A Dynamic Geometry-Based Shared Space Interaction Framework for Parallel Scientific Applications[*]

Li Zhang and Manish Parashar

The Applied Software Systems Laboratory, Rutgers University,
94 Brett Road, Piscataway, NJ 08854
{emmalily, parashar}@caip.rutgers.edu

Abstract. While large-scale parallel/distributed simulations are rapidly becoming critical research modalities in academia and industry, their efficient and scalable parallel implementations present many challenges. A key challenge is the dynamic and complex communication/coordination patterns required by these applications, which depend on states of the phenomenon being modeled and are determined by the specific numerical formulation, the domain decomposition and/or sub-domain refinement algorithms used, and are known only at runtime. In this paper, we present a dynamic geometry-based shared-space interaction framework for scientific applications. The framework provides the flexibility of shared-space coordination models while enabling scalable implementations. The design, prototype implementation and experimental evaluation using an adaptive multi-block oil reservoir simulation are presented.

Keywords: parallel scientific applications, dynamic geometry-based shared space, communication locality, scalability, tuple space, Hilbert space filling curve.

1 Introduction

Large-scale parallel/distributed simulations are playing an increasingly important role in science and engineering and are rapidly becoming critical research modalities in academia and industry. With the increasing scale of parallel systems and sophistication of application formulations and numerical techniques, emerging applications offer the potential for accurately simulating physically realistic models of complex phenomena and providing dramatic insights into complex applications such as interacting black holes and neutron stars, formations of galaxies, subsurface flows in oil reservoirs and aquifers, and dynamic response

[*] The research presented in this paper is supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS), EIA-0120934 (ITR), ANI-0335244 (NRT), CNS-0305495 (NGS) and by DOE ASCI/ASAP via grant number 82-1052856.

of materials to detonation. However, the phenomena being modeled by these applications and their implementations are inherently multi-phased, dynamic, and heterogeneous in time, space, and state. Combined with the complexity and scale of the underlying parallel/distributed system, efficient and scalable implementations of these applications present many challenges.

A key challenge is the dynamic and complex communication/coordination patterns required by these applications. These communication/coordination patterns depend on states of the phenomenon being modeled and are determined by the specific numerical formulation, domain decomposition and/or sub-domain refinement algorithms used, and are known only at runtime. Implementing these communication/coordination patterns using commonly used parallel programming frameworks is non-trivial. Message passing frameworks such as MPI require matching sends and receives to be explicitly programmed for each interaction. Frameworks based on shared address spaces provide higher-level abstractions that can support dynamic interactions. However scalable implementation of global shared address spaces remains a challenge.

Tuple spaces provide a very flexible and powerful mechanism for extremely dynamic communication and coordination patterns [1]. In the model, processes interact using an associative shared tuple space. A tuple is a sequence of fields, each of which has a type and contains a value. The producer of a message formulates the message as a tuple and places it into the tuple space. The consumer(s) can associatively look up relevant tuples using pattern matching on the tuple fields. The tuple space model provides two fundamental advantages: simplicity and flexibility. The communicating nodes need not care about who produced or will consume a tuple. Furthermore, the communicating processes do not have to be temporally or spatially synchronized. This decoupling feature automatically supports for dynamic communication/coordination. However, scalable implementation of tuple spaces remains a challenge. In a pure tuple space environment, all the communication passes through a central tuple space with relatively slow associative lookup mechanisms [2], which is an inherent bottleneck impeding scalability and efficiency.

In this paper, we present the design, implementation and evaluation of an interaction framework for scientific applications that address the challenges outlined above. The proposed framework supports the flexibility and dynamism of a tuple-based environment while enabling scalable implementations. It builds on two key observations: (a) formulations of most scientific and engineering applications are based on geometric multi-dimensional domains (e.g., grid or mesh) and (b) interactions in these applications are typically between entities that are geometrically close in this domain (e.g., neighboring cells, nodes or elements). Rather than implementing a general and global associative space, we enable the dynamic creation of transient geometry-based interaction spaces, each of which is localized to a sub-region of the overall geometric domain. The interaction space is defined to cover a closed region of the application domain described by an interval of coordinates in each dimension, and can be identified by any set of coordinates contained in the region. It can then be used to share objects between nodes corresponding to that region. Nodes do not have to know of or synchronize with each other. The semantics of sharing is similar to traditional tuple space models.

The prototype implementation of the proposed model complements existing interaction frameworks (e.g., MPI, OpenMP) and provides a scalable geometry-based shared-space for dynamic runtime coordination and localized communication. It uses the Hilbert Space Filling Curve (SFC), a locality preserving recursive mapping from a multi-dimensional coordinate space to a 1-dimensional index space, to construct a distributed directory structure that enables efficient registration and lookup of objects in the shared-space. The prototype is evaluated using a parallel adaptive multi-block oil reservoir simulation [3]. Experimental results demonstrate system scalability, low space operation overheads, and that the performance is comparable to a pure message passing system.

The rest of the paper is organized as following. Section 2 presents a driving application and its interaction requirements. Section 3 presents the dynamic geometry-based shared space model. Section 4 presents design of the interaction framework. Section 5 presents the prototype implementation and experimental evaluation. Section 6 discusses related work and Section 7 draws a conclusion for our work.

2 A Driving Application: Parallel Adaptive Multi-block Oil Reservoir Simulation

In this section we use the parallel multi-block oil reservoir simulation as the driving application to motivate the interaction framework presented in this paper. In these simulations, the oil reservoir is discretized as a series of blocks and interfaces between blocks. The target domain consists of a coupled system of highly nonlinear transient partial differential equations. Its geometrical and geological features induce a multi-block decomposition so that each block is discretized by cell-centered finite differences on logically rectangular grids. Flux

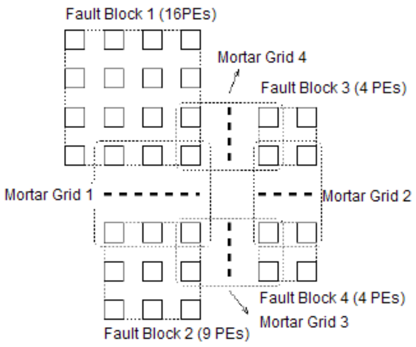

Fig. 1. 2-D view of decomposed domains with interface sharing [3]

matching conditions are imposed on the interfaces and a non-overlapping domain decomposition algorithm is exploited so that solving the interface problem only requires in-block solves and an exchange of interface values between neighboring blocks [3]. We refer to the face sharing described above as neighbor-neighbor relationship. Fig. 1 presents a 2-D view of decomposed domains in a 2-dimensional coordinate system. From the figure we can observe that communication between blocks in this particular environment is highly localized and is on the interfaces of neighboring blocks. The challenge presented in this application is that when the decomposed sub-blocks are distributed across nodes in the system, locating the processor assigned to a neighboring block is non trivial especially when dynamic load-balancing is used.

3 Dynamic Geometry-Based Shared Space(DGSS) Model

DGSS builds on the tuple spaces model. Communication entities interact with each other by sharing objects using a virtual shared space. However there is conceptual difference between the DGSS model and the general tuple space model. A general tuple space spans the entire problem domain, is accessible to all nodes in computing environments, and is associated with a generic tuple matching scheme. DGSS defines a dynamic shared space that is based on geometric regions within the application domain. It enables interactions that are localized to a geometric region by sharing objects in the DGSS based on geometry-associative semantics. DGSS supports for dynamic and flexible interaction/coordination while enabling scalable realizations based on the geometric nature of the computational domain and the local nature of communications, which are typical of most scientific applications. The geometric nature is due to the observation that formulations of scientific applications are based on a geometric descretization of the physical domain. Communication locality is due to the observation that interaction and coordination are defined by problem domains, and are typically local to sub-regions of the domain. Consequently, operations on an object shared in a DGSS only require communication within the DGSS. Coordinates from the geometric domain define the geometry-associative semantics for retrieving/storing objects from/to the spaces. DGSS is dynamic in the sense that it is created/destroyed at runtime and is constructed on top of a dynamic set of nodes that may change as the communication-integrated sub-domain changes. Through the model we automate the communication setup procedure among partitioned tasks, thus releasing programmers from the complicated and tiresome work of manually arranging all coordination patterns for every node during application development, facilitate localized communication, thus ensuring scalability of the model and benefit most scientific applications through its support for geometry-associative semantics.

4 Design of the DGSS Framework

4.1 DGSS Architecture

The DGSS architecture consists of two components: a distributed directory structure that enables locating shared spaces based on geometric relationships, and the dynamic shared-spaces that are associated with geometric regions. The distributed directory is constructed as a distributed table. The index of the table is generated by mapping the multi-dimensional problem domain to a 1-dimensional index space using Hilbert space-filling curves, which is then partitioned and distributed across the nodes in the system. A space-filling curve (SFC) is a continuous mapping from a d-dimensional space to a 1-dimensional space. The d-dimensional space is viewed as a d-dimensional cube, which is mapped onto a line such that the line passes once through each point in the volume of the cube, entering and exiting the cube only once [4]. Using this mapping, a point in the cube can be described by its spatial or d-dimensional coordinates, or by the length along the 1-dimensional index measured from one of its ends. The construction of SFCs is recursive. An important property of SFCs is locality preserving. Points that are close together in the 1-dimensional space are mapped from points that are close together in the d-dimensional space. As the index space is partitioned across the nodes in the system, each node is responsible for an interval of the index space and the region of the computational domain corresponding to this interval. The node manages information regarding the creation, deletion and memberships of any DGSS in this region. Note that the hash table will be typically sparsely populated and that the shared spaces are not uniformly distributed across the index space. As a result, load-balancing is used while partitioning the index space across the nodes. The mapping of a 2-dimensional domain using the Hilbert SFC and the creation of a distributed directory are illustrated in Fig. 2.

To create or access a shared space corresponding to a region in the computational domain, the region is first translated to interval(s) in the 1-d index space and these intervals are used to locate the processor where information about the space is maintained. The process of locating corresponding directory node is efficient, requiring only local computation. An interval tree is used to store index intervals corresponding to already created and registered shared spaces at each

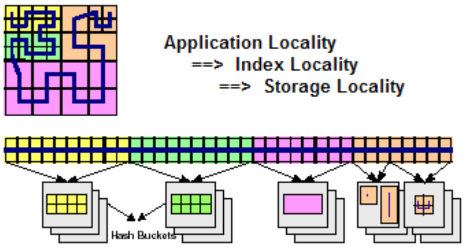

Fig. 2. Directory structure using Hilbert SFC [3]

node, and to detect geometric relationships between new and already registered spaces.

4.2 DGSS-Based Shared Storage

The DGSS-based shared storage is created to span a dynamic set of nodes based on the geometric relationship of their regions of interaction. Multiple DGSS-based shared spaces can co-exist in the system and each node can be part of more than one space. Physically, the shared space is replicated on each of the participating nodes and consistency is maintained using a combination of update propagation and multiple-versioned objects. Update propagation refers to propagating changes to a shared object to every node caching the object within the space. Since each DGSS-based shared space only spans a geometrically localized communication zone, it typically spans a small number of nodes and update propagation does not result in significant overheads. Multiple-versioned objects allows shared objects to have multiple co-existing versions, which can improve parallelism by enabling nodes to access and update different versions of the same object without synchronization.

4.3 DGSS Interface

The DGSS framework interface defines operators to allow nodes to join/leave a space and to access the space. The creation/destruction of a space is a non-collective operation and nodes can join and leave a space at runtime. A node joins a space by registering its interaction region described using geometric coordinates. If the region overlaps with an existing region, the querying node joins the existing space and the region covered by the space is redefined to be a union of the two regions, and the membership of the space is updated. If it does not overlap with any of the existing regions, a new shared space is dynamically created. A node leaves a space by de-registering itself. When the last node associated with a space de-registers, the space is destroyed.

The space access operators are similar to those provided by tuple space systems such as Linda [1], with the exception of the "eval" function, which is not supported. The space access operators are listed in Table 1. Given the geometry-based access semantic defined by DGSS, the search process for a finite region should uniquely return zero or one object from the shared space. This is unlike a generalized tuple space, which may have multiple matches.

Table 1. DGSS Interface

Interface Operators	Function Description	Linda Correspondence
get	A "get" operation moves an object from a DGSS to requesting node. Further "get" requests on the object are blocked until it is "put" back to DGSS	in
put	A "put" operation moves an object from requesting node to a DGSS.	out
read	A "read" operation copies an object to requesting node without removing it from DGSS. Multiple "read" operations can occur simultaneously.	rd
register	"register" is provided to register an object with DGSS. Based on registered geometric information, a pointer pointed at an existing DGSS or a new DGSS will be returned.	n/a

5 Implementation and Performance Evaluation of a Prototype System

5.1 Prototype Implementation

We have developed a prototype DGSS interaction framework. The implementation uses multi-threading. At application startup, a *DGSS-daemon* thread is created within the user application process on each node. This daemon handles registration requests by retrieving and updating local directory entries, and object access requests if the node is part of a DGSS. Besides the *DGSS-daemon*, the other key component is *DGSS-storage*, which is created to store shared objects. To create a DGSS at runtime, nodes in a sub-domain will register a geometric interaction region of interest with the underlying distributed directory layer. On receiving the registration request, the *DGSS-daemon* retrieves its local directory to determine whether the region of interest intersects with an existing DGSS or if a new DGSS should be created. The *DGSS-daemon* returns a pointer to the existing/new DGSS, which is then used for further space interactions. Current implementation needs to statically define a startup server, which is known a priori to all nodes in the computing environment. Table 2 lists a code sample showing how a node starts up a space daemon, joins a DGSS by registering an object and shares the object with other nodes through the DGSS within a computation loop.

Table 2. Pseudo Code Series Calling DGSS Interface

```
/*In the pseudo code series we first create a runtime DGSS by calling space-initiation function,
register an object with the DGSS and insert an object into it. Then a loop that takes the object from
the DGSS, performs local computation and updates the object, and puts the object back to the DGSS
is executed until loop condition becomes invalid. After that the object is de-registered from DGSS*/
/*Create DGSS by calling space-initiation function*/
SPACE* space=system_init(node id, shared space bootstrap server ip);
Initiate a local object;
/*Register an object with the DGSS*/
space->register(object geometric description);
/*Insert the object into the DGSS*/
space->put(object geometric description, object, object version number);
while(number of iterations<maximum number of iterations){
/*Get the object which has been updated by other nodes sharing it from the DGSS*/
space->get(object geometric description, object, object version number);
perform local computation, update object and its version number;
/*After performing local computation and updating the object, put it back to the DGSS*/
space->put(object geometric description, object, object version number);
}
/*De-register the object from the DGSS*/
space->deregister(object geometric description );
```

5.2 Experimental Evaluation

We have constructed a simulated oil reservoir environment to evaluate performance and scalability of the prototype system. In the simulation, we assumed

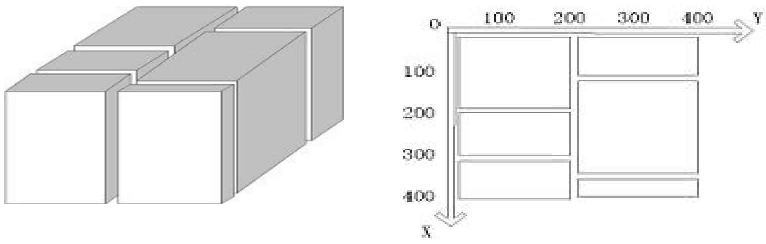

Fig. 3. 3-D/2-D view of the simulated oil reservoir experiment environment

the whole problem domain is mapped to a geometry model of a series of 6 blocks and 5 interfaces in a 3-dimensional coordinate system as shown in Fig. 3.

Four of shared interfaces have a size of 200*400 grid points and the fifth has 400*400 grid points. The data attached to each point is of type double. Blocks are decomposed at runtime into smaller blocks and assigned to nodes across a cluster of workstations. Thus possibly a node owns only a small partition of one block and its associated interface or possibly no associated interfaces, e.g., it is a central part of a block. The simulation is run on a 64-node Beowulf cluster connected by a high speed 100 MB LAN.

a) Performance Evaluation

The execution times for "register" "get" and "put" operations are measured for a range of system sizes, upto 64 nodes. Two observations result: First, as the system size increases, application grid blocks were partitioned into a larger number of partitions of smaller sizes. Consequently, the corresponding shared interfaces were also smaller in size. Second, not all nodes were assigned shared interfaces and so, not all nodes were part of a DGSS. Fig. 4 shows the execution time of each primitive. Lines with different colors represent experiments on different system sizes. The figure shows that the time for the "register" operation varies from 0.06428 second to 3.10842 seconds. This is because in the experiment all nodes that share an interface register that interface almost at the same time. Thus these registration requests nearly simultaneously reach the directory nodes that should handle them and are processed sequentially to guarantee consistency of the registration process. As a result, the execution time for "register" operation includes the time that a request blocks waiting for response, which can increase as system size increases. This potential bottleneck can be removed using dynamic load balancing. The times required for "get" and "put" operations are much smaller and comparatively stable, as seen from the figure. Further, the times required for these operations decrease as system size grows due to the two observations mentioned above. Of the three operations, "register" has the highest cost. However, note that each shared interface is registered only once in the application.

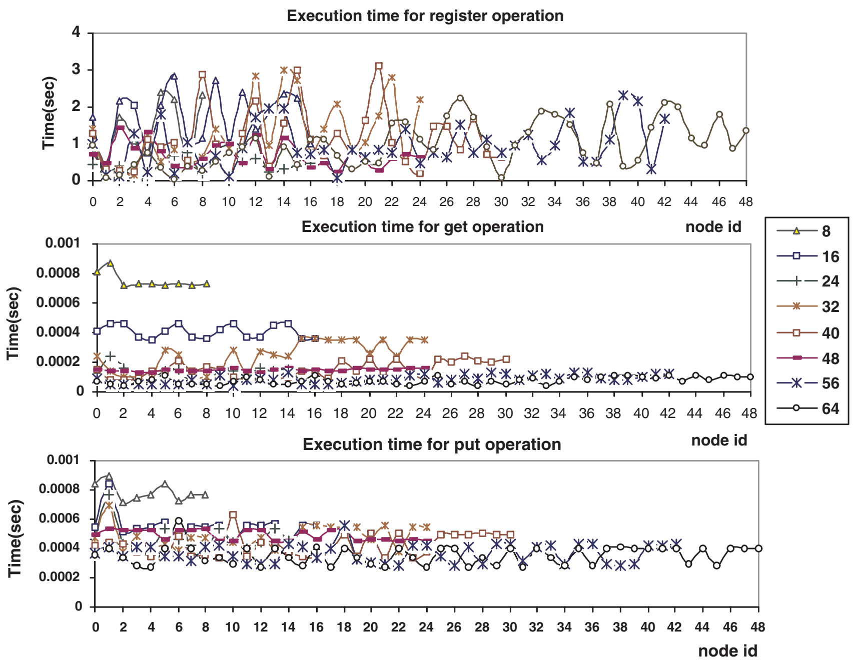

Fig. 4. Execution time for “register”,“get” and “put” operations

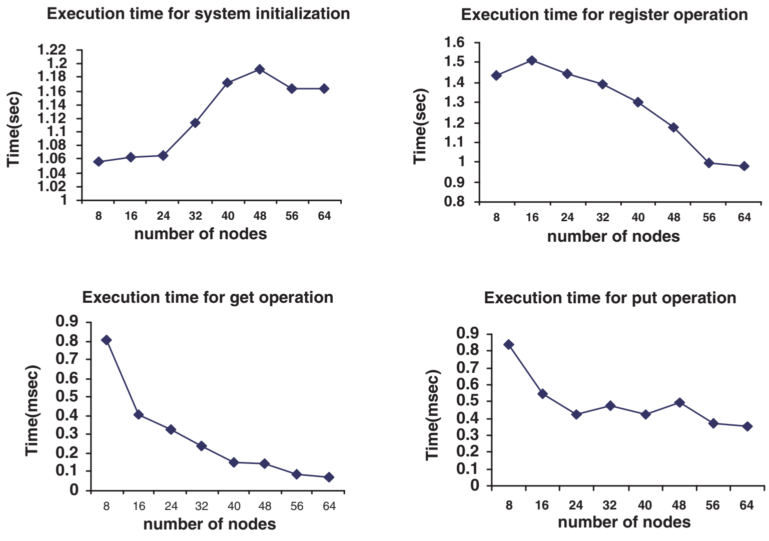

Fig. 5. Execution times for “init”,“register”,“get”and “put” operations

b) Scalability Evaluation

To evaluate system scalability, we averaged the execution time for each operation on different nodes for different system sizes. These results are plotted in Fig. 5. The figure shows that only system startup time increases as system size increases. Execution times for "register", "get" and "put" operations decrease as system size increases. The reason for increasing startup time is that the startup phase uses server-client communication to collect necessary network information from all nodes. As system size increases, the server becomes a communication bottleneck causing the startup time to increase. The other three operations scale well, which is because (a) they operate with DGSS which includes only a small number of processors, and (b) as the system size increases, the size of shared interface corresponding to a DGSS reduces and thus the "register", "get" and "put" operations operate on objects of smaller sizes.

6 Related Work

Several other projects also base their frameworks on tuple space concept such as Sun's JavaSpaces and IBM's Tspace. Sun's JavaSpaces combines Java with tuple spaces while IBM's Tspace emphasizes the integration of tuple space with database systems. These systems are quite complex and over-weighted for High Performance Computing. A lightweight Java Taskspaces framework for scientific computing on computational Grids [2] is a work similar to ours. The framework constructs lightweight shared taskspaces for node pairs that are assigned with tasks in problem sub-domains with neighbor-neighbor relationship. Because of the particular type of applications targeted, space sharing mechanism is supported by building direct communication channels between two nodes in a node pair. However the framework is limited to only one specific type of communication locality while our model addresses communication locality in general.

7 Conclusion

This paper presented a DGSS interaction framework to facilitate dynamic interaction/coordination in large-scale parallel scientific applications. The framework exploits geometric structure of the application domain and its communication locality to provide the flexibility and dynamic of shared space interaction models while enabling their scalable implementations. A DGSS is virtually shared among a group of nodes in a problem sub-domain and provides a powerful mechanism for dynamic complex interaction/coordination. The framework complements (and can co-exist with) existing interaction infrastructures (e.g. MPI). A prototype implementation and experimental evaluations were presented. Experimental results using a multi-block adaptive oil reservoir simulation show system scalability and small overheads of interface operations.

References

1. Nicholas Carriero, David Gelernter. Linda in context. Communications of the ACM, Volume 32, Issue 4, pp.444-458, April 1989, ISSN:0001-0782.
2. H. De Sterck, R.S.Markel, T.Pohl, U.Rude. A lightweight Java Taskspaces framework for scientific computing on computational grids. The eighteenth annual ACM symposium on applied computing, March 2003, Melbourne, Florida, USA. pp.1024-1030, 2003, ISBN:1-58113-624-2
3. M. Parashar and I. Yotov. An Environment for Parallel Multi-Block, Multi-Resolution Reservoir Simulations. Proceedings of the 11th International Conference on Parallel and Distributed Computing Systems (PDCS 98), Chicago, IL, International Society for Computers and their Applications (ISCA), pp.230-235, September 1998.
4. Cristina Schmidt, Manish Parashar. Flexible Information Discovery in Decentralized Distributed Systems. 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), June 22-24,2003, Seattle, Washington, pp.226, 2003, ISBN:0-7695-1965-2.