or a comma. [string]

- `hardening_parameters_scale_unit` The unit of the each hardening parameter. This parameter also provides a method to scale up or scale down the hardening parameter of each yield surfaces. $[F/L^2]$

#### 205.3.4.17 Modeling, Material Model: Drucker-Prager Nonassociated Material Model that Matches $G/G_{max}$ Curves

The command is:

```
add material # <.> type DruckerPragerMultipleYieldSurfaceGoverGmax
  mass_density = <M/L^3>
  initial_shear_modulus = <F/L^2>
  poisson_ratio = <.>
  initial_confining_stress = <F/L^2>
  reference_pressure = <F/L^2>
  pressure_exponential_n = <.>
  cohesion = <F/L^2>
  dilation_angle_eta = <.>
  dilation_scale = <.>
  total_number_of_shear_modulus = <.>
  GoverGmax = <string>
  ShearStrainGamma = <string>
```

Command Example is

```
add material # 1 type DruckerPragerMultipleYieldSurfaceGoverGmax
  mass_density = 0.0*kg/m^3
  initial_shear_modulus = 3E8 * Pa
  poisson_ratio = 0.0
  initial_confining_stress = 1E5 * Pa
  reference_pressure = 1E5 * Pa
  pressure_exponential_n = 0.5
  cohesion = 0. * Pa
  dilation_angle_eta =1.0
  dilation_scale = 0.0
  total_number_of_shear_modulus = 9
  GoverGmax =
  "1,0.995,0.966,0.873,0.787,0.467,0.320,0.109,0.063"
  ShearStrainGamma =
  "0,1E-6,1E-5,5E-5,1E-4, 0.0005, 0.001, 0.005, 0.01";
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of the material $[F/L^2]$
- `poisson_ratio` is the constant Poisson-ratio [dimensionless]
- `initial_confining_stress` is the initial confining (mean) pressure $[F/L^2]$
- `reference_pressure` is the reference pressure for the initial modulus. This parameter is usually 101kPa. $[F/L^2]$

- `pressure_exponential_n` is the exponential number of the pressure dependent modulus. [dimensionless]
- `cohesion` is the attraction force is the soil. [$F/L^2$]
- `dilation_angle_eta` controls the dilation and compaction of the material. When the stress ratio is smaller than this parameter, plastic compaction takes place. When the stress ratio is greater than this parameter, the plastic dilation takes place. [dimensionless]
- `dilation_scale` controls the rate of the dilation or compaction in the plastic flow. For this automatic G/Gmax match, the dilation scale has to be zero, which means only deviatoric plastic flow is allowed. If the users want to have volumetric dilation, they can match the G/Gmax manually with the other DruckerPragerMultipleYieldSurface command. [dimensionless]
- `total_number_of_shear_modulus` is the total number of shear modulus, including the initial maximum shear modulus. The total number of yield surface is one less than the total number of shear modulus. Namely, (N+1) areas are divided by N surfaces. [dimensionless]
- `GoverGmax` is the G/Gmax from experiments, including the initial shear modulus. Namely, the first element should be 1.0. Each element is dimensionless. The input should be separated by a blank space or a comma. [string]
- `ShearStrainGamma` is the shear strain $\gamma$ corresponding to the GoverGmax. Note that $\gamma = 2\varepsilon$ when the input is prepared. The first element should be 0.0 corresponding to the initial shear modulus. Each element is dimensionless. The input should be separated by a blank space or a comma. [string]

#### 205.3.4.18 Modeling, Material Model: Rounder Mohr-Coulomb Nonassociated Multi-Yield Surface Material Model

The command is:

```
add material # <.> type RoundedMohrCoulombMultipleYieldSurface
  mass_density = <M/L^3>
  elastic_modulus = <F/L^2>
  poisson_ratio = <.>
  initial_confining_stress = <F/L^2>
  reference_pressure = <F/L^2>
  pressure_exponential_n = <.>
  cohesion = <F/L^2>
  RMC_shape_k =<.>
  dilation_angle_eta = <.>
  dilation_scale = <.>
  total_number_of_yield_surface = <.>
  sizes_of_yield_surfaces = <string>
  yield_surface_scale_unit = <F/L^2>
  hardening_parameters_of_yield_surfaces = <string>
  hardening_parameters_scale_unit = <F/L^2>;
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `elastic_modulus` is the elastic modulus of the material $[F/L^2]$
- `poisson_ratio` is the constant Poisson-ratio [dimensionless]
- `initial_confining_stress` is the initial confining (mean) pressure $[F/L^2]$
- `reference_pressure` is the reference pressure for the initial modulus. This parameter is usually 101kPa. $[F/L^2]$
- `pressure_exponential_n` is the exponential number of the pressure dependent modulus. [dimensionless]
- `cohesion` is the attraction force is the soil. $[F/L^2]$
- `RMC_shape_k` controls the shape of the rounded Mohr-Coulomb yield surface. [dimensionless]
- `dilation_angle_eta` controls the dilation and compaction of the material. When the stress ratio is smaller than this parameter, plastic compaction takes place. When the stress ratio is greater than this parameter, the plastic dilation takes place. [dimensionless]
- `dilation_scale` controls the rate of the dilation or compaction in the plastic flow. [dimensionless]

- `total_number_of_yield_surface` is the total number of yield surfaces. [dimensionless]
- `radiuses_of_yield_surface` is the radius list of multiple yield surfaces. This parameter gives the radiuses of each yield surface from the smallest to the biggest. This parameter should be a string which contains the dimensionless radiuses. The radiuses should be separated by a blank space or a comma. [string]
- `radiuses_scale_unit` is the unit of the each yield surface. This parameter also provides a method to scale up or scale down the radiuses of each yield surfaces. $[F/L^2]$
- `hardening_parameters_of_yield_surfaces` is the hardening parameters corresponding to each yield surface. This parameter should be a string which contains the dimensionless hardening parameters. The hardening parameters should be separated by a blank space or a comma. [string]
- `hardening_parameters_scale_unit` The unit of the each hardening parameter. This parameter also provides a method to scale up or scale down the hardening parameter of each yield surfaces. $[F/L^2]$

#### 205.3.4.19 Modeling, Material Model: Tsinghhua Liquefaction Material Model

The command is:

```
add material # <.> type TsinghuaLiquefactionModel
  mass_density = <M/L^3>
  poisson_ratio = <.>
  initial_confining_stress = <F/L^2>
  liquefaction_G0 = <.>
  liquefaction_EXPN = <.>
  liquefaction_c_h0 = <.>
  liquefaction_mfc = <.>
  liquefaction_mdc = <.>
  liquefaction_dre1 = <.>
  liquefaction_Dre2 = <.>
  liquefaction_Dir = <.>
  liquefaction_Alpha = <.>
  liquefaction_gamar = <.>
  liquefaction_pa = <.>
  liquefaction_pmin = <.>
```

Command Example is

```
add material # 1 type TsinghuaLiquefactionModel
  mass_density = 0.0*kg/m^3
  poisson_ratio = 0.1
  initial_confining_stress = 1E5 *Pa
  liquefaction_G0 = 800
  liquefaction_EXPN = 0.5
  liquefaction_c_h0 = 1.0
  liquefaction_mfc = 1.2
  liquefaction_mdc = 0.4
  liquefaction_dre1 = 0.5
  liquefaction_Dre2 = 1500
  liquefaction_Dir = 0.1
  liquefaction_Alpha = 0.01
  liquefaction_gamar = 0.01
  liquefaction_pa = 1E5
  liquefaction_pmin = 100 ;
```

where

- `mass_density` is the mass density of material $[M/L^3]$
- `poisson_ratio` is the constant Poisson ratio [dimensionless]
- `initial_confining_stress` is the initial confining (mean) pressure $[F/L^2]$
- `liquefaction_G0` is initial modulus scale at the reference pressure. For medium dense soil, G0 is 800. [dimensionless]

- `liquefaction_EXPN` is the exponential number of the pressure dependent modulus. [dimensionless]
- `liquefaction_c_h0` is the plastic modulus coefficient. This parameter should be determined by the G/Gmax curve. When the G/Gmax curve is hyperbolic, h is 1.2. The range of h is 0.7-1.2 [dimensionless].
- `liquefaction_mfc` is the slope of the failure surface in p-q plane. The range of $M_{f,c}$ is 1.4-1.8 [dimensionless].
- `liquefaction_mdc` is the slope of the phase transition surface in p-q plane. The range of $M_{d,c}$ is 0.3-1.0 [dimensionless].
- `liquefaction_dre1` is the accumulation coefficient of the reversible dilatancy. This parameter is usually 0.4 [dimensionless].
- `liquefaction_Dre2` is the release coefficient of the reversible dilatancy. This range of $d_{re,2}$ is 1000-1500 [dimensionless].
- `liquefaction_Dir` is the coefficient of irreversible dilatancy. The parameter $d_{ir}$ controls the initial slope of the irreversible strain development with respect to the number of reversible loadings. Intuitively, when $d_{ir}$ is bigger, the soil becomes liquefaction faster. The parameter $d_{ir}$ can be around 0.2 [dimensionless].
- `liquefaction_Alpha` is the limit of the irreversible strain. Intuitively, $\alpha$ controls the maximum strain after the liquefaction. The parameter $\alpha$ can be around 0.03 [dimensionless].
- `liquefaction_gamar` is the maximum shear strain length in one liquefaction loading. Intuitively, this parameter controls the maximum strain size of one loop. This parameter can be around 0.05 [dimensionless].
- `liquefaction_pa` is the reference pressure. Usually, this parameter is 10000 [dimensionless].
- `liquefaction_pmin` is the minimum pressure in the calculation. If the pressure is smaller than $p_{min}$ during the calculation, the pressure will be set to $p_{min}$. This parameter can be 1. Increasing this parameter can avoid the potential numerical errors on small numbers [dimensionless].

205.3.4.20 Modeling, Material Model: SANISand Material Model, version 2004

The command is:

```
add material # <.> type sanisand2004
   mass_density = <M/L^3>
   e0 = <.>
   sanisand2004_G0 = <.>
   poisson_ratio = <.>
   sanisand2004_Pat = <.>
   sanisand2004_p_cut = <.>
   sanisand2004_Mc = <.>
   sanisand2004_c = <.>
   sanisand2004_lambda_c = <.>
   sanisand2004_xi = <.>
   sanisand2004_ec_ref = <.>
   sanisand2004_m = <.>
   sanisand2004_h0 = <.>
   sanisand2004_ch = <.>
   sanisand2004_nb = <.>
   sanisand2004_A0 = <.>
   sanisand2004_nd = <.>
   sanisand2004_z_max = <.>
   sanisand2004_cz = <.>
   initial_confining_stress = <F/L^2>;
```

where

- `MaterialNumber`: Material tag
- `mass_density` is the mass density of material $[M/L^3]$
- `sanisand2004_e0` initial void ratio [ ]
- `sanisand2004_G0` normalized elastic shear modulus [ ]
- `poisson_ratio` Poisson's ratio [ ]
- `sanisand2004_Pat` atmospheric pressure $[F/L^2]$
- `sanisand2004_p_cut` pressure cut-off ratio $[F/L^2]$
- `sanisand2004_Mc` Critical stress ratio at triaxial compression [ ]
- `sanisand2004_c` tension-compression strength ratio $c = M_e/M_c$ [ ]
- `sanisand2004_lambda_c` parameter for critical state line [ ]
- `sanisand2004_xi` parameter for critical state line [ ]

- `sanisand2004_ec_ref` reference void for critical state line [ ]
- `sanisand2004_m` opening of the yield surface [ ]
- `sanisand2004_h0` bounding surface parameter [ ]
- `sanisand2004_ch` bounding surface parameter [ ]
- `sanisand2004_nb` bounding surface parameter [ ]
- `sanisand2004_A0` dilatancy parameter [ ]
- `sanisand2004_nd` dilatancy parameter [ ]
- `sanisand2004_z_max` maximum $z$ fabric parameter [ ]
- `sanisand2004_cz` fabric hardening parameter [
- `initial_confining_stress` is the initial confining stress $p = -1/3\sigma_{ii}$ and it is positive in compressions (since there is that – (minus) sign in front of sum of normal stresses ($\sigma_{ii}$ indicial notation summation convention applies) that are positive in tension [stress].

More on this material model can be found in section 104.6.11 on Page 255 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

Important note: This material model should be used together with explicit constitutive algorithms, e.g. `Forward_Euler` or `Forward_Euler_Subincrement`. For better result, it is suggested to apply strain increments, or sub-increments, smaller than 1e-4.

205.3.4.21 Modeling, Material Model: SANISand Material Model, version 2008

The command is:

```
add material # <.> type sanisand2008
    mass_density = <M/L^3>
    e0 = <.>
    sanisand2008_G0 = <.>
    sanisand2008_K0 = <.>
    sanisand2008_Pat = <.>
    sanisand2008_k_c = <.>
    sanisand2008_alpha_cc = <.>
    sanisand2008_c = <.>
    sanisand2008_lambda = <.>
    sanisand2008_ec_ref = <.>
    sanisand2008_m = <.>
    sanisand2008_h0 = <.>
    sanisand2008_ch = <.>
    sanisand2008_nb = <.>
    sanisand2008_A0 = <.>
    sanisand2008_nd = <.>
    sanisand2008_p_r = <.>
    sanisand2008_rho_c = <.>
    sanisand2008_theta_c = <.>
    sanisand2008_X = <.>
    sanisand2008_z_max = <.>
    sanisand2008_cz = <.>
    sanisand2008_p0 = <F/L^3>
    sanisand2008_p_in = <F/L^3>
    algorithm = explicit (or) implicit
    number_of_subincrements = <.>
    maximum_number_of_iterations = <.>
    tolerance_1 = <.>
    tolerance_2 = <.>;
```

where

- `MaterialNumber`: Number of the ND material to be used ;
- `Algorithm`: Explicit (=0) or Implicit (=1) ;
- `rho`: density ;
- `e0`: initial void ratio at zero strain ;
- `G0`: Reference elastic shear modulus [stress];
- `K0`: Reference elastic bulk modulus [stress];
- `sanisand2008_Pat`: atmospheric pressure for critical state line ;

- `sanisand2008_k_c`: cut-off factor; for $p < k_c P_{at}$, use $p = k_c P_{at}$ for calculation of $G$; (a default value of $k_c = 0.01$ should work fine) ;
- `sanisand2008_alpha_cc`: critical state stress ratio ;
- `sanisand2008_c`: tension-compression strength ratio ;
- `sanisand2008_lambda`: parameter for critical state line ;
- `sanisand2008_xi`: parameter for critical state line ;
- `sanisand2008_ec_ref`: reference void for critical state line, ; $e_c = e_r lambda(p_c/P_{at})^x i$ ;
- `sanisand2008_m`: opening of the yield surface ;
- `sanisand2008_h0`: bounding surface parameter ;
- `sanisand2008_ch`: bounding surface parameter ;
- `sanisand2008_nb`: bounding surface parameter ;
- `sanisand2008_A0`: dilatancy parameter ;
- `sanisand2008_nd`: dilatancy parameter ;
- `sanisand2008_p_r`: LCC parameter ;
- `sanisand2008_rho_c`: LCC parameter ;
- `sanisand2008_theta_c`: LCC parameter ;
- `sanisand2008_X`: LCC parameter ;
- `sanisand2008_z_max`: fabric parameter ;
- `sanisand2008_cz`: fabric parameter ;
- `sanisand2008_p0`: yield surface size ;
- `sanisand2008_p_in` ;
- `number_of_subincrements` number of subincrements in constitutive simulation
- `maximum_number_of_iterations` maximum number of iterations

- `tolerance_1` Explicit: tolerance for intersection point (distance between two consecutive points) Implicit: yield function tolerance
- `tolerance_2` Implicit: residual tolerance

More on this material model can be found in Section 104.6.12 on Page 262 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

205.3.4.22 Modeling, Material Model: Cosserat Linear Elastic Material Model

The command is:

```
add material # <.> type Cosserat_linear_elastic_isotropic_3d
    mass_density = <M/L^3>
    lambda = <F/L^2>
    mu = <F/L^2>
    chi = <F/L^2>
    pi1 = <F>
    pi2 = <F>
    pi3 = <F>
    ;
```

- `MaterialNumber` unique material Number.
- `mass_density` the density of the material.
- `lambda`, `mu`, `chi`, `pi1`, `pi2`, `pi3` are the 6 Cosserat elastic constants (Eringen, 2012).

The relations between elastic constants is as follows Eringen (2012). Note the Young's modulus and the Poisson's ratio are different from the classical elasticity:

- Young's modulus $E = (2\mu + \chi)(3\lambda + 2\mu + \chi)$.
- Shear modulus $G = \mu + 1/2\chi$.
- Poisson's ratio $\nu = \lambda/(2\lambda + 2\mu + \chi)$.
- Characteristic length for torsion $l_t = ((\pi_2 + \pi_3)/(2\mu + \chi))^{1/2}$.
- Characteristic length for bending $l_b = (\pi_3/2(2\mu + \chi))^{1/2}$.
- Coupling number $N = (\chi/2(\mu + \chi))$
- Polar ratio $\Phi = (\pi_2 + \pi_3)/(\pi_1 + \pi_2 + \pi_3)$

According to Eringen Eringen (2012), the 6 elastic constants should satisfy the following conditions

$$\begin{aligned} 3\lambda + 2\mu + \chi \geq 0\ , \quad 2\mu + \chi \geq 0\ , \quad \chi \geq 0\ , \\ 3\pi_1 + \pi_2 + \pi_3 \geq 0\ , \quad \pi_3 + \pi_2 \geq 0\ , \quad \pi_3 - \pi_2 \geq 0\ . \end{aligned} \tag{205.1}$$

#### 205.3.4.23 Modeling, Material Model: von Mises Cosserat Material Model

The command is:

```
add material # <.> type Cosserat_von_Mises
    mass_density = <M/L^3>
    lambda = <F/L^2>
    mu = <F/L^2>
    chi = <F/L^2>
    pi1 = <F>
    pi2 = <F>
    pi3 = <F>
    plastic_internal_length = <L>
    von_mises_radius = <F/L^2>
    isotropic_hardening_rate = <F/L^2>
    ;
```

- `MaterialNumber` unique material Number.
- `mass_density` the density of the material.
- `lambda`, `mu`, `chi`, `pi1`, `pi2`, `pi3` are the 6 Cosserat elastic constantsEringen (2012).
- `plastic_internal_length` is the characteristic length in the plasticity.
- `von_mises_radius` is radius of the unified yield surface of force-stress and couple-stress.
- `isotropic_hardening_rate` is the rate of isotropic hardening.

#### 205.3.4.24 Modeling, Material Model: Uniaxial Linear Elastic, Fiber Material Model

The command is:

```
add material # <.> type uniaxial_elastic
    elastic_modulus = <F/L^2>
    viscoelastic_modulus = <mass / length / time> ;
```

where

- `MaterialNumber` unique material Number.
- `elastic_modulus` elastic modulus of the material.
- `viscoelastic_modulus` damping tangent.

More on this material model can be found in Section ?? on Page ?? in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

As the name implies, `uniaxiale_elastic` material model works with uniaxial element only. For 3D elements, for example solid brick elements, please use 3D material models, for example, `linear_elastic_isotropic_3d`

#### 205.3.4.25 Modeling, Material Model: Stochastic Uniaxial Linear Elastic Model

The command is:

```
add material # <.> type stochastic_uniaxial_elastic uncertain_elastic_modulus = ↩
    random variable # <.> elastic_modulus_scale_unit = <F/L^2>;
```

where

- `uncertain_elastic_modulus` specify uncertain elastic modulus of the material through a defined random variable.
- `elastic_modulus_scale_unit` specify the unit scale factor that would be multiplied with the polynomial chaos coefficients of the random variable.

As the name implies, `stochastic_uniaxial_elastic` material model works with stochastic uniaxial element only.

For example:

```
add material # 1 type stochastic_uniaxial_elastic uncertain_elastic_modulus = ↩
    random variable # 1 elastic_modulus_scale_unit = 1*Pa;
```

Add material #1 as `stochastic_uniaxial_elastic` material with uncertain elastic modulus characterized by the polynomial chaos coefficients of random variable 1 and scale factor $1 * Pa$.

#### 205.3.4.26 Modeling, Material Model: Stochastic Uniaxial Nonlinear Armstrong Frederick Model

The command is:

```
add material # <.> type stochastic_uniaxial_Armstrong_Frederick
constitutive triple product # <.>
armstrong_frederick_ha = random variable # <.>
armstrong_frederick_ha_scale_unit = <F/L^2>
armstrong_frederick_cr = random variable # <.>
```

or

```
add material # <.> type stochastic_uniaxial_Armstrong_Frederick
constitutive triple product # <.>
armstrong_frederick_ha = random variable # <.>
armstrong_frederick_ha_scale_unit = <F/L^2>
armstrong_frederick_cr = random variable # <.>
polynomial_chaos_terms_ha = <.>
polynomial_chaos_terms_cr = <.>
polynomial_chaos_terms_incremental_strain = <.>;
```

Note that the difference between these two commands is that the first command would by default use the full polynomial chaos (PC) bases defined in the provided `constitutive triple product` for probabilistic constitutive modeling. The second command would support user-specified number of polynomial chaos terms for uncertain `armstrong_frederick_ha`, `armstrong_frederick_cr` and `incremental_strain`. This enables users to perform truncation of PC bases for probabilistic constitutive modeling.

The command input parameters are:

- `constitutive triple product #` specifies the ID of the triple product, that would be used in probabilistic constitutive updating. In stochastic finite element method (FEM), the first and second PC basis for this triple product should come from the joint PC representation of uncertain parameters `armstrong_frederick_ha` and `armstrong_frederick_cr`. The third PC basis for this triple product should come from the PC representation of uncertain FEM system response, e.g., uncertain structural displacement.

- `armstrong_frederick_ha = random variable #` specifies the uncertain Armstrong Frederick parameter $ha$ through a defined random variable.

- `armstrong_frederick_ha_scale_unit` specifies the unit scale factor that would be multiplied with the polynomial chaos coefficients of the random variable of uncertain Armstrong Frederick parameter $ha$.

- `armstrong_frederick_cr = random variable #` specifies the uncertain Armstrong Frederick parameter $cr$ through a defined random variable.
- `polynomial_chaos_terms_ha` specifies the number of polynomial chaos basis of uncertain $ha$ involved in the probabilistic constitutive updating.
- `polynomial_chaos_terms_cr` specifies the number of polynomial chaos basis of uncertain $cr$ involved in the probabilistic constitutive updating.
- `polynomial_chaos_terms_incremental_strain` specifies the number of polynomial chaos basis of uncertain incremental strain $d\epsilon$ involved in the probabilistic constitutive updating.

As the name implies, `stochastic_uniaxial_Armstrong_Frederick` material model works with stochastic uniaxial element only.

For example:

```
add material # 1 type stochastic_uniaxial_Armstrong_Frederick
constitutive triple product # 1
armstrong_frederick_ha = random variable # 1
armstrong_frederick_ha_scale_unit = 1*Pa
armstrong_frederick_cr = random variable # 2
polynomial_chaos_terms_ha = 10
polynomial_chaos_terms_cr = 10
polynomial_chaos_terms_incremental_strain = 30;
```

Add material # 1 as `stochastic_uniaxial_Armstrong_Frederick` material with triple product # 1 for probabilistic constitutive updating.

Uncertain parameter $ha$ is characterized by random variable # 1 using scale unit $1*$Pa.

Uncertain parameter $cr$ is characterized by random variable # 2. The number of polynomial chaos basis for uncertain parameters $ha$, $cr$ and incremental strain in probabilistic constitutive updating are 10, 10 and 30, respectively.

205.3.4.27 Modeling, Material Model: Uniaxial Nonlinear Concrete, Fiber Material Model, version 02

The command is:

```
add material # <.> type uniaxial_concrete02
    compressive_strength = <F/L^2>
    strain_at_compressive_strength = <.>
    crushing_strength = <F/L^2>
    strain_at_crushing_strength = <.>
    lambda = <.>
    tensile_strength = <F/L^2>
    tension_softening_stiffness = <F/L^2>;
```

- `compressive_strength` compressive strength.
- `strain_at_compressive_strength` strain at compressive strength.
- `crushing_strength` crushing strength.
- `strain_at_crushing_strength` strain at crushing strength.
- `lambda` ratio between unloading slope at epscu and initial slope.
- `tensile_strength` tensile strength.
- `tension_softening_stiffness` tension softening stiffness (absolute value) (slope of the tension softening branch).

More on this material model can be found in Section ?? on Page ?? in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.28 Modeling, Material Model: Faria-Oliver-Cervera Concrete Material

The command is:

```
add material No (or #) <material_number>
    type FariaOliverCerveraConcrete
    elastic_modulus = <F/L^2>
    poisson_ratio = <.>
    tensile_yield_strength = <F/L^2>
    compressive_yield_strength = <F/L^2>
    plastic_deformation_rate = <.>
    damage_parameter_Ap = <.>
    damage_parameter_An = <.>
    damage_parameter_Bn = <.>
```

where

- `No (or #)<material_number>` is a unique material integer number (does not have to be sequential, any unique positive integer number can be used).
- `type FariaOliverCerveraConcrete` is the material type.
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material.
- `tensile_yield_strength` is the tensile yield strength $[F/L^2]$
- `compressive_yield_strength` is the compressive yield strength $[F/L^2]$

#### 205.3.4.29 Modeling, Material Model: Plane Stress Layered Material

The command is:

```
add material No (or #) <element_number>
    type PlaneStressLayeredMaterial
    number_of_layers = <.>
    thickness_array = <string>
    thickness_scale_unit = <L>
    with material # <string>
    ;
```

where

- `No (or #)<material_number>` is a unique material integer number (does not have to be sequential, any unique positive integer number can be used).
- `type PlaneStressLayeredMaterial` is the material type.
- `number_of_layers` is the number of layers in this layered material. For reinforced concrete wall element, this will be just 3 layers, inside/confined concrete, reinforcement, and outside/unconfined concrete.
- `thickness_array` is the thickness ratio of each individual material.
- `thickness_scale_unit` set the length unit and the scale factor for the thickness of the layered material.
- `material # <string>` is the string of predefined individual material tags.

205.3.4.30 Modeling, Material Model: Uniaxial Nonlinear Steel, Fiber Material Model, version 01

The command is:

```
add material # <.> type uniaxial_steel01
   yield_strength = <F/L^2>
   elastic_modulus = <F/L^2>
   strain_hardening_ratio = <.>
   a1 = <.>
   a2 = <.>
   a3 = <>
   a4 = <.> ;
```

- `yield_strength` yield strength.
- `elastic_modulus` initial elastic tangent.
- `strain_hardening_ratio` strain-hardening ratio (ratio between post-yield tangent and initial elastic tangent).
- `a1, a2, a3, a4` isotropic hardening parameters
  - `a1`: isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic strain of a2*(fy/Ep). ;
  - `a2`: isotropic hardening parameter (see explanation under a1) ;
  - `a3`: isotropic hardening parameter, increase of tension yield envelope as proportion of yield strength after a plastic strain of a4*(fy/Ep) ;
  - `a4`: isotropic hardening parameter (see explanation under a3) ;

More on this material model can be found in Section ?? on Page ?? in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

205.3.4.31 Modeling, Material Model: Uniaxial Nonlinear Steel, Fiber Material Model, version 02

The command is:

```
add material # <.> type uniaxial_steel02
    yield_strength = <F/L^2>
    elastic_modulus = <F/L^2>
    strain_hardening_ratio = <.>
    R0 = <.>
    cR1 = <.>
    cR2 = <.>
    a1 = <.>
    a2 = <.>
    a3 = <>
    a4 = <.> ;
```

- `yield_strength`: yield strength ;
- `elastic_modulus`: initial elastic tangent ;
- `strain_hardening_ratio`: strain-hardening ratio (ratio between post-yield tangent and initial elastic tangent) ;
- `R0, cR1, cR2`: control the transition from elastic to plastic branches. Recommended values: R0=between 10 and 20, cR1=0.925, cR2=0.15 ;
- `a1, a2, a3, a4`: isotropic hardening parameters ;
  - `a1`: isotropic hardening parameter, increase of compression yield envelope as proportion of yield strength after a plastic strain of a2*(Fy/E). ;
  - `a2`: isotropic hardening parameter (see explanation under a1) ;
  - `a3`: isotropic hardening parameter, increase of tension yield envelope as proportion of yield strength after a plastic strain of a4*(Fy/E) ;
  - `a4`: isotropic hardening parameter (see explanation under a3) ;

More on this material model can be found in Section ?? on Page ?? in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.32 Modeling, Material Model: Plane Stress Plastic Damage Concrete Material

This is a plane stress version of the plastic damage concrete model developed by Faria et al. (1998). This material was implemented as part of the endeavor to model reinforced concrete shells, plates and shear walls. It should only be used together with Inelastic Layered Shell Section and 4 Node Shell NLDKGQ/Xin-Zheng-Lu, see page 915.

The command is:

```
add material No (or #) <material_number> type PlasticDamageConcretePlaneStress
    elastic_modulus = <F/L^2>
    poisson_ratio = <.>
    tensile_yield_strength = <F/L^2>
    compressive_yield_strength = <F/L^2>
    plastic_deformation_rate = <.>
    damage_parameter_Ap = <.>
    damage_parameter_An = <.>
    damage_parameter_Bn = <.>
```

where

- `No (or #)<material_number>` is a unique material integer number (does not have to be sequential, any unique positive integer number can be used).
- `elastic_modulus` is the elastic modulus of material $[F/L^2]$
- `poisson_ratio` is the Poisson's ratio material.
- `tensile_yield_strength` is the tensile yield strength $[F/L^2]$
- `compressive_yield_strength` is the compressive yield strength $[F/L^2]$
- `plastic_deformation_rate` governs the post-yield hardening modulus in the effective (undamaged) space and the plastic strain rate
- `damage_parameter_Ap` governs the tensile fracture energy and affects the ductility of the tensile response
- `damage_parameter_An` governs the softening behavior of concrete in compression, it changes the ductility but does not alter the peak strength
- `damage_parameter_Bn` governs the softening behavior of concrete in compression, it changes both the ductility and the peak strength

#### 205.3.4.33 Modeling, Material Model: Plane Stress Rebar Material

This is a plane stress version of the Uniaxial Nonlinear Steel material. This material was implemented as part of the endeavor to model reinforced concrete shells, plates and shear walls. This model should be used together with Inelastic Layered Shell Section and 4 Node Shell NLDKGQ/Xin-Zheng-Lu, see page 915.

The command is:

```
add material No (or #) <material_number> type PlaneStressRebarMaterial
   with uniaxial_material # <.>
   angle = <degree> ;
```

where

- `No (or #)<material_number>` is a unique material integer number (does not have to be sequential, any unique positive integer number can be used).
- `with uniaxial_material #` is the material tag of predefined uniaxial steel material
- `angle` is the angle of uniaxial steel rebars. The angle is 0 along the direction formed by the first two nodes of a 4 Node Shell element.

#### 205.3.4.34 Modeling, Nodes: Adding Nodes

Nodes can be added to the finite element model.

The command is:

```
add node # <.> at (<L>,<L>,<L>) with <.> dofs;
```

For example:

```
add node No 1 at (1.0*m, 2.5*m, 3.33*m) with 3 dofs;
```

adds a node number 1 at coordinates $x = 1.0m$, $y = 2.5m$ and $z = 3.33m$ with 3 dofs. The nodes can be of 3dofs $[u_x, u_y, u_z]$, 4dofs $[]u_x, u_y, u_z, p]$ (u-p elements) , 6dofs $[u_x, u_y, u_z, r_x, r_y, r_z]$ (beams and shells) and 7 dofs $[u_x, u_y, u_z, p, U_x, U_y, U_z]$ (upU element) types. Description of output for nodes of different dof types can be found in section 206.6

#### 205.3.4.35 Modeling, Nodes: Adding Stochastic Nodes

Nodes can be added to the stochastic finite element model. Different from deterministic finite element analysis, nodes in stochastic FEM should specify the number of polynomial chaos (PC) terms for each physical nodal degree of freedom (dof).

The command is:

```
add node # <.> at (<L>,<L>,<L>) with <.> dofs polynomial_chaos_terms = <.>;
```

Where:

- `polynomial_chaos_terms` specifies the number of polynomial chaos terms for each physical nodal dof.

The stochastic nodes can also be added as:

```
add node # <.> at (<L>,<L>,<L>) with <.> dofs polynomial_chaos_terms as random field # <.>;
```

Which specifies the number of polynomial chaos terms for each physical nodal dof using the number of Hermite PC basis of a defined random field.

For example:

```
add node # 1 at (1.0*m, 0.0*m, 0.0*m) with 3 dofs polynomial_chaos_terms = 10;
```

Add a node # 1 at coordinates $x = 1.0m$, $y = 0.0m$ and $z = 0.0m$ with 3 physical dofs. For each physical dof, the number of terms for polynomial chaos expansion is 10.

```
add node # 1 at (1.0*m, 0.0*m, 0.0*m) with 3 dofs polynomial_chaos_terms as random field # 2;
```

Add a node # 1 at coordinates $x = 1.0m$, $y = 0.0m$ and $z = 0.0m$ with 3 physical dofs. For each physical dof, the number of terms for polynomial chaos expansion is equal to the number of PC basis of random field # 2.

#### 205.3.4.36 Modeling, Nodes: Define Nodal Physical Group

Physical Group for nodes can be defined as well.

The command is:

```
define physical_node_group "string";
```

For example:

```
define physical_node_group "my_new_node_group";
```

this would create a new physical_node_group with name "my_new_node_group".

Description of output for physical groups can be found in section 206.5.5

#### 205.3.4.37 Modeling, Nodes: Adding Nodes to Nodal Physical Group

Already created nodes can be added to the (any) physical_node_group.

The command is:

```
add nodes (<.>,<.>,...) to physical_node_group "string";
```

For example:

```
add nodes (1,2,3) to physical_node_group "my_new_node_group";
```

this would add node tag (1,2 and 3) to already created physical_node_group "my_new_node_group". Please note that the nodes (1,2 and 3) must be added to the model before they are added to the physical_node_group.

Description of output for physical groups can be found in section 206.5.5

#### 205.3.4.38 Modeling, Nodes: Removing Nodal Physical Group

Already defined node physical group physical_node_group can be removed.

The command is

```
remove physical_node_group "string";
```

For example:

```
remove physical_node_group "my_new_node_group";
```

this would delete the physical_node_group "my_new_node_group".

#### 205.3.4.39 Modeling, Nodes: Print Nodal Physical Group

Printing already defined nodal physical grouop physical_node_group is possible too.

The command is:

```
print physical_node_group "string";
```

For example:

```
print physical_node_group "my_new_node_group";
```

this would print the information about physical_node_group "my_new_node_group".

```
PHYSICAL_NODE_GROUP my_new_node_group
[1 2 3]
```

#### 205.3.4.40 Modeling, Nodes: Removing Nodes

Nodes can be removed from the finite element model, for example during excavation, removal of finite elements.

The command is:

```
remove node No (or #) <.>;
```

For example:

```
remove node # 1;
```

#### 205.3.4.41 Modeling, Nodes: Adding Nodal Mass, for 3DOFs and/or 6DOFs

Nodal mass can be added to nodes with 3 DOFs and/or 6DOFs. This is in addition to nodal mass that is obtained from finite elements.

The command for 3DOFs nodes (truss, solids, wall) is:

```
add mass to node # <.>
mx = <M>
my = <M>
mz = <M>;
```

Simularly, the command for 6DOFs nodes (beams and shells) is:

```
add mass to node # <.>
mx = <M>
my = <M>
mz = <M>
Imx = <M*L^2>
Imy = <M*L^2>
Imz = <M*L^2>;
```

#### 205.3.4.42 Modeling, Finite Element: Adding Finite Elements

The basic structure for adding any finite element is:

```
add element No (or #)
   type <finite_element_type>
   with nodes (<.>, ..., <.>)|
   {element dependent parameters};
```

Choices for `finite_element_type` are listed below

#### 205.3.4.43 Modeling, Finite Element: Define Finite Element Physical Group

Physical group for finite elements can be defined.

The command is:

```
1 define physical_element_group "string";
```

For example:

```
1 define physical_element_group "my_new_element_group";
```

this would create a new physical_element_group with name "my_new_element_group".

Description of output for physical groups can be found in Section 206.5.5.

#### 205.3.4.44 Modeling, Finite Element: Adding Elements to Physical Element Group

Finite elements, that already exist in the finite element domain, can be added to the physical_element_group.

The command is:

```
add elements (<.>,<.>,...) to physical_node_group "string";
```

For example:

```
add elements (1,2,3) to physical_node_group "my_new_node_group";
```

this would add elements with tags/numbers (1,2 and 3) to already created physical_element_group "my_new_element_group". Please note that the elements (1,2 and 3) must be added to the model before they are added to the physical_element_group.

Description of output for physical groups can be found in Section 206.5.5.

#### 205.3.4.45 Modeling, Finite Element: Remove Physical Finite Element Group

Finite elements can also be removed from the physical_element_group.

The command is:

```
remove physical_element_group "string";
```

For example:

```
remove physical_element_group "my_new_element_group";
```

this would delete the physical_element_group "my_new_element_group".

#### 205.3.4.46 Modeling, Finite Element: Print Physical Finite Element Group

Details of the physical_element_group can be printed.

The commands is:

```
print physical_element_group "string";
```

For example:

```
print physical_element_group "my_new_element_group";
```

this would print the information about physical_element_group "my_new_element_group".

```
PHYSICAL_ELEMENT_GROUP my_new_element_group
[1 2 3]
```

#### 205.3.4.47 Modeling, Finite Element: Remove Finite Element

Finite elements can be removed, for example if modeling requires excavation, removal of finite elements and nodes.

The command is:

```
remove element # <.>;
```

For example,

```
remove element # 1;
```

#### 205.3.4.48 Modeling, Finite Element: Truss Element

The command is:

```
add element No (or #) <element_number> type truss
           with nodes (n1, n2)
           use material No (or #) <material_number>
           section_area <section_area> [unit];
           mass_density <mass_density> [unit];
```

where

- `No (or #)<element_number>` is a unique element integer number (does not have to be sequential, any unique positive integer number can be used)
- `type truss` is the element type
- `with nodes (n1, n2)` are the 2 nodes (node numbers) defining this element
- `use material No (or #)` is the material number which makes up the element. Material has to be a uniaxial material, and it can be either elastic or one of the elastic-plastic materials defined for uniaxial behavior.
- `section_area` is the cross section area $[L^2]$

Description of output by this element can be found in Section 206.8.1. more on this finite element can be found in Section 102.6 on Page 126 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.49 Modeling, Finite Element: Kelvin-Voigt Element

The command is:

```
add element # <.> type Kelvin_Voigt
    with nodes (<.>, <.>)
    axial_stiffness = <F/L>
    axial_viscous_damping = <F/L*T>;
```

where

- `No (or #)<element_number>` is a unique element integer number, that does not have to be sequential, any unique positive integer number can be used
- `type Kelvin_Voigt` is the element type
- `with nodes (n1, n2)` are the 2 nodes (node numbers) defining this element
- `axial_stiffness` represents the stiffness in the axial direction, $[F/L]$
- `axial_viscous_damping` represents the viscosity, or viscous damping coefficient, in the axial direction, $[F/L * T]$

Note: Nodes defining this element cannot be at the same location, that is, this is a two node element and direction of this element is calculated from two distinct locations/coordinates of nodes.

205.3.4.50 Modeling, Finite Element: Inerter Element

The command is:

```
add element # <.> type Inerter
   with nodes (<.>, <.>)
   inertance = <M>;
```

where

- `No (or #)<element_number>` is a unique element integer number, that does not have to be sequential, any unique positive integer number can be used
- `type Inerter` is the element type
- `with nodes (n1, n2)` are the 2 nodes (node numbers) defining this element
- `inertance` represents the inertance in the axial direction, $[M]$

Note: Nodes defining this element cannot be at the same location, that is, this is a two node element and direction of this element is calculated from two distinct locations/coordinates of nodes.

#### 205.3.4.51 Modeling, Finite Element: Shear Beam Element

The command is:

```
add element # <.> type ShearBeam
  with nodes (<.>, <.>)
  cross_section = <l^2>
  use material # <.>;
```

where

- `No (or #)<element_number>` is a unique element integer number (does not have to be sequential, any unique positive integer number can be used)
- `with nodes (n1, n2)` are the 2 nodes (node numbers) defining this element. NOTE: element is supposed to be alligned along vertical, Z direction !!
- `use material No (or #)` is the material (LT-based material) number which makes up the element.
- `section_area` is the cross section area $[L^2]$

Description of output by this element can be found in Section 206.8.3. more on this finite element can be found in Section 102.9 on page 135 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).

#### 205.3.4.52 Modeling, Finite Element: Stochastic Shear Beam Element

Add stochastic shear beam element for stochastic finite element analysis.

```
add element # <.> type stochastic_shear_beam with nodes (<.>, <.>)
  use material # <.>
  triple product # <.>
  cross_section = <L^2>
  mass_density = <M/L^3>;
```

where

- `No (or #)<element_number>` is a unique element integer number (does not have to be sequential, any unique positive integer number can be used).
- `with nodes (n1, n2)` are the 2 nodes (node numbers) defining this element.
- `use material No (or #)` is the stochastic uniaxial material number that makes up the element.
- `triple product #` specifies the ID of the triple product, that would be used in the formation of elemental stochastic stiffness matrix. In stochastic finite element method (FEM), the first PC basis for this triple product should come from the PC representation of uncertain element stiffness. The second and third PC basis for this triple product should come from the PC representation of uncertain FEM system response, e.g., uncertain structural displacement.
- `section_area` is the cross section area $[L^2]$.
- `mass_density` is the density $[M/L^3]$.

For example:

```
add element # 1 type stochastic_shear_beam with nodes (1, 2) use material # 1 ↩
   triple product # 1 cross_section = 1*m^2 mass_density = 2000*kg/m^3;
```

Add a stochastic shear beam element # 1 with stochastic nodes 1 and 2 using stochastic uniaxial material # 1.

The cross section of the element is 1 $m^2$ and mass density is 2000 $kg/m^3$.

### 205.3.4.53 Modeling, Finite Element: Elastic Beam–Column Element

The command is:

```
add element # <.> type beam_elastic with nodes (<.>, <.>)
    cross_section = <L^2>
    elastic_modulus = <F/L^2>
    shear_modulus = <F/L^2>
    torsion_Jx = <length^4>
    bending_Iy = <length^4>
    bending_Iz = <length^4>
    mass_density = <M/L^3>
    xz_plane_vector = (<.>, <.>, <.> )
    joint_1_offset = (<L>, <L>, <L> )
    joint_2_offset = (<L>, <L>, <L> );
```

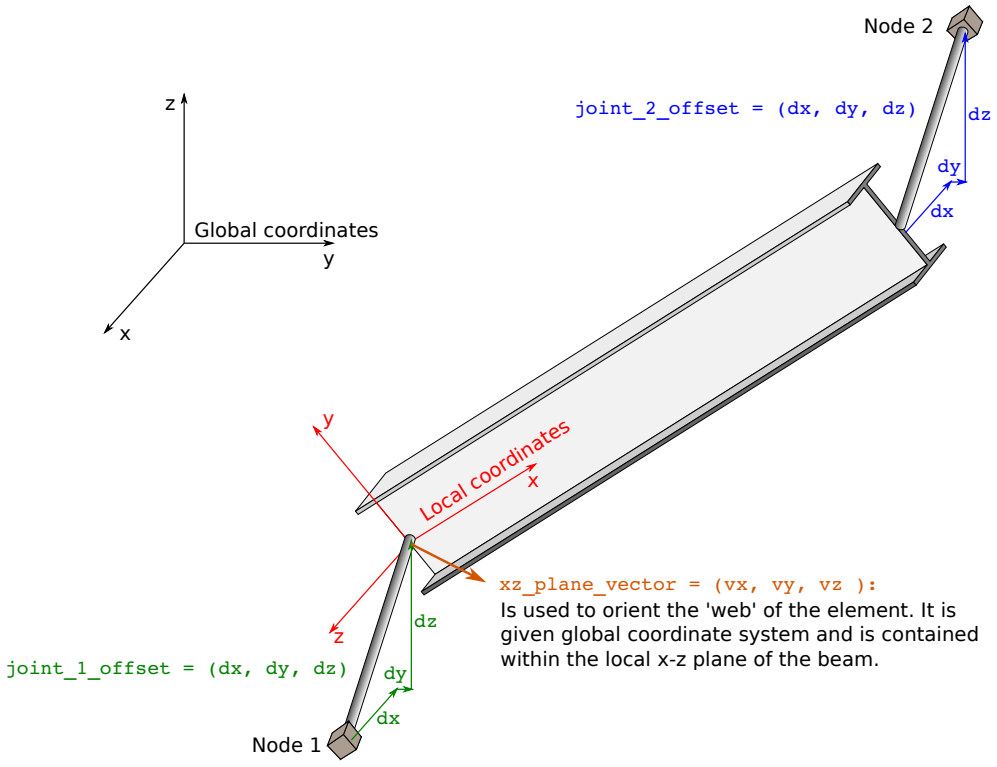



Figure 205.3: Beam Element, sketch of main geometric components.

where

- `No (or #)<element_number>` is a unique element integer number (does not have to be sequential, any unique positive integer number can be used)
- `type beam_elastic` is the element type

- `with nodes (n1, n2)` are the 2 nodes (node numbers) defining this element
- `cross_section` is the cross section area, $[L^2]$
- `elastic_modulus` elastic modulus of the material which makes up the beam, $[F/L^2]$
- `shear_modulus` shear modulus of the material which makes up the beam, $[F/L^2]$
- `torsion_Jx` cross section polar (torsional) moment of inertia, $[L^4]$
- `bending_Iy` cross section moment of inertia about local $y$ axis, $[L^4]$
- `bending_Iz` cross section moment of inertia about local $z$ axis, $[L^4]$
- `mass_density` mass per unit volume of the material, $[M/L^3]$
- `xz_plane_vector` a vector which defines the orientation of the local (beam coordinate system) xz plane in global coordinates. NOTE: Please make sure that your `xz_plane_vector` is a bit away from the actuall local $x$ axes, the axes that runs along the beam elelent, in order to prevent numerical problems that might appear when vector cross products are performed inside the program... It is suggested that your `xz_plane_vector` be closer to local z axes... See Figure 205.4 on Page 901 for more in depth explanation of `xz_plane_vector`.
- `joint_1_offset` vector defining the rigid offset between end of beam and connection node 1, $[L]$
- `joint_2_offset` vector defining the rigid offset between end of beam and connection node 2, $[L]$

Description of output by this element can be found in Section 206.8.4

more on this finite element can be found in Section 102.7 on Page 126 in Lecture Notes by Jeremić et al. (1989-2025) (Lecture Notes URL).