Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Parallel Computing 29 (2003) 1539–1562
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Solving irregularly structured problems based on distributed object model

Yudong Sun [a,*], Cho-Li Wang [b]

[a] School of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, UK
[b] Department of Computer Science and Information Systems, The University of Hong Kong, Pokfulam Road, Hong Kong

Received 15 January 2003; accepted 15 May 2003

Abstract

This paper presents a distributed object model called MOIDE (multi-threading object-oriented infrastructure on distributed environment) for solving irregularly structured problems. The model creates an adaptive computing infrastructure for developing and executing irregular applications on distributed systems. The infrastructure allows dynamic reconfiguration to match the evolution of irregular computation and available system resources. A unified communication mechanism is built to integrate different communication paths on heterogeneous systems to support efficient communication. Autonomous load scheduling approach is proposed for dynamic load balancing. A runtime system is developed to implement MOIDE-based computing. Applications including N-body problem, ray tracing, and conjugate gradient are developed to demonstrate the advantages of the model.

Keywords: Irregularly structured problem; Distributed object model; Distributed system; Adaptive computing infrastructure; N-body

1. Introduction

Irregularly structured problems are the applications whose computation and communication patterns are input-dependent, unstructured, and evolving during computation [10,21,26]. Many applications in scientific and engineering computing fields

* Corresponding author.
E-mail addresses: yudong.sun@ncl.ac.uk (Y. Sun), clwang@csis.hku.hk (C.-L. Wang).

0167-8191/$ - see front matter
doi:10.1016/j.parco.2003.05.006

such as astrophysics, fluid dynamics, sparse matrix computation, system modeling and simulation, computer graphics, etc. can be classified as irregularly structured problems.

Irregularly structured problems possess different irregularities. In general, the *irregular and dynamically evolving data distribution* in these problems results in *non-predetermined computation pattern* and *workload*. The high data-dependency in some of the problems even complicates the task decomposition in parallel and distributed computing. The irregular data distribution and computation also produce *irregular communication pattern*, especially on distributed systems. The irregular communication produces high overhead that severely impairs the performance of computation. These irregularities need different approaches to tackle, such as special data structure, dynamic task decomposition and load balancing, and efficient communication mechanism.

With the rapid advance of high-performance computers and networking technologies, distributed systems have been providing cost-effective environment for the parallel and distributed computing of large-scale applications. As a powerful methodology, *distributed object computing* integrates the object-oriented technique with networking [15,29]. Flexible computing infrastructure can be built based on distributed objects to support efficient computing for irregularly structured problems. Various methods can be designed and integrated on the object-based infrastructure to resolve the irregularities in different applications. The methods include runtime system reconfiguration, dynamic task scheduling, efficient intra- and inter-object communication mechanism. The polymorphism of distributed objects allows the infrastructure adaptive to irregular computation pattern and distributed system architecture.

This paper presents a distributed object model called MOIDE (multi-threading object-oriented infrastructure on distributed environment) for solving irregularly structured problems. The MOIDE model creates an adaptive computing infrastructure called *hierarchical collaborative system* (HiCS) by distributed objects and multi-threads for developing and executing applications on distributed heterogeneous systems. The MOIDE also provides a uniform programming model that is independent of particular system architecture for developing applications. In the past, the researches on irregularly structured problems concentrated on designing the algorithms for specific problems. Differently, our MOIDE model aims to establish a generic computing infrastructure for solving irregularly structured problems efficiently on any distributed system. Based on the model, varied mechanisms and approaches can be developed and integrated for different applications.

Dynamic reconfiguration can be performed on the HiCS at runtime in response to the evolution of computation pattern and available system resources to enhance the computing capability of the infrastructure. A *unified communication mechanism* is built to seamlessly integrate the local data sharing and remote messaging (i.e., the remote method invocation between distributed objects) to implement efficient communication on heterogeneous systems. *Autonomous load scheduling* is proposed as an approach for the dynamic load balancing in irregular computation.

Three irregularly structured applications are developed to demonstrate the utilization and advantages of the MOIDE model. The *N-body* method gives a concrete example of designing data structures and algorithms for irregularly structured problems based on the model. The *ray tracing* method adopts the autonomous load scheduling to achieve high parallelism in computation. The *conjugate gradient* method implements efficient vector communication by means of the unified communication mechanism.

The rest of the paper is organized as follows. Section 2 introduces the irregularly structured problems. Section 3 describes the MOIDE model and the implementation. Section 4 presents the *N*-body method. Section 5 presents the ray tracing method. Section 6 presents the conjugate gradient method. Section 7 discusses the related work and Section 8 concludes the paper.

2. Irregularly structured problems

Irregularly structured problems exist in different fields. These problems have different irregularities. However, they have a common characteristic of irregular data distribution that in turn generates irregular computation and communication patterns [10,21,27]. Generally, irregularly structured problems can be defined as following.

Definition 1. An irregularly structured problem is an application whose computation and communication patterns are input-dependent, unstructured, and evolving during computation.

Usually, these problems are large-scale, compute-intensive and/or communication-intensive applications [10,22,25,26,34]. The main features of the problems are:

- unstructured and dynamically evolving data distribution,
- non-predetermined computational workload,
- unstructured and high communication requirement.

These irregular features result in difficulties to the development of high-performance parallel and distributed algorithms for the problems. Different approaches can be devised to resolve the irregular computation and communication. For example, complicated *data structures* such as special forms of trees and graphs can be employed to represent the unstructured and evolving data distribution and interrelationship [1,6,17]. The data structures should be flexible to perform task decomposition and support data sharing in distributed computing.

The non-predetermined computational workload requires *dynamic load balancing* for the computation. A load balancing strategy is closely related to the computation pattern of an application [9,33]. Global load redistribution is suitable for the

applications with high data-dependency. Runtime task allocation is suitable for the applications with low data-dependency.

In a distributed system, inter-process communication is usually accomplished through message passing with high communication latency. The high communication overhead in irregularly structured problems severely constrains the performance of computation. *Efficient communication mechanism* is required to improve the communication efficiency [9,22]. The communication mechanism should harness the flexible inter-object interaction methods and the architecture of a distributed system to implement efficient communication. Communication-efficient algorithms should also be designed to reduce the communication overhead in irregular problems.

A distributed system usually contains heterogeneous nodes with different architecture. A *unified computing infrastructure* is required to seamlessly integrate the system resources and provide a uniform environment for the development and execution of applications. The infrastructure should be able to adaptively map the applications onto underlying systems [19] so as to fully utilize the system resources and implement high-performance computing.

Irregularly structured problems include the applications in different fields. For example, *N-body problem* studies the evolution of a physical system containing a great number of particles (bodies) [5,26,27,31,32]. The bodies impose force influences on each other that causes continuous body motion. Many physical systems exhibit such a behavior in astrophysics, plasma physics, molecular dynamics, fluid dynamics, etc. The irregularity of N-body problem exists in the non-uniform body distribution and thus the non-predetermined workload in the complicated computation of force influences. A distributed N-body method should balance the workload among the processes using a wise task decomposition strategy. The problem also produces heavy communication overhead to propagate the information of the bodies. A communication strategy is needed to reduce the overhead.

Ray tracing is a graph rendering algorithm [8,25] that generates an image from the description of the objects in a scene. Primary rays emitted from a viewpoint pass through a screen and enter the scene. When hitting an object, a ray is reflected to each light source to check if it is shielded from the light source. If not, the light contribution is calculated from the light source to a pixel on the screen. The rays also spawn new rays by the reflection on the objects. The rendering is recursively performed on the new rays. This is an irregularly structured problem as the generation of new rays is non-predetermined and the workload of rendering each pixel is highly diverse.

Sparse matrix computation broadly exists in scientific and engineering computing such as solving sparse linear systems and partial differential equations. Due to the unstructured data density in sparse matrix, the computational workload is unbalanced in parallel computation. Unstructured vector communication is also included in the computation. The *conjugate gradient* (CG) is an iterative method for solving large sparse linear systems [14]. It includes vector reduction and transposition operations that incur high communication overhead. In this paper, the three irregular applications are implemented based on the MOIDE model.

3. MOIDE: a distributed object model

3.1. Adaptive computing infrastructure

3.1.1. System architecture

The fundamental structure of the MOIDE model is the *Hierarchical Collaborative System* (HiCS). It is a runtime computing infrastructure constructed with distributed objects and multi-threads on the hosts to run an application. Fig. 1 shows the structure of a hierarchical collaborative system built on four hosts. The HiCS consists of four objects, one per host. The object on *Host* 0 is called *compute coordinator* that is the first object created on the host where an application is launched for execution. The compute coordinator acts as the system initiator and coordinator. It uses the *host selection mechanism* to select other hosts available in the underlying distributed system to run the application together. The criterion for the host selection takes into account the computing power and the workload of a host. The host that can present the highest performance will be selected first. The selection criterion is defined as:

$$\text{performance} = \frac{\text{power}_i}{\text{workload}_i}$$

where power_i and workload_i denote the computing power and the current workload on host i.

The compute coordinator instantiates an object called *compute engine* on each of the selected hosts and allocates a computational task of the application to it. All *compute engines* execute the tasks in parallel. The compute coordinator is responsible for coordinating the computing procedure on the compute engines. It also executes a computational task to participate in the collaborative computation.

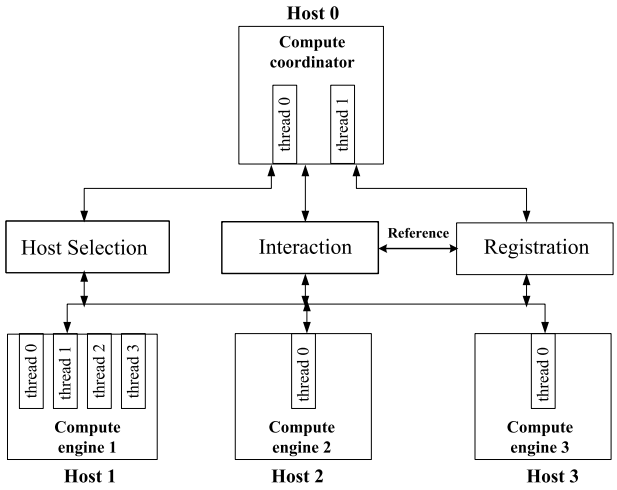

Fig. 1. Hierarchical collaborative system (HiCS).

If a host is a multi-processor, e.g., an SMP (symmetric multi-processing) node, the *object* (referring to the *compute coordinator* or *compute engine* hereinafter) created on the node will generate multi-threads (or called *threads*) inside in correspondence with the multi-processors. Assume that in Fig. 1, *Host* 0 is a dual-processor node; *Host* 1 is a quad-processor node; *Host* 2 and *Host* 3 are single-processor nodes. Thus, the compute coordinator spawns two threads. The *compute engine* 1 spawns four threads. The multi-threads in an object are distributed to run on the multi-processors in parallel. In the compute coordinator, the original thread (i.e., *thread* 0) acts as the system initiator and coordinator. Other threads work in the identical way to receive and process computational tasks. The *registration mechanism* records the references to all objects and threads in the HiCS. The objects and threads can locate each other through the references in the registration mechanism. The *interaction mechanism* implements the inter-object and inter-thread communication.

3.1.2. Multi-threaded computing objects

Constructed with distributed objects and the associated threads, the hierarchical collaborative system presents a two-level structure. The upper level contains the objects of compute coordinator and compute engines. The lower level is composed of the threads in each of the objects. The HiCS is an adaptive infrastructure as the threads are generated based on the architecture of the selected hosts. Threads are the light-weight objects that consume less system resources in comparison with the high-weight objects (i.e., the compute engines). Running multi-threads, instead of multiple objects, on a multi-processor can improve the computational efficiency.

The multi-threads in an object can work in two modes: *cooperative mode* and *independent mode* depending on the computation pattern of an application. In cooperative mode, the compute coordinator allocates one computational task per compute engine based on the computing power of the host. If a host is a multi-processor, its computing power is the total power of all processors. The threads in an object collaboratively process a computational task. The cooperative mode is suitable to run the applications with high data-dependency such as the N-body problem (see Section 4). This mode can maintain high data locality in a multi-threaded object and reduce the inter-object communication.

In independent mode, the multi-threads in an object work independently. Each thread runs as a compute engine and processes an individual computational task. The computational tasks are allocated to each of the threads based on the computing power of the associated processor. Nevertheless, the threads in an object still occupy fewer resources than multiple compute engines. The independent mode is suitable for the applications with low data-dependency in which low communication is required between the threads. The threads can execute the computational tasks in parallel. The ray tracing method in Section 5 adopts the independent mode.

3.1.3. Unified communication mechanism

The two-level structure of HiCS provides two communication paths. *Shared-data access* is the efficient way through local memory for the communication between the threads within an object. *Remote messaging* is the way for the communication

between distributed objects. The MOIDE model integrates these two paths into a *unified communication mechanism*. The mechanism can choose one of the paths to complete the communication with respect to the locations of the communication peers.

The unified communication mechanism is transparent to the applications. The applications do not need to distinguish the different paths when invoking communication operations in the programs. A *uniform communication interface* is built atop the unified communication mechanism. It provides a library of communication primitives. The applications can call the primitives in identical format, regardless of the potential communication paths. In the primitives, the sender and receiver are specified by the logical identifiers (IDs) without any indication of the locations. A logical ID is assigned to an object or a thread on creation. The logical ID is registered in the registration mechanism along with the reference of the object or thread. When executing a communication primitive, the unified communication mechanism finds the locations of the communication peers according to the references of them. The proper communication path can be determined based on the locations. The unified communication mechanism and interface are included in the interaction mechanism of the HiCS.

Fig. 2 depicts the unified communication mechanism and interface built on the HiCS in Fig. 1. Each thread is identified with a logical ID (ID0–ID7). All threads can call the uniform communication interface. For example, a pair of threads can call the primitive *exchIntArray*() to exchange array of integers:

exchIntArray(*int send ID*, *int recv ID*, *int* [] *send_buf*, *int* [] *recv_buf*,
int send_len, *int recv_len*, *int status*)

send ID logical ID of sender
recv ID logical ID of receiver
send_buf & *recv_buf* send buffer and receive buffer

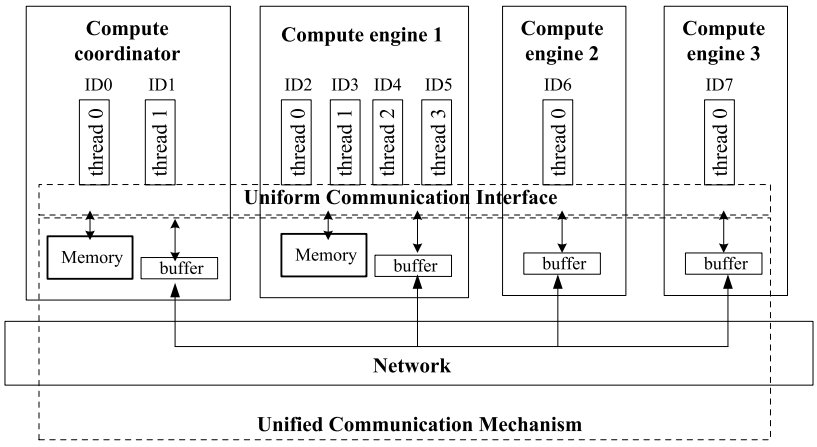

Fig. 2. Unified communication mechanism.

send_len & *recv_len* amount of data in send buffer and receive buffer
status status of communication

The unified communication mechanism integrates the memory and buffers in each host and the system-wide network to form a two-level communication structure. Assume that in Fig. 2, the thread ID2 is the sender that calls a communication primitive. If the receiver ID given in the primitive refers to a thread in the same compute engine (i.e., ID3–ID5), the unified communication mechanism will complete the communication locally by shared-data access. If the receiver is a thread in another object, the communication will be fulfilled by remote messaging through the network. The unified communication mechanism can improve the communication efficiency on the two-level structure of the HiCS.

3.1.4. Dynamic reconfiguration

A HiCS is built on the selected hosts that are predicated to provide the best performance. However, the hosts can be simultaneously occupied by many users and applications. The real performance of the hosts is variable at runtime due to the occurrence of other users and applications. Moreover, the workload of irregular computation may evolve during the execution. In response to the changes in the computational workload and the available resources, the HiCS can perform dynamic reconfiguration to alter its structure or the selected hosts to enhance the performance in computation.

As an object-oriented and multi-threaded infrastructure, the HiCS has the polymorphism to support dynamic reconfiguration. It can conduct *system expansion* to incorporate additional compute engine that is created on a new host to handle the excessive workload appeared at runtime. The system expansion can be made in two directions: *horizontal* and *vertical*. The *horizontal expansion* is performed by the compute coordinator that adds a new compute engine to the HiCS. The new compute engine works in the same way as the existing compute engines to execute the computational task assigned by the compute coordinator. The *vertical expansion* is performed by an overloaded compute engine. If a compute engine is highly overloaded to be a bottleneck in the collaborative computation, it can decide by itself to attach an additional compute engine to share its workload. The additional compute engine is created on a new host. It works under the control of the overloaded one. The new compute engine is invisible to the compute coordinator and other compute engines. The logical structure of the HiCS is unchanged but the computing power has been improved. In Fig. 3, the new engine on *Host* 5 represents a horizontal expansion. The assistant engine on *Host* 6 is attached to the *compute engine* 3 as a result of vertical expansion.

A host may also become overloaded due to the simultaneous occupation by other users and applications. In this case, the compute coordinator can perform *host replacement* to replace the overloaded host with a new host and transfer the computational task from the overloaded host to the new one. The compute coordinator selects a spare or least loaded host in the underlying system and creates a new compute engine on it. Then, the computational task being executed by the compute

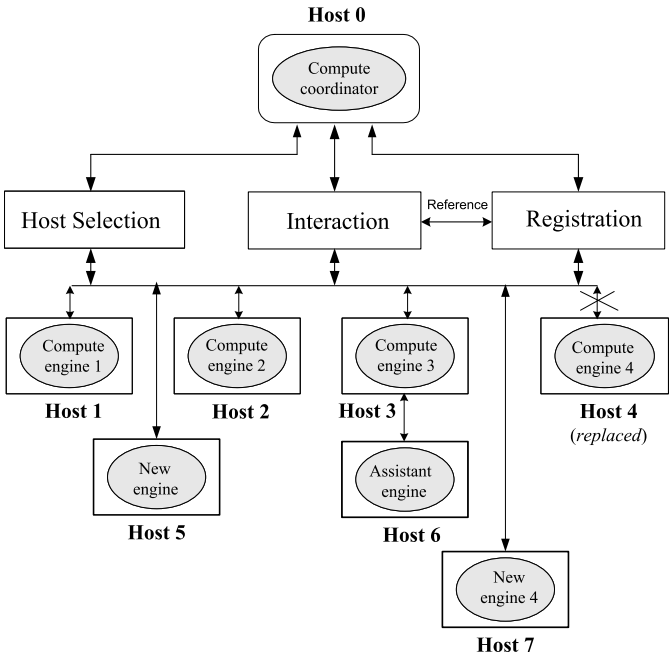

Fig. 3. Dynamic reconfiguration.

engine on the overloaded host is transferred to the new compute engine on the new host. The old compute engine is then terminated and removed from the HiCS. In Fig. 3, the *new engine* 4 on *Host* 7 replaces the old *compute engine* 4 on *Host* 4 by host replacement.

The registration mechanism of the HiCS should be updated accordingly in dynamic configuration. In horizontal expansion, the reference of the new compute engine is added to the registration mechanism. In vertical expansion, no change should be made to the registration. In host replacement, the reference of the old compute engine is replaced by the reference of the new compute engine.

3.2. Uniform programming model

Upon the adaptive computing infrastructure, the MOIDE model provides a uniform programming model for user to develop applications. An application developed on the model is independent of any specific system architecture. Nevertheless, it can be mapped to the system at runtime by creating a HiCS based on the system architecture. Fig. 4 shows the composition of the uniform programming model. The main components are two classes specified for the compute coordinator and compute engine. *Codr* is the class of compute coordinator. It calls the class *StartEngine* at first that performs host selection and creates the compute engines and the HiCS on the selected hosts. Then, it instantiates the class of the application code *Appl* to run

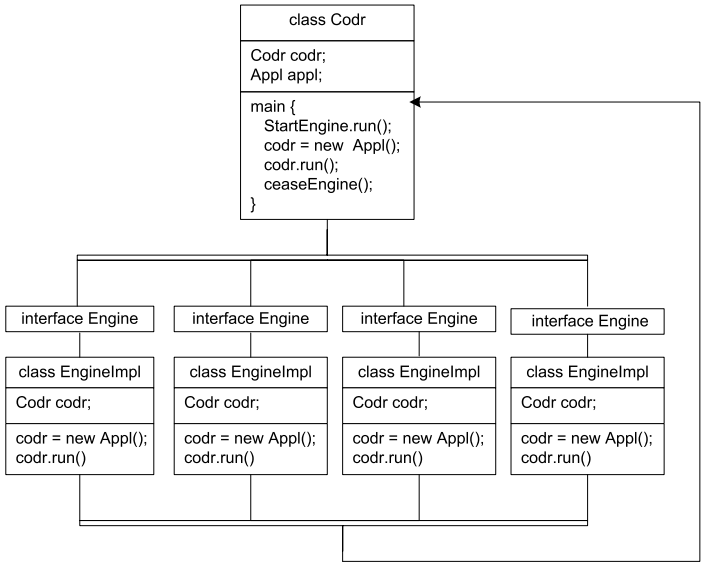

Fig. 4. Uniform programming model.

the computational task. When the execution of an application is completed, the compute coordinator calls the *ceaseEngine* () method to terminate the compute engines and finalize the collaborative computation on the HiCS.

The compute engine is an object that is remotely created on another host by the compute coordinator. According to the convention of distributed object programming, the compute engine is defined as an interface *Engine* and the implementation of the interface *EngineImpl*. The interface declares the methods involved in remote method invocation. The implementation specifies the attributes of a compute engine and the code of the methods declared in the interface. The compute coordinator remotely invokes the construction method in *EngineImpl* to instantiate a compute engine. When running on a multi-processor node, the object of compute coordinator or compute engine will generate a group of threads inside. Then, each compute engine or thread (if existent) instantiates the application class *Appl* and runs the code to execute the computational task.

3.3. Implementation

A runtime system called *MOIDE runtime* is developed to implement the MOIDE-based computation on distributed systems. The MOIDE runtime provides the fundamental classes and mechanisms specified by the MOIDE model. The runtime is implemented in Java and RMI [12]. It provides a platform-independent environment on heterogeneous systems.

Fig. 5 shows the components of the MOIDE runtime. The runtime specifies the templates of the classes *Codr*, *Engine*, *EngineImpl*, and *StartEngine*. With the prede-

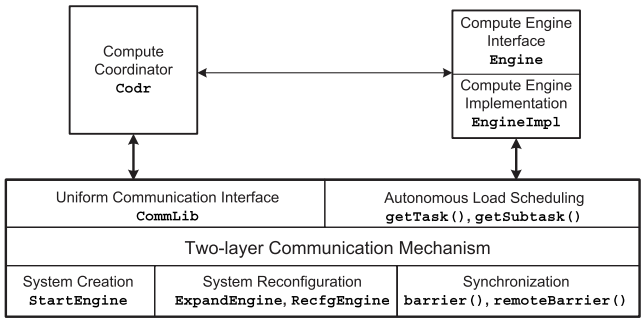

Fig. 5. MOIDE runtime system.

fined templates, a programmer only needs to write the application code, i.e., the class *Appl*, and let it be called in the compute coordinator and compute engine. The multi-threads are created in an object as native threads [18] that can be distributed to run on the multi-processors in parallel with the support of operating systems. The *StartEngine* obtains the resource information from a cluster monitoring tool called *ClusterProbe* [16], which is running on a server to detect the available hosts in the underlying system and report the parameters such as the number of processors, memory space, and current workload of the hosts.

The MOIDE runtime implements the unified communication mechanism and the uniform communication interface. The interface provides a library of communication primitives, called *CommLib*. The following primitives are provided in the library:

send() send data to an object or a thread
receive() receive data from an object or a thread
broadcast() broadcast data to all objects and threads
exchIntArray() exchange array of integers between two objects or threads
exchDoubleArray() exchange array of doubles between two objects or threads
scatter() scatter data to all objects and threads
allReduce() all-to-all reduction on data among all objects and threads

Synchronization is required to coordinate the computing procedure on distributed objects and threads. The MOIDE runtime provides two synchronization methods: (1) the local synchronization method *barrier*() synchronizes the threads within an object; (2) the global synchronization method *remoteBarrier*() imposes system-wide synchronization on all objects and threads. The local synchronization is implemented by an integral barrier manipulated by the threads in an object. The threads exclusively increment the value of the barrier when calling *barrier*(). Local synchronization is achieved when each thread has incremented the barrier once. The global synchronization includes the synchronizations on two levels. On the thread level, a local synchronization *barrier*() is performed by the threads in each object. Then, each object enters a locally synchronized state and waits for the invocation from the

compute coordinator. On the object level, the compute coordinator iteratively polls the state of each compute engine. If all compute engines have reached the locally synchronized state, the global synchronization is accomplished. Then, the compute coordinator signals all compute engines to continue the computation after the synchronization point.

The MOIDE runtime provides the classes for dynamic reconfiguration. The class *ExpandEngine* implements system expansion including horizontal expansion and vertical expansion. The class *RecfgEngine* implements host replacement. To perform any kind of system reconfiguration, an object (compute coordinator or compute engine) executes the operations of host selection and object creation as what the compute coordinator did in the creation of the HiCS. The object calls *StartEngine* to select a new host and calls the construction method in *EngineImpl* to create a compute engine on the host. The new compute engine takes part in the computation by calling the class *Appl*.

In system expansion, the main methods for horizontal and vertical expansions are the same. In horizontal expansion, however, the compute coordinator needs to register the reference of the new compute engine to the registration mechanism and informs other compute engines of the existence of the new member. In vertical expansion, the reference of the new compute engine will not be registered but the compute engine that creates the new engine should inform the compute coordinator about the enhancement of its computing power. The compute coordinator will allocate a computational task to that compute engine later on according to the enhanced computing power. In the *RecfgEngine*, the compute coordinator calls the *ceaseEngine*() method to terminate the old compute engine and replaces the reference of the old compute engine with the new one in the registration mechanism.

The MOIDE runtime supports *autonomous load scheduling* that can evenly distribute the non-predetermined workload of an irregular application to all objects and threads. The autonomous load scheduling does not require a dedicated task scheduler. All computational tasks are maintained in a task pool on the side of the compute coordinator. Each object or thread autonomously fetches a task from the pool on demand. By the autonomous task fetching, the computational workload can be gradually distributed to the objects and threads during the execution and the workload can be automatically balanced on them without load balancing operation.

The autonomous load scheduling can be implemented by making use of the *single-sided* feature of remote method invocation. The inter-object communication can be activated on one side of the sender or the receiver without the communication operation issued on the other side. By the single-sided feature, a compute engine or a thread can take a task from the task pool by itself without the direct involvement of the compute coordinator. As a result, the computations are carried out asynchronously on all objects and threads so that the highest parallelism can be realized in the computation.

The MOIDE runtime provides the *getTask*() method for an object or a thread to fetch a task from global task pool and the *getSubtask*() method for a thread to get a subtask from local subtask queue in a compute engine. The autonomous load sched-

uling is suitable for the applications with low data-dependency. For more information about the implementation of the MOIDE runtime, please refer to [30].

4. Distributed *N*-body method

A distributed N-body method is developed based on the MOIDE model. N-body is a compute-intensive and communication-intensive problem that includes high data-dependency. The straightforward method for N-body problem computes pair-wide force influence that produces a high computational complexity as $O(N^2)$. To reduce the complexity, hierarchical methods are designed to compute the approximated force influence based on the fact that a body requires gradually less data, in less precision, from the bodies that are farther away [26,27]. The Barnes–Hut method [5] is a hierarchical approach using a tree structure to represent the body distribution in a space.

Parallel N-body methods can be designed based on the sequential Barnes–Hut method. Singh proposed a parallel N-body method based on shared-address-space model in [26,27]. In this method, concurrent processes collaboratively build a Barnes–Hut tree (BH tree in short) in shared memory and each process computes the force influences on a subset of bodies by concurrently traversing on the tree. This method is only suitable for the shared-memory systems like SMP machines. Another example is Salmon and Warren's method designed for distributed-memory system [32]. This method recursively divides a space into domains. A local essential tree is built for each domain. As each body needs a fraction of the BH tree in the computation of force influences, the local essential tree is the union of the tree fractions required by all bodies in a domain. The method uses data keys and hash table to map the cells of local essential trees to the memory locations. Each cell is assigned with a key that is generated from the coordinate of its location in the space. The key is translated to a memory location through the hash table. The construction of local essential trees and the search in the hash table are time-consuming operations.

Our distributed N-body method on the MOIDE model is derived from the Barnes–Hut method. The method is featured with a *distributed tree structure*. The task decomposition, force computation, and dynamic load balancing are all implemented based on the distributed tree structure. The first step of the method is decomposing a space into domains. Each domain is assigned to an object that is responsible for computing the force influences on the bodies in the domain. The space decomposition is realized by partitioning the BH tree. According to the specification of the BH tree, a body is inserted to the tree according to its location in the space so that the bodies in neighbor are inserted to the leaves adjacent in the tree. With such a structure, the BH tree assures the Peano–Hilbert ordering of the bodies on the path of depth-first traversal [26,27]. The partitioning of the tree divides the leaves into subsets. The Peano–Hilbert ordering can guarantee that the tree partition by a depth-first traversal can assign the neighboring bodies (on the leaves) to the same subset of leaves. The subsets of leaves correspond to the domains in the space.

The number of bodies in a subset is proportional to the computing power of the target host.

Fig. 6(a) shows an N-body problem in 2D space. To run this problem on four hosts (e.g., two quad-processor nodes and two dual-processor nodes), the compute coordinator builds the BH tree and makes the tree partition as shown in Fig. 6(b). The leaves are partitioned into four subsets (subset A–D). The order of the leaves in the depth-first traversal is from the left-most leaf to the right-most leaf, which is equivalent to the Peano–Hilbert ordering of the bodies in the space as shown in Fig. 6(a) from the start point to the end. The subsets A–D correspond to the four domains A–D in the space. As there are totally 48 bodies in the space, a domain allocated to a quad-processor node contains 16 bodies and a domain to a dual-processor node contains 8 bodies.

After the space decomposition, the compute coordinator allocates one domain per compute engine. Each compute engine builds a *subtree* for the domain assigned to it using the same approach of building a BH tree and computes the force influences on the bodies of the domain. All subtrees form a *distributed tree structure* as shown in Fig. 6(c). On a SMP node, the multi-threads work in the cooperative mode. They collaboratively build a subtree and each thread computes the force influences on a portion of the bodies.

To compute the force influences, an object needs the data of the subtrees on other objects. If all subtrees are broadcasted among the objects, high communication overhead will be incurred. In order to share the information of the subtrees at low communication cost, a *partial subtree* scheme is designed based on the property of the

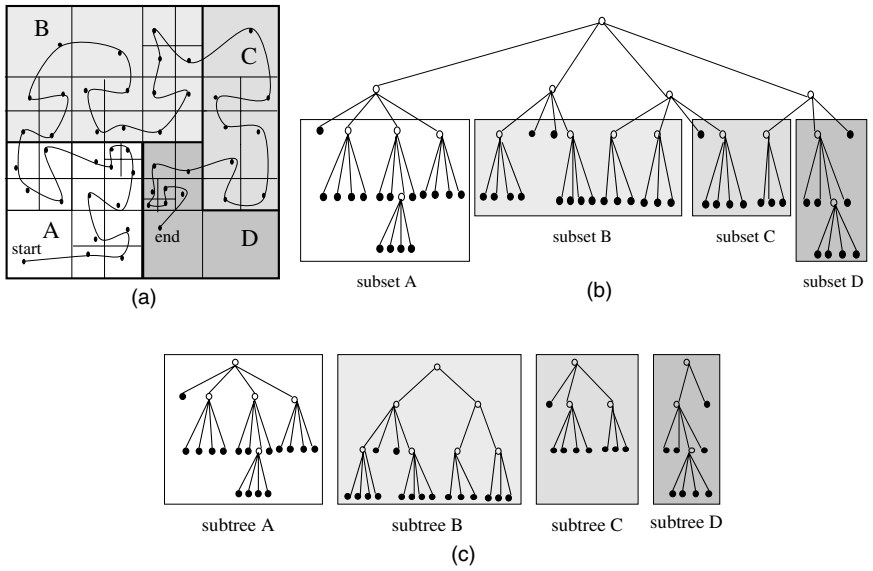

Fig. 6. Space decomposition and distributed tree structure: (a) space decomposition; (b) tree partitioning; (c) distributed tree structure.

BH tree. In the tree, a cell (not leaf) is *the center of mass* in a subspace. The center of mass can represent the accumulative force influence to a remote body from all bodies in the subspace (i.e., the leaves beneath it) provided that the distance from the center of mass to the remote body satisfies the condition: $l/d < \theta$, where l is the width of the subspace; d is the distance from the center of mass to the remote body; θ is a user-defined accuracy parameter (between 0 and 1.0). That means if a body is far enough from the center of mass, the force influences imposed on the body from all bodies in a subspace can be approximated by the force influence from the center of mass. This feature can be harnessed to reduce the data propagation. Instead of a complete subtree from another object, an object requires only a fraction, called *partial subtree*, of the subtree. In fact, each object requires different fraction of a subtree, depending on the distance between two corresponding domains. Thus, an object needs to construct different partial subtrees from its own subtree and distribute them to other objects.

A partial subtree usually contains fewer cells if the distance between two corresponding domains are far from each other. As the partial subtrees are built based on the same distance condition as in the force computation, they can provide other objects with most of the data required in the force computation. However, if an object needs more data than a partial subtree supplies, it has to access the complete subtree in another object by remote method invocation. Even so, the partial subtree scheme can still reduce the overall communication overhead [30]. The partial subtree scheme is an improvement of our previous scheme in [31], where a partial subtree was simply a duplication of the top levels of a subtree and it was irrelevant to the distance condition. The modified partial subtree scheme presented here can undoubtedly provide more essential data to the objects and reduce the remote subtree access.

Although the idea of the partial subtree scheme is similar to the local essential tree in Salmon and Warren's method, our scheme is more suitable for distributed-memory systems. The scheme of local essential tree needs to examine the distance between each pair of the bodies between two domains. The construction of local essential tree should be more time-consuming than the partial subtree. These costs might not be a significant problem in their method because it was implemented on supercomputers. On the other hand, our N-body method aims to run on distributed systems where the communication cost is a decisive factor to the overall performance. The partial subtree scheme can effectively reduce the communication overhead. Moreover, the motivation of using keys and hash table in their method came from the difficulty in representing a distributed tree using the pointers in traditional languages such as FORTRAN 90 and HPF, especially when referring to the cells in a separate memory space on anther processor. Differently, our method is based on the object-oriented paradigm and implemented in Java. The distributed tree structure can be readily constructed by object references. The transmission of partial subtrees can be easily implemented by remote method invocation.

The N-body method simulates the evolution of a physical system by the iterative computation of force influences and the resultant body motion. Although the bodies move forward in small pace, the body motion will eventually lead to imbalanced body distribution and therefore imbalanced computational workload in the domains. Our N-body method adopts a load balancing strategy that re-decomposes

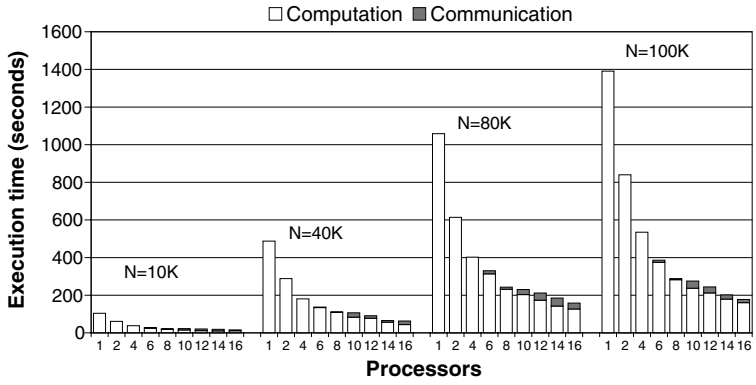

Fig. 7. Execution time breakdowns of the N-body method.

the space to generate new domains with balanced workload. The workload in a domain is measured in the amount of the computational operations performed for the bodies in the domain. The compute coordinator periodically examines the workload in each domain. If the workload in some domain exceeds a predefined ratio of the average workload, the compute coordinator will perform space re-decomposition using the same approach as the space decomposition shown in Fig. 6. The new domains are allocated to the compute engines. The force computation will continue based on the new domains.

The distributed N-body method has been tested on a cluster of four SMP nodes installed with the MOIDE runtime system. The SMPs are quad-processor nodes linked by Fast Ethernet switch. Fig. 7 shows the execution time breakdowns with 10K to 100K bodies. In the cases of one to four processors, one SMP node is used with one object and a certain number of threads on it. The threads share a BH tree and no partial subtree will be built. Therefore, there is not communication cost in these cases. Apparent communication overhead emerges when more than one SMP node is involved. The objects on the SMP nodes need to propagate partial subtrees and perform remote subtree access when required. As Fig. 7 shows, the communication time remains at a low level in the total execution time. No significant growth of the communication time occurs when increasing the number of processors and the number of bodies. The proportion of the communication time in the total execution time even decreases when the number of bodies increases. The results manifest that the distributed tree structure with the partial subtree scheme provides a communication-efficient data structure for the distributed N-body method.

5. Ray tracing

Ray tracing is a graph rendering algorithm that generates an image from the description of the objects in a scene. Parallel ray tracing methods usually partition the

image into blocks and render them in parallel. Dynamic load balancing is required as the workload of rendering each block is non-predetermined and highly diverse.

Parallel ray tracing methods are generally based on the message passing paradigm such as the methods in [8,24]. These methods use the master/slave scheduling to allocate the blocks. A master process acts as a dedicated load scheduler. It keeps on detecting the requests for block allocation from other processes and allocates one-block-a-time to them. So, its execution time is mostly spent in waiting. Some task allocation schemes allow the master process to perform the block rendering as well and check the incoming requests at specified moments. In this approach, the requests from other processes may not get immediate response and those processes have to wait in idle.

The ray tracing method based on the MOIDE model can use the autonomous load scheduling for block allocation so that all objects and threads can be devoted to the rendering operations. The multi-threads work in the independent mode. As the rendering of a block is independent from other blocks, a compute engine or a thread can autonomously fetch a new block from the global task pool once it has finished the rendering of the previous block. With the autonomous block fetching and independent rendering, the computation and communication are conducted asynchronously on all objects and threads. Hence, dynamic load balancing can be automatically realized and the parallelism in the computation can be fully exploited.

The autonomous load scheduling is also different from the task-stealing method in [25]. The task-stealing method includes an initial task allocation to the processes. When a process runs out of the initial tasks, it inquires other processes to obtain extra tasks. Differently, the autonomous load scheduling does not require any initial task allocation, neither task reallocation. The computation can start and proceed completely in asynchrony on all objects and threads. So, the autonomous load scheduling can generate higher parallelism than the task-stealing method.

The ray tracing method has been tested on the cluster of four quad-processor SMP nodes. The autonomous load scheduling is compared with the master/slave scheduling. Fig. 8(a) shows that the autonomous load scheduling results in lower execution time than the master/slave scheduling method.

To analyze the performance of the autonomous load scheduling in detail, Fig. 8(b) shows the execution time breakdowns associated with each of the threads. When $P = 4$ (four processors in one SMP node), the four threads exist in the same object. In this case, no communication is required and the computational workload is automatically balanced on all threads. When $P = 8$, two SMP nodes are used with four threads each. The four threads on the left reside in the compute coordinator. These threads can fetch the blocks locally. No communication is needed for them. The four threads on the right belong to the compute engine on another SMP node. They need to perform remote communication to send a rendered block back and fetch a new block from the global task pool. So, the communication time occurs on them.

The difference in the communication times between the threads in different objects is also the result of the single-sided feature and the implementation of the remote method invocation mechanism RMI. Distributed objects do not interact with each other directly but through their agents—the *skeleton* on the compute coordinator

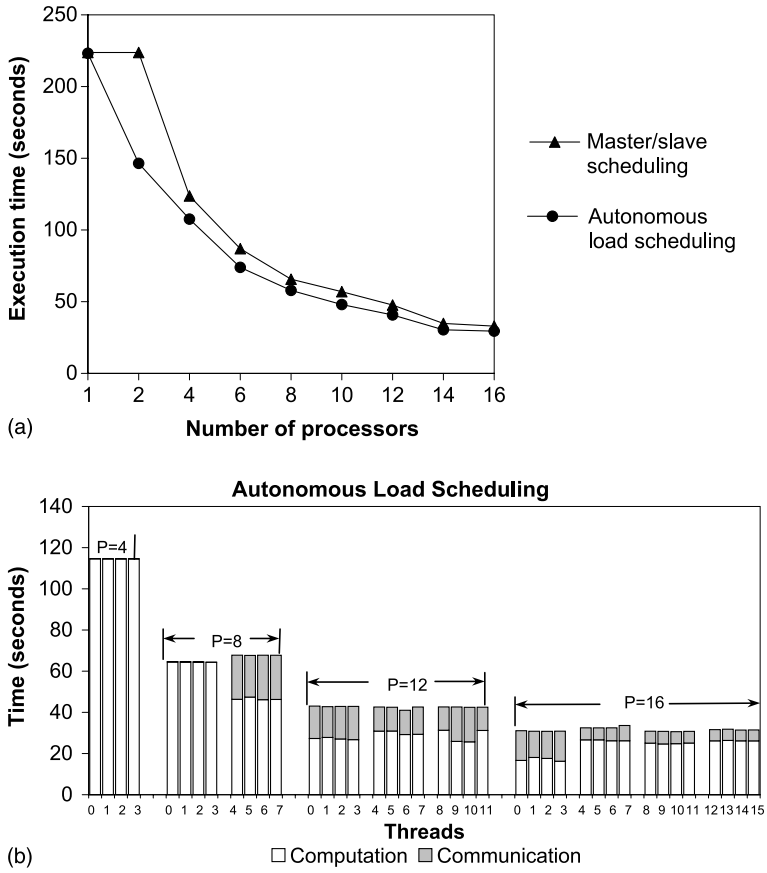

Fig. 8. Execution time of the ray tracing methods.

and the *stub* on a compute engine. The operation of block fetching from the compute engines does not produce direct interference to the computation on the compute coordinator. Consequently, the communication time is negligible on the compute coordinator when $P = 8$. However, the interaction between the skeleton and the stub does affect the computation on the compute coordinator. The compute coordinator has to preempt a processor to the skeleton for it to process the block fetching. The processing time of the skeleton is counted as the communication time of the compute coordinator. As a result, obvious communication time appears on the threads in the compute coordinator when $P = 12$ and 16. These threads need to preempt the processors more frequently to the skeleton to process the block fetching from the threads in remote compute engines. Nevertheless, the total communication time on the side of the compute coordinator remains lower than the total communication time on all compute engines. Fig. 8(b) also shows that the execution time is almost balanced on all threads. This phenomenon reflects the automatic load balancing achieved by the autonomous load scheduling.

6. Conjugate gradient

The conjugate gradient (CG) is an iterative method for solving large sparse linear system $Ax = b$ [14]. It computes the approximated solution x by the iteration:

$$x_k = x_{k-1} + \alpha_k p_k$$

where α_k is a scalar step size and p_k is the direction vector.

Parallel CG method contains vector communication operations such as vector reduction and transposition. The performance of the method is determined by the communication efficiency. We implement a CG method based on the MOIDE model to utilize the unified communication mechanism. The CG method is modified from the CG code in the NAS Parallel Benchmarks (NPB) [4]. In our method, all threads work in the independent mode. They call the vector communication primitives provided by the uniform communication interface. The unified communication mechanism in turn implements the communication.

The CG method has been tested on the cluster of four quad-processor SMP nodes. Fig. 9(a) shows the execution time breakdowns with the matrix size $n \times n$. The communication time increases along with the increase of the number of processors, especially when more than one SMP node is involved, due to the remote messaging. However, the unified communication mechanism can accelerate the vector communication and therefore reduce the overall execution time with the increase of the processors. Fig. 9(b) shows the speedups calculated from the execution time. High speedup can be achieved on large matrix size.

As a comparison, we use a single-threaded CG method to demonstrate the disadvantage of sole remote messaging communication. Instead of creating a multi-threaded compute engine, the single-threaded method creates multiple compute engines (i.e., high-weight objects) on a SMP node. All communications between the compute engines on the same or different SMP nodes are implemented through remote messaging. Fig. 10(a) shows the execution time breakdowns of the single-threaded method on two quad-processor SMP nodes. Compared with the results in Fig. 9(a), the single-threaded method incurs excessive communication overhead that leads to high execution time. The execution time on eight processors (two SMP nodes) is even worse than that on four processors (one SMP node). The speedups in Fig. 10(b) also show the poor performance of the single-threaded method.

7. Related work

The wide use of distributed systems for high-performance computing has been attracting remarkable research efforts in developing computing infrastructures and environments on them. The related work includes the generic programming paradigms for distributed computing and the specific methodologies for solving irregularly structured problems.

AppLeS (Application Level Scheduler) project [28] provides the mechanisms and paradigms for the resource configuration and load scheduling of the applications on

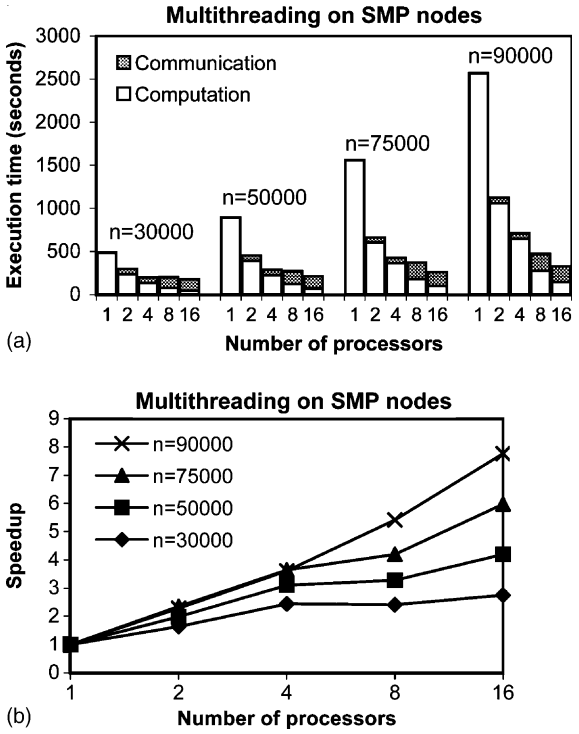

Fig. 9. Performance of the CG method based on MOIDE model: (a) execution time breakdowns; (b) speedups.

distributed heterogeneous system. The application-level scheduling agents are the mechanisms for scheduling individual applications based on their static and dynamic information and the available resources. A parallel ray tracing application is implemented based on the master/slave scheduling to study the application scheduling policies [24]. As a comparison, the focus of our MOIDE model is on the development of an adaptive computing infrastructure on heterogeneous systems. The applications developed on the MOIDE model can be adaptively mapped to the hosts at runtime to utilize the architectural features. Unlike the master/slave scheduling approach in AppLeS, our autonomous load scheduling approach can realize dynamic load balancing and exploit high parallelism in irregular computation without any dedicated task scheduler and load balancing operation.

KeLP (Kernel Lattice Parallelism) [2] is a programming model for implementing portable scientific applications on distributed-memory computers. It provides a set of programming abstractions to represent the data layout and data motion patterns in block-structured scientific computing on SMP clusters. The KeLP run-time system is implemented as a C++ library to support the general blocked data decompositions and manage the low-level implementation details such as message-passing, processes, threads, synchronization, and memory allocation. On contrary, the MOIDE model

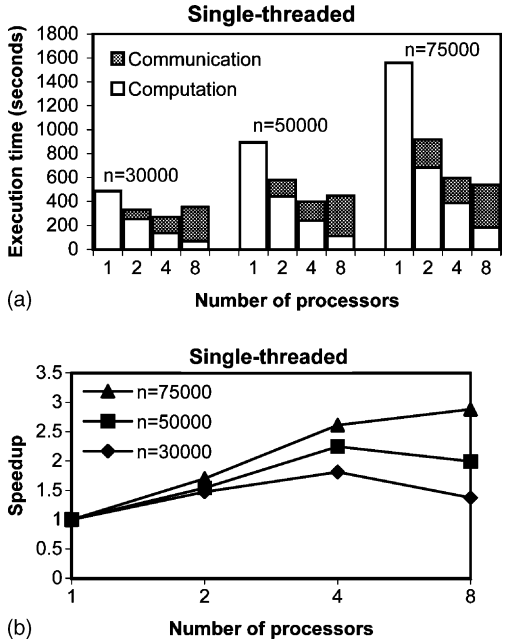

Fig. 10. Performance of single-threading CG method: (a) execution time breakdowns; (b) speedups.

is a general-purpose model that provides the mechanisms such as the hierarchical collaborative system and the unified communication mechanism for solving irregularly structured applications and other applications on distributed heterogeneous systems. The model provides the flexibility for the developers to implement varied approaches on it for different applications.

SIMPLE model [3] develops the methodology for high-performance programming on clusters of SMP nodes. The methodology is based on a small kernel of collective communication primitives that makes use of the hybrid shared-memory on a SMP node and the message passing paradigms. The communication primitives are provided in three modules: (1) the Internode Communication Library (ICL) provides an MPI-like small kernel for inter-node communication; (2) the SMP Node Library contains the communication primitives for SMP node; (3) the SIMPLE Communication Library is built upon the ICL and SMP Node Libraries. Differently, our MOIDE model builds a unified communication mechanism on heterogeneous systems at runtime based on the architecture of the hosts. Applications can identically call the communicate primitives that will be efficiently accomplished by the unified communication mechanism.

JavaParty [13,20] is a programming layer on top of Java and RMI for easy porting of multi-threaded Java programs to distributed environments such as clusters of workstations. It extends Java with a new class modifier *remote* by which remote classes and instances can be created and accessed in uniform way on any node in

distributed environment. It supports automatic object distribution for load balancing and supports object migration to reduce communication. Irregular applications such as a geophysical method *Veltran* are implemented to demonstrate the performance of JavaParty [11]. The objective of JavaParty is to improve the reusability and portability of Java code. Although the idea of the MOIDE model is similar to JavaParty, our model aims at creating an adaptive computing infrastructure on distributed heterogeneous system. The applications developed on the model can be adaptively mapped to the architecture of underlying system to achieve efficient computation and communication on the system. The MOIDE model provides the mechanisms such as dynamic reconfiguration, unified communication mechanism, and autonomous load scheduling to support the high-performance solutions for irregularly structured problems as well as other applications. The MOIDE runtime system is also a portable environment. It provides the reusable templates, communication library, and implementations for the development and execution of the applications on varied systems.

The IPA (Irregular Parallel Algorithms) project proposes *nested data parallelism* to express the irregular computations and investigates the incorporation of nested data parallelism in the programming languages such as FORTRAN 95/HPF and Java [7,21]. Two approaches are used to cope with the load imbalance in the computations: (1) using fine-grained threads and thread migration to balance the load; (2) flattening nested parallelism through compilation techniques to create an unnested data-parallel computation to perform the correct amount of work. A runtime support library based on FORTRAN intrinsic functions and HPFLIB routines supports the data-parallel computations on supercomputers. Our MOIDE model is a more flexible object-oriented computing infrastructure on distributed systems. Various algorithms can be developed to exploit the parallelism in irregular computation and enhance the communication efficiency based on both the computation pattern and the system architecture.

The Scandal [23] project develops a portable, interactive environment for programming a wide range of supercomputers. It proposes a parallel language *NESL* as a portable interface for programming a variety of parallel and vector supercomputers and as a basis for designing data-parallel algorithms. It also develops fast implementations of parallel algorithms for various irregular problems on different parallel machines to study the methods for mapping the algorithms onto existing parallel machines and communication topologies as well as the efficient implementations. Compared with Scandal, the MOIDE model also possesses architecture-independence and adaptability. Furthermore, the object-oriented features make the MOIDE model more powerful to adaptively implement high-performance computation on heterogeneous system.

8. Conclusions

MOIDE is a distributed object model for solving irregularly structured problems. It creates an adaptive computing infrastructure and provides a uniform program-

ming model on distributed heterogeneous systems. Applications can be developed on the uniform model and adaptively mapped to the system architecture at runtime. The computing infrastructure supports dynamic reconfiguration to meet the evolution of computational requirement and available resources. The unified communication mechanism integrates shared-data access and remote messaging to support the efficient communication on heterogeneous systems. Autonomous load scheduling is proposed as an approach for dynamic load balancing to exploit high parallelism in irregular computation. A runtime system provides a platform-independent environment to implement MOIDE-based computing.

In the future work, we will extend the MOIDE model to wide-area environment. The research will concentrate on improving the scalability of the model on hundreds to thousands of computer nodes across geographically distributed sites. To coordinate the collaborative computations on them, the model should be expanded to a multi-level hierarchy in order to organize the autonomous computing in distributed domains. The system coordination and communication mechanisms should be modified to efficiently support the interactions between the domains. New task scheduling strategies should be designed to support the load balancing on wide-area environment, which should take into account the remote data access and long-latency communication as well as the varied performance of the resources. New programming methodologies, algorithms, and data structures should be developed for the applications in wide-area environment.

References

[1] I. Alfuraih, S. Aluru, S. Goil, S. Ranka, Parallel construction of k–d tree and related problems, in: Proc. 2nd Workshop on Solving Irregular Problems on Distributed Memory Machines, 1996.
[2] S. Baden, S. Fink, A programming methodology for dual-tier multicomputers, IEEE Trans. Software Eng. 26 (2000) 212–226.
[3] D. Bader, J. JáJá, SIMPLE: a methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs), J. Parallel Distrib. Comput. 58 (1999) 92–108.
[4] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, M. Yarrow, The NAS parallel benchmarks 2.0, Technical Report NAS-95-020, NASA Ames Research Center, 1995. Available from <http://www.nas.nasa.gov/Software/NPB/>.
[5] J. Barnes, P. Hut, A hierarchical $O(N \log N)$ force-calculation algorithm, Nature 324 (1986) 446–449.
[6] S. Bhatt, M. Chen, J. Cowie, C. Lin, Object-oriented support for adaptive methods on parallel machines, Sci. Comput. 2 (1993) 179–192.
[7] B. Blount, S. Chatterjee, M. Philippsen, Irregular parallel algorithms in Java, in: Proc. IRREGULAR'99, Lecture Notes in Computer Science, vol. 1586, Springer-Verlag, 1999, pp. 1026–1035.
[8] A. Fava, E. Fava, M. Bertozzi, MPIPOV: a parallel implementation of POV-Ray based on MPI, in: Proc. Euro PVM/MPI'99, Lecture Notes in Computer Science, vol. 1697, Springer-Verlag, 1999, pp. 426–433.
[9] R. Garmann, Locality preserving load balancing with provably small overhead, in: Proc. IRREGULAR'98, Lecture Notes in Computer Science, vol. 1457, Springer-Verlag, 1998, pp. 80–91.
[10] T. Gautier, J. Roch, G. Villard, Regular versus irregular problems and algorithms, in: Proc. IRREGULAR'95, Lecture Notes in Computer Science, vol. 980, Springer, 1995, pp. 1–25.
[11] M. Jacob, M. Philippsen, M. Karrenbach, Large-scale parallel geophysical algorithms in Java: a feasibility study, Concurrency: Practice Experience 10 (1998) 1143–1154.

[12] Java remote method invocation-distributed computing for Java, White Paper. Available from <http://java.sun.com/marketing/collateral/javarmi.html>.
[13] JavaParty. Available from <http://www.ipd.uka.de/JavaParty/>.
[14] V. Kumar, A. Grama, A. Gupta, G. Karypis, Solving sparse systems of linear equations, in: Introduction to Parallel Computing: Design and Analysis of Algorithms, Benjamin Cummings, 1994, pp. 433–435.
[15] M. Lewis, A. Grimshaw, The core Legion object model, in: Proc. HPDC'96, IEEE Computer Society, 1996, pp. 551–561.
[16] Z. Liang, Y. Sun, C. Wang, ClusterProbe: an open, flexible and scalable cluster monitoring tool, in: Proc. IWCC'99, IEEE Computer Society, 1999, pp. 261–270.
[17] P. Liu, J. Wu, A framework for parallel tree-based scientific simulations, in: Proc. ICPP'97, IEEE Computer Society, 1997, pp. 137–144.
[18] S. Oaks, H. Wong, Java Threads, second ed., O'Reilly, Sebastopol, CA, 1999.
[19] L. Oliker, R. Biswas, R. Strawn, Parallel implementation of an adaptive scheme for 3D unstructured grids on the SP2, in: Proc. IRREGULAR'96, Lecture Notes in Computer Science, vol. 1117, Springer, 1996, pp. 35–47.
[20] M. Philippsen, M. Zenger, JavaParty—transparent remote objects in Java, Concurrency: Practice Experience 9 (1997) 1225–1242.
[21] J. Prins, S. Chatterjee, M. Simons, Irregular computations in Fortran-expression and implementation strategies, Sci. Program. 7 (1999) 313–326.
[22] V. Ramakrishnan, I. Scherson, Executing communication-intensive irregular programs efficiently, in: Proc. IRREGULAR'2000, Lecture Notes in Computer Science, vol. 1800, Springer-Verlag, 2000, pp. 457–468.
[23] Scandal project. Available from <http://www.cs.cmu.edu/~scandal/>.
[24] G. Shao, R. Wolski, F. Berman, Performance effects of scheduling strategies for master/slave distributed applications, in: Proc. PDPTA'99, CSREA, Sunnyvale, CA, 1999.
[25] J. Singh, A. Gupta, M. Levoy, Parallel visualization algorithms: performance and architectural implications, IEEE Comput. 27 (1994) 45–55.
[26] J. Singh, J. Hennessy, A. Gupta, Implications of hierarchical N-body methods for multiprocessor architectures, ACM Trans. Comput. Syst. 13 (1995) 141–202.
[27] J. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, Load balancing and data locality in adaptive hierarchical N-body methods: Barnes–Hut, fast multipole, and radiosity, J. Parallel Distribut. Comput. 27 (1995) 118–141.
[28] S. Smallen, W. Cirne, J. Frey, F. Berman, R. Wolski, M. Su, C. Kesselman, S. Young, M. Ellisman, Combining workstations and supercomputers to support grid applications: the parallel tomography experience, in: Proc. HCW'2000, IEEE Computer Society, 2000, pp. 241–252.
[29] N. Soundarajan, On the specification, inheritance, and verification of synchronization constraints, in: Proc. FMOODS '97, Chapman & Hall, London, 1997.
[30] Y. Sun, A Distributed Object Model for Solving Irregularly Structured Problems on Distributed Systems, Ph.D. Thesis, Department of Computer Science and Information Systems, The University of Hong Kong, 2001. Available from <http://www.csis.hku.hk/~ydsun/thesis/>.
[31] Y. Sun, Z. Liang, C. Wang, Distributed particle simulation method on adaptive collaborative system, Future Gener. Comput. Syst. 18 (2001) 79–87.
[32] M. Warren, J. Salmon, A parallel hashed oct-tree N-body algorithm, in: Proc. Supercomputing'93, IEEE Computer Society, 1993, pp. 12–21.
[33] J. Watts, S. Taylor, Practical dynamic load balancing for irregular problems, in: Proc. IRREGULAR'96, Lecture Notes in Computer Science, vol. 1117, Springer, 1996, pp. 299–306.
[34] S. Woo, M. Ohara, E. Torrie, J. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proc. ISCA'95, IEEE Computer Society, 1995, pp. 24–36.