COMCO--98001806

Modern Industrial Simulation Tools:
Kernel-Level Integration of High Performance
Parallel Processing,
Object-Oriented Numerics, and Adaptive
Finite Element Analysis
(Final Report)

U.S. Department of Energy CRADA
Solicitation No. DE-SA-03-93DP400010
Project Period: July 16, 1993 – September 30, 1997

Prepared By:

Computational Mechanics Company, Inc. (COMCO)
7800 Shoal Creek Blvd., Suite 290E, Austin, Texas 78757

Principal Investigators: Manas K. Deb, Stephen R. Kennon

Team Members: C. Berry, M. Deb, S. Kennon, T. Liszka, J.
Meyering, J. Pickett, S. Sharma

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

Modern Industrial Simulation Tools:
Kernel-Level Integration of High Performance
Parallel Processing,
Object-Oriented Numerics, and Adaptive
Finite Element Analysis
(Final Report)

U.S. Department of Energy CRADA
Solicitation No. DE-SA-03-93DP400010
Project Period: July 16, 1993 – September 30, 1997

Prepared By:

Computational Mechanics Company, Inc. (COMCO)
7800 Shoal Creek Blvd., Suite 290E, Austin, Texas 78757

Principal Investigators: Manas K. Deb, Stephen R. Kennon

Team Members: C. Berry, M. Deb, S. Kennon, T. Liszka, J.
Meyering, J. Pickett, S. Sharma

DISCLAIMER

Portions of this document may be illegible electronic image products. Images are produced from the best available original document.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Contents

1	Executive Summary	1
1.1	General	1
1.2	Specific Goals	2
1.3	Achievements	2
1.4	Unfinished	3
I	Overview	4
2	Objectives	5
3	Accomplishments	11
3.1	Research	11
3.2	Technology	11
3.3	Development	12
3.4	Task Fulfillment	14
II	Research	15
4	Object-oriented Technology	16
4.1	Analysis	16
4.2	Abstractions	17
4.2.1	Topology	18
4.2.2	Finite Element Solvers	18
4.2.3	Applications Programming Interface	21
4.3	Numerics	21

5	Phlex-Linda	26
5.1	Motivation	26
5.2	Hierarchical Resource Model (HRM)	27
5.3	Master Slave Paradigm	30
5.4	Parallelization Techniques	30
5.4.1	Batching	30
5.4.2	Coloring	30
5.4.3	Domain Decomposition	31
5.5	An Example Using HRM	31
5.6	The Linda Model	32
5.7	Linda Implementation	33
6	Parallelization	34
6.1	Object Handles	34
6.1.1	Functional Requirements	35
6.1.2	Class Structure	36
6.2	Domain Decomposition	39
6.3	Input/Output (IO)	46
7	Finite Element Technology	51
7.1	Topology	51
7.1.1	Topology entities	51
7.1.2	Topology Glue	53
7.2	Green Elements	53
7.2.1	QUAD green element (4 to 9 nodes)	54
7.2.2	Hierarchical shape functions for p-elements	58
7.2.3	Extension to 3d	62
7.3	Adaptivity	64
7.3.1	Recursive algorithm	78
7.3.2	Details of member functions	79
7.3.3	Deadlocks in 3D Adaptation	82
III	PHLEX2000 Development	90
8	Overall Philosophy	91
8.1	Generic Kernel	92

8.2	Librarization	92
9	Applications Interface	94
9.1	The Coeff Class	95
9.2	Domain	98
9.2.1	Interface	100
9.2.2	Field, SubField, and Component Classes	100
9.2.3	Mesh	105
9.2.4	BC	107
9.3	PDE	109
9.4	SolutionProcess	110
9.4.1	Solver	112
9.5	Application Development	115
9.6	An Example: Heat Conduction Application	116
9.6.1	Application Class	118
9.6.2	PDEcoeff Class	121
9.6.3	BCs	124
10	Finite Element Solvers	134
10.1	General	134
10.1.1	Basic Solver Components	136
10.1.2	Finite Element Integration	136
10.1.3	Finite Element Assembly	139
10.2	Linear Solvers	140
10.3	NonLinear Solvers	144
10.3.1	Picard Nonlinear Solver	148
10.3.2	Newton Nonlinear Solver	149
10.4	Transient Solvers	150
10.4.1	The Time Marcher	150
10.4.2	Newmark- β Time Integration	151
10.4.3	Euler Time Integration	153
11	Miscellaneous	155
11.1	FieldVector	155
11.2	GlobalVector	157
12	A Prototype Application	160

List of Figures

2.1	Major Technical Goals	7
2.2	Team Technology Contribution	8
2.3	R&D and Technology Transfer	9
2.4	Teaming and Funding Relationships	10
3.1	PHLEX2000 Architecture	13
4.1	Class Diagram for Topo Entity	19
4.2	Class Diagram for Topo Function	20
4.3	Class Diagram for Solver Hierarchy	22
4.4	API Class Diagram	23
5.1	An Example of Hierarchical Resource Model (HRM) Application.	29
6.1	Object Handles: Process ownership	40
6.2	Object Handles: Floated ownership, local handle	41
6.3	Object Handles: Process ownership, non-local handle	42
6.4	Object Handles: Data Migration	43
6.5	Parallel IO Components	47
6.6	Persistent Database	48
6.7	Persistent Address	49
6.8	Parallel IO Mechanism	50
7.1	Hierarchical definition of a Quad	52
7.2	General configuration of green Quad element	55
7.3	Possible configurations of green Quad elements	56
7.4	Base Shape Functions	57
7.5	Modified Shape Functions	59

7.6	Base Hierarchical Shape Functions	60
7.7	Divided Edge Hierarchical Shape Functions	61
7.8	Various configurations of the face divided into two small ones. (a) No middle node; (b) and (d) Center node on the face, no edge middle node (illegal); (c) and (e) No center face node, edge has middle node (divided edge)	63
7.9	Quad Face Parametrization	64
7.10	Triangle Face Parametrization	65
7.11	Edge Refinement	68
7.12	Triangle Refinement Rules	70
7.13	Triangle Refinement Patterns	71
7.14	Triangle Refinement along one direction	72
7.15	Isotropic Triangle Refinement	73
7.16	Quad Refinement Rules	74
7.17	Quad Refinement Patterns	75
7.18	Quad Refinement along one direction	76
7.19	Isotropic Quad Refinement	77
7.20	3D Adaptation - Basic Schema	83
7.21	Decomposition of face and edge operations into refine and un- refine parts of the hex adaptation loop	84
7.22	A multidirectional refinement example	85
7.23	A multidirectional unrefinement example	86
7.24	Deadlock on the quad face of a Hex	88
7.25	Implementation of two hex adaptations (2-to-2 and 4-to-4) under deadlock	89
10.1	Layered Design of Finite Element Solvers	135
10.2	Global and Element-by-element Integrals	137
10.3	Finite Element Assemblers	141
10.4	Finite Element Matrices	142
10.5	Implementation of DirectSolver	143
10.6	Class Diagram for DenseSolver	145
12.1	Plate In Axial Tension - Problem Schematic	161
12.2	Initial Coarse Mesh	162
12.3	Transverse Displacements (Coarse Mesh)	163
12.4	Solution-Adaptive Mesh	164

12.5 Transverse Displacements (Adapted Mesh)	165
--	-----

Chapter 1

Executive Summary

1.1 General

A cooperative R&D effort between industry and the US government, this project, under the HPPP (High Performance Parallel Processing) initiative of the Dept. of Energy, started the investigations into parallel object-oriented (OO) numerics. The basic goal was to research and utilize the emerging technologies to create a physics-independent computational kernel for applications using adaptive finite element method. The industrial team included Computational Mechanics Co., Inc. (COMCO) of Austin, TX (as the primary contractor), Scientific Computing Associates, Inc. (SCA) of New Haven, CT, Texaco and CONVEX. Sandia National Laboratory (Albq., NM) was the technology partner from the government side. COMCO had the responsibility of the main kernel design and development, SCA had the lead in parallel solver technology and guidance on OO technologies was Sandia's main expertise in this venture. CONVEX and Texaco supported the partnership by hardware resource and application knowledge, respectively. As such, a minimum of fifty-percent cost-sharing was provided by the industry partnership during this project.

The project started in July of 1993. Due to budget reductions, the government funding was cut short which brought abrupt conclusion to the R&D activities of the project. However, a significant amount of targeted research was undertaken before the funding cut. In fact, a two-dimensional prototype computational kernel was built and tested. The current report describes the

R&D activities and provides some details about the prototype kernel and example applications.

1.2 Specific Goals

As mentioned above, the project set out to develop a physics independent finite element based computational kernel capable of parallel execution. For higher computational reliability and efficiency (defined as computation cost for a given solution accuracy), control of solution error using selective mesh adaptation is important. The target code was to have *hp*-adaptive capabilities i.e. be able handle simultaneous manipulation of *both* mesh size (*h*) and local interpolation orders (*p*). The new code, tentatively named *PHLEX2000* has been modeled after the existing adaptive finite element kernel of COMCO, called PHLEX, as far as the overall functionalities are concerned.

The project had three major goals which are enumerated as follows:

1. Research and development of a new OO, parallel, physics-independent, 2D and 3D *hp*-adaptive finite element kernel, PHLEX2000.
2. Development of a network parallel version of existing PHLEX kernel.
3. Migration of existing PHLEX applications to the new PHLEX2000 kernel.

The second goal was set in order to satisfy more immediate parallel processing needs of the CRADA partners as well to gain additional understanding of the challenges relevant to parallelization of an adaptive finite element kernel. It was decided that network parallelization will use Linda tools/libraries from SCA in order to handle data communication between compute processors.

1.3 Achievements

On the technical front, in spite of the early termination of the project, many of the research goals were accomplished. The research findings, the authors

believe, have direct applicability in design and implementation of general-purpose and complex computational kernels. While more details will be presented in the body of the report, here we summarize the key technologies researched/developed during this project:

1. Formal separation of topology from geometry and solution spaces.
2. Hierarchical definition of entities.
3. Methodologies for constraint-free adaptivity.
4. Adaptive algorithms amenable to parallelization.
5. Highly flexible and intuitive applications programming interface.
6. Development of finite element abstractions based on object-oriented design principles.
7. Hierarchical resource model for network parallelization of finite element codes.

These technologies were used to develop a prototype computational kernel and a few example finite element applications.

1.4 Unfinished

As mentioned earlier, due to premature termination of the government funding, certain activities could not be completed. In some cases, these include activities that were already in progress (e.g., items 2 and 3 in the following list). The major items that remain incomplete are:

1. Three dimensional adaptivity
2. Integration of MPP research into PHLEX2000 kernel
3. Numerical experiments with large scale problems

Although incomplete, the key issues and major observations made during the R&D activities on some of the above items are summarized in this report.

Part I

Overview

Chapter 2

Objectives

The overall objective of this project was to utilize a proper combination of emerging technologies in the area of object-oriented numerics and parallel computing in order to develop the core technical components of a 'physics-independent' *hp*-adaptive finite element computational kernel. In an effort to leverage existing knowledge and expertise, another key objective was to bring about cooperation between private industries and appropriate research divisions of US government (DOE) laboratories.

As dependence on scientific simulations increases for engineering design and decision making so does the need for reliable computational algorithms. Once a physical system has been adequately represented via a system of partial differential equations (PDEs) the next task is to discretize the computational domain and numerically solve the governing PDEs for such a discretization. Among many prevailing discretization methods, finite elements have gained industrial acceptance as one of the most flexible ones. However, as with any discretized method, errors due to finite size (i.e. h) or the local interpolation order (i.e. p) of the (finite) elements, some discretization error is unavoidable. The best that can be done is to be able to estimate such errors and have the ability to control them in some efficient manner. One approach that has been shown to be effective is that of *a posteriori* error estimation coupled with simultaneous adjustment of h and p in regions where error is high. COMCO's existing finite element library, PHLEX, incorporates such capabilities. Thus PHLEX functionalities were chosen to be a model for the next generation finite element kernel (tentatively named PHLEX2000).

For large scientific computational problems, parallel computing is viewed

as a key enabling strategy. Thus it was strongly desirable to investigate the parallel implementation aspects of PHLEX2000. However, for a computational kernel to be general-purpose, be mesh adaptive and parallelizable meant that highly complex code need to designed, developed, tested and maintained. This led to the consideration of employing some of the emerging trends in object-oriented numerics (OON). Sandia National Laboratories (SANDIA) laboratories had been engaged in R&D involving parallel and OO numerics. SANDIA was included in the CRADA so as to facilitate transfer of such technologies from DOE to private industry. Scientific Computing Associates (SCA) and Convex Computer Corporation (CONVEX) provided additional technology and support in the area of parallel computing.

When finished, some of the current applications (such as computational fluid dynamics, oil reservoir simulation, etc.) running on existing PHLEX, were to be transfered to PHLEX2000. The participation of Lockheed-Martin and Texaco Exploration and Production Division (TEXACO) was mainly to assist migration and testing of such application modules. In an effort to gain additional experience as well as to meet the needs of potential customers requiring parallel versions of these applications, one of the tasks was dedicated to the parallelization of existing PHLEX library. It was decided to use a tool such Linda (from SCA) to accomplish network parallelization of PHLEX.

Figure 2.1 shows the major technology objectives and their evolution. Figure 2.2 summarizes the key technology contributions of various team members of this CRADA. Technology transfer plan and the funding relationships are portrayed figures 2.3 and 2.4, respectively.

Figure 2.1: Major Technical Goals

Team Technology Contributions

Computational Mechanics Co. Inc.:

- h-p Adaptivity
- Error Estimation (Asymptotically Correct, Residual)
- Applications Expertise
 - o Reservoir Simulation (UniSim)
 - o Computational Fluid Dynamics (P3/CFD)
 - o Solid Mechanics (PHLEX/Solids)

Scientific Computing Associates, Inc.:

- Parallel Programming Language (C-, C++-, and FORTRAN-Linda)
- Parallel Programming Expertise
- Linear Solvers (MP-PCGPAK2)

Convex:

- Parallel Machines (Exemplar)
- Parallel Applications Development Expertise

Sandia National Laboratories:

- "Scaled Parallelism" - rebuke of Amdahl's Law
- Object-Oriented Numerics
- Applications Expertise

Texaco Corp.:

- Software Applications Need
- Testing, User Feedback
- Reservoir Simulation

Lockheed-Martin Corp.:

- Software Applications Need
- Testing, User Feedback
- CFD

Figure 2.2: Team Technology Contribution

Figure 2.3: R&D and Technology Transfer

Figure 2.4: Teaming and Funding Relationships

Chapter 3

Accomplishments

3.1 Research

An important component of this project was to research methods for applying new software development technology towards producing innovative solutions to industrial simulation problems. To this end, research was conducted in the following areas:

- Object Oriented Numerics
- Object Oriented Analysis
- High performance computing
 - Parallelization of computation on MPP systems
 - Parallel IO
 - Persistence of data
- h-p adaptive Finite Element Technology

3.2 Technology

Based on the above research, following technologies were developed and refined:

- h-p adaptive *green element*
- parallelizable mesh adaptivity
- object-oriented finite-element methodology

3.3 Development

A two dimensional, h-p adaptive finite element computational kernel was developed. This kernel, called PHLEX2000, is based on sound object oriented analysis and design principles. The architecture of the kernel is shown in figure 3.1.

Following are some of the highlights from the object oriented development undertaken during the project:

- A framework for application development enforced by abstract application interface.
- A highly flexible and intuitive applications development interface. The interface is built around powerful abstractions from the problem space like Domain, PDE, Solver, BC, etc.
- SolutionProcess abstraction which allows specification of complex solution sequences involving multiple domains and PDEs.
- A layered solver architecture that simplifies writing new solvers of various kinds. Each layer basically represents a different level of abstraction.
- A solver hierarchy based on encapsulation of distinct abstractions arrived at after thorough analysis. This class design allows maximum reuse of solver components in various mix and match solution sequences.
- Reusable finite element integration and assembly mechanism which allows efficient generation of systems of linear equations from the underlying discretization, even for complex cases.

Figure 3.1: PHLEX2000 Architecture

3.4 Task Fulfillment

Here is the list of tasks reproduced from the original solicitation along with our annotations showing what was completed and to what degree:

Task 1. Development of object-oriented, adaptive finite element kernel technology using existing team technology wherever possible.

This task was completed for two dimensional analyses and a kernel was produced. Three dimensional adaptivity was being developed at the time funding was stopped. Hence 3D work is not operational.

Task 2. Development of technology for distributed, object-oriented computing using existing team technology wherever possible.

Research was done on various aspects of parallelization and adaptive algorithms suitable for MPP were developed and used in the kernel.

Task 3. Merging of the above two phases to produce an object-oriented parallel, adaptive, FE kernel.

The distributed technology did not make it into the kernel because of funding cuts and modified statement of work.

Task 4. Network-parallel optimization of existing PHLEX kernel.

The current PHLEX kernel was made network parallel using Linda distributed programming paradigm and hierarchical resource model.

Task 5. Applications development and/or enhancement, using existing team technology where applicable.

Two applications from solid mechanics and fluid dynamics were built using the 2D PHLEX2000 kernel.

Task 6. Testing, validation and optimization of the techniques.

The kernel was tested and validated using the above mentioned applications. The kernel was optimized by using C++ techniques and object oriented design patterns.

Task 7. Large problem solutions.

This task was left out because of Task 3 remaining incomplete.

Part II

Research

Chapter 4

Object-oriented Technology

The object oriented paradigm provides a powerful, new way of expressing coding solutions to many problems. These solutions are based on modeling objects from the problem space and expressing their interactions through object behaviors as compared to the traditional techniques based on procedural programming and data modeling. The Object-Oriented-Design (OOD) model provides better flexibility, reuse and extensibility without (significant) modification.

4.1 Analysis

There are three major principles of object oriented design and analysis:

1. **The open/closed principle (Bertrand Meyer):** Software entities (classes, modules, etc) should be open for extension, but closed for modification. Designing modules whose behavior can be modified without making source code modifications to that module is what yields the benefits of OOD.
2. **The Liskov Substitution Principle:** Derived classes must be usable through the base class interface without the need for the user to know the difference. This rule is a logical extension of the open/closed principle. Consider function F that uses type T . Given S a subtype of T , F should be able to use objects of type S without knowing it. In fact, any subtype of T should be substitutable as an argument of F . If this

were not true, then F would have to have tests to determine which of the various subtypes it was using; and this will break the open/closed principle!

3. **Principle of Dependency Inversion:** Details should depend upon abstractions. Abstractions should not depend upon details. This is the direction that all dependencies should take in an object oriented program. All high level functions and data structures should be utterly independent of low level functions and data structures.

In the current project, we have tried to adhere to these principles as best as possible.

4.2 Abstractions

The hardest part of object oriented development is that of identifying the set of abstractions which would give maximum benefit in terms of flexibility and reuse. Intimate familiarity of the COMCO team with the mechanics of complex hp-adaptive finite element technology and that of the SANDIA team with various parallel finite element applications helped us in analyzing and designing abstractions that were highly functional and concise.

The various OOD efforts that were undertaken during this project can be broken down into three main categories:

1. Topology - includes topo entities, topo functions, and mesh adaptivity
2. Finite Element Solvers - includes numerical integration, assembly of systems of equation and their solution
3. Applications Programming Interface - PDE, domain, bcs, etc.
4. Parallelization - includes IO, distribution, persistence, and domain decomposition.

We explain these efforts in the following subsections. Note that all class diagrams are presented in Booch Notation ([13]).

4.2.1 Topology

Recognition of topology as an independent concept is one of the major accomplishments of this project. Separation of topology and encapsulation of all functionality related to it has provided a powerful and flexible means to defining complex geometric and solution spaces. In traditional finite element codes, both geometry and solution carried as parts of *topology*. Adding complexity to either of these two typically introduces a ripple effect throughout the code associated with any of these spaces and ultimately making it hard to even understand their complex interactions and to maintain/enhance the code reliably.

Once the abstraction of topology was extracted, it was easy to come up with the orthogonal concept of topological functions which represent variation of some space. These two concepts can be used together to define geometric spaces or solution spaces of arbitrary complexity. Figures 4.1 and 4.2 show the class diagrams of the hierarchies for both these concepts. Further details on these concepts are presented in chapter 7.

For example, it is possible to define a domain consisting of various topological entities (quads, triangles, hexes) with geometric spaces of one type and solution spaces of another type – containing both continuous as well as piecewise variations.

4.2.2 Finite Element Solvers

At the core of a typical finite element simulator is some kind of an *equation solver*. In PHLEX2000, the term *solver* is being used to mean something much more general than just a linear equations solver. Solvers in PHLEX2000 take on all responsibilities of finite element integration, assembly and the final solution.

We know from our experiences with various applications that real simulations (where computational efficiency is a real concern) can require complex combinations of different linear equation solvers and complicated sequences of solution algorithms. For example, a computational fluid dynamics (CFD) problem may require solving multiple nonlinear systems using iterative linear solvers with preconditioners, all inside a time integrator.

A thorough analysis of a general finite-element solution process lead us to the solver hierarchy shown in figure 4.3. While analyzing this hierarchy, we

Figure 4.1: Class Diagram for Topo Entity

Figure 4.2: Class Diagram for Topo Function

had to tackle the classic OOD question of using the right kind of relationship out of the two – *is-a* or *has-a*, many times. While the first represents the *subtype relation*, the second represents *containment*. For example, a *NonLinearSolver* is a *MatrixEqnSolver* but it contains (i.e., *has-a*) an instance of a *LinearSolver*. A *TransientMatrixSolver* is a *TransientEqnSolver* and it may contain either a *LinearSolver* or a *NonLinearSolver*.

4.2.3 Applications Programming Interface

In order to provide a simple and intuitive interface to the application developers, we developed abstractions which correspond closely to the concepts in the problem domain (i.e., engineering simulations). In a typical engineering problem requiring a numerical solution of one or more partial differential equations (PDEs), one needs to define the particulars of the equation itself (i.e. the details of the coefficients corresponding to the various terms in the *PDE*), define the computational domain (*Domain*) over which the PDE is applicable, prescribe the conditions on the solution (and some of the derivatives, as appropriate) on the domain boundary (*BC*). Since the derivatives will be approximated numerically, one also needs to specify a valid discretization of the domain (i.e. the *Mesh*). Keeping with these natural concepts, in the current design, classes such as *Domain*, *Mesh*, *Interface*, *PDE*, *BC*, etc. were defined. Since we wanted to give the application developer high flexibility in terms of defining his problem, most of the interface consisted of *abstract classes*. Figure 4.4 gives a brief look at some of these classes making up the interface. The PHLEX2000 kernel works off of these abstract classes and applications have the ability to derive from these interface classes and customize for their implementation. Here full use is made of the polymorphism provided by C++ virtual function mechanism. Chapter 9 describes in detail the classes involved in the interface.

4.3 Numerics

One of the goals of the project was to provide powerful and efficient Object-Oriented-Numerics (OON). There is a general feeling among various scientific and engineering developers that one has to sacrifice speed and efficiency when developing using an OO technology. It is true that some extra price may have

Figure 4.3: Class Diagram for Solver Hierarchy

Figure 4.4: API Class Diagram

to be paid if whenever robust generalizations of any methodology is sought. We do however feel that this downgrading of efficiency can be minimized by incorporating the following considerations:

1. A 'lean' design requiring minimal overhead for workflow/data organization
2. A good understanding of the subtleties of the particular OO language being used (in this case C++)
3. Selection of proper tools for numerical work

The first factor comes into play due to generation of *temporaries* under various circumstances involving argument passing by value, implicit type conversion, etc. The temporaries can be eliminated in most circumstances by careful consideration of the code. For example, when invoking methods on an object, one should try to pass arguments by reference and not by value (in most cases). Similarly, one should avoid instances where compiler might do an implicit type conversion by invoking user defined default constructors or copy constructors.

Using the right tools for numerical work is very important for efficiency. We chose to incorporate the following off-the-shelf packages:

1. Standard Template Library (STL) from ObjectSpace
2. Lapack++

The standard template library ([12]) provides a fairly extensive collection of commonly used data structures and algorithms that are needed for any good size application. Instead of writing these foundation classes from scratch, using this commercial, supported library helped us concentrate on our higher level work instead of getting bogged down into the details of very low-level data structures.

Lapack++ ([11]) is the object oriented version of the well known Lapack suite of numerical algorithms. The core of the library is still the Fortran coded Lapack, with a C++ layer wrapped around it. This organization retains the computational efficiency of the original library while providing a means of interacting with an object-oriented code.

In addition to the above, we also used commercial products such *Purify* and *Quantify* ([10]) from Pure Software Inc.). *Purify* is a powerful tool for identifying and eliminating common errors, including run time errors and memory leaks. *Quantify*, on the other hand, helps in determining the computational bottlenecks by giving detailed statistics from the run-time performance of the code.

Chapter 5

Phlex-Linda

One of the major task of this project was the development of a *network parallel* version of the existing PHLEX kernel using the *Linda programming model*. The effort towards this goal consisted of two main components:

- Development of a generic framework for network parallelization, and
- Actual implementation using Linda

In the following we first describe the development of this framework followed by some discussion of the Linda implementation.

5.1 Motivation

Some of the key requirements of a portable and efficient parallel programs may be listed as (cf. James Browne in [1]):

- parallel semantics must be readily accessible to the translator
- parallel semantics must be architecture independent
- parallel semantics must be at a higher level of abstraction than the execution environments

In order to motivate the design of a programming model that will systematically capture the best of abilities of all the participating computer

systems in a heterogeneous network, we first considered some related work in standard parallel programming paradigms. For low level encapsulations work done in [2] or use of 'abstract data types' (ADTs) [3] were studied. Some of the basic philosophies regarding high level encapsulations as found in [4, 5, 6, 7] were also considered. It was clear from our research that, a successful programming model/tool for heterogeneous networks needs to be:

- network topology independent at the application programming level
- architecture adaptive at the execution level

The overall scheme must be *scalable*. In addition, in our particular case, the implementation should also be amenable to easy use by application developers utilizing the PHLEX kernel. This led to the design of a *Hierarchical Resource Model (HRM)* which addresses all these issues. We describe this model in some more detail in the following section.

5.2 Hierarchical Resource Model (HRM)

Most of the widely used parallel systems (MIMD type) can be classified under these categories (based on their architectural characteristics):

1. Heterogeneous workstation clusters.
2. Shared memory (hardware, not simulated, e.g., SGI Power Challenge).
3. Hypernode based machines (e.g., Convex Exemplar).
4. General purpose MIMD machines such IBM SP2s, CRAY T3D/T3E
5. Special MPP machines Intel Paragon, Hypercube.

Of these, the hypernode based architectures, wherein each distributed node itself consists of a number of processors with shared memory access, seems to us to be the most generic and scalable. The two levels of memory latency provided by such an architecture will likely be the model of the future – the first level provided by multiple CPU chips placed on the same die and the second level provided by connections between such assemblies. The low

latency shared memory of a hypernode can effectively support fine-grained parallelism available within parts of a task that is itself distributed over the hypernodes to make use of the coarse grained parallelism available in it.

The above mentioned system organization may be viewed as a superset of other classes of parallel/distributed architectures. For efficiency, it is required that our programming model match the architectural model at some suitable level. The model chosen here will be able to make use of much more generic organization of hardware. This model, which will be called the *Hierarchical Resource Model* (HRM), for resolution of the computational tasks, considers the individual computing nodes as available computing resources and does not make any assumptions about their nature.

In the HRM, a given domain is mapped on to a given *resource*. The resource (object) makes decisions regarding what kind of strategy to use to make best use of the compute power. Consider the case where the resource is a heterogeneous cluster. The resource may decide to use a domain decomposition strategy with message passing using PVM/MPI to exploit parallelism. The subdomains are mapped to the nodes of the network, without making any assumptions about the nature of the individual nodes themselves. A node of the network may be a shared memory four processor Power Challenge. Since the resource at that node knows what kind of a resource it is, it can choose, for example, to use a *coloring scheme* for the equations belonging to the subdomain to exploit fine grained parallelism. Numerous other hierarchical combinations are possible (figure 5.1).

Our model thus allows us to arm ourselves with various generic and specific parallelization/optimization techniques without losing portability or scalability. These techniques can become a part of a library which could be modularly modified or replaced to take advantage of any new multiprocessor architectures or software advancements. Some of the most generic techniques which will be applicable to the mentioned classes of parallel systems will be *batching* and *coloring* and often a combination of the two. These strategies have proven to be critical in obtaining good cache and vector register performances.

CP1 - Communication Primitive 1 (e.g. MPI)
 CP2 - Communication Primitive 2 (e.g. Linda)

Figure 5.1: An Example of Hierarchical Resource Model (HRM) Application.

5.3 Master Slave Paradigm

A Master-Slave paradigm is inherent in the HRM model. Whenever a resource consists of more than one processor, there needs to be a master process. *It should be noted that only a process is designated as a master and not a processor.* Though there is no restriction that the master be a processor, however, for reasons of scalability, this will usually be the case.

In HRM, any participating process is capable of working in both the master and slave modes. In the slave mode, it simply performs the given compute tasks. In the master mode, it decomposes the problem appropriately for its slaves, hands down the current tasks and is required to correctly synchronize all parallel operations between its slaves and its masters. In general, there will be multiple master-slave levels. The process at the top of the hierarchy is a true master while those at the lowest levels are true slaves.

This dual capability of a process to switch between Master and Slave modes, gives the model added flexibility. Any process belonging to intermediate layers of the hierarchy can switch communication modes depending on to whom it is talking. It can, for example, use MPI to communicate with its neighbors and its parent, and use Linda to communicate with its slaves.

5.4 Parallelization Techniques

5.4.1 Batching

It is well known that performance of a code on any machine can be substantially enhanced when cache is usefully employed (i.e., cache misses are minimal). In PHLEX, all element computations are performed by employing a *batching* strategy. A batch is a set of elements whose data can fit into the cache. All element routines are oriented to work on batches of elements. By appropriately selecting a batch length which closely matches the optimal vector length/cache size of the machine, performance can be significantly improved.

5.4.2 Coloring

Medium to fine grain parallelism can be obtained in finite element applications by coloring the elements of a domain. All elements belonging to a color

do not share any information and hence can be processed simultaneously. Coloring mechanism can be fruitfully employed in two situations — to take advantage of vector-parallel machines like Crays and to parallelize on shared memory multiprocessors.

5.4.3 Domain Decomposition

In a distributed environment, domain decomposition (DD) is the natural approach for achieving parallel scalability for finite element applications. The problem domain is physically distributed among processors so as to reduce the amount of communication (due to data locality). This approach fits naturally into the HRM model (a hierarchical decomposition of domains can be mapped directly on to a hierarchy of resources). Coloring and batching can then be used within the nodes of the distributed network to further elevate performance. There are commercial tools available for performing optimal domain decomposition to maximize data locality and minimize interface communication, such as CHACO [8] and TOP/DOMDEC [9].

5.5 An Example Using HRM

Following is a conceptual pseudo-code for Conjugate Gradient method — an important iterative linear solver, written specifically for parallel finite element applications.

```
p = r = residual = f - K.x
loop iter = 1,...
  u = K.p
  be = p.u
  b = GLOBAL_SCALAR_SUM(be)
  x = x + b.p
  r = r - b.u
  de = Ke.r
  d = GLOBAL_VECTOR_SUM(de)
  rho = r.d
  eta = GLOBAL_SCALAR_SUM(rho)
  if (eta < gamma) exit
end loop
```

The important point to note here is the occurrence of two parallel operations denoted by GLOBAL_SCALAR_SUM and GLOBAL_VECTOR_SUM. These represent, respectively, a sum of similar scalar values from all the subdomains and a similar sum for a vector. The latter gets simplified in FE applications to be just a correction of values at neighboring subdomain interfaces.

Within the framework of HRM, the GLOBAL_SCALAR_SUM may be represented as follows (in C code) :

```
double GLOBAL_SCALAR_SUM (value)
{
    if (hasServers) {
        recv_from_servers(&value);
    }
    if (hasMaster) {
        send_to_master(&value);
        recv_from_master(&value);
    }
    if (hasServers) {
        send_to_servers(&value);
    }
    return value;
}
```

The dual modality of an HRM process described earlier is clearly demonstrated here. The functionality is obvious from the code. Similar functionality will be needed for GLOBAL_VECTOR_SUM.

5.6 The Linda Model

Linda consists of a small number of powerful operations that may be integrated into a conventional base language, yielding a dialect that supports parallel programming. Thus, for example, C with the addition of the Linda operations becomes the parallel programming language C-Linda.

The Linda model is a *memory* model. Linda has an associative logically-shared memory called *tuple space* that consists of a collection of logically-ordered sets of data called *tuples*. There are two kinds of tuples resident in

tuple space (TS): *process tuples*, which are under active evaluation, and *data tuples*, which are passive. Process tuples (which are all executing simultaneously) accomplish work by generating, using, and consuming data tuples. Once a process tuple has finished executing, it turns into a data tuple (its result), indistinguishable from other data tuples.

There are four basic tuple-space operations: *out*, *eval*, *in* and *rd*. *out*(*t*) causes the tuple *t* to be evaluated by the executing process and added to TS; the executing process continues immediately. *eval*(*t*) is the same as *out*(*t*), except that *t* is evaluated after rather than before it enters tuple space; *eval* implicitly creates new processes to perform the evaluation. *in* and *rd* both retrieve data from TS. *in*(*s*) causes some tuple *t* that associatively matches the *template s* to be withdrawn from TS. *rd*(*s*) is identical to *in*(*s*), except that the matching tuple *t* remains in TS.

Tuple space is an associative memory. Tuples have no addresses; they are selected by *in* or *rd* on the basis of any combination of their field values. Thus the five-element tuple (*A, B, C, D, E*) may be referenced as "the five element tuple whose first element is *A*," or as "the five-element tuple whose second element is *B* and fifth is *E*" or by any other combination of element values. To read a tuple using the first description, one would write *rd*(*A*, ?*w*, ?*x*, ?*y*, ?*z*). In this case *A* is an actual parameter to be matched against, and *w* through *z* are formals whose values will be filled in from the matched tuple. To read it using the second and the fifth description one would write *rd*(?*v*, *B*, ?*x*, ?*y*, *E*). Associative matching is in fact more general than this: formal parameters (or "wild cards") may appear in tuples as well as in templates, and matching is sensitive to the types as well as the values of tuple fields.

5.7 Linda Implementation

The HRM model described previously was incorporated in PHLEX and implemented using the C and Fortran Linda programming languages. A domain decomposition approach was used to divide the initial domain into as many subdomains as the number of workstations desired for a specific execution on the network. Farhat's decomposition algorithm was used for this purpose. The *save-and-restart* mechanisms available in PHLEX were modified and used for packing and unpacking of subdomain data.

Chapter 6

Parallelization

Active-Message/Shared-Memory concept is used for the following activities:

1. Initialization (global objects, etc.)
2. Start/Restart (i.e. reading in the computational mesh)
3. Adaptation
4. Migration (tactical and strategic)

Alias approach is used for all (explicit) number-crunching activities like matrix-vector product.

6.1 Object Handles

For parallelization in a generic distributed environment, the concept of *object handles* was researched. The basic idea is to develop a kind of a *smart pointer* or *handle* for all those objects which may need to be distributed. If such objects are always accessed through these handles, it should be possible to access local and remote objects transparently. During the research phase of PHLEX2000, such a concept was refined and implemented. In the following we present a summary description of these efforts.

6.1.1 Functional Requirements

The design criteria for developing reference handling can be itemized as follows:

- First and foremost, should be robust, reliable and concise.
- Data referencing should be transparent to coherency ("freshness").
⇒ Floating ownership
- Locality of reference must be transparent to all classes 'except' those responsible for it.
⇒ Handles
- Should be transparent to Datatype, but provide type safety.
⇒ Templates
⇒ All objects inherit from a common base class — the *Phlex_Object*.
- No penalty for Serial machines.
⇒ Handles not handle pointers.
⇒ Single Dereference using CPP and in-lining on Serial machines.
- Data should be available on demand, and user code should be straight C++.
⇒ Overloading the dereference operators ($\rightarrow$, $*$)
- No pointer "swizzling"
⇒ Exchange addresses between processors — the *GlobalAddress*.
- Should not rely upon system-dependent services, but be able to exploit them, if needed.
⇒ No reliance upon *mprotect*, signals, or hardware faulting.
- Should provide garbage collection services (automatic memory management).
⇒ Reference counting.
- Should easily facilitate the *forced migration of data* between processors for load balancing, *in an asynchronous mode*.

- Domain Decomposition should need to migrate only the data-containers e.g., "elements", and all other associates data should move "naturally".
- Should be able to easily overlay schemes for shared ownership.
 $\Rightarrow$ Handles point to data directly embedded with reference information.
 e.g., Aliasing schemes, Protection flags (RO, RW), Merge operator.
- Should be able to easily overlay schemes for improving efficiency.
 e.g., Paging schemes, buffering messages.
 CPP Generated Serial versions embedded in protected code areas, etc.

The above design criteria has following design implications for the overall project:

- No copy constructors or destructors for general classes, unless one takes great care to explicitly invoke copy constructors or destructors for any included data, particularly Handles.
- In general, Handles will always be used in place of pointers. (e.g., Arrays of Handles, etc.)

6.1.2 Class Structure

```
template < class T >
Phlex_GlobalAddress {
private :
    Phlex_ProcessorID proc_id_;
    T * address_;
};

template < class T >
Phlex_Handle {
public:
    Phlex_Handle ();                                // Constructor
    ~Phlex_Handle ();                                // Destructor
    Phlex_Handle (const Phlex_Handle& ht);           // Copy Constructor
    Phlex_Handle& operator= (const Phlex_Handle& ht); // Assignment operator
```

```

Phlex_Handle& operator= (T* tp);           // Assignment operator

void bind (Phlex_ProcessorID pid, T *tp); // Bind to a new piece of data
void nullify (void);                     // Bind to NULL

T& operator* ();           // 'dereference' a handle pointer
T* operator-> (void);       // 'dereference' a handle

size_t size (void);        // sizeof T
int handle_is_local (void);

// Migration/copy etc. routines
void migrate (Phlex_ProcessorID to_proc_id);
void process_handles (void);

void copy (Phlex_ProcessorID to_proc_id);
void send (Phlex_ProcessorID to_proc_id, SendType send_type);
Phlex_Handle * recv (int message_tag, int remote_proc, int n_bytes);

// Message functions
void poll (void);
void poll_and_sync (void);
void process_non_sync_message (int message_tag,
int remote_proc, int n_bytes);
void synchronize (void);

private :
    GlobalAddress< T > ga_ ;
};

class Phlex_Object {
public:
    int rep_is_local (void);
    void get_data (void * remote_address,
int remote_proc_id,
void * local_address,
size_t Tsize,

```

```

RefCountAction handle_ref_count_action,
                Phlex_HandleTypeID handle_id );

Phlex_Object * process_request (int message_tag,
int remote_proc,
int n_bytes);
private:
    Phlex_GlobalAddress<Phlex_Object> global_address_;

    int handle_ref_count_; // Number of references by handles
    int rep_ref_count_;    // Number of forwarding references by Phlex_Objects
};

class Elem : public Phlex_Object {
    .....
};

class Phlex_MessagePasser {
public:
    static Phlex_MessagePasser mp;

    static int enrolled_;
    static int initialized_;

    static ofstream fs_;
    static char      buf_[100];
    static int       my_proc_id_;
    static int       my_tid_;
    static int       tids_[MAX_NPROC];
    static int       n_procs_;

    // The first function that must be called on entry to any process
    // It returns my_proc_id
    static int  enroll (void);

    // Send a single buffer
    static void send      (int to_proc, void *data, int size, int msgtag);

```

```

// Receive of a single buffer
static void recv      (int msgtag, int from_proc, void *buf, int n_bytes);

// Following are for multiple calls to pack the buffer before
// sending.
static void init_buffer (void);
static void pack_buffer (void *data, int size);
static void send_buffer (int to_proc, int msgtag);
static int  recv_buffer (int msgtag, int from_proc);
static void unpack_buffer (void *data, int size);

static void broadcast  (int num_proc, int *procs,
                        void *data, int size, int msgtag);

static int  probe_any  (int *msgtag, int *proc_id_l, int *n_bytes);
static int  probe      (int *msgtag, int *proc_id_l, int *n_bytes,
                        const int num_msg_look_for = 0,
                        const int *msg_look_for_array = NULL,
                        const int num_proc_look_for = 0,
                        const int *proc_look_for_array = NULL);

static void init      (int argc, char **argv, int n_procs_in,
                        int debug_mode = 0,
                        const char **hosts = (const char **)0);
static void fini      (void);
};

```

6.2 Domain Decomposition

The functional specification for a 'static' domain decomposition (DD) tool for parallel computation may be stated as: given a set of finite elements, decompose the domain into sub-sets that are assigned to processors of a parallel computer such that:

Single Process Ownership ► handle is local
 ► data is local

Figure 6.1: Object Handles: Process ownership

Figure 6.2: Object Handles: Floated ownership, local handle

Figure 6.3: Object Handles: Process ownership, non-local handle

Step 1. Migrate()

Step 2. Process_handles()

Migration of Data

Figure 6.4: Object Handles: Data Migration

1. two adjacent finite elements are either
 - (a) on the same processor, or
 - (b) on neighboring processors (defined as 'close-by'; in reality, we weaken this restriction a little to allow neighboring processors to mean any that can communicate)
2. the computational load is 'balanced' across the processors
3. the amount of communications between processors is minimized

The above static domain decomposition problem (SDD) is a subset of the more general 'dynamic DD problem' (DDD). The DD problem may be stated as: given an initial DD, improve it to minimize load imbalance such that communications cost is also minimized. This can be seen by the fact that any SDD can be achieved by solving a DDD where the initial guess is a trivial DD (e.g. all elements are assigned to the same processor). The DDD problem (and thus the SDD problem) should be solvable with scalable parallel algorithms that can be run on parallel computers.

The DD problem involves both the assignment of elements as well as their constitutive parts (i.e. all the TopoEntities) to processors. We foresee this occurring by moving elements (and their children as may be produced due to mesh adaptivity, sub-topo-entities, etc.) as single blocks between processors.

In the following we present the public interface of the DomainDecomposer class developed keeping in mind the above requirements:

```
class Phlex_DomainDecomposer {
public:
    enum TopoType
    {
        T_VERTEX = Phlex_TopoEntity::T_VERTEX,
        T_EDGE = Phlex_TopoEntity::T_EDGE,
        T_FACE = Phlex_TopoEntity::T_FACE,
        T_VOLUME = Phlex_TopoEntity::T_VOLUME,
        T_END
    };
    typedef int Connection[4][4];
```

```
Phlex_DomainDecomposer (int num_procs_in, Phlex_MeshDB *mesh_db_in);

void read_initial_mesh_description ();    // parallel I/O here
void initialize_DD ();

enum StaticDDType {
    TRIVIAL,
    GREEDY,
    METIS_P,
    METIS_K
};

// do the initial domain decomposition
void initial_DD(StaticDDType static_DD_type);

// do a single dynamic update of the DD
void do_one_dynamic_DD_update (void);

enum MoveType {
    ENTITY_ALONE      = 1 << 0,
    ENTITY_PLUS_SUB_ENTITIES = 1 << 1,
    ENTITY_PLUS_CHILDREN   = 1 << 2
};

void mark_for_move (Phlex_TopoEntity *element,
    int from_proc, int to_proc,
    MoveType move_type);

void do_moves (int from_proc, int to_proc);

double load ();                //returns load on this processor's domain
double comm_cost_estimate (); //returns communication cost estimate

// defines relative weights of comm. vs. load costs
void set_cost_function (VFP cost_func);

enum OptimizationMethod {
```

```
    DIFFUSION,  
    CG,  
    SIM_ANNEALING  
};  
  
void set_optimization_method (OptimizationMethod m);  
  
void set_iteration_params (int num_iter, double tolerance);  
int check_convergence ();  
  
void mesh_to_graph (int level, int te_type_i, int te_type_j, int val);  
int mesh_to_graph (int level, int te_type_i, int te_type_j);  
  
int num_procs ();  
};
```

6.3 Input/Output (IO)

Input/output is handled through the IOPort mechanism. An IOPort is a logical place on the MP machine that can accept I/O. There may be several processors that are assigned to a given IOPort. IOPort's define how information comes into and out of the machine. Mappers are needed to map from changed IOPorts (ie. on restart/save).

If VMID (VirtualMachineID) is the same, then IOPorts are used as is; else, objects are remapped as they come in based on some segmentation/decomposition. A basic mapper takes a given PermanentID and assigns it to a new processor, independent of the actual domain decomposition.

File

- Unix byte-stream
- Methods:
 - lseek, read, write
- open as FILE* (fopen) or as file descriptor (open)

Database

- logical grouping of several files viewed as a single file addressable by aPersistentAddress
- methods:
 - lseek(PersistentAddress offset, SeekType whence),
 - read, write
- must perform an lseek before a read/write

IOPort:

- logical place to perform I/O
- associated with a processor that is the 'owner'
- has a list of processors that are attached to this IOPort
- used by ParallelIO

ParallelIO:

- provides for parallel lseek, read and write
- methods
 - parallel_lseek, parallel_read, parallel_write
 - parallel_lseek_read, parallel_lseek_write

PersistentAddress:

- an offset into a Database that is allowed to be larger than a regular offset (64 vs. 32bits)

Figure 6.5: Parallel IO Components

Database

- logical grouping of several files viewed as a single file addressable by aPersistentAddress
- methods:
 - lseek(PersistentAddress offset, SeekType whence),
 - read, write
- must perform an lseek before a read/write

Figure 6.6: Persistent Database

Figure 6.7: Persistent Address

IOPort:

- logical place to perform I/O
- associated with a processor that is the 'owner'
- has a list of processors that are attached to this IOPort
- used by ParallelIO

ParallelIO:

- provides for parallel lseek, read and write
- methods
 - `parallel_lseek`, `parallel_read`, `parallel_write`
 - `parallel_lseek_read`, `parallel_lseek_write`

Figure 6.8: Parallel IO Mechanism

Chapter 7

Finite Element Technology

7.1 Topology

One of the major accomplishments of the current project has been the recognition of topology as an independent concept in its own right. Separation of topology and encapsulation of all functionality related to it has provided a powerful and flexible means for defining complex geometric and solution spaces. Earlier, adding complexity to either of these two spaces used to introduce a ripple effect throughout the code associated with any of these spaces ultimately making it hard to even understand their complex interactions.

In Phlex2000, topology is represented as a combination of two distinct concepts – topological entities (TE) and topological functions (TF). Topological entities concern themselves with the shapes and patterns while the topological functions evaluate prescribed relations in the context of the topological entities. We describe these concepts in the following subsections.

7.1.1 Topology entities

Topological entities (topoentity or TE) in PHLEX2000 are defined in a hierarchical fashion. Each independent topoentity consists of a collection of lower level subentities. For example, a *QuadFace* topoentity consists of a face entity and two kinds of subentities – edges and vertices, four of each. Figure 7.1 shows this description pictorially.

Figure 7.1: Hierarchical definition of a Quad

7.1.2 Topology Glue

Due to the recursive definition of topological entities (TEs), the topological function at any point within a topoentity is computed as the sum of incremental contributions from all the sub topo-entities suitably blended and added to its own blended contribution.

For example, consider that the current TE is a Volume, xyz is $\{xi, eta, zeta\}$ and the Volume is asked to evaluate a function at xyz . Then we consider each of the sub-topo-entity's in turn (i.e. faces, edges and vertices), map xyz to each sub-topo-entity's local coordinate, evaluate its incremental value, blend using the blending functions, then sum it with this Volume's own increment.

7.2 Green Elements

In PHLEX2000 we have used a different scheme to handle h-adaptivity than that used in the existing PHLEX kernel. The existing PHLEX technology uses a 'constraint based scheme', wherein extra nodes introduced along sides of unbroken elements are constrained to have values dictated by the 'true' nodes at the corners of the elements. This approach has several disadvantages – the constrained nodes do not resolve the solution since they are not true degrees of freedom, though they do add to the size of the equation system to be solved; these constraints can form long chains and introduce sequentiality in both adaptivity and solution processes thus inhibiting parallelism.

PHLEX2000 uses an approach based on *Green elements* or transition elements which does away with the need to introduce constraints. These elements are based on the extension of the 'Gupta' scheme.

The basic idea is to define shape functions for each node inside the element using piecewise continuity only, i.e. shape functions may have $C1$ discontinuities inside the elements (not only at element boundaries). This results in the need to pseudo refine each green element into several small sub-elements for the purpose of proper integration. Inside each sub-element all shape functions are fully continuous (C infinity) and standard Gauss integration rules can be used. Thus sub-elements are sub-integration-domains. It is important to say clearly that in this scheme all nodes are pure Lagrange nodes, i.e. value at each node represents the value of continuous function, or (which is equivalent) each shape function is one at its own node and zero at

all other nodes.

The Gupta scheme gives explicit formulas for each shape function. Portions of these formulas depend on whether some of the 'green' nodes exists. We present some details of few of the green element formulations, as illustrative examples. In particular, we discuss:

- anisotropic h-refinement in 2d (Quad)
- anisotropic h-refinement in 3d (Quad face, Hex element)
- anisotropic p-enrichment in 2d (Quad)
- anisotropic p-enrichment in 3d (Quad face, Hex element)

This will be followed by (limited) application of the scheme to triangular faces and prism/tet 3d elements.

Only the consistency requirements and the shape functions will be discussed here. The adaptation algorithms, and the computations necessary to preserve solution upon refinement are discussed elsewhere.

7.2.1 QUAD green element (4 to 9 nodes)

General configuration of a green quad element is shown in figure 7.2. Note that there can be five additional optional nodes that can be added to the 'regular' four noded quad element.

After considering rotation and symmetry, twelve unique configurations are possible for a green quad element employing optional nodes as shown in figure 7.3.

To simplify the algorithm we restrict the allowable refined configurations to those for which the shape functions for corners nodes are modified by (the existence of) center node and by adjacent center edge nodes only, while edge nodes shape functions are modified only by center node. This makes algorithm elegantly hierarchical (which will be important in 3d case) but it makes configurations (c), (e), and (f) illegal. Use of these configurations will also prevent exact approximation of the (bi)linear solution. Configurations (g), (h), and (j) are consistent but the suggested algorithm will not create them (see below for the discussion of quad face in 3d case, which may use center node on the edge in the same location as center node on the face).

Figure 7.2: General configuration of green Quad element

The rule for consistent configuration is then as follow: "the center node is required if any two adjacent edges exist". The rule is justified on the basis of figure 7.5. Two adjacent nodes 'clip' the edge shape function such that the shape function is zero at the center point, so there has to be an independent degree of freedom at that point to assure proper interpolation of arbitrary linear function.

The shape functions will be expressed in terms of standard $(x, y) = (-1, 1)$ master element. x_i, y_i will denote coordinates of the node.

The base shape functions are shown in figure 7.4.

The corner base function :

$$Ng = 0.25(x + x_i)(y + y_i)(x_i y_i) \quad (7.1)$$

The edge base function :

$$Ne = 0.5x_i(x + x_i)abs(y)(for y_i = 0.0)0.5y_i(y + y_i)abs(x)(for x_i = 0.0) \quad (7.2)$$

The center base function :

$$Nf = (abs(x) - 1)(abs(y) - 1) \quad (7.3)$$

Figure 7.3: Possible configurations of green Quad elements

Corner shape function

Edge shape function

Center shape function

Base shape functions

----- Internal discontinuity

Figure 7.4: Base Shape Functions

Modified shape functions are shown in figure 7.5. These functions are general (they should work for all configurations above). The algorithm to build them (presented below) assumes that inconsistent configurations ((c), (e), (f)) are adjusted by adding a center node.

The corner adjusted by one edge:

$$N_{ge} = N_g - 1/2N_e \quad (7.4)$$

The corner adjusted by two edges

$$N_{gee} = N_g - 1/2(N_e + N_e) + 1/4N_f \quad (7.5)$$

The corner adjusted by center (in absence of edge nodes)

$$N_{gf} = N_g - 1/4N_f \quad (7.6)$$

The edge adjusted by center

$$N_{ef} = N_e - 1/2N_f \quad (7.7)$$

If corner is adjusted by edge(s) then the center node is irrelevant.

7.2.2 Hierarchical shape functions for p-elements

As it is customary with hierarchical p-elements, we assume that interpolation within the element is defined by the 'bubble' functions, ie. the shape functions which are zero on the boundary (and outside of) the element. In such case the interelement continuity requirements are easily satisfied if the remaining higher order shape functions are defined on the edges and shared between the elements. Please note, that the higher order shape functions are incremental, not Lagrange type. (i.e. they produce additional solution on top of standard (bi/tri-)linear shape functions, their degrees of freedom do not have the interpretation of the value of unknown function at the point, and the shape functions do not sum to 1.0).

For two dimensional Quad element, the center shape functions are the same as for standard hierarchical p-element (figure 7.6). The edge shape functions are built from one-dimensional shape functions extended into the element by simple linear blending. If the edge has center point we do not allow it to have any p-approximation, but we treat it like two smaller edges, each having its own independent p-approximation. The important part of the requirement is that subdivided edge should exactly approximate polynomial of order p if both edges are of at least order p (figure 7.7).

Corner clipped by one edge

Corner clipped by two edges

Corner clipped by center (no edges)

Edge clipped by center

Modified shape functions

Figure 7.5: Modified Shape Functions

Figure 7.6: Base Hierarchical Shape Functions

Edge shape functions, $p=2,2$

Mixed approximation edge shape function, $p=3,2$

Quadratic function approximation by divided edge

Divided edge hierarchical shape functions

Figure 7.7: Divided Edge Hierarchical Shape Functions

7.2.3 Extension to 3d

For 3d Hex element (as well as 3 faces of Prism element) foregoing discussions are applicable to individual faces. The standard interpolation requirements still apply but there are additional considerations to be taken account because the solution between two neighboring elements has to be continuous also in the case when the neighbors are not the same size.

The face between elements can be broken into two or four smaller ones, and the continuity requirements will be satisfied if the approximation on each small face will be completely defined by the true edges of the small face and by possible center node. Thus the presence of edge or edges has to modify appropriately the face shape functions, and in general, central face node is independent of any of such edges (so it is possible to have edge dividing the face and have or not the central node on the face, figure 7.8). In the actual algorithm if the face is split, then possible center node is always assigned to the edge, not to the face (i.e. edge has to be divided), although the resulting approximation and shape functions are equivalent.

The existence of edge dividing the face requires that the p-shape functions from the edge be 'clipped' (ie. limited to the small face), and the center bubble function has to be replaced by two bubble functions on small faces. (The general rule is: If any entity (edge, face, volume) is subdivided then its p-nodes are transferred to small entities)

Linear shape functions and higher-order shape functions from edges propagate into the face (and into the hex) using different rules:

- Corner nodes and Edge center nodes have their shape functions clipped (blend to zero at middle of the total length) if there is an center node on the face.
- Edge p-functions are clipped when there is a center edge on the face (parallel to this edge).

Consider, for example, the element face in configuration (c) in figure 7.8: p-shape functions from bottom edges and that from top edge are linearly blended over the half of element size (they vanish at middle edge), while all p-functions from side edges propagate all the way across the element.

Figure 7.8: Various configurations of the face divided into two small ones. (a) No middle node; (b) and (d) Center node on the face, no edge middle node (illegal); (c) and (e) No center face node, edge has middle node (divided edge)

Figure 7.9: Quad Face Parametrization

7.3 Adaptivity

Here we describe mixed element adaptivity technology based on marking and synchronization.

Each object has two fields describing its current and projected state. Refinement pattern describes only $n-1$ dimensional boundary of object (i.e. for volume, the refinement of edges is not visible)

In general the states are:

none no projected state yet

refined means this object will be refined or father refined to more sons

no_change current and projected states are identical

Figure 7.10: Triangle Face Parametrization

unrefined means that this will be destroyed and father may be potentially refined to less sons

join_all which is the true unrefinement (my father becomes active)

Refined and unrefined have substates (eg. refined into 2 sons, refined into 4 sons, etc.) The hierarchy is ordered with following order: none, unrefined, no_change, refined. Each object starts marked as none, and can only upgrade mark to higher state. The same rule applies to MERGE in MPP synchronization (resulting state is the greatest of all merged states). This rule ensures that all refinements are always enforced, additional refinements are always acceptable, unrefinements are performed only if they do not require any other unrefinements (no edge may be unrefined unless explicitly scheduled for unrefined by error indicator).

NOTE1 - Actually the above hierarchy is not so precise: in order to resolve potential deadlocks we need 'unrefine_all' to have higher priority than any refinements.

NOTE2 - The reason to handle 'green' or irregular elements is to limit the propagation of refinements (i.e. avoid the cases when single cut propagates all the way to the boundaries. In tet/triad mesh this can be accomplished without introducing irregular elements, in all other cases they are necessary. The consistency rules presented below are based on this requirement and on 1-to-2 rule (which limits the complexity of shape functions).

At each adaptation pass the following algorithm is implemented:

1. All objects are marked as 'none'.
2. All edges are marked using current error indicator (or any other similiar function).
3. All active faces and volumes are asked to verify its consistence with their boundary objects (i.e. faces verify based on edges, volumes - on faces). If the state is inconsistent then own state is adjusted and boundary objects are asked to upgrade to higher refinement. Depending on the type of object upgrade may consist of partial unrefine and then refine to bigger number of sons.
4. Step 2 is repeated till all objects are no longer changing.

5. In MPP all Aliases are merged and operation repeated until no merge is needed. Alternatively all marks may be propagated in synchronous mode.
6. All objects are refined to its projected state (edges, then faces and volumes).
7. Resulting structure may be inconsistent (new faces/volumes may require further changes) so the step 2 is repeated.
8. After the refinement phase all non-active elements are considered for unrefinement. To do this the object must have at least one boundary marked for unrefinement. All possible unrefined patterns are checked and the best one accepted, if more than one equivalent pattern is acceptable then one is selected in arbitrary way (eg. random). All boundaries not-refined in this pattern are marked as none, if any of them changed state (::Mark() returned true) or new projected status changed, then the step 7. will be repeated.
9. All objects with unrefine mark are unrefined (first volumes then faces and edges - the order is essential to keep the database consistent).
10. Projected states are reset to none.

This algorithm assumes that the global (per cpu) iterator is used for the consistency checks in multiple passes (till no change is reported). Alternative algorithm using recursive correction is described below.

In the following 'dir' is '-0' '-1' in Quad, and '-0' '-1' '-2' in Hex.

The states for the edge are:

unrefined father will be active

no_change

refined

Figure 7.11 shows the complete process of breaking an edge into two.

The states for Triad or Quad face are

unrefined

Edge Breaking Rules:

EDGE TopoFunction	Child TopoFunction
SuperIncomplete	RefinedIncomplete (attached_to_ = parent)
Complete	RefinedIncomplete (attached_to_ = parent)
StdIncremental	StandardIncremental (needs DOF computation)
MixedIncremental (undefined)	
Complete	RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete	RefinedIncomplete (attached_to_ = parent)

Figure 7.11: Edge Refinement

unrefined-dir father will be refined in direction dir, while currently father has 4 sons or 2 sons in different direction

no_change

refined-dir split in 2 sons in direction dir

refined all split in 4 sons (after possible unrefinement from refined-dir state)

Triad is always regular, Quad face can have arbitrary number of irregular (mid-edge) nodes preserving 1-to-2 rule.

The states for Hex, Prism, Pyramid, and Tet are

unrefined

unrefined-dir father will have 2 sons in direction dir

unrefined-4-dir father will have 4 sons in direction dir

unrefined-all father will have 8 sons in direction dir

no_change

refined-dir split into two sons

refined-4-dir split into four sons

refined-all split into eight sons

Hex and Prism have 7/8 possible refinement patterns. In addition, all faces in Hex, and in prism can be irregular (1-to-2 rule).

Tet is always regular, it has following allowable refinement patterns:

- one edge cut: 2 sons, 6 cases
- three edges on one face cut: 4 sons, 4 cases
- all edges cut: 8 sons, 1 case

TriFace 2-Cut Breaking Rules:

TriFace TopoFunction	TriFace-Child TopoFunction
SuperIncomplete	RefinedIncomplete (attached_to_ = parent)
Complete	RefinedIncomplete (attached_to_ = parent)
StdIncremental	StandardIncremental (needs DOF computation)
MixedIncremental	RefinedIncomplete (attached_to_ = parent)
Complete	RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete	RefinedIncomplete (attached_to_ = parent)

TriFace TopoFunction	New EDGE (en) TopoFunction
SuperIncomplete	SuperIncomplete(attached_to_ = q0 or q1)
Complete	SuperIncomplete(attached_to_ = q0 or q1)
StdIncremental	StandardIncremental (needs DOF computation)
MixedIncremental	SuperIncomplete(attached_to_ = q0 or q1)
Complete	SuperIncomplete(attached_to_ = q0 or q1)
RefinedIncomplete	SuperIncomplete(attached_to_ = q0 or q1)

TriFace 4-Cut Breaking Rules:

TriFace TopoFunction	TriFace-Child TopoFunction
SuperIncomplete	RefinedIncomplete (attached_to_ = parent)
Complete	RefinedIncomplete (attached_to_ = parent)
StdIncremental	StandardIncremental (needs DOF computation)
MixedIncremental	RefinedIncomplete (attached_to_ = parent)
Complete	RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete	RefinedIncomplete (attached_to_ = parent)

TriFace TopoFunction	New EDGE (en) TopoFunction
SuperIncomplete	SuperIncomplete(attached_to_ = q0 or q1)
Complete	SuperIncomplete(attached_to_ = q0 or q1)
StdIncremental	StandardIncremental (needs DOF computation)
MixedIncremental	SuperIncomplete(attached_to_ = q0 or q1)
Complete	SuperIncomplete(attached_to_ = q0 or q1)
RefinedIncomplete	SuperIncomplete(attached_to_ = q0 or q1)

Figure 7.12: Triangle Refinement Rules

Figure 7.13: Triangle Refinement Patterns

Figure 7.14: Triangle Refinement along one direction

Figure 7.15: Isotropic Triangle Refinement

Quad 2-Cut Breaking Rules:

QUAD TopoFunction

SuperIncomplete
Complete
StdIncremental
MixedIncremental
Complete
RefinedIncomplete

QUAD-Child TopoFunction

RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete (attached_to_ = parent)
StandardIncremental (needs DOF computation)
RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete (attached_to_ = parent)

QUAD TopoFunction

SuperIncomplete
Complete
StdIncremental
MixedIncremental
Complete
RefinedIncomplete

New EDGE (en) TopoFunction

SuperIncomplete(attached_to_ = q0 or q1)
SuperIncomplete(attached_to_ = q0 or q1)
StandardIncremental (needs DOF computation)
SuperIncomplete(attached_to_ = q0 or q1)
SuperIncomplete(attached_to_ = q0 or q1)
SuperIncomplete(attached_to_ = q0 or q1)

Quad 4-Cut Breaking Rules:

QUAD TopoFunction

SuperIncomplete
Complete
StdIncremental
MixedIncremental
Complete
RefinedIncomplete

QUAD-Child TopoFunction

RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete (attached_to_ = parent)
StandardIncremental (needs DOF computation)
RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete (attached_to_ = parent)
RefinedIncomplete (attached_to_ = parent)

QUAD TopoFunction

SuperIncomplete
Complete
StdIncremental
MixedIncremental
Complete
RefinedIncomplete

New EDGE (en) TopoFunction

SuperIncomplete(attached_to_ = q0 or q1)
SuperIncomplete(attached_to_ = q0 or q1)
StandardIncremental (needs DOF computation)
SuperIncomplete(attached_to_ = q0 or q1)
SuperIncomplete(attached_to_ = q0 or q1)
SuperIncomplete(attached_to_ = q0 or q1)

Figure 7.16: Quad Refinement Rules

One into Two

Two into Four

One into Four

Figure 7.17: Quad Refinement Patterns

Figure 7.18: Quad Refinement along one direction

Figure 7.19: Isotropic Quad Refinement

A total of 11 out of 64 possible patterns are allowed, but propagation is restricted due to fully non-structured mesh.

Pyramid is the most restrictive - if any of the non-base edges is refined then only refined-all state is allowed, if any of the base edges is refined then the opposite edge has to be refined also. Thus pyramid fully propagates the refinements, unless its neighbors are not-pyramids.

All objects have a member function *intA :: Mark(state)*, where return value is true if resulting projected state changed (or any of boundary objects returned true when Mark'ed). Due to the strict ordering of the states asking to mark to lower state is immediately aborted, as is Mark() when state is already marked. Mark() to higher or equal (but different) state requires full reevaluation of all consistency. Usually Mark() to equal state enforces promotion to next common state (i.e. refined-xi combined with refined-eta results in refined-all).

7.3.1 Recursive algorithm

When all objects know their higher order neighbors (i.e. edges know all its faces, and faces know both its volumes) then the algorithm may be optimized for speed using recursion: Every Mark() operation if the state changed, invokes Mark() for all high neighbors with the status 'none', which should be interpreted by each object as 'something changed: verify again' message. Note that this recursion uses only immediate up-neighbors (edges do not notify volumes, nodes have no connectivity at all) and only significant cost is related to edges pointing to faces (a face has only two neighbors which are useful anyway).

In this algorithm there is always one sweep through faces and then one through volumes. Inter-cpu MERGE may force to repeat this iteration, however fully synchronous algorithm (no Aliases but all Mark()'s always performed on true object with blocking message) may be effective in this case. In all described versions of the algorithm there is a global barrier between phases 2, 5, 7 and 8 (check/refine/check/unrefine). No Alias approach may be more efficient as it involves less global barriers than others, which require global barrier before each Merge operation.

7.3.2 Details of member functions

For each type of object (Edge/Face/Elem)

Reset_status() clears mark to none

Mark (!=none) if status == none for Edge
is the message from adapter : status is changed and no other action
taken return status != no_change

Mark(!=none) otherwise
is the info from the up: I need you to be at least in this state if
this is not satisfied edge promotes itself and sends Mark(none) to
all connected up-objects (i.e. from edge to faces, from face to
elems)

Mark(none) is a message from down-object or from global iterator:
verify that you are consistent, promote if necessary, send Mark()'s
up and down if appropriate.

Refine() - if your status is refine then do it now,
if status is inconsistent
then program is buggy, return 1 if anything changed

Unrefine() - if your status is unrefine then do it now, if status is
inconsistent then program is buggy, return 1 if anything changed

Global loop:

1. Reset() all objects
2. Adapt edges == Mark(based on error) if result == 0 exit
3. Mark(none) all faces - if result == 0 goto 5
4. Refine() all objects and goto 3
5. Unrefine() all objects - if result non-zero then goto 3

6. Internal test

Logic details of Mark(action) function In all cases if status is none then input parameter is copied into status Mark is always called on valid element, which is one not having any grandsons. (valid in the 1-to-2 rule). Marking for non-active object is void unless action is refine, in which case it is passed down to remove 'unrefine' marks from sons. Status member is valid only for active objects.

Action == none is always invalid

Mark(no_change) == don't unrefine - so it has to be stored.

Mark always returns 1 if there were any changes in status, except none;no_change. All recursive changes are also reported, so if there was a change in object, or in object boundaries, then it has to call all owners (objects for which it is the boundary, i.e. edge calls faces etc) if the global algorithm is recursive.

Mark on edge

```
if (state == none) state = action;return 0;
```

cur state	action:	unref,	no_action,	refine
unref		-	no_action	refine
no_action		-	-	refine
refine		-	-	-

(-) - return 0,

Mark on face

```
old_state = state; // to compare at exit for change
if (state == none) state = action;
new_mark = Current_state() // what is my lowest consistent state
if ( new_mark|status|action > no_change ) then we have to refine
    { if there are two different refines then upgrade to refine_all; }
else if any of them is no_change - upgrade to no_change
    else if no father upgrade to no_change;
        else { consider unrefine:
            father_mark = father->Current_state()
```

```
        status = unrefine(father_mark) }

if ( old_dsate != status ) changes = 1;

for each edge Mark(properly) add result to 'changes'

if ( changes != 0 ) notify all owners;return 1;
else return 0
```

3D Adaptivity: Issues and Strategies

The adaptivity strategies implemented for the 2D version of the OO kernel were extended to handle 3D topology. At the termination of development a few items in the 3D topology adapter and the hook-up to the OO kernel remained unfinished (ref. Section 'Unfinished'). In this section, we briefly describe the important issues that had to be coped with and the strategies that were followed.

The basic considerations for 3D topology adaptivity were similar to their counterparts in 2D. One important goal was to maintain parallelizability. Entities mark or examine only their bounding subentities and their parents during the marking phase. After the marking phase, each grade of subentity (i.e. edge, face) is refined separately; then the hex elements are adapted; and finally the subentities (face, edge) are unrefined by grade. Throughout, entities interact only with close relatives, internal subentities, and, during the marking phase, bounding subentities, so that the whole process is well-suited to parallel computation.

Another design goal was to avoid 'deadlock', a situation in which an entity cannot be adapted to the conditions imposed upon it with a single refinement operation. Deadlock conditions were eliminated by implementing refinements that break entities in multiple directions and by separating marking from refinement. Further details on handling of deadlocks are given the next section.

Each multidirectional hex refinement or unrefinement is implemented by cutting hexes in two repeatedly or joining pairs of hexes repeatedly. This limits the complexity of the low-level operations that actually split and join topo entities, topo functions, and boundary conditions. To maintain the

simplicity of refine and unrefine operations while allowing adaptation to be localized, a 2:1 rule was imposed at the interfaces between entities.

In order to implement 3D adaptation, volume and hex topo entity classes were created, and the set of refinement and unrefinement marks was expanded to include the additional adaptation operations possible with hexes (see Figures 7.20 and 7.21 for schematics on the overall 3D adaptation strategies). Initial versions of a complete set of hex adaptation operations (refinements and unrefinements) were written. A volume marking function for adding explicit marks to those marks consistent with the projected status of the volume's faces was implemented; this required creating a "consistency checker" for hexes. A hex will be kept green until one of the following conditions requires its refinement:

- all marks are consistent with refinement in a direction
- any boundary edge or face is marked for second level refinement
- any boundary face is marked for refinement in four

Many changes to quad and face objects were required because these objects were now subentities of and interfaces between hexes. Major changes were needed primarily for handling of deadlocks. Face refine and unrefine operations had to be split from one another so that faces would be refined before hex adaptation and unrefined only after hex adaptation. A new kind of face, the "stepparent face", was implemented to accommodate intermediate states in hex adaptation and 4:2 interfaces between hexes.

Examples of a typical refinement operation and typical unrefinement operation are given in Figures 7.22 and 7.23.

7.3.3 Deadlocks in 3D Adaptation

The deadlock is the situation where two elements on both sides of the face have inconsistent (not-aligned) cuts. The deadlock may happen only in anisotropic 3d adaptation.

These are two examples of deadlock:

- face is a quad and elements on both sides are cut once - one horizontally, another vertically.

Figure 7.20: 3D Adaptation - Basic Schema

Marks and actions for faces.

MARK	PARENT BREAK	PARENT MARK	REFINE ACTION	UNREFINE ACTION
refine_0	NA	NA	1->2	none
refine_1	NA	NA	1->2	none
refine_all	NA	NA	1->4	none
unrefine_all	refine_0	refine_all	2->4	clean stepparents
unrefine_all	refine_1	refine_all	2->4	clean stepparents
unrefine_0	refine_1	refine_0	2->4	4->2
unrefine_0	refine_all	refine_0	add stepparents	4->2
unrefine_1	refine_0	refine_1	2->4	4->2
unrefine_1	refine_all	refine_1	add stepparents	4->2
join_all	refine_0	NA	none	2->1
join_all	refine_1	NA	none	2->1
join_all	refine_all	NA	none	4->1

Marks and actions for edges.

MARK	PARENT BREAK	PARENT MARK	REFINE ACTION	UNREFINE ACTION
refine_all	NA	NA	refine	none
unrefine_all	refine_all	NA	none	unrefine

Figure 7.21: Decomposition of face and edge operations into refine and unrefine parts of the hex adaptation loop

2→4

Pair of 2→1 ops. (Discard 1
internal stepface, unrefine another.)

Join followed by 1→4. (Discard
original internal face.)

Figure 7.22: A multidirectional refinement example

4→2

Pair of 2→1 ops. (Discard one
internal stepface,

Join followed by 1→2. (Discard
two internal stepfaces.)

Figure 7.23: A multidierctional unrefinement example

- face is triad and one element has two sons, while another has four.

Resolution of deadlocks requires, in general, postponing actual refinements and unrefinements until the ultimate state of the mesh is known to be free of deadlocks and 2:1 violations. For this reason, all conflicts are resolved in a marking phase before adaptation proper. To remove the deadlock, an element on one side of the interface will be refined to a higher degree. For the examples above:

- The first element will be cut in four perpendicular to the quad face, and two additional faces will be built for the faces of the children of the second element. See Figure 7.24
- The first element will be cut in four across the triad face.

Handling of hex deadlocks After some experimentation, a new kind of face was introduced to handle the 4:2 interface induced by hex deadlock resolution, the stepparent face. Stepparent faces have children that are members of the family tree but are not themselves proper members of the family tree; they require special handling during face marking, refinement and unrefinement and when examined in hex consistency checking; and they are green. Such quads are stepparents of two of the children of a quad broken in four. To support the use of stepparent faces, reference counting was added so that stepparents and their children would be destroyed or retained properly during face unrefinement.

In order to get hex adaptation and especially deadlock resolution to work, the 2D code had to be extensively modified. In the 2D code, face refinement and unrefinement operations had been fused together and were all performed in the face refine portion of the adaptation loop. When these methods were carried over to 3D, there were synchronization problems because hex faces were unrefined before the hexes they were attached to. To solve these problems, quad refine and unrefine operations were split.

In some cases, the initial versions of hex adaptation code followed the model of 2D adaptation too closely. Some of the resulting operations modified subentities on boundaries, often when the hex passed through intermediate states during the operation. This caused conflicts between operations on neighboring hexes. All changes to boundary subentities were removed from

Attached Hex
split horizontally

Attached Hex
split vertically

Figure 7.24: Deadlock on the quad face of a Hex

2→2

Face refine loop breaks 2 faces in 4 with stepparents. Hex operation joins followed by splitting in the new direction.

4→4

Implemented as two 2→2 operations
Must unrefine? one midface, destroy
another, and create a new one.

Figure 7.25: Implementation of two hex adaptations (2-to-2 and 4-to-4) under deadlock

hex operations, including the creation of stepfaces, which is now performed during the face refine phase. The initial version of the hex consistency checker examined projected edge status only, which is not a reliable indicator of projected face status. A new consistency checker was written which examines the face status directly.

Figure 7.25 a schematic of two hex adaptations (2:2 and 4:4) are depicted that show how the deadlocks were corrected.

Part III

PHLEX2000 Development

Chapter 8

Overall Philosophy

Phlex2000 is based on the premise that most physical problems solved using finite elements can be formulated in a similar fashion. One of the requirement of such a premise is that a simulation problem be representable as a second order partial differential equation which can then be cast into a 'weak' (or variational) form. Many of the real life physical problems either directly conform to this form or can be reformulated (in terms of secondary variables) to appear in this form. The main advantage of such an view is that it decouples the PDE formulation from the discretization used. It thus allows encapsulation of the (often very complex) discretization mechanisms like the finite element method or the finite difference technique without tying them to any particular physical problem formulation. This approach is thus inherently object-oriented and promotes reuse of the discretization technology while solving a wide variety of physical applications.

Phlex2000 encapsulates the core finite element technology and the secondary components that go along with it. Through a general applications programming interface (API), application developers can code arbitrary applications (as long as they conform to the form mentioned above) and solve them using finite element method.

All of the classes in the API are abstract. This means that - kernel deals only with the interfaces provided by these classes - an application needs to override these classes by deriving from them which also allows it to add extra functionality peculiar to the application.

8.1 Generic Kernel

The Phlex2000 kernel is composed of many distinct components which are needed to solve finite element problems. Following object oriented principles, each of these components encapsulates a clearly identifiable concept. Most of these components are represented by abstract base classes thus making behaviour more important than the state.

8.2 Librarization

The kernel abstraction is represented by the class *Phlex_Kernel*. This class is implemented as a singleton, i.e., only one instance of this class is allowed to exist ([14]). The interface for the class is very simple since the kernel is assumed to be the master driver for the whole finite element process. Here's the class interface:

```
class Phlex_Kernel : Phlex_Object {
public:
    // implemented as a singleton
    static Phlex_Kernel* Kernel();
    ~Phlex_Kernel();

    void execute(Phlex_Application* app);

protected:
    Phlex_Kernel();
};
```

PHLEX2000 also provides an abstract class to represent applications that will be written using the kernel. The *Phlex_Application* class defines this interface on which the *Phlex_Kernel* depends. When writing any new application, the author must derive from this base class and provide implementations for all of the (pure) virtual methods. Here's the class definition:

```
class Phlex_Application : Phlex_Object {
public:
    // implemented as a singleton
```

```

static Phlex_Application* Application();
~Phlex_Application();

virtual void setupFields(Phlex_Domain* domain) = 0;
virtual void setupBCs(Phlex_Domain* domain) = 0;
virtual void ReadBCs() = 0;
virtual void ReadMaterials() = 0;
virtual void Init() = 0;
virtual void Driver(Phlex_Domain* domain) = 0;
virtual void ErrorEst(Phlex_Domain* domain) = 0;
virtual void Adapter(Phlex_Domain* domain) = 0;
virtual void Gui() = 0;
virtual void Postproc() = 0;

void name(const char*& name_in);
const char* name() const;

protected:
    Phlex_Application();
    static Phlex_Application* application_;
};

```

Note that this class is also implemented as a singleton since it wouldn't make sense to have more than one instance of an application.

An application needs to create an instance of the *Kernel* class and invoke the method *execute()* on it at an appropriate time. This method then drives the whole finite element process. For example, here's a code snippet from a solid mechanics application:

```

void main()
{
    Phlex_Application* app = SolidsApp::createSolidsApp("Solids");
    Phlex_Kernel* k = Phlex_Kernel::Kernel();
    k->execute(app);
}

```

Chapter 9

Applications Interface

The interface presented to the applications consists of a few abstract concepts directly related to the problem space; these concepts are independent of the considerations for the solution space (i.e. no dependence on the type of discretization used). In PHLEX2000, there are three major orthogonal concepts that an application developer uses to describe complex problems adequately. These are:

- Domain
- Pde
- SolutionProcess

The idea is to describe the characteristics of the physical domain through objects of class Domain and the formulation of the underlying partial differential equations through the use of Pde objects. SolutionProcess objects can then be used to associate particular solution sequences to combinations of Domain and Pde objects. As we will see later in this chapter, identification and encapsulation of these fundamental (and independent) concepts allows easy expression of complex simulation problems. We will now describe these concepts in detail, followed by some examples of applications.

9.1 The Coeff Class

As has been mentioned earlier, Phlex2000 requires specification of the partial differential equations to be solved in a particular form. The PDE should be written in a variational (or weak) form such that no term has higher derivatives than one. The general format of the (linear) variational form that PHLEX can solve is given by:

$$\int_{\Omega} a_{p,q}^{i,j} \frac{\partial^p v_i}{\partial x^p} \frac{\partial^q u_j}{\partial x^q} d\Omega + \int_{\Gamma} b_{k,l}^{i,j} \frac{\partial^k v_i}{\partial n^k} \frac{\partial^l u_j}{\partial n^l} ds = \int_{\Omega} f_p^i \frac{\partial^p v_i}{\partial x^p} d\Omega + \int_{\Gamma} g_k^i \frac{\partial^k v_i}{\partial n^k} ds$$

where,

$p, q \in \{0, 1, 2, 3\}$, 0 means no derivative, $\{1, 2, 3\}$ mean derivatives with respect to $\{x, y, z\}$

$k, l \in \{0, n, t, s\}$, 0 means no derivative, $\{n, t, s\}$ are the normal (n) and two tangential directions on the boundary (t, s)

$i, j \in \{1, 2, \dots, ncomp\}$, where $ncomp$ is the number of equations in the PDE which is equal to the number of solution components (i.e. $\{u, v, w, p\}$, $ncomp = 4$).

For example, solving the problem

$$-\nabla^2 U + c * U = F$$

in Ω , and

$$\frac{\partial U}{\partial n} = G$$

on boundary of $\Omega = \Gamma$, would give (for $ncomp = 1$ and $G = 0$):

$$a_{1,1}^{1,1} = a_{2,2}^{1,1} = a_{3,3}^{1,1} = 1.0$$

$$a_{0,0}^{1,1} = c$$

$$f^1 = F$$

and all other coefficients zero.

The `Phlex_Coeff` class provides a means of specifying the coefficients for second order PDEs as required by Phlex2000 kernel. Both PDE objects as well as BCs need to be configured with appropriately derived `PDEcoeff` objects. In addition, various solver algorithms can supply their own specialized Coeffs to formulate the linear system of equations.

Here's the interface for this class:

```

class Phlex_Coeff : public Phlex_Object {
public:
    Phlex_Coeff();
    Phlex_Coeff(int ncomp);
    Phlex_Coeff(const Phlex_Coeff& orig);
    Phlex_Coeff& operator=(const Phlex_Coeff& orig);
    virtual ~Phlex_Coeff();

    virtual void reset_matrix();
    virtual void reset_vector();

    virtual void set_matrix_haves();
    virtual void set_vector_haves();

    const bool has_matrix_entry(const int i, const int j) const;
    const bool has_vector_entry(const int i) const;
    const Phlex_NumMatDouble& matrix_entry(const int i, const int j) const;
    const Phlex_NumVecDouble& vector_entry(const int i) const;
};

```

To elicit coefficients for PDEs in the required manner from applications, an abstract class *Phlex_PDEcoeff* has been derived from *Phlex_Coeff*. Applications are thus required to derive from this class and provide their own definitions of the pure virtual functions declared in *Phlex_PDEcoeff*.

```

class Phlex_PDEcoeff : public Phlex_Coeff {
public:
    // The following coeffs are based on the general, 2nd order (in time)
    // equilibrium equation represented as
    //      M.a + C.v + K.u = f
    // where u = primary variable
    //      v = first time derivative of u
    //      a = second time derivative of u

    // Note that a matrix and a vector coefficient can be requested
    // simultaneously in the following (numbering) scheme. For example,
    // one would request evaluation of both MASS and RESIDUAL as

```

```

// coef.eval(ipd, el, (MASS+RESIDUAL));

enum PDEcoeffType
{NO_REQUEST      = 0,
 INVALID_REQUEST = 1 << 0,
 STIFFNESS       = 1 << 1, // K
 MASS            = 1 << 2, // M
 DAMPING         = 1 << 3, // C
 // vector requests below
 SOURCE          = 1 << 4, // f
 FUNCTIONAL      = 1 << 5, // = F() = LHS(u,v,a) - f
 STIFFNESSxV     = 1 << 6, // = K*V
 MASSxV          = 1 << 7, // = M*V
 DAMPINGxV       = 1 << 8  // = C*V
};

// Note: a qualifier is meaningful only in certain contexts
enum PDEcoeffQualifier {NONE, TANGENT};

Phlex_PDEcoeff(int ncomp);
Phlex_PDEcoeff(const Phlex_PDEcoeff& orig);
Phlex_PDEcoeff& operator=(const Phlex_PDEcoeff& orig);
virtual ~Phlex_PDEcoeff();

// compute the requested coeff and store in matrix/vector
virtual void eval(const Phlex_IntegrationPointData& ipd,
                  const Phlex_DataVectorIDs& dv,
                  const Phlex_TopoEntityFull& el,
                  const PDEcoeffType request,
                  const PDEcoeffQualifier qualifier) = 0;

const PDEcoeffType& LastCoeffRequested(bool forMatrix) const;

// this signature is different from that in the base class
virtual void set_matrix_haves(const PDEcoeffType request,
                              const Phlex_PDEcoeff::PDEcoeffQualifier
                              qualifier = NONE);

```

```

virtual void set_vector_haves(const PDEcoeffType request,
                             const Phlex_PDEcoeff::PDEcoeffQualifier
                             qualifier = NONE);

// this provides a list of sub_fields on which the coeffs
// depend. Even though the primary variable can only
// be of either continuous or piecewise type, it could possibly
// have dependencies on both types of variables.
void dependencies(const Phlex_Vector<int>& dep_in,
                 Phlex_FEField::SpaceType type=Phlex_FEField::CONTINUOUS);
const Phlex_Vector<int>& dependencies
    (Phlex_FEField::SpaceType type=Phlex_FEField::CONTINUOUS) const;

// specify which of the dependent subfields need to be evaluated at IPs
// the vectors should contain yes(1) or no(0)
void eval_at_int_pt(const Phlex_Vector<int>& vec,
                   Phlex_FEField::SpaceType type=Phlex_FEField::CONTINUOUS);
void eval_derivs_at_int_pt
    (const Phlex_Vector<int>& vec,
     Phlex_FEField::SpaceType type=Phlex_FEField::CONTINUOUS);
const Phlex_Vector<int>& eval_at_int_pt
    (Phlex_FEField::SpaceType type=Phlex_FEField::CONTINUOUS) const;
const Phlex_Vector<int>& eval_derivs_at_int_pt
    (Phlex_FEField::SpaceType type=Phlex_FEField::CONTINUOUS) const;
};

```

The PDEcoeffs are always asked for a particular topological entity at a given integration point.

9.2 Domain

A Domain is an abstraction that defines the properties of the space on which the given PDE is to be solved. This includes the mesh, field description, boundary conditions and material properties.

Domains are recursively defined so that a domain could have multiple regions or subdomains. Interfaces among these subdomains are treated as

specialized domains. This means that one could actually define interface PDEs and solve them on these interfaces.

A parent domain doesn't have any interfaces, only the child domains point to a common interface object. An interface is always defined between a pair of interacting domains.

There is a global domain (defined by Type T_GLOBAL) that is meant to mean all the elements of the entire mesh. It is created by the kernel from input data. The global domain is implemented as a singleton, it can be accessed but not created by an application.

```
class Phlex_Domain : public Phlex_Object {
public:
    enum DomainType {T_GLOBAL, T_SUB_DOMAIN};

    static Phlex_Domain* createDomain(DomainType t=T_GLOBAL,
                                      const char name[]="");

    virtual ~Phlex_Domain();

    Phlex_Domain* parent() const;
    void addRegion(Phlex_Domain* region);
    Phlex_Domain* Region(int i) const;
    int NoOfRegions() const;

    void field(Phlex_FEField* field);
    Phlex_FEField *field() const;

    void addBC(Phlex_BCtype* bc);
    Phlex_BCtype* BC(int i) const;
    int NoOfBCs() const;
    int BCindex(const String& bctype) const;

    void addMaterial(Phlex_Material* mat);
    Phlex_Material* Material(int i) const;
    int NoOfMaterials() const;

    void addInterface(Phlex_Interface* interface);
    Phlex_Interface* Interface(int i) const;
```



```

    int NoOfInterfaces() const;

    void Mesh(Phlex_Mesh* mesh);
    Phlex_Mesh* Mesh() const;

    Phlex_FieldVector* fieldVector();

protected:
    Phlex_Domain();
};

```

Descriptions of the various components that define a Domain follow.

9.2.1 Interface

An interface represents the region between two subdomains or regions. It is a specialization of Domain since all domain like operations can be required on an interface. This gives us the ability to actually specify a PDE and a solutionProcess over it. Special operations may be needed to correlate the degrees of freedom on the two sides of an interface.

```

class Phlex_Interface : public Phlex_Domain {
public:
    Phlex_Interface (Phlex_Domain *, Phlex_Domain *);
    void relation (Phlex_Domain *, Phlex_Domain *) = 0;
protected:
    Phlex_Domain * domain1;
    Phlex_Domain * domain2;
};

```

9.2.2 Field, SubField, and Component Classes

Field, Subfield and Component classes are used to describe the characteristics of solution and geometry spaces associated with a domain.

Field

In our design we recognize two distinct spaces: geometry and solution. A "field" is used to represent elements of these spaces. An application can only define solution spaces. The geometric space is automatically created by the kernel from input data and attached to the global domain.

A field has the following properties:

1. a field essentially represents an element of the space.
2. a field is composed of sub-field(s).
3. a field can have two types of subspaces (or sub-fields) – those with 'continuous' and/or 'piecewise' variation.
4. the grouping of sub-fields defines a single field.
5. each field has a topo-function attached to it which describes the manner of variation within the space e.g., linear Lagrange, quadratic Lagrange, incremental Lagrange, etc. A different topo-function can be associated with each of the two types (continuous or piecewise) of subfields.
6. a field can have derived quantities defined by a derived field.
7. a derived field has a parent field.
8. a subfield can have multiple components. Each of these components can be of type scalar, vector, matrix, tensor, etc.

A field will be associated with a domain. It can have multiple subfields, but they are part of the same SOLUTION space. Only one SOLUTION field can be active at a time for a domain. A SOLUTION field can have two kinds of subfields - CONTINUOUS and PIECEWISE, with their own specific variation. These subfields can refer to each other freely (as specified via the PDE formulation).

A field is actually a signature of a space. A specific space is created using this signature for a Domain. Hence, a field object can be associated with multiple domains. This also means that a different set of "degrees-of-freedom" (dofs) unique to a domain exists for each specified field object.

Subfields of a Field are usually defined keeping in mind the problem formulation. For example, in a two-phase flow through porous media problem the solution field may consists of the variables – pressure (P), velocity (V) and saturation (S). Depending on the formulation, these could be treated as a single subfield if there is only one PDE involving all variables or as three different subfields if there is a PDE for each variable.

Let us describe the above example more thoroughly to show how the field setup can be used meaningfully. Suppose that there are the following possibilities in the above-mentioned flow problem:

1. There is a single PDE coupling all variables. In this case, there will be a single subfield with three components. Components corresponding to P and S will be of type SCALAR while that for V will be of type VECTOR.
2. There are two PDEs, one for P and the other for V and S. Here, we will have two subfields. First subfield will have one component of type SCALAR while second subfield will consist of two components of types SCALAR and VECTOR.
3. There are three PDEs, one each for P, V and S. There will be three subfields with one component each of types SCALAR, VECTOR, SCALAR.

Here is the public interface of the Field class:

```
class Phlex_FEFIELD : public Phlex_Object {
public:
    static const int FULL_FIELD = -1;
    enum SpaceType {CONTINUOUS=0, PIECEWISE=1};

    Phlex_FEFIELD(const String& name_in,
                  const Phlex_Vector<Phlex_FESubField*>* sub_fields_cont,
                  Phlex_BasisClassType::Type tftype_cont =
                      Phlex_BasisClassType::BCL_IL,
                  const Phlex_Vector<Phlex_FESubField*>* sub_fields_piece = 0,
                  Phlex_BasisClassType::Type tftype_piece =
                      Phlex_BasisClassType::BCL_IL);
```

```

Phlex_FEFIELD(const Phlex_FEFIELD& cp);
virtual ~Phlex_FEFIELD();

const String& name() const;
const Phlex_FESubField* sub_field(int i, SpaceType type=CONTINUOUS) const;
int num_sub_fields(SpaceType type=CONTINUOUS) const;
Phlex_BasisClassType::Type tf_type(SpaceType type=CONTINUOUS) const;

int n_components_field(SpaceType type=CONTINUOUS) const;
int n_values_field(SpaceType type=CONTINUOUS) const;
bool have_space(SpaceType type) const;
};

```

SubField

A sub-field is described by a name and a description of its components.

```

class Phlex_FESubField : public Phlex_Object {
public:
    Phlex_FESubField(const String& name_in,
                     Phlex_Vector<Phlex_ComponentDesc*>* compDesc_in);
    Phlex_FESubField(const Phlex_FESubField& cp);
    Phlex_FESubField& operator=(const Phlex_FESubField& cp);
    virtual ~Phlex_FESubField ();

    const String& name() const;
    const Phlex_ComponentDesc* component(const int i) const;
    int n_components() const;
    Phlex_ComponentDesc::Type component_type(const int i) const;
    int n_values_component(const int i) const;

    void MaxIntOrder(const unsigned int setTo);
    int MaxIntOrder() const;

    void ReduceIntOrder(const int reduceBy);
    int ReduceIntOrder() const;
};

```

```

    int n_values() const;
};

```

In certain situations, one may want to associate special integration requirements with some of the state variables. This can be easily accomplished within the field mechanism by defining such state variables as separate sub-fields and by the use of member functions:

- MaxIntOrder (setTo)
- ReduceIntOrder (reduceBy)

By setting the values in reduceBy to -ve, one can also increase the integration order, if desired.

Component

A component is defined as a set of numerical values that all have the same units (defined by the Unit class). The set of scalars (if more than one) can be 'grouped' as a vector (CT_VECTOR), matrix (CT_MATRIX) or tensor (CT_TENSOR) by defining the dimension of the quantity with the DIMENSION data field.

```

class Phlex_ComponentDesc : public Phlex_Object {
public:
    static const int MAX_ORD = 4;
    enum Type { CT_SCALAR=0,
                CT_VECTOR=1,
                CT_MATRIX=2,
                CT_TENSOR_3=3,
                CT_TENSOR_4=4,
                CT_END};

    Phlex_ComponentDesc();
    Phlex_ComponentDesc(const String& name_in,
                        const Phlex_Unit& unit_in,
                        const Type& type_in = CT_SCALAR);

```

```

Phlex_ComponentDesc(const Phlex_ComponentDesc& cd);
Phlex_ComponentDesc& operator=(const Phlex_ComponentDesc& cd);
virtual ~Phlex_ComponentDesc();

// inlined modifiers
void name(const String& name_in);
void type(const Type& type_in);
void unit(const Phlex_Unit& unit_in);
void ncomp(const unsigned int dimension, const unsigned int ncomp);

const String& name() const;
const Phlex_Unit& unit() const;
Type type() const;
unsigned int order() const;
unsigned int size(const unsigned int dimension) const;

// Total number of (scalar) values defining the component.
int n_values(void) const;
}

```

9.2.3 Mesh

A mesh is a representation of the underlying spatial discretization. This class provides means of operating on this discretization. It is assumed that discretization is represented only by finite elements and not by topo-entities. However, in case of an Interface mesh, one may need a mesh defined by topo-entities. In this case, one can derive from Mesh.

The Mesh class provides iterators for traversing the finite elements, topo entities and bcs for the domain object. The public interface of this class will provide, in a uniform manner, mesh traversal for both the kernel as well as the applications (if needed) while hiding the implementation details.

```

class Phlex_Mesh : public Phlex_Object {
public:
    Phlex_Mesh();
    virtual ~Phlex_Mesh();

```

```
void is_modified(bool flag);
bool is_modified() const;
int version() const;

void update(); // creates active elements list

void addInitialElement(Phlex_Element* elem);
void resetInitialElements();
Phlex_Element* nextInitialElement();
int n_InitialElements() const;

void resetElements();
Phlex_Element* nextElement();
int n_Elements() const;

void addVertex(Phlex_Vertex* vertex);
void resetVertices();
Phlex_Vertex* nextVertex();
int n_Vertices() const;

void addDOF(Phlex_DOF* dof);
void resetDOFs();
Phlex_DOF* nextDOF();
int n_DOFs() const;

// actual bc objects
void addBC(Phlex_BC* bc, int bctype_index);
void resetBCs(int bctype_index);
Phlex_BC* nextBC(int bctype_index);
int n_BCs(int bctype_index) const;

void addPointBC(Phlex_PointBC* pbc);
void resetPointBCs();
Phlex_PointBC* nextPointBC();
int n_PointBCs() const;
}
```

9.2.4 BC

All BC objects are instances of class *Phlex_BC*, however, applications do not need to deal with this class. Kernel creates BC objects from the description provided by the application. For applications, there are two abstract classes available to describe their boundary conditions — *Phlex_PDEcoef* and *Phlex_BCdata*. An application derives subclasses from both of these base classes corresponding to the different kinds of BCs it needs. While subclasses of *Phlex_PDEcoef* are used to supply the coefficients of the second order PDE, subclasses of *Phlex_BCdata* are used to describe the input data for the BC (i.e., the data read from an input file describing the mesh). There is yet another class *Phlex_BCtype*, which is used by the application to create a complete description of the BCs it needs for passing on to the kernel. Here is the description of these classes:

```
class Phlex_BC : public Phlex_Object {
public:
    static Phlex_BC* lookup_bc(const String& bc_instance_name);
    static void print_bcs();

    Phlex_BC(const String& name,
              const Phlex_BCtype* type,
              Phlex_BCdata* data);

    Phlex_BC(const Phlex_BC& orig);
    Phlex_BC& operator=(const Phlex_BC& orig);
    virtual ~Phlex_BC();

    const String& name() const;
    const Phlex_BCtype* bc_type() const;
    Phlex_BCdata* bc_data() const;
    void print();

    void add_applied_to(Phlex_TopoEntity* te, int sub_te);
    void remove_applied_to(Phlex_TopoEntity* te, int sub_te);

    const int n_applied_to(void) const;
    Phlex_TopoEntity* applied_to(int i, int &sub_te) const;
```



```
protected:
    Phlex_BC();
};

class Phlex_BCtype : public Phlex_Object {
public:
    static const int MAX_BCtypes = 50;
    enum Bctype {ESSENTIAL, NATURAL};

    Phlex_BCtype(const String& name, Phlex_PDEcoeff* coef,
                  Phlex_BCdata* data, Bctype type = ESSENTIAL);
    Phlex_BCtype(const Phlex_BCtype& orig);
    Phlex_BCtype& operator=(const Phlex_BCtype& orig);
    ~Phlex_BCtype();

    Phlex_BCdata* createData(const Phlex_Vector<double>& input_data);
    const String& name() const ;
    Bctype type() const;
    Phlex_PDEcoeff* coeff() const ;
    void print() const;

    static Phlex_BCtype* lookup_bctype(const String& bctype);

protected:
    Phlex_BCtype();
};

class Phlex_BCdata : public Phlex_Object {
public:
    Phlex_BCdata();
    Phlex_BCdata(const Phlex_BCdata& orig);
    Phlex_BCdata& operator=(const Phlex_BCdata& orig);
    virtual ~Phlex_BCdata();

    virtual Phlex_BCdata*
        create_data(const Phlex_Vector<double>& input_data) const = 0;
```

```

virtual void eval(const Phlex_IntegrationPointData* ipd,
                  const Phlex_TopoEntity* te,
                  const Phlex_TopoEntity* sub_te) = 0;

virtual void print() const;
};

```

BCs have to be provided by PDE, since they are subfield specific. The application adds the required/active BCs for a PDE by their names. The actual BC lists are generated by kernel by matching those names to the specific BC instance names in the list of BCs in Mesh object. It is kernel's responsibility to create two separate instances of a BC if it is shared among two domains (or subdomains).

9.3 PDE

This class is representative of a physical PDE (Partial Differential Equation). A PDE is defined by specifying the primary variable and dependent sub_fields, and is configured with a specific PDEcoeff object which delivers the correct PDE coefficients at an integration point. This class is defined as an abstract class so that the application is forced to derive from the base class and provide the methods needed for defining PDE coefficients.

In addition, the user can also specify the particular primary state variables associated with this PDE (if there are multiple subfields). This is done by specifying an integer index corresponding to the subfield in the solution field.

In many cases, the primary variable(s) being solved for also depend on other field variables (which are not primary to this PDE). A list of such dependencies can also be specified.

There might also be specific BCs associated with a PDE. One also needs to specify the types of BCs associated with this PDE.

```

class Phlex_PDE : public Phlex_Object {
public:
    // Sets the coefficient object and the primary state variable
    // for this PDE; the int denotes the index of sub_field in FieldDesc.

```

```

Phlex_PDE(Phlex_PDEcoeff& coeff,
          const Phlex_FEFfield& field,
          int primary_var = 0,
          Phlex_FEFfield::SpaceType type=Phlex_FEFfield::CONTINUOUS);
~Phlex_PDE();

// returns the primary_variable & its type
int primary_variable(Phlex_FEFfield::SpaceType& type) const;
Phlex_PDEcoeff& Coeff() const;

// Add/remove specific BCtypes by names
void addBC(const String& bcname);
void removeBC(const String& bcname);

int numBC() const;
int numBC(Phlex_BCtype::BCtype type) const;
const String& bc(int i, Phlex_BCtype::BCtype type) const;
};

```

9.4 SolutionProcess

A SolutionProcess class allows an application to specify complicated solution processes in an intuitive manner. Here is the basic class definition:

```

class Phlex_SolutionProcess : public Phlex_Object {
public:
    Phlex_SolutionProcess(Phlex_PDE* pde,
                          Phlex_Domain* domain,
                          Phlex_EqnSolver* solver);

    // pde & domain are not required while specifying a strategy
    Phlex_SolutionProcess(Phlex_EqnSolver* solver,
                          Phlex_Vector<Phlex_SolutionProcess*>* strategy);

    Phlex_SolutionProcess(const Phlex_SolutionProcess& sp);
    Phlex_SolutionProcess& operator=(const Phlex_SolutionProcess& sp);
};

```

```

virtual ~Phlex_SolutionProcess();

void solve(); // recursively invokes solver(strategy)
Phlex_PDE* pde() const {return pde_;};
Phlex_Domain* domain() const {return domain_;};
Phlex_EqnSolver* solver() const {return solver_;};
};

```

The general, active interface for a SolutionProcess is through specification of the triplet — PDE, domain, solver. These three orthogonal concepts can be glued together by the developer according to his needs.

The second interface allows for a recursive definition of a SolutionProcess. 'Strategy' corresponds to a sequence of actions to be performed inside a given 'solver' loop. Hence, while 'Strategy' makes sense inside a time marcher or a nonlinear solver, it makes no sense inside a linearSolver.

Here is a general example of this construct's usage. Let's say that an application requires following solver structure:

```

time loop {
  nonlinear loop {
    pde1
    Skyline linearSolver

    pde2
    Sparse linearSolver
  }
}

```

With the SolutionProcess scheme, this would be described as

```

SkylineSolver Skyline();
SparseSolver Sparse();

SolutionProcess SP1(pde1, domain, Skyline);
SolutionProcess SP2(pde2, domain, Sparse);
vector<SolutionProcess> linearSolves;

```

```

linearSolves[0] = SP1;
linearSolves[1] = SP2;
NewtonSolver Newton();
SolutionProcess nonlin_solver(Newton, linearSolves);

TimeMarcher tm();
vector<SolutionProcess> nonlinSolves;
nonlinSolves[0] = nonlin_solver;

SolutionProcess solver(tm, nonlinSolves);

solver.solve();

```

The SolutionProcess scheme allows for following situations:

1. Application defines its own solution strategy, only uses kernel's linear-Solver. In this case, PDE coeffs correspond to app's chosen strategy.
2. Application uses kernel's time marcher and nonlinear solvers but provides coeffs appropriate for this combination of solvers. In other words, the solvers just provide looping mechanisms.
3. Application completely depends upon kernel's solvers. In this case, the nonlinear and time solvers would manipulate coeffs to provide the appropriate equation system to the linear solver.

9.4.1 Solver

An abstract solver class hierarchy is provided from which specific solvers are derived. The base class for all solvers is:

```

class Phlex_EqnSolver : public Phlex_Object {
public:
    Phlex_EqnSolver(Phlex_AppHook* apphook=0);
    virtual ~Phlex_EqnSolver() {};
    virtual void solve(Phlex_PDE* pde, Phlex_Domain* domain) = 0;
    virtual void solve(Phlex_Vector<Phlex_SolutionProcess*>* strategy) = 0;

```

```
Phlex_AppHook* appHook();
};
```

The important thing to note here is that each solver includes a pointer to an object of class AppHook. The class AppHook provides the functionality of a user defined hook in the solution process. An abstract base AppHook class is defined as:

```
class Phlex_AppHook : public Phlex_Object {
public:
    Phlex_AppHook();
    virtual void before() = 0;
    virtual void after() = 0;
};
```

Objects of this class provide a means of applying application specific actions as requested by app. developer. Since the class is abstract, a developer is required to derive from this class and specify his problem specific actions to be applied every time a particular solver is invoked. The before and after methods get invoked in the beginning and at the end of a solver. A developer may derive his AppHook as:

```
class MyAppHook : public Phlex_AppHook {
public:
    MyAppHook(Domain,pde1,pde2,pde3);
    void before();
    void after();
private:
    Domain *d1_;
    PDE *p1_, *p2_, *p3_;
};
```

He can then compute derived quantities, keep track of the solution's quality, etc. Note however, that one is not required to supply an AppHook if the default solver functionality is deemed sufficient (in which case, there is no requirement to derive from AppHook either).

More complex AppHooks may be required for some solvers in special circumstances. A nonlinear solver would generally provide for a Nonlinear-AppHook object which is a special abstract AppHook class defined as:

```

class NonlinearAppHook : public Phlex_AppHook {
public:
    NonlinearAppHook();
    virtual void isConverged() = 0;
}

```

NonlinearAppHook essentially adds extra functionality to the base class description. User's can now opt to use kernel's nonlinear solvers with their own convergence criteria. In some circumstances a NonlinearAppHook object is actually required to be specified e.g., when there are multiple PDEs or multiple solution spaces specified (since the kernel can not figure out what convergence criteria to use in these complex situations). An example of such a complex case follows (from Unisim):

```

class MyNonlinearAppHook : public NonlinearAppHook {
public:
    MyNonlinearAppHook(Domain,pde1,pde2,pde3);
    void before();
    void after();
    bool isConverged();
private:
    double n1,n2,n3;
    double n4,n5,n6;
    bool firstTime=yes;
    Domain *domain;
    PDE *pde1, *pde2, *pde3;
}

```

```

void
MyNonlinearAppHook::before()
{
    if (firstTime) {
        n1 = domain.Solution.2norm(pde1->sub_field);
        n2 = domain.Solution.2norm(pde2->sub_field);
        n3 = domain.Solution.2norm(pde3->sub_field);
    }
    else {

```

```
    n1 = n4; n2 = n5; n3 = n6;
  }
}

void
MyNonlinearAppHook::after()
{
    n4 = d1.Solution.2norm(pde1->sub_field);
    n5 = d1.Solution.2norm(pde2->sub_field);
    n6 = d1.Solution.2norm(pde3->sub_field);
}

bool
MyNonlinearAppHook::isConverged() {
    return (n1 < n4 && n2 < n5 && n3 < n6);
}
```

9.5 Application Development

There is a top level abstract class provided by kernel called `Phlex_Application` from which an application is supposed to derive. Here's the class description:

```
class Phlex_Application {
public:
    // implemented as a singleton
    static Phlex_Application* Application();
    ~Phlex_Application();

    virtual void setupFields(Phlex_Domain* domain) = 0;
    virtual void setupBCs(Phlex_Domain* domain) = 0;
    virtual void ReadBCs() = 0;
    virtual void ReadMaterials() = 0;
    virtual void Init() = 0;
    virtual void Driver(Phlex_Domain* domain) = 0;
    virtual void ErrorEst(Phlex_Domain* domain) = 0;
    virtual void Adapter(Phlex_Domain* domain) = 0;
```



```

virtual void Gui() = 0;
virtual void Postproc() = 0;

void name(const String& name_in);
const String& name() const;
}

```

The member functions of this base class essentially represent application specific functionality that the kernel expects from a derived application. The kernel works polymorphically on a pointer of type *PhlexApplication*, invoking application specific functions.

The `main()` is written by the application, as in

```

int main()
{
    Phlex_Application * app = new ResSimulator ();
    Phlex_Kernel k();

    k.run(app);
}

```

9.6 An Example: Heat Conduction Application

This is an example of a nonlinear heat conduction application built using PHLEX2000. The partial differential equation is a non-linear Poisson's equation derived from Fourier's law of heat conduction,

$$q_n = -k \frac{\partial u}{\partial n} = -k(\nabla u \cdot \mathbf{n}) \quad (9.1)$$

where:

- q_n -is the normal heat flux (the rate of heat flow per unit area).
- k -is the thermal conductivity of the material.
- $\mathbf{n}$ -is the unit vector normal to the surface area.
- u -is the temperature.

Now if we examine the net heat flowing out through the faces of a differential cube, we find that:

$$-\nabla \cdot (k \nabla u) = Q \quad (9.2)$$

where:

Q is the rate at which heat is being generated per unit volume inside the differential cube.

The nonlinearity introduced in this problem comes from assuming that the thermal conductivity is a nonlinear function of temperature, i.e.

$$k = \hat{k}u(x, y, z)$$

or

$$q_n = -\hat{k}u \frac{\partial u}{\partial n} = -\hat{k}u(\nabla u \cdot n).$$

Substituting this temperature dependent conductivity into Eq. (9.2) gives

$$-\nabla \cdot (\hat{k}u \nabla u) = Q \quad (9.3)$$

where Q is a volumetric heat source.

The boundary conditions that we will use here are the prescribed temperatures on Γ_u

$$u = \hat{u}(x, y, z)$$

and prescribed heat flux on Γ_t

$$-\hat{k}u \frac{\partial u}{\partial n} = q_0(x, y, z).$$

We describe here the development of this application in a step by step manner in phlex2000.

The application needs to derive from and provide concrete member function implementations for following classes:

1. Application
2. PDE
3. NaturalBC
4. EssentialBC

9.6.1 Application Class

```
class HeatApp : public Phlex_Application {
public:
    static Phlex_Application* createHeatApp(const String& name);
    ~HeatApp();

    void setupFields(Phlex_Domain* domain);
    void setupBCs(Phlex_Domain* domain);
    void ReadBCs();
    void ReadMaterials();
    void Init();
    void Driver(Phlex_Domain* domain);
    void ErrorEst(Phlex_Domain* domain);
    void Adapter(Phlex_Domain* domain);
    void Gui();
    void Postproc();

protected:
    HeatApp();
}
```

setupFields Implementation

```
void
HeatApp::
setupFields(Phlex_Domain* domain)
{
    // set units
    Phlex_Unit unit_len = Phlex_Unit::length;

    // Component Description
    Phlex_ComponentDesc* temp_comp =
        new Phlex_ComponentDesc("Temperature", unit_len);

    Phlex_Vector<Phlex_ComponentDesc*>* temp_desc =
        new Phlex_Vector<Phlex_ComponentDesc*>(1);
```

```

    (*temp_desc)[0] = temp_comp;

    // SubField Description
    Phlex_FESubField* temp_field =
        new Phlex_FESubField("Temperature Field", temp_desc);

    Phlex_Vector<Phlex_FESubField*>* sub_fields =
        new Phlex_Vector<Phlex_FESubField*>(1);
    (*sub_fields)[0] = temp_field;

    // Set up the solution field
    Phlex_FEField* soln_field =
        new Phlex_FEField("Solution Field", sub_fields);

    domain->field (soln_field);
}

setupBCs Implementation

void
HeatApp::
setupBCs(Phlex_Domain* domain)
{
    Phlex_BCtype* bctype;

    // we have only two types of BCs for now

    bctype = new Phlex_BCtype("Neumann", new PneumanBcoeff(),
                               new PneumanBCdata(), Phlex_BCtype::NATURAL);
    domain->addBC(bctype);

    DirichBcoeff* bcoeff = new DirichBcoeff();
    Phlex_Vector<int> dep_vec(1);
    dep_vec[0] = 0;
    bcoeff->dependencies(dep_vec);

    bctype = new Phlex_BCtype("Dirichlet", bcoeff,

```

```

                                new DirichBCdata(), Phlex_BCtype::ESSENTIAL);
    domain->addBC(bctype);
}

```

Driver Implementation

The driver essentially sets up the PDEs and the solution process. In this case, the solver has this structure:

```

void
HeatApp::
Driver(Phlex_Domain* domain)
{
    HeatPDEcoeff pde_coeff;
    Phlex_Vector<int> dep_vec(1);
    dep_vec[0] = 0;
    pde_coeff.dependencies(dep_vec);

    Phlex_PDE* pde = new Phlex_PDE(pde_coeff, *(domain->field()));
    pde->addBC("Neumann");
    pde->addBC("Dirichlet");

    // specify initial solution
    Phlex_FEField::SpaceType type = Phlex_FEField::CONTINUOUS;
    int pv = pde->primary_variable(type);
    Phlex_FieldVector* soln = domain->fieldVector();
    Phlex_GlobalVector* u = soln->extract(pv,false,type);
    *u = 1;
    soln->replace(*u,pv,type);
    delete u;

    int maxiter = 20; // max no of nonlinear iterations
    Double tol = 1.0e-5; // convergence tolerance

    // use dense linear solver
    Phlex_DirectSolver dense;

```

```

    // use newton nonlinear solver
    Phlex_NewtonNL newton(&dense);
    newton.max_iterations(maxiter);
    newton.convergence_tolerance(tol);
    newton.initialGuess(Phlex_NonLinearSolver::CURRENT_SOLUTION);

    // set up the solution process
    Phlex_SolutionProcess sp(pde, domain, &newton);

    // fire the solver
    sp.solve();

    // accept the solution (vector)
    soln->updateDatabase();

    delete pde;
}

```

9.6.2 PDEcoeff Class

```

class HeatPDEcoeff : public Phlex_PDEcoeff {
public:
    HeatPDEcoeff();
    HeatPDEcoeff& operator=(const HeatPDEcoeff& cp);
    HeatPDEcoeff(const HeatPDEcoeff& cp);
    ~HeatPDEcoeff();

    void eval(const Phlex_IntegrationPointData& ipd,
              const Phlex_DataVectorIDs& dv,
              const Phlex_TopoEntityFull& el,
              const PDEcoeffType request,
              const PDEcoeffQualifier qualifier);

    void set_matrix_haves(const PDEcoeffType request,
                          const Phlex_PDEcoeff::PDEcoeffQualifier
                          qualifier = NONE);
    void set_vector_haves(const PDEcoeffType request,

```

```

        const Phlex_PDEcoeff::PDEcoeffQualifier
            qualifier = NONE);
};

```

Here's the implementation:

```

void
HeatPDEcoeff::
eval(const Phlex_IntegrationPointData& ipd,
     const Phlex_DataVectorIDs& dv,
     const Phlex_TopoEntityFull& el,
     const PDEcoeffType request,
     const PDEcoeffQualifier qualifier)
{
    Double k = 1.0;
    Double q = 0.0;

    // get current solution vector
    int iu = dv.soln();
    Double Ulast = ipd.solution(iu,0,0,0,0);

    switch (matrix_request(request)) {
    case NO_REQUEST:
        break;
    case STIFFNESS:
        matrix_[1][1](0,0) = k*Ulast;
        matrix_[2][2](0,0) = k*Ulast;
        if (qualifier == TANGENT) {
            Double Uxlast = ipd.solution(iu,0,0,0,1); // x deriv
            Double Uylast = ipd.solution(iu,0,0,0,2); // y deriv
            matrix_[1][0](0,0) = k * Uxlast;
            matrix_[2][0](0,0) = k * Uylast;
        }
        break;
    default:
        cerr << "matrix_request = " << matrix_request(request) << endl;
        Phlex_Error(1,0,"HeatPDEcoeff:: cannot evaluate this matrix\n");
    }
}

```

```

};

switch (vector_request(request)) {
case NO_REQUEST:
    break;
case SOURCE:
    vector_[0](0) = q;
    break;
case FUNCTIONAL:
    {
        Double Uxlast = ipd.solution(iu,0,0,0,1); // x deriv
        Double Uylast = ipd.solution(iu,0,0,0,2); // y deriv
        vector_[1](0) = k * Ulast * Uxlast;
        vector_[2](0) = k * Ulast * Uylast;
    }
    break;
default:
    cerr << "vector_request = " << vector_request(request) << endl;
    Phlex_Error(1,0,"HeatPDEcoeff:: cannot evaluate this vector\n");
};
}

void
HeatPDEcoeff::
set_matrix_haves(const PDEcoeffType request,
                  const Phlex_PDEcoeff::PDEcoeffQualifier qualifier)
{
    int i, j;
    for(i=0; i<4; i++) {
        for(j=0; j<4; j++) {
            have_a_[i][j] = false;
        }
    }
    if (matrix_request(request) == STIFFNESS) {
        have_a_[1][1] = true;
        have_a_[2][2] = true;
        if (qualifier == TANGENT) {

```



```

        have_a_[1][0] = true;
        have_a_[2][0] = true;
    }
}

void
HeatPDEcoeff::
set_vector_haves(const PDEcoeffType request,
                  const Phlex_PDEcoeff::PDEcoeffQualifier qualifier)
{
    for(int i=0; i<4; i++)
        have_f_[i] = false;

    switch (vector_request(request)) {
    case SOURCE:
        have_f_[0] = true;
        break;
    case FUNCTIONAL:
        have_f_[1] = true;
        have_f_[2] = true;
        break;
    default:
        break;
    }
}

```

9.6.3 BCs

Natural BCs

Definitions:

```

class PneumanBcoeff : public Phlex_PDEcoeff {
public:
    PneumanBcoeff();
    PneumanBcoeff& operator=(const PneumanBcoeff& cp);

```

```

PneumanBcoeff(const PneumanBcoeff& cp);
~PneumanBcoeff();

void eval(const Phlex_IntegrationPointData& ipd,
          const Phlex_DataVectorIDs& dv,
          const Phlex_TopoEntityFull& el,
          const PDEcoeffType request,
          const PDEcoeffQualifier qualifier);

void set_matrix_haves(const PDEcoeffType request,
const Phlex_PDEcoeff::PDEcoeffQualifier
qualifier = NONE);
void set_vector_haves(const PDEcoeffType request,
const Phlex_PDEcoeff::PDEcoeffQualifier
qualifier = NONE);

protected:
private:
};

class PneumanBCdata : public Phlex_BCdata {
public:
    PneumanBCdata();
    PneumanBCdata(const PneumanBCdata& orig);
    PneumanBCdata& operator=(const PneumanBCdata& orig);
    ~PneumanBCdata();

    Phlex_BCdata* create_data(const Phlex_Vector<double>& input_data) const;
    void eval(const Phlex_IntegrationPointData* ipd,
              const Phlex_TopoEntity* te,
              const Phlex_TopoEntity* sub_te);
    void print() const;
    Double value(int icomp) const;

protected:
    void set_values(const Phlex_Vector<double>& input_data);

```

```
private:
    Phlex_NumVecDouble values_;
};
```

Implementation:

```
void
PneumanBcoeff::
eval(const Phlex_IntegrationPointData& ipd,
     const Phlex_DataVectorIDs& dv,
     const Phlex_TopoEntityFull& el,
     const PDEcoeffType request,
     const PDEcoeffQualifier qualifier)
{
    PneumanBCdata* data = (PneumanBCdata*)ipd.bcddata();

    switch (vector_request(request)) {
    case NO_REQUEST:
        break;
    case SOURCE:
        {
            // The data is assumed to be supplied in n-t (0-1) system for
            // the boundary, but kernel needs in X-Y.
            const Phlex_NumMatDouble& vec = ipd.direction_vector();
            vector_[0](0) = - vec(0,0)*data->value(0);
            vector_[0](1) = - vec(0,1)*data->value(0);
        }
        break;
    case FUNCTIONAL:
        {
            const Phlex_NumMatDouble& vec = ipd.direction_vector();
            vector_[0](0) = vec(0,0)*data->value(0);
            vector_[0](1) = vec(0,1)*data->value(0);
        }
        break;
    default:
        cerr << "vector_request = " << vector_request(request) << endl;
```

```
        Phlex_Error(1,0,"PneumanBcoeff:: cannot evaluate this vector\n");
    };
}

void
PneumanBcoeff::
set_matrix_haves(const PDEcoeffType request,
    const Phlex_PDEcoeff::PDEcoeffQualifier qualifier)
{
    int i, j;
    for(i=0; i<4; i++) {
        for(j=0; j<4; j++) {
            have_a_[i][j] = false;
        }
    }
}

void
PneumanBcoeff::
set_vector_haves(const PDEcoeffType request,
    const Phlex_PDEcoeff::PDEcoeffQualifier qualifier)
{
    for(int i=0; i<4; i++)
        have_f_[i] = false;

    switch (vector_request(request)) {
    case SOURCE:
    case FUNCTIONAL:
        have_f_[0] = true;
        break;
    default:
        break;
    }
}

Phlex_BCdata*
PneumanBCdata::

```

```
create_data(const Phlex_Vector<double>& input_data) const
{
    PneumanBCdata* data = new PneumanBCdata();
    data->set_values(input_data);
    return data;
}

void
PneumanBCdata::
eval(const Phlex_IntegrationPointData* ipd,
      const Phlex_TopoEntity* te,
      const Phlex_TopoEntity* sub_te)
{
    // constant values, nothing to be done
}

void
PneumanBCdata::
set_values(const Phlex_Vector<double>& input_data)
{
    assert(input_data.size() >= 1);
    values_.resize(1);
    values_[0] = input_data[0];
}

Double
PneumanBCdata::
value(int icomp) const
{
    return values_[icomp];
}

void
PneumanBCdata::
print() const
{
    cout << "PneumanBCdata: values = " << values_[0]
```

```

    << " " << values_[1] << endl;
}

```

Essential BCs

Definitions:

```

class DirichBcoeff : public Phlex_PDEcoeff {
public:
    DirichBcoeff();
    DirichBcoeff& operator=(const DirichBcoeff& cp);
    DirichBcoeff(const DirichBcoeff& cp);
    ~DirichBcoeff();

    void eval(const Phlex_IntegrationPointData& ipd,
              const Phlex_DataVectorIDs& dv,
              const Phlex_TopoEntityFull& el,
              const PDEcoeffType request,
              const PDEcoeffQualifier qualifier);

    void set_matrix_haves(const PDEcoeffType request,
                          const Phlex_PDEcoeff::PDEcoeffQualifier
                          qualifier = NONE);
    void set_vector_haves(const PDEcoeffType request,
                          const Phlex_PDEcoeff::PDEcoeffQualifier
                          qualifier = NONE);

protected:
private:
};

class DirichBCdata : public Phlex_BCdata {
public:
    DirichBCdata();
    DirichBCdata(const DirichBCdata& orig);
    DirichBCdata& operator=(const DirichBCdata& orig);
    ~DirichBCdata();
}

```

```

Phlex_BCdata* create_data(const Phlex_Vector<double>& input_data) const;
void eval(const Phlex_IntegrationPointData* ipd,
          const Phlex_TopoEntity* te,
          const Phlex_TopoEntity* sub_te);
void print() const;
Double value(int icomp) const;

protected:
    void set_values(const Phlex_Vector<double>& input_data);

private:
    Phlex_NumVecDouble values_;
};

```

Implementation:

```

void
DirichBCcoeff::
eval(const Phlex_IntegrationPointData& ipd,
     const Phlex_DataVectorIDs& dv,
     const Phlex_TopoEntityFull& el,
     const PDEcoeffType request,
     const PDEcoeffQualifier qualifier)
{
    DirichBCdata* data = (DirichBCdata*)ipd.bcddata();

    switch (matrix_request(request)) {
    case NO_REQUEST:
        break;
    case STIFFNESS:
        matrix_[0][0](0,0) = 1.0;
        break;
    default:
        cerr << "matrix_request = " << matrix_request(request) << endl;
        Phlex_Error(1,0,"HeatPDEcoeff:: cannot evaluate this matrix\n");
    };
}

```

```
switch (vector_request(request)) {
case NO_REQUEST:
    break;
case SOURCE:
    vector_[0](0) = data->value(0);
    break;
case FUNCTIONAL:
    {
        int iu = dv.soln();
        Double Ulast = ipd.solution(iu,0,0,0,0);
        vector_[0](0) = Ulast - data->value(0);
    }
    break;
default:
    cerr << "vector_request = " << vector_request(request) << endl;
    Phlex_Error(1,0,"DirichBcoeff:: cannot evaluate this vector\n");
};
}

void
DirichBcoeff::
set_matrix_haves(const PDEcoeffType request,
    const Phlex_PDEcoeff::PDEcoeffQualifier qualifier)
{
    int i, j;
    for(i=0; i<4; i++) {
        for(j=0; j<4; j++) {
            have_a_[i][j] = false;
        }
    }
    if (matrix_request(request) == STIFFNESS)
        have_a_[0][0] = true;
}

void
DirichBcoeff::
```



```
set_vector_haves(const PDEcoeffType request,
  const Phlex_PDEcoeff::PDEcoeffQualifier qualifier)
{
  for(int i=0; i<4; i++)
    have_f_[i] = false;

  switch (vector_request(request)) {
  case SOURCE:
  case FUNCTIONAL:
    have_f_[0] = true;
    break;
  default:
    break;
  }
}

Phlex_BCdata*
DirichBCdata::
create_data(const Phlex_Vector<double>& input_data) const
{
  DirichBCdata* data = new DirichBCdata();
  data->set_values(input_data);
  return data;
}

void
DirichBCdata::
eval(const Phlex_IntegrationPointData* ipd,
      const Phlex_TopoEntity* te,
      const Phlex_TopoEntity* sub_te)
{
  // constant values, nothing to be done
}

void
DirichBCdata::
set_values(const Phlex_Vector<double>& input_data)
```

```
{
    assert(input_data.size() >= 1);
    values_.resize(1);
    values_[0] = input_data[0];
}

Double
DirichBCdata::
value(int icomp) const
{
    return values_[icomp];
}

void
DirichBCdata::
print() const
{
    cout << "DirichBCdata: values = " << values_[0] << endl;
}
```

Chapter 10

Finite Element Solvers

10.1 General

Solvers are based on a layered design (figure 10.1) which represents various levels of abstraction. This structure allows easy addition of new solver algorithms since the lower level abstractions necessary to express them are already provided. Classes such as LHS, RHS, LHSXVector are the components required to code any linear equations solver. These components fully encapsulate all the complexity of the finite element engine and clients of these classes need to deal only with the properties of these objects.

Figure 4.3 shows the solver class hierarchy in Booch notation. The hierarchy is rooted in the abstract base class Phlex_EqnSolver (please note that the prefix Phlex_ hasn't been shown in the class diagram for clarity though, in actual code, it is prepended to each of the class names in the figure). This class presents the most generic interface in terms of Domain and PDE classes.

This design allows easy extension of solver category by derivation from the abstract classes provided. Reuse is evident in many cases from the class diagram. For example, the NonLinearSolver class uses an instance of LinearSolver class (by reference). Since the LinearSolver class is abstract, it means that at execution time, any of the derived linear solvers may be used with any derivative of non-linear solver. Similarly, a TransientSolver can use either a NonLinearSolver or a LinearSolver instance.

Figure 10.1: Layered Design of Finite Element Solvers

10.1.1 Basic Solver Components

There are following classes of basic solver components:

- Lhs
- Rhs
- LhsRhs
- LhsXVector
- Residual

Here are some examples of where these abstractions are extremely useful:

- Dense Solver: uses LhsRhs
- Conjugate Gradient Solver: uses Residual and LhsXVector
- Preconditioner: uses Lhs
- Nonlinear GMRES: uses Function

10.1.2 Finite Element Integration

In the finite element method, PDEs are integrated over the finite elements using numerical integration rules; these element level integrals, via an assembly procedure, provide appropriate contributions to the global integral over the entire domain. We have identified and encapsulated the abstractions involved in this process to provide high reuse and flexibility. These abstractions are:

1. The type of integral to be evaluated e.g., scalar, vector, etc. There are two class hierarchies representing this abstraction, rooted in the abstract base classes *Phlex_GlobalIntegral* and *Phlex_EBEIntegral* (figure 10.2).
2. The integration rule to be used. The abstract base class representing this abstraction is *Phlex_Quadrature*. PHLEX2000 defines two derived classes – *Phlex_NewtonCotesQuadrature* and *Phlex_GaussLegendreQuadrature* for integration rules.

Figure 10.2: Global and Element-by-element Integrals

The (concrete) class `FEIntegrator` encapsulates the (complex) functionality of evaluation of a finite element integral. The class uses objects of various disparate classes like *Domain*, *PDE*, *SolverCoeff*, *Quadrature* and *GlobalIntegral*. Clients of this class can ask for evaluation of the integral over interior of the domain or over different portion of the domain boundaries (e.g., those containing either natural or essential boundary conditions). The interface of the `FEIntegrator` is simple:

```
class Phlex_FEIntegrator : public Phlex_Object {
public:
    Phlex_FEIntegrator(const Phlex_Domain* domain,
                      const Phlex_PDE* pde,
                      Phlex_SolverCoeff* solverCoef,
                      Phlex_SolverCoeff::SolverCoeffType request,
                      const Phlex_Quadrature* quadrature,
                      Phlex_GlobalIntegral* integral);
    ~Phlex_FEIntegrator();
    void eval_interior();
    void eval_natural_bcs();
    void eval_essential_bcs();
};
```

Here's some implementation detail for the interior integral evaluation:

```
void
Phlex_FEIntegrator::
eval_interior()
{
    Phlex_Mesh* mesh = domain_->Mesh();
    Phlex_PDEcoeff& pdeCoef = pde_->Coeff();
    solverCoef_->set_matrix_haves(pdeCoef, request_);
    solverCoef_->set_vector_haves(pdeCoef, request_);
    CoeffData coef(&pdeCoef, solverCoef_, request_);

    Phlex_FEField::SpaceType pv_type;
    int pv = pde_->primary_variable(pv_type);
```

```

// set the no of components
Phlex_TopoEntity::TFType tf_type = Scope::TF_Type;

const Phlex_FESubField* subfield = domain_>field()->sub_field(pv,pv_type);
Phlex_IntegrationPointData ipd(domain_>field());
Phlex_EBEIntegral* ebe = globalIntegral_>create_ebe();

const Phlex_Element* elem;
int n_dofs;
Phlex_TopoFunction* tf;
Phlex_NumVecDouble xez(3);
Phlex_NumVecDouble values;
Phlex_NumMatDouble derivs;
Phlex_Vector<Phlex_DOF *> dofs;

mesh->resetElements();
while ((elem = mesh->nextElement()) != 0) {
    tf = ((Phlex_TopoEntityFull*)*elem)->topo_function(tf_type);
    n_dofs = tf->basis_functions ((Phlex_TopoEntityFull*)*elem, tf_type,
                                Phlex_TopoFunction::DOFS,
                                xez, values, derivs, dofs);

    ebe->reset();
    ebe->dofs(dofs);
    elem->integrate(*subfield, coef, quadrature_, ipd, *ebe);

    // add element contribution to global
    *globalIntegral_ += *ebe;
}
delete ebe;
}

```

10.1.3 Finite Element Assembly

A finite element assembler performs integration over a domain (using *FEIntegrator* objects) so as to produce the integral in the form required by the clients of this class. For example, a linear solver may use a *MatrixAssembler* object to generate a system of equations in matrix form by integration over

a domain. The class diagram for assembler hierarchy is shown in figure 10.3. The abstract base class *Phlex_FEAssembler* has the following interface:

```
public:
    Phlex_FEAssembler(const Phlex_Domain* domain,
                      const Phlex_PDE* pde,
                      Phlex_SolverCoeff::SolverCoeffType request);
    virtual ~Phlex_FEAssembler();
    virtual void assemble(solverFlags::IntegralActionType action);

protected:
    virtual Phlex_GlobalIntegral* integral() = 0;
};
```

Matrix assembly is special since there are various ways of storing the matrix due to considerations of efficiency and performance. *Phlex_MatrixAssembler* is an abstract class derived from *Phlex_FEAssembler* which needs to be specialized (after derivation) for different matrix types. Not surprisingly, there is a parallel hierarchy for different matrix types as shown in figure 10.4. A *DenseAssembler*, for example, uses an object of type *DenseMatrix* to deal with the integral.

10.2 Linear Solvers

Linear equation solvers form part of the core of any finite element simulator. In *Phlex2000*, the class *Phlex_LinearEqnSolver* forms the abstract base of all linear solvers and represents following behavior:

1. generation of a system of linear equations from the underlying finite element discretization.
2. solution of the system.
3. updating the solution space of the PDE and domain.

As the hierarchy in figure 4.3 shows, both iterative and direct linear solvers are derived from this base class. We will now describe in some detail

Figure 10.3: Finite Element Assemblers

Figure 10.4: Finite Element Matrices

```

void
Phlex_DirectSolver::
solve(Phlex_PDE* pde, Phlex_Domain* domain)
{
    Phlex_FEFIELD::SpaceType pv_type;
    int pv = pde->primary_variable(pv_type);
    Phlex_FieldVector* fv = domain->fieldVector();

    int ncomp = domain->field()->sub_field(pv, pv_type)->n_values();
    Scope::Field_Type = pv_type;
    Phlex_MatrixAssembler* assembler;
    Phlex_AlgebraicEqnSolver* eqnSolver;
    Phlex_SolverCoeff* coef = createSolverCoeff(ncomp);

    if (coef) coef->Push();
    assembler = rep->createAssembler(pde, domain);

    Phlex_LhsRhs LhsRhs(*assembler);
    LhsRhs.eval();

    if (coef) {coef->Pop(); delete coef;}
    eqnSolver = rep->createEqnSolver(LhsRhs.matrix());
    eqnSolver->factorize();

    Phlex_GlobalVector* rhs = LhsRhs.vector();
    Phlex_GlobalVector solution(*rhs);
    eqnSolver->backsolve(*rhs, &solution);

    // transfer this vector to domain's field vector
    fv->replace(solution,pv,pv_type);

    delete assembler;
    delete eqnSolver;
}

```

Figure 10.5: Implementation of DirectSolver

the implementation of base class *Phlex_DirectSolver* in terms of the abstractions provided by the kernel (refer to figure 10.5).

As shown earlier, the layered architecture of PHLEX2000 makes expression of the *DirectSolver* quite simple since most of the finite element related operations is abstracted away into the basic solver components – LHS and RHS. In this case, we use objects of class *LhsRhs* for efficiency, though LHS/RHS would have been equally valid (but less efficient).

It is clear from figure 10.5 that the algorithm for direct solver is not dependent on any details specific to special kind of assemblers or matrices. *DirectSolver* is an abstract class and some of the methods used in describing the above algorithm are implemented by classes derived from *DirectSolver*. For example, the derived class *DenseSolver* provides implementation of *createAssembler* and *createEqnSolver* to create objects special to its structure. The class diagram in figure 10.6 gives the static description of a dense linear equations solver. It captures most of the class dependencies.

10.3 NonLinear Solvers

The base class *Phlex_NonLinearSolver* provides the basic nonlinear equation solving mechanism through the implementation of *solve* method. The class is abstract and concrete nonlinear solvers derived from it are required to supply specific implementations of the protected pure virtual methods used in describing the general solution algorithm. The interface of the *Phlex_NonLinearSolver* class is given below:

```
class Phlex_NonLinearSolver : public Phlex_MatrixEqnSolver {
public:
    enum InitialGuessType {ZERO, CURRENT_SOLUTION};

    Phlex_NonLinearSolver(Phlex_LinearSolver* linearSolver,
                          Phlex_NonLinearAppHook* appHook = 0);
    Phlex_NonLinearSolver(Phlex_NonLinearAppHook* appHook = 0);

    virtual ~Phlex_NonLinearSolver();
    virtual void solve(Phlex_PDE* pde, Phlex_Domain* domain);
    virtual void solve(Phlex_PDE* pde, Phlex_Domain* domain,
```

Figure 10.6: Class Diagram for DenseSolver

```

        Phlex_TransientMatrixSolver* transient);
virtual void solve(Phlex_Vector<Phlex_SolutionProcess*>* strategy);

void initialGuess(InitialGuessType gtype);
InitialGuessType InitialGuess();
Phlex_LinearSolver* linearSolver();
void max_iterations(int maxIter);
int max_iterations();
void convergence_tolerance(Double tol);
Double convergence_tolerance();
int iterations();

protected:
virtual void initialize(Phlex_PDE* pde, Phlex_Domain* domain) = 0;
virtual void pre_linearSolve_eval(Phlex_PDE* pde, Phlex_Domain* domain) = 0;
virtual void post_linearSolve_eval(Phlex_PDE* pde, Phlex_Domain* domain) = 0;
virtual bool isConverged() = 0;
virtual void finish(Phlex_PDE* pde, Phlex_Domain* domain) = 0;

void soln_label(int label);
int soln_label() const;
};

```

Here's the *Phlex_NonLinearSolver*'s implementation of the *solve*:

```

Phlex_NonLinearSolver::
solve(Phlex_PDE* pde, Phlex_Domain* domain)
{
    Phlex_LinearSolver* linsol = linearSolver();
    Phlex_FieldVector* fv = domain->fieldVector();
    bool fv_status = fv->isActive();
    fv->isActive(true); // make the FieldVector active for use by coeffs
    soln_label(fv->label()); // save the label for use by coeffs

    int iters_ = 0;
    Phlex_NonLinearAppHook* appHook =
        static_cast<Phlex_NonLinearAppHook*> (this->appHook());
}

```

```

Phlex_FEFIELD::SpaceType pv_type;
int pv = pde->primary_variable(pv_type);
int ncomp = domain->field()->sub_field(pv, pv_type)->n_values();
Phlex_SolverCoeff* coef = createSolverCoeff(ncomp);

if (coef) coef->Push();
initialize(pde, domain);
while (!isConverged() && iters_ < max_iterations()) {
    cout << "Iteration: " << iters_ << endl;
    if(appHook) appHook->before();
    pre_linearSolve_eval(pde, domain);
    linsol->solve(pde, domain);
    post_linearSolve_eval(pde, domain);
    if(appHook) appHook->after();
    ++iters_;
}
finish(pde, domain);
if (coef) {coef->Pop(); delete coef;}

if (isConverged()) {
    cout << "\nNonlinear Solver converged in " << iters_ << " iterations\n";
}
else {
    cout << "\nNonLinear Solver couldn't converge in "
        << iters_ << " iterations\n";
}

fv->updateDatabase();
fv->isActive(fv_status);
}

```

Two nonlinear solvers have been provided in the kernel – Picard and Newton, whose descriptions follow:

10.3.1 Picard Nonlinear Solver

```

class Phlex_PicardNL : public Phlex_NonLinearSolver {
public:
    Phlex_PicardNL(Phlex_LinearSolver* linearSolver = 0,
        Phlex_NonLinearAppHook* appHook = 0);
    ~Phlex_PicardNL();

protected:
    void initialize(Phlex_PDE* pde, Phlex_Domain* domain);
    void pre_linearSolve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
    void post_linearSolve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
    void finish(Phlex_PDE* pde, Phlex_Domain* domain);
    bool isConverged();

    Phlex_SolverCoeff* createSolverCoeff(int ncomp);

    class PicardCoeff : public Phlex_SolverCoeff {
    public:
        PicardCoeff(Phlex_PicardNL& solver, int ncomp);
        void eval(Phlex_PDEcoeff& pdeCoeff,
            SolverCoeffType request,
            const Phlex_IntegrationPointData& ipd,
            const Phlex_TopoEntityFull& el,
            const Phlex_PDEcoeff::PDEcoeffQualifier qualifier
                = Phlex_PDEcoeff::NONE);
        void set_matrix_haves(Phlex_PDEcoeff& pdeCoeff,
            SolverCoeffType request,
            const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
        void set_vector_haves(Phlex_PDEcoeff& pdeCoeff,
            SolverCoeffType request,
            const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
    };
};

```

10.3.2 Newton Nonlinear Solver

```

class Phlex_NewtonNL : public Phlex_NonLinearSolver {
public:
    Phlex_NewtonNL(Phlex_NonLinearAppHook* appHook = 0,
        bool full_formulation = false);

    Phlex_NewtonNL(Phlex_LinearSolver* linearSolver,
        Phlex_NonLinearAppHook* appHook = 0);

    Phlex_NewtonNL(bool full_formulation,
        Phlex_LinearSolver* linearSolver,
        Phlex_NonLinearAppHook* appHook = 0);
    ~Phlex_NewtonNL();

protected:
    void initialize(Phlex_PDE* pde, Phlex_Domain* domain);
    void pre_linearSolve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
    void post_linearSolve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
    void finish(Phlex_PDE* pde, Phlex_Domain* domain);
    bool isConverged();

    Phlex_SolverCoeff* createSolverCoeff(int ncomp);

    class NewtonCoeff : public Phlex_SolverCoeff {
    public:
        NewtonCoeff(Phlex_NewtonNL& solver, int ncomp);
        void eval(Phlex_PDEcoeff& pdeCoeff,
            SolverCoeffType request,
            const Phlex_IntegrationPointData& ipd,
            const Phlex_TopoEntityFull& el,
            const Phlex_PDEcoeff::PDEcoeffQualifier qualifier
                = Phlex_PDEcoeff::NONE);
        void set_matrix_haves(Phlex_PDEcoeff& pdeCoeff,
            SolverCoeffType request,
            const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
        void set_vector_haves(Phlex_PDEcoeff& pdeCoeff,

```

```

    SolverCoeffType request,
    const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
};
};

```

10.4 Transient Solvers

The transient solvers implement the policies of time integration corresponding to a particular algorithm. The functionality of advancing in time has been abstracted away into a *TimeMarcher* class and so the *TransientSolver* class needs to use an instance of that class. We have implemented two time integration algorithms in the kernel — Newmark- β and Euler. Their descriptions follow:

10.4.1 The Time Marcher

```

/*
  This class encapsulates the logic of marching in time.
  An object of this class is to be used as follows:

  Phlex_TimeMarcher tm(0.0, 14.5, 0.3);
  tm.begin();
  while (! tm.end()) {
    tm.step();
    if (tm.currentTime() > 2.5) {
      cout << "currentTime = " << tm.currentTime() << endl;
      cout << "step size = " << tm.currentStepSize() << endl;
    }
  }
*/

class Phlex_TimeMarcher : public Phlex_Object {
public:
    typedef Double Time;

    Phlex_TimeMarcher();

```

```

Phlex_TimeMarcher(Time Tbegin, Time Tend);
Phlex_TimeMarcher(Time Tbegin, Time Tend, Time Tstep);
virtual ~Phlex_TimeMarcher();

virtual void begin();
virtual bool end();
virtual void step(); // advances the currentTime
virtual void step(Time stepSize); // for variable step sizes

Time beginTime();
Time endTime();
Time currentTime();
Time currentStepSize();
Time nextTime();
};

```

10.4.2 Newmark- β Time Integration

```

class Phlex_NewmarkBeta : public Phlex_TransientMatrixSolver {
public:
    Phlex_NewmarkBeta(Phlex_TimeMarcher& timeMarcher,
        Phlex_TransientAppHook* appHook = 0);

    Phlex_NewmarkBeta(Phlex_TimeMarcher& timeMarcher,
        Phlex_LinearSolver* solver,
        Phlex_TransientAppHook* appHook = 0);

    Phlex_NewmarkBeta(Phlex_TimeMarcher& timeMarcher,
        Phlex_NonLinearSolver* solver,
        Phlex_TransientAppHook* appHook = 0);

    virtual ~Phlex_NewmarkBeta();

    virtual void predict(Phlex_PDE* pde, Phlex_Domain* domain);
    virtual void correct(Phlex_PDE* pde, Phlex_Domain* domain);

    void beta(Double bta) {beta_ = bta;};

```

```

Double beta() {return beta_;};
void gamma(Double gma) {gamma_ = gma;};
Double gamma() {return gamma_;};

```

protected:

```

virtual void initialize(Phlex_PDE* pde, Phlex_Domain* domain);
virtual void pre_Solve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
virtual void post_Solve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
virtual void finish(Phlex_PDE* pde, Phlex_Domain* domain);

```

```

// computes initial acceleration
virtual void computeInitial(Phlex_PDE* pde, Phlex_Domain* domain,
    Phlex_MatrixEqnSolver* solver);

```

```

Phlex_SolverCoeff* createSolverCoeff(int ncomp);

```

```

class NewmarkBetaCoeff : public Phlex_SolverCoeff {
public:
    NewmarkBetaCoeff(Phlex_NewmarkBeta& solver, int ncomp);
    void eval(Phlex_PDEcoeff& pdeCoeff,
        SolverCoeffType request,
        const Phlex_IntegrationPointData& ipd,
        const Phlex_TopoEntityFull& el,
        const Phlex_PDEcoeff::PDEcoeffQualifier qualifier
            = Phlex_PDEcoeff::NONE);

```

```

    void eval_init_acc(Phlex_PDEcoeff& pdeCoeff,
        SolverCoeffType request,
        const Phlex_IntegrationPointData& ipd,
        const Phlex_TopoEntityFull& el,
        const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);

```

```

    void set_matrix_haves(Phlex_PDEcoeff& pdeCoeff,
        SolverCoeffType request,
        const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
    void set_vector_haves(Phlex_PDEcoeff& pdeCoeff,
        SolverCoeffType request,

```

```

    const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
};
};

```

10.4.3 Euler Time Integration

```

/*
The Euler method uses these relations
  1.  $U(t+a.dt) = a.U(t+dt) + (1-a).U(t)$ 
  2.  $V(t+a.dt) = (U(t+dt) - U(t))/dt$ 
The (first order) equilibrium eqn is written at time (t+a.dt)
       $C.V(t+a.dt) + K.U(t+a.dt) = f$  ... linear
or,    $F(U(t+a.dt), V(t+a.dt)) = 0$  ... nonlinear
*/

class Phlex_Euler : public Phlex_TransientMatrixSolver {
public:
    Phlex_Euler(Double theta,
                Phlex_TimeMarcher& timeMarcher,
                Phlex_TransientAppHook* appHook = 0);

    Phlex_Euler(Double theta,
                Phlex_TimeMarcher& timeMarcher,
                Phlex_LinearSolver* solver,
                Phlex_TransientAppHook* appHook = 0);

    Phlex_Euler(Double theta,
                Phlex_TimeMarcher& timeMarcher,
                Phlex_NonLinearSolver* solver,
                Phlex_TransientAppHook* appHook = 0);

    virtual ~Phlex_Euler();

    Double theta() {return theta_;};

protected:
    void initialize(Phlex_PDE* pde, Phlex_Domain* domain);

```

```
void pre_Solve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
void post_Solve_eval(Phlex_PDE* pde, Phlex_Domain* domain);
void predict(Phlex_PDE* pde, Phlex_Domain* domain);
void correct(Phlex_PDE* pde, Phlex_Domain* domain);
void finish(Phlex_PDE* pde, Phlex_Domain* domain);

Phlex_SolverCoeff* createSolverCoeff(int ncomp);

class EulerCoeff : public Phlex_SolverCoeff {
public:
    EulerCoeff(Phlex_Euler& solver, int ncomp);
    void eval(Phlex_PDEcoeff& pdeCoeff,
              SolverCoeffType request,
              const Phlex_IntegrationPointData& ipd,
              const Phlex_TopoEntityFull& el,
              const Phlex_PDEcoeff::PDEcoeffQualifier qualifier
                = Phlex_PDEcoeff::NONE);
    void set_matrix_haves(Phlex_PDEcoeff& pdeCoeff,
                          SolverCoeffType request,
                          const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
    void set_vector_haves(Phlex_PDEcoeff& pdeCoeff,
                          SolverCoeffType request,
                          const Phlex_PDEcoeff::PDEcoeffQualifier qualifier);
};
```

Chapter 11

Miscellaneous

11.1 FieldVector

A FieldVector is a (discrete) dof (i.e. degree-of-freedom) representation of a field. A FieldVector always represents a complete field. It's properties are:

- A FieldVector can be created only by applying a field on a mesh.
- One cannot create an empty FieldVector, though copies can be made (deep copy).
- FieldVectors are not supposed to be used heavily for computational purposes instead they should be used as a repository from which to fetch or put into the selective portions of a field representation.
- GlobalVectors can be extracted from a FieldVector or a FieldVector can be updated by a GlobalVector.
- A FieldVector can be used to update a domain's database.
- All active FieldVectors are considered for evaluation at int. pts. (as dictated by PDEcoeff objects)
- Each FieldVector has a unique label assigned to it which can be queried.

```
class Phlex_FieldVector : public Phlex_Object {  
public:
```



```

Phlex_FieldVector(const Phlex_FEFld* field, const Phlex_Mesh* mesh);

// following use deep copy
Phlex_FieldVector(const Phlex_FieldVector& orig);
Phlex_FieldVector& operator=(const Phlex_FieldVector& orig);

virtual ~Phlex_FieldVector();
Phlex_FieldVector& operator=(Double value);

// methods for looping over all FieldVectors
static void resetFieldVectors();
static Phlex_FieldVector* nextFieldVector();

// extract methods
Phlex_GlobalVector* extract
    (const Phlex_FESubField* subField, bool readOnly=false) const;
Phlex_GlobalVector* extract
    (int subField_index, bool readOnly=false,
     Phlex_FEFld::SpaceType type=Phlex_FEFld::CONTINUOUS) const;
Phlex_GlobalVector* extract
    (int subField_index, int component_index, bool readOnly=false,
     Phlex_FEFld::SpaceType type=Phlex_FEFld::CONTINUOUS) const;
Phlex_GlobalVector* extract
    (int subField_index, int component_index, int array_index,
     bool readOnly=false,
     Phlex_FEFld::SpaceType type=Phlex_FEFld::CONTINUOUS) const;

// replace by GlobalVector data
void replace(const Phlex_GlobalVector& gv,
            int subField_index,
            Phlex_FEFld::SpaceType type=Phlex_FEFld::CONTINUOUS);
void replace(const Phlex_GlobalVector& gv,
            int subField_index, int component_index,
            Phlex_FEFld::SpaceType type=Phlex_FEFld::CONTINUOUS);
void replace(const Phlex_GlobalVector& gv,
            int subField_index, int component_index, int array_index,
            Phlex_FEFld::SpaceType type=Phlex_FEFld::CONTINUOUS);

```

```

// database communication
void updateDatabase();

// properties of a FieldVector
int label() const;
void isActive(bool trueFalse);
bool isActive() const;

protected:
    Phlex_FieldVector();
};

```

11.2 GlobalVector

A *GlobalVector* is used to get a restricted view of a *FieldVector*. It is mostly used for computations, while a *FieldVector* is mostly used for communication between database and solver. A *GlobalVector* can be created either from a *FieldVector* or by specifying sizes. In the latter case, no special functionality is associated with the vector. Various numerical and mathematical operations can be performed on *GlobalVectors*. This class is optimized for such usage. *ReadOnly* attribute is useful especially in conjunction with *FieldVector*. We save on extra copying when not needed. Each *GlobalVector* maintains links to the *FieldVector* from which it is extracted. A *GlobalVector* has a label which is used to access integration point data by *PDEcoeffs*.

Various solvers extract out *GlobalVectors* from the *FieldVector* (associated with a particular coupling) corresponding to the primary variable of the PDE being solved. Only *GlobalVectors* marked as active are to be processed by *FEIntegrator*.

Instances of this class are used in two different situations:

- as descendants of *GlobalIntegral* during vector integration and
- as free standing global variables in higher level code.

```

class Phlex_GlobalVector : public Phlex_GlobalIntegral {
public:

```

```

Phlex_GlobalVector(int ndof, int ncomp);
Phlex_GlobalVector(const Phlex_GlobalVector& orig, bool readOnly=false);
Phlex_GlobalVector& operator=(const Phlex_GlobalVector& orig);
virtual ~Phlex_GlobalVector();

virtual void reset();
virtual Phlex_GlobalIntegral& operator+=(const Phlex_EBEIntegral& ebe);
Phlex_GlobalVector& operator+=(const Phlex_EBEvector& ebe);
Phlex_NumMatDouble* gather(const Phlex_Vector<int>* ids) const;

virtual Phlex_EBEIntegral* create_ebe(int el_ndof=DEFAULT_ELEM_NDOFS);

Double operator() (int dof, int comp) const;
void operator() (int dof, int comp, Double value);
bool readOnly() const;
int ndof() const;

Phlex_GlobalVector& operator=(Double scalar);

Phlex_GlobalVector operator+(const Phlex_GlobalVector& gv) const;
Phlex_GlobalVector operator-(const Phlex_GlobalVector& gv) const;
Phlex_GlobalVector& operator+=(const Phlex_GlobalVector& gv);
Phlex_GlobalVector& operator-=(const Phlex_GlobalVector& gv);

Phlex_GlobalVector operator*(Double scalar) const;
Phlex_GlobalVector operator/(Double scalar) const;
Phlex_GlobalVector& operator*=(Double scalar);
Phlex_GlobalVector& operator/=(Double scalar);

Double dot(const Phlex_GlobalVector& gv) const;
Double norm_inf() const;
Double norm_L2() const;

Phlex_GlobalVector* extract(int icomp);
void replace(Phlex_GlobalVector* gv, int icomp);

```

protected:

```
Phlex_GlobalVector& scatter(const Phlex_EBEvector& ebe);

// data must be laid out (ndof,ncomp), fortran style
Phlex_GlobalVector(const Double* data, int ndof, int ncomp,
    bool readOnly=false);

// return a copy of data laid out (ndof,ncomp), fortran style
// in user allocated array
void fetchData(Double* data) const;
};
```

Chapter 12

A Prototype Application

Using the prototype kernel developed during this project, several example applications, from solid and fluid mechanics, were built and tested. One of the key aspects that was noted was the ease and compactness with which a fresh non-trivial application could be built on such an object-oriented finite element kernel. Also, some of the new technologies, particularly related to mesh adaptivity were successfully tested.

In this section we show some results from a solid mechanics application built on top of PHLEX2000 kernel. The problem is posed as a plate in axial tension. The structure is assumed fixed at one end. A schematic of the problem is given in Figure 12.1.

The solution to this problem is straightforward except for the singularities at the fixed end. Typically mesh refinement strategies yield improved results for such situations. Starting with an arbitrary coarse mesh as shown in Figure 12.2 and using an *a-posteriori* error estimator, PHLEX2000 adapted the mesh and improved the solution. In Figure 12.3 the contours of transverse displacements are plotted. Figures 12.4 and 12.5 present the final adapted mesh and the corresponding solution, respectively. The mesh refinement near the fixed-end singularities is noteworthy.

]

Figure 12.1: Plate In Axial Tension - Problem Schematic

Figure 12.2: Initial Coarse Mesh

Figure 12.3: Transverse Displacements (Coarse Mesh)

Figure 12.4: Solution-Adaptive Mesh

Figure 12.5: Transverse Displacements (Adapted Mesh)

Bibliography

- [1] Wolfe, M. (Moderator), Banerjee, P., Browne, J., Dennis, J., Dietz, H., Dongarra, J., Koelbel, C., Loveman, D., Nicolau, A. and Otto, S. (Participants), "Panel: Program Portability Across Parallel Architectures SIMD/MIMD/SPMD/Distributed/...", Proc. 8th Intl. Parallel Processing Symp., Cancun, Mexico, pp 658-661, April 26-29, 1994.
- [2] Lo, V.M. and Rajopadhye, S., "OREGAMI: Software Tools for Mapping Parallel Computations to Parallel Architectures", Proc. 1990 Intl. Conf. Parallel Processing, Vol.II, pp 88-92, CRC Press, 1990.
- [3] Gadkari, S., "Layered Encapsulations for Portable Parallel Programming", Proc. 1993 Intl. Conf. Parallel Processing, Vol.II, pp 275-278, CRC Press, 1993.
- [4] Crivelli, S. and Jessup, E.R., "A Programming Paradigm for Distributed-Memory Computers", Proc. Sixth SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, pp 883-886, 1993.
- [5] Dietz, H.G., Cohen, W.E. and Grant, B.K., "Would You Run It Here... Or There? (AHS: Automatic Heterogeneous Computing)", Proc. 1993 Intl. Conf. Parallel Processing, Vol.II, pp 217-221, CRC Press, 1993.
- [6] Krommer, A.R. and Uberhuber, C.W., "Scalable Software Based On Architecture Adaptive Algorithms", pp 840-849, Proc. Sixth SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, Philadelphia, PA, 1993.

- [7] Huang, Y. and Paker, Y., "Data Flow Approach to Topological Independence", *Parallel Processing in Engineering Applications*, R.A.Adey, Editor, Springer-Verlag, pp 1990.
- [8] Hendrickson, Bruce and Leland, Robert, "An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations", SAND92-1460, Sandia National Laboratories, 1992.
- [9] Farhat, C. and Simon, H. D., "TOP/DOMDEC: A Software Tool for Mesh Partitioning and Parallel Processing", Center for Space Structures and Controls, University of Colorado, Boulder, Colorado, May 1993.
- [10] Purify User's Guide, Pure Software Inc., 1309 South Mary Avenue, Sunnyvale, CA 94087.
- [11] LAPACK++ V. 1.0, High Performance Linear Algebra User's Guide, April 1994, Oak Ridge National Laboratory, University of Tennessee, Knoxville.
- [12] STL ToolKit, Version 1.1, ObjectSpace, Inc., 14881 Quorum Drive, Suite 400, Dallas, TX 75240.
- [13] Object-Oriented Analysis and Design, with Applications, 2nd edition, Grady Booch, Addison Wesley, ISBN 0-8053-5340-2
- [14] Design Patterns, Gamma et al, Addison Wesley, ISBN 0-201-63361-2