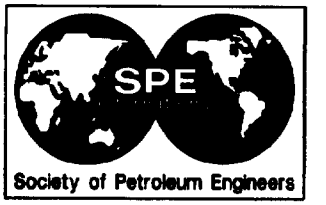

SPE 37977

A New Generation EOS Compositional Reservoir Simulator: Part II – Framework and Multiprocessing

Manish Parashar, John A. Wheeler, Gary Pope, Kefei Wang, Peng Wang
The University of Texas at Austin

This paper was prepared for presentation at the 1997 SPE Reservoir Simulation Symposium held in Dallas, Texas, 8–11 June 1997

Abstract

This paper describes the design and implementation of a Problem Solving Environment (PSE) for developing parallel reservoir simulators that use multiple fault blocks, multiple physical models and dynamic locally adaptive mesh-refinements. The objective of the PSE is to reduce the complexity of building flexible and efficient parallel reservoir simulators through the use of a high-level programming interface for problem specification and model composition, object-oriented programming abstractions that implement application objects, and distributed dynamic data-management that efficiently supports adaptation and parallelism. This work is presented in two parts. In Part I (SPE 37979), we describe the mathematical formulation and discuss numerical solution techniques, while this Part II paper we address framework and multiprocessing issues.

Introduction

The primary goal of the new generation of reservoir simulators currently under development is to support realistic, high-resolution reservoir studies with a million or more grid elements on massively parallel computers. Key requirements for these simulators include the ability to handle multiple physical models, generalized well management, multiple fault blocks, and dynamic, locally adaptive mesh-refinements.

In this paper we describe the design and development of a Problem Solving Environment (PSE) for developing parallel reservoir simulators. Our research is motivated by the multi-disciplinary structure of the new generations simulators and the inherent complexity of their implementation. In a typical reservoir simulator on the order of 20,000 lines of code may be required to support a physical model. It is difficult and inefficient for individual researchers to develop such a framework before even beginning to test their ideas for a physical model. Further, a large percentage of this framework which deals with managing adaptively, multiple fault blocks, parallelism, data distribution, and dynamic load-balancing is not directly related to the physical problem and can be reused. Clearly, a common infrastructure that provides computational support and parallel adaptive data-management will result in enormous savings in development and coding effort and will make individual research at universities and commercial labs more efficient. The PSE presented in this papers provides such an infrastructure. Its primary objectives are:

1. To alleviate the complexity of implementing new reservoir simulators due to parallel data-management, dynamic mesh-refinement, multiple fault blocks, by defining appropriate data-structures, programming abstractions and high-level programming interfaces.
2. To provide a user interface for problem specification and model composition in the form of keyword input.
3. To provide a general framework for integrating input/output, visualization, and interactive experimentation with applications computation.
4. To use the most appropriate language and technology for individual tasks in the PSE. For example, to implement low level data-management in C, to build programming abstractions in an object-oriented language like C++, and to develop computational kernels in FORTRAN.

The PSE design is based on a clean separation of concerns across its functionality and the definition of hierarchical levels of abstraction based on this separation. The overall schematic of the design is shown in Figure 1.

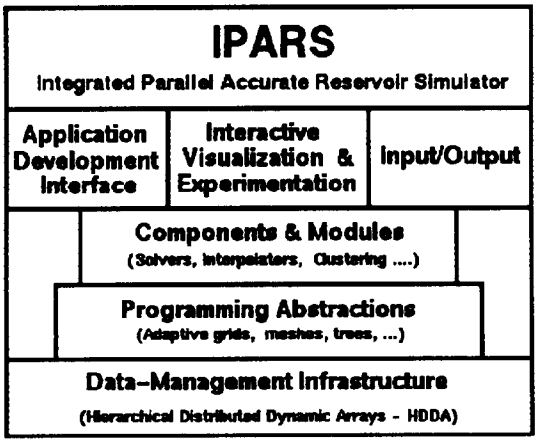

Figure 1. Problem Solving Environment for Parallel Reservoir Simulation

The lowest layer of the PSE is a Hierarchical Distributed Dynamic Array (HDDA). HDDA provides pure array semantics to hierarchical, dynamic and physically distributed data, and has been successfully used as a data-management infrastructure for a variety of parallel adaptive algorithms such adaptive mesh-refinement and multigrid methods, h-p adaptive finite element methods and adaptive multiplole method. HDDA objects encapsulate distribution, dynamic load-balancing, communications, and consistency management. The next layer, programming abstractions, adds application semantics to HDDA objects and implements application objects such as grids, meshes and trees. This layer provides object-oriented programming abstractions that can be used to directly implement parallel adaptive algorithms. The intermediate layers of the PSE implement application specific methods and components. The topmost layer is a high-level application programming interface (API) customized to parallel reservoir simulation. It provides keyword input support for problem and model specification, generalized units conversion, and incorporates a FORTRAN interpreter to allow code fragments as part of the user input data.

The PSE has been implemented by integrating two independent software sub-systems: IPARS, an Integrated Parallel Accurate Reservoir Simulator and DAGH, a data-management infrastructure for Distributed Adaptive Grid Hierarchies. DAGH implements the lower two layers of the PSE and has been extended to support multiple fault blocks. IPARS covers the uppers layers of the PSE and provides the high-level API. Input/output and visualization capabilities have been integrated in to the fundamental data-structures at the lowest level of the PSE. In the following sections we first introduce the IPARS and DAGH subsystems and then describe their integration.

IPARS: Integrated Parallel Accurate Reservoir Simulator

The objective of the IPARS[1] infrastructure is to provide a high-level framework for the development of a new generation of reservoir simulators as well as to support extension and porting of existing simulators.

The IPARS framework supports three-dimensional transient flow of multiple phases containing multiple components plus immobile phases (rock) and adsorbed components. Phase densities and viscosities may be arbitrary functions of pressure and composition or may be represented by simpler functions (e.g. constant compressibility). The initial system is isothermal but an effort will be made later to incorporate nonisothermal calculations.

Porosity at standard conditions and permeability may vary with location in an arbitrary manner. Permeability is initially a diagonal tensor but a full tensor capability may be added later.

Relative permeability, and capillary pressure are functions of saturations and rock type but not directly functions of location. Rock type is a integer function of location. A Stone model for three-phase relative permeability is built into the framework and other similar models can be added as needed. Any dependence of relative permeability and capillary pressure on phase composition and pressure is left to the individual physical models.

The reservoir consists of one or more fault blocks. Each fault block has an independent user-defined coordinate system and gravity vector. Flow between fault blocks, however, can occur only through a common flat face. The primary grid imposed on each fault block is a logical cube but may be geometrically irregular. Currently, the framework supports both rectangular grids and corner-point grids. Dynamic grid refinement of the primary grid on each fault block is supported by the framework but also must be supported by the individual physical models. Grid elements may be keyed out to efficiently represent irregular shapes and impermeable strata.

Our concept for calculating flow between fault blocks has been successfully tested in two and three dimensions and is being incorporated into the framework.

The IPARS framework supports an arbitrary number of wells each with one or more completion intervals. A well may penetrate more than one fault block but a completion interval must occur in a single fault block. On parallel machines, well grid elements may be assigned to more than one processor. The framework assigns primary responsibility for a well calculation to one of these processors and communicates well data between processors. For each well element, the framework also provides estimates of the permeability normal to the wellbore, the geometric constant in the productivity index, and the length of the open interval. Other well calculations are left to the individual physical models. The framework will eventually do well management calculations that include surface facilities but this code is not yet in place. We anticipate a major effort will be required to efficiently carry out well management calculations on multiprocessor machines since these calculations are inherently scalar.

A checkpoint/restart capability is included in the IPARS design. Restart file(s) are independent of the number of processors but may be machine dependent (binary files).

Timestep recalculation is included in the basic design as its need is inevitable and the cost of reprogramming to include the

capability later would be excessive. However, robust numerical techniques rather than timestep cutting should be emphasized.

General purpose 2-dimensional function utilities are provided. The 2-dimensional utilities include:

1. Piecewise constant
2. Piecewise linear
3. Quadratic splines with optional poles
4. Cubic splines with optional poles
5. User define functions (FORTRAN code in the input data).

The N dimensional function utilities may eventually be added.

Free-form keyword input is used for direct data input to the computation stage of the simulator. The keyword input file(s) are explicitly defined to serve as an output file for a graphical front end or geostatistical grid and property generator. The keyword file is an ASCII file.

Multiple levels of output are provided in the simulator. These will range from selective memory dumps for debugging to minimal output for automatic history matching.

Internally the simulator uses a single set of units chosen to minimize multiplication by conversion factors and facilitate interpretation of debug output. Externally, the user may choose any physically correct units; if he or she wants to specify production rates in units of cubic furlongs per fortnight, the simulator will determine and apply the appropriate conversion factor. The simulator also provides a default set of external units.

Implementation Strategy

The overall structure of IPARS consists of 3 layers:

1. Executive Layer
2. Work Routines
3. Data-Management Layer

Executive Layer:

The executive layer consists of routines that direct the overall course of the simulation. These routines do not directly access grid element data or the work routines.

Work Routines:

Work routines are typically FORTRAN subroutines that actually perform grid-element computations. These are invoked indirectly by routines in the executive layer via the data-management interface.

Data-Management Layer:

The data-management layer handles the distribution of grid across processing nodes, the allocation of local storage, communication scheduling, dynamic reallocation and dynamic load-balancing. The data-management layer is also responsible for checkpoint/restart, input/output and visualization.

Two separate implementations of the IPARS framework have been completed. The first uses customized memory management implemented in classical C. In this implementation, the grid system is distributed among the processors such that each processor is assigned a subset of the total grid system. Only one array spans the entire grid system on each processor; this is a two byte integer array that defines division of the grid among processors. The subgrid assigned to a processor is surrounded by a "communication" layer of grid elements that have been assigned to other processors. This layer is one element wide for a 7-point finite difference stencil but may be made wider to support higher-order stencils. The framework provides a routine that updates array data in the communication layer using data from the processors to which elements are actually assigned. This update routine currently supports three finite difference stencils but is organized to support an arbitrary number of stencils. This implementation does not handle adaptive refinement. Collection and organization of all output data, including restart data, is handled by a single processor. The input data file is processed separately by all processors.

The second implementation is built on top of the DAGH object-oriented data-management infrastructure, and uses its functionality to support adaptive refinements, checkpoint/restart, input/output and visualization.

Framework Status

Both implementations of the IPARS framework have been completed and are currently being tested. Supported platforms include high-performance parallel systems like the Cray T3E, IBM SP2, SGI Origin 2000, and Intel Paragon as well networked work-stations.

Testing the IPARS Framework

To test the IPARS framework including memory management and parallel computation, we are porting an existing chemical flooding simulator, UTCHEM[2], into it. We selected UTCHEM largely because of its capability of modeling very complex chemcial processes.

UTCHEM is a general purpose three-dimensional, multicomponent, multiphase compositional chemical flooding simulator that is used for a wide variety of applications by a large number of organizations. On the other hand, UTCHEM cannot handle complex geometries and the standard version does not run on parallel processors. An earlier version was ported to MPPs and problems with a large number of gridblocks successfully run but many new features have been added during the past couple of years. Thus, we decided that porting the current version of UTCHEM to the IPARS framework would make sense.

The porting effort is underway but progressing slowly because of necessary code reorganization. For example, there are 1082 COMMON statements in the program representing 107 distinct COMMON blocks. In addition, UTCHEM was optimized for vector machines and thus uses a single subscript to represent location; IPARS, on the other hand, is intended for irregular geometries and multiple processor machines and thus uses three indexes to represent location. Reorganizing the simulator is requiring a significant effort.

DAGH: Distributed Adaptive Grid Hierarchy

DAGH[3] (Distributed Adaptive Grid Hierarchy) is an object-oriented data-management infrastructure for distributed adaptive computations hierarchical adaptive mesh refinement and multigrid techniques.

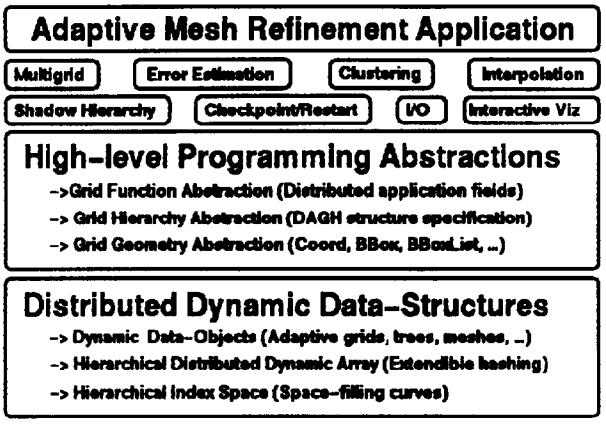

Figure 2 DAGH - An Overview

Figure 2 shows the overall structure of DAGH. It consists of two key components:

1. Distributed dynamic data-structures that efficient implement distributed adaptive grid hierarchies.
2. Programming abstractions that can be used to directly express adaptive computations on dynamic grid hierarchies.

Distributed Dynamic Data-Structures[4]

The fundamental data-structure underlying dynamically adaptive methods based on hierarchical adaptive-mesh refinements is a dynamic hierarchy of successively and selectively refined grids, or, in the case of a parallel implementation, a Distributed Adaptive Grid Hierarchy (DAGH). The efficiency of parallel/distributed implementations of these methods is then limited by the ability to partition the DAGH at run-time so as to expose all inherent parallelism, minimize communication and synchronization overheads, and balance load. A critical requirement while partitioning the adaptive grid hierarchy is the maintenance of logical locality, both across different levels of the hierarchy under expansion and contraction of the adaptive grid structure, and within partitions of grids at all levels when they are partitioned and mapped across processors. The former enables efficient computational access to the grids while the latter minimizes the total communication and synchronization overheads.

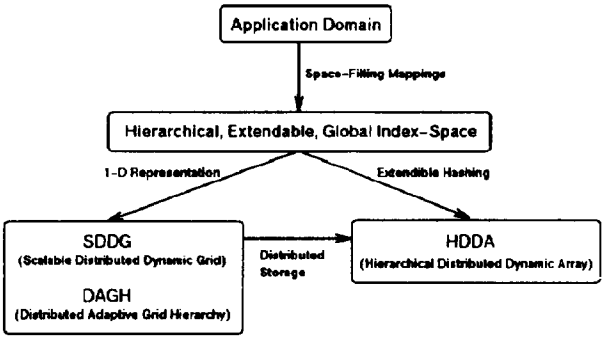

Figure 3. Constructing Distributed Dynamic Data-Structures

The construction of distributed and dynamic data-structures is outlined in Figure 3. The construction begins by defining a hierarchical and extendible global index-space. This is derived directly from the application domain using space-filling mapping[5] which are computationally efficient, recursive mapping from N-dimensional space to 1-dimensional space (see Figure 4). The generated index space is hierarchical in that each index may itself be an index space, and extendible in that it can dynamically expand and contract. Application domain locality is encoded in the index-space by the space-filling mappings and this locality is maintained through its expansion and contraction. The global index-space is used as a basis for application domain partitioning, as a global name-space for name resolution, and for communication scheduling.

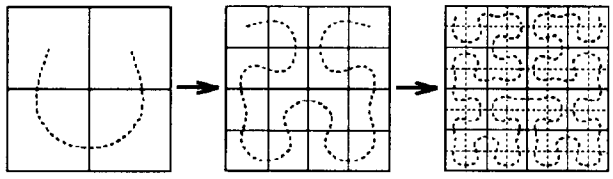

Figure 4. Space-Filling Mappings

The next step adds dynamic storage to the global index-space to create a Hierarchical Distributed Dynamic Array (HDDA). The HDDA provides pure array semantics to hierarchical, dynamic and physically distributed data. It uses extendible hashing[6] techniques, with the global index-space as the hash key-space, to translate index locality to storage locality. Further, this locality is preserved under index-space expansion and contraction. HDDA objects encapsulate distribution, dynamic load-balancing, communications and consistency management.

The final step constructs 2 fundamental application objects on top of the index-space and HDDA.

1. A Scalable Distributed Dynamic Grid (SDDG) which is a single grid in an adaptive grid hierarchy.
2. A Distributed Adaptive Grid Hierarchy (DAGH) which is a dynamic collection of SDDGs implementing an entire adaptive grid hierarchy.

The design of these data-structures uses a linear representation of the hierarchical, multi-dimensional grid structure generated using the global index-space defined above. Operations on the grid hierarchy such as grid creation, grid refinement or coarsening, grid partitioning and dynamic re-partitioning, are then efficiently defined on this one-dimensional representation. The self-similar (i.e. recursive) nature of index-space is exploited to maintain locality across levels of the grid hierarchy. SDDG and DAGH representations are described below.

SDDG Representation

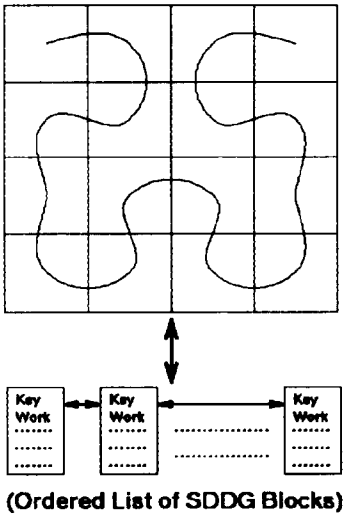

Figure 5. SDDG Representation

A multi-dimensional SDDG is represented as a one dimensional ordered list of SDDG blocks. The list is obtained by first blocking the SDDG to achieve the required granularity, and then ordering the SDDG blocks by mapping them to a span of the global index-space. The granularity of SDDG blocks is system dependent and attempts to balance the computation-communication ratio for each block. Each block in the list is assigned a cost corresponding to its computational load. In case of an AMR (adaptive mesh-refinement) scheme, computational load is determined by the number of grid elements contained in the block and the level of the block in the AMR grid hierarchy. The former defines the cost of an update operation on the block while the latter defines the frequency of updates relative to the base grid of the hierarchy. Figure 5 illustrates this representation for a 2-dimensional SDDG.

Partitioning a SDDG across processing elements using this representation consists of appropriately partitioning the SDDG block list so as to balance the total cost at each processor. Since space-filling curve (used to generated the global index-space) mappings preserve spatial locality, the resulting distribution is comparable to traditional block distributions in terms of communication overheads.

DAGH Representation

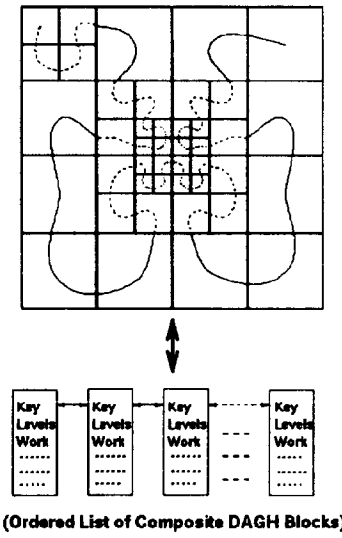

Figure 6. DAGH Representation

The DAGH representation starts with a simple SDDG list corresponding to the base grid of the grid hierarchy, and appropriately incorporates newly created SDDGs within this list as the base grid gets refined. The resulting structure is a composite list of the entire adaptive grid hierarchy. Incorporation of refined component grids into the base SDDG list is achieved by exploiting the recursive nature of global index-space: For each refined region, the SDDG sub-list corresponding to the refined region is replaced by the child grid's SDDG list. The costs associated with blocks of the new list are updated to reflect combined computational loads of the parent and child. The DAGH representation therefore is a composite ordered list of DAGH blocks where each DAGH block represents a block of the entire grid hierarchy and may contain more than one grid level; i.e. inter-level locality is maintained within each DAGH block. Each DAGH block in this representation is fully described by the combination of the space-filling index corresponding to the coarsest level it contains, a refinement factor, and the number of levels contained. Figure 6 illustrates the composite representation for a two dimensional grid hierarchy.

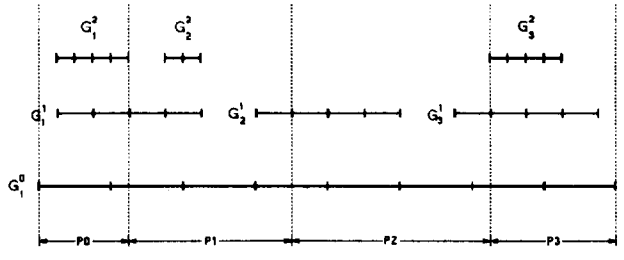

Figure 7. DAGH Composite Distribution

The AMR grid hierarchy can be partitioned across processors by appropriately partitioning the linear DAGH representation[7]. In particular, partitioning the composite list to

balance the cost associated with each processor results in a composite decomposition of the hierarchy. The key feature of this decomposition is that it minimizes potentially expensive inter-grid communications by maintaining inter-level locality in each partition. A composite decomposition of a 1-dimensional DAGH is shown in Figure 7. Note that each numbered unit in this figure is a composite block of some architecture dependent granularity. Other distributions of the grid hierarchy can also be generated using the above representation. For example, a distribution that decomposes each grid separately is generated by viewing the DAGH list as a set of SDDG lists.

Object-Oriented Programming Abstractions[8]

DAGH provides three fundamental programming abstractions that can be used to express parallel adaptive computations based on adaptive mesh refinement and multigrid techniques (see Figure 8).

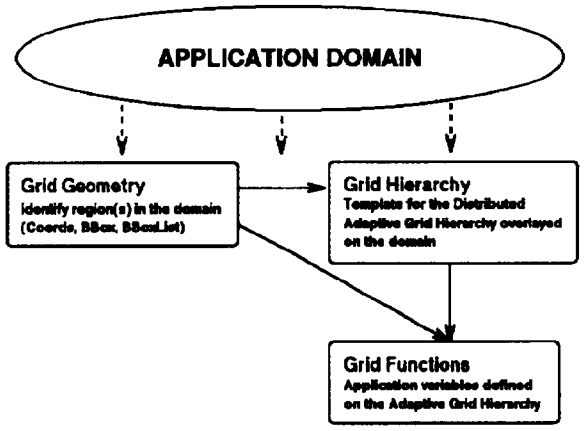

Figure 8. Programming Abstraction for Parallel Adaptive

The Grid Geometry abstractions represent regions in the computational domain and provide an intuitive means for addressing the regions and directing computations. A Grid Hierarchy abstraction abstracts an entire distributed adaptive grid hierarchy (a directed acyclic graph comprising of a dynamic number of levels of grid resolution, and a dynamic number of component grids at each level of resolution). Applications can directly index, operate on, and refine component grids within the hierarchy independent of its current structure and distribution. Parallelization issues such as partitioning and dynamic load balancing are encapsulated within Grid Hierarchy objects. The final abstraction is the Grid Function which represents an application variable defined on the distributed, hierarchical computational domain. A Grid Function object allocates distributed storage for the variable it represents according to the structure of the dynamic grid hierarchy, and enables the variable to be locally manipulated as simple FORTRAN 77/90 arrays. All user-level operations defined on these abstractions are independent of the structure of the dynamic grid or its distribution. Data-partitioning, load-balancing, communications and synchronization operations are transparent to the end user.

Input/Output and Visualization Support

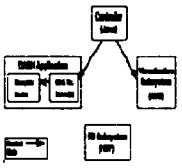

Figure 9. DAGH Input/Output & Visualization Support

Figure 9 outlines DAGH input/output and visualization support. DAGH dedicates a set of processors as IO servers. These processors listen for IO requests from processors performing computations and are responsible for data collection, data collation, and actual file or stream IO. All overheads associated with input/output are thus off-loaded from the compute nodes. Current visualization support enables DAGH data-structures to directly communicate with the AVS[1] visualization system for interactive multiresolution visualization. DAGH base data structures are being extended to directly interact with a Java controller to provide computational steering capabilities.

Programming Interface

DAGH supports a coarse grained data-parallel programming model where parallelism is achieved by operating on regions of the grid hierarchy in parallel. The DAGH programming interface is developed so as to enable computations to performed by traditional FORTRAN 77 and FORTRAN 90 kernels. Consequently, all storage is maintained in FORTRAN compatible fashion.

DAGH Performance

DAGH performance is found to be comparable to and often better than conventional implementation (using FORTRAN) for non adaptive application[3]. Further dynamic partitioning and load-balancing overheads are less than 10% of the time spent in the computational kernels.

Current Status

The DAGH infrastructure has been implemented as a C++ class library on top of the MPI communication system. The system is currently operational on a number of high-performance platforms including the SGI Origin 2000, Cray T3E and the IBM SP2 and on networked workstations. DAGH has been used an enabler for muti-disciplinary research. DAGH based applications are summarized in Figure 10.

[1] Advanced Visual Systems Inc.

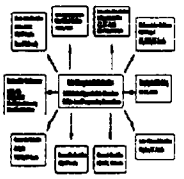

Figure 10. DAGH Applications

More information about DAGH can be obtained from http://www.ticam.utexas.edu/~parashar/public_html/DAGH/

Integrating IPARS and DAGH

Integration of IPARS with DAGH essentially consists of replacing the IPARS data-management layer with DAGH as shown in Figure 11.

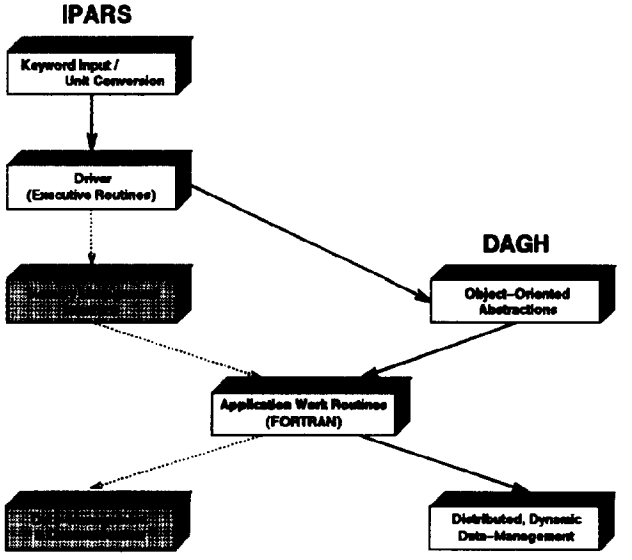

Figure 11 Integrating IPARS with DAGH

The resulting PSE provides combines the high-level application specific capabilities provided by IPARS with the dynamic data-management capabilities provided by DAGH.

Supporting Multiple Fault Blocks

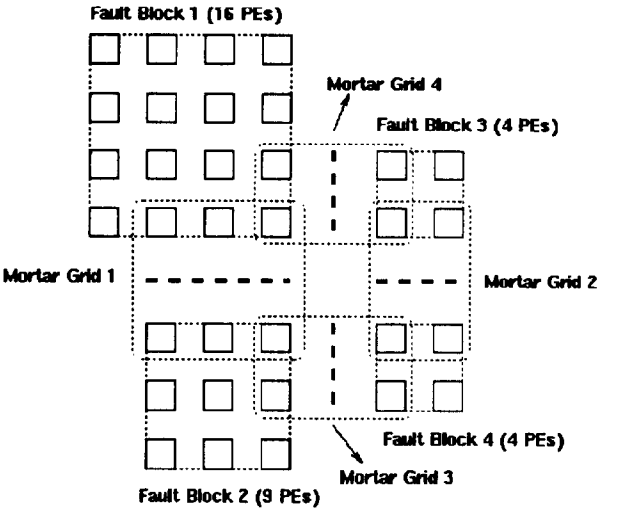

Figure 12 Multiple Fault Block Support

The IPARS + DAGH PSE provides a 2 level combined MIMD[2] - SPMD[3] programming model to support multiple fault block. At the top level processors are divided in groups and these groups operate on different fault blocks in an MIMD fashion. With in a processor group, computations on a single fault block proceed using the SPMD programming model. Each fault block is implemented as a complete grid hierarchy and can be independently refined. Special "mortar" grids are defined to couple adjoining fault blocks. These grids exists on the intersection of the adjacent faces of the two blocks and are shared by processors responsible for the block faces. Communication between blocks and the mortar grids is handled automatically by the PSE. Figure 12 outlines the support for multiple fault. This picture shows 4 fault blocks and 4 mortar grids. The dotted oval boxes represent processors sharing a mortar grid.

A Sample Application using Adaptive Grid Refinement

[2] Multiple Instruction Multiple Data
[3] Single Program Multiple Data

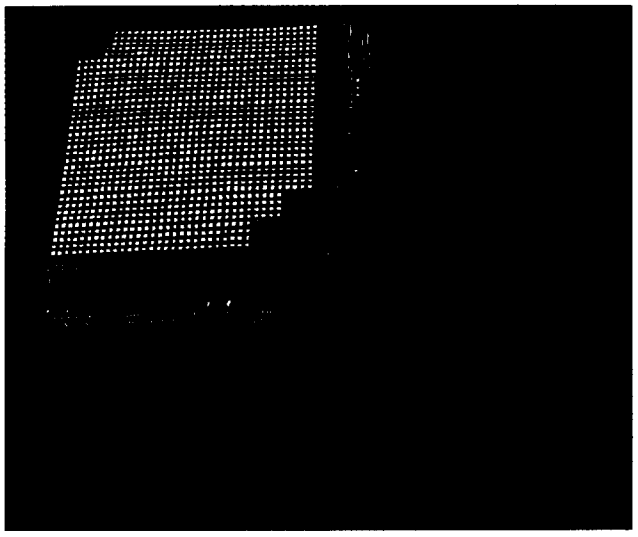

Figure 13 2-D Buckley Leverett Solution (4 Level of Refinement)

Figure 13 shows the solution for a Buckley Leverett quarter five spot test problem where a uniformly constant velocity field is imposed and injection takes place in the top left had corner while outflow occurs in the corner cell that is diagonally opposite. This figure shows time step 70 with 4 levels of refinement.

Conclusions

In this paper we described the design and implementation of a problem solving environment to support the development of a new generation of reservoir simulators capable of realistic, high-resolution reservoir studies with a million or more grid elements on massively parallel multicomputer systems. The simulators can handle multiple physical models, generalized well management, multiple fault blocks, and dynamic, locally adaptive mesh-refinements. The PSE reduces the complexity of building flexible and efficient parallel reservoir simulators through the use of a high-level programming interface for problem specification and model composition, object-oriented programming abstractions that implement application objects, and distributed dynamic data-management that efficiently supports adaptation and parallelism. The PSE is currently operational on a number of high-performance computing platforms including the IBM SP2, CRAY T3E and SGI Origin 2000, as well as networked workstations.

Acknowledgements

We gratefully acknowledge the financial support of the U.S. Department of Energy ER/MICS and ER/LTR programs that funded Argonne National Laboratory (ANL) and the University of Texas as part of the Advanced Computational Technology Initiative (ACTI) project New Generation Framework for Petroleum Reservoir Simulation. (ANL contract 951522401 to UT). The authors wish to thank the other participants of this project and in particular Todd Arbogast, Clint Dawson, Mojdeh Delshad, Kamy Sepehrnoori, Mary Wheeler and Ivan Yotov of the University of Texas and Satish Balay, L.C. McInnes, Tom Morgan and Barry Smith of Argonne National Laboratory and our industrial partners. We also wish to acknowledge partial support of this research under the Department of Energy grant DE-FG07-96ER14720.

References

1. Wheeler, J.A.: "IPARS Simulator Framework Users Guide," Technical Report, University of Texas at Austin, Austin, TX.
2. Delshad, M., G.A. Pope, and K. Sepehrnoori: "A Compositional Simulator for Modeling Surfactant Enhanced Aquifer Remediation, 1 Formulation", J. Contaminant Hydrology 23 (1996) 303-327.
3. Parashar, M., and Browne, J.C.: "DAGH Users Guide" Technical Report, Department of Computer Sciences, University of Texas at Austin, Austin, TX. (Available at http://www.ticam.utexas.edu/~parashar/public_html/DAGH/)
4. Parashar, M. and Browne, J.C.: "Distributed Dynamic Data-Structures for Parallel Adaptive Mesh Refinement," Proceedings for the International Conference on High Performance Computing, India, December, 1995, 22-27.
5. Sagan, H.: "Space-Filling Curves," Springer-Verlag, 1994.
6. Fagin, R.: "Extendible Hashing - A Fast Access Mechanism for Dynamic Files," ACM TODS, 4:315-344, 1979.
7. Parashar, M. and Browne, J.C.: "On Partitioning Dynamic Adaptive Grid Hierarchies," Proceedings of the 29th Annual Hawaii International Conference on System Sciences, Hawaii, January, 1996, 604-613.
8. Parashar, M. and Browne, J.C.: "Object Oriented Programming Abstractions for Parallel Adaptive Mesh-Refinement," Presentation at Parallel Object-Oriented Methods and Applications Workshop, Santa Fe, February, 1996.