1000-9825/2002/13(03)0342-12 ©2002 Journal of Software 软 件 学 报 Vol.13, No.3

A Distributed Object Based Framework for Parallel Computations*

LI Guo-dong, ZHANG De-fu

(*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093, *China*)

E-mail: dong@dislab.nju.edu.cn

http://www.nju.edu.cn

Received April 5, 2001; accepted July 30, 2001

Abstract: The computational and compositional features are very important while constructing parallel software for the workstation clusters. However, lack of suitable supporting environment for parallel software development makes most existing distributed parallel software systems very weak in these two aspects, especially in the compositional feature. In this paper, a distributed object based framework for parallel computation is proposed. The goal of the framework is to achieve high efficiency for parallel computing, to construct a mechanism to encapsulate and reuse parallel programs, and to guarantee load balancing and fault tolerance. The framework is a four-layer model that includes an object-group layer and a mobile object layer. The experimental results verify the efficiency of the scheme.

Key words: distributed object; mobile object; parallel computing; workstation clusters; framework

Parallel computing has become the natural medium for solving a wide variety of computationally intensive problems in a very fast and efficient manner. Currently, more and more distributed software systems base their platforms on heterogeneous connected workstation clusters. However, the inherent complexity of many parallel computing applications makes development using existing parallel programming paradigms both time-consuming and error-prone.

Recently, many parallel computing environments such as MPI[1] and PVM[2] have been proposed, but these environments just provide a message-passing library to be used by the programmer, the programmer must synchronize the communications between processes (or threads) by himself. The drawbacks of using this mechanism are: (1) Similar to using assemble language to write sequential programs, writing parallel program is very difficult and coding is often out of control. (2) The problems of load balancing and fault tolerance are not taken into consideration. (3) Software is hard to be reused.

The adoption of object-oriented (OO) technology is currently a major trend in the software industry. OO technology emphasizes the importance of reusability, maintainability, flexibility and modularity in the software development process, thus enhancing the quality of the software and reducing its development cost. The parallel computing applications are inherently complex in nature, and are further complicated by communication, concurrency and synchronization issues, therefore, combining technique of distributed object with parallel software development is a natural demand.

* Supported by the National High Technology Development 863 Program of China under Grant No. 863-306-ZT06-04-4 (国家863高科技发展计划)

LI Guo-dong was born in 1975. He is an MS candidate at the Department of Computer Science and Technology, Nanjing University. He received a B.S. degree in computer science from Zhongshan University in 1997. His research interests are parallel and distributed computing. **ZHANG De-fu** was born in 1937. He is a professor and doctoral supervisor of Nanjing University. His current research areas include parallel processing and distributed systems.

In this paper we present a software system that greatly eases the burden placed on application developers to implement and maintain building blocks for parallel adaptive applications. Our software system assists application programmers to address the following fundamental issues in parallel computing:

· Distributed Object and Mobile Object: Computation or information processing in object-oriented (OO) framework is represented as a sequence of message passing among objects. Thus, the decomposition of a system into a collection of concurrently executable objects is flexible and the resulting system structures become very natural. Distributed OO programming/system, with concepts of dynamic binding, information hiding and message passing, is a powerful programming and design methodology. Meanwhile, mobile object extend remote procedure calls (RPCs) by letting a object to be executed at the most appropriate site among a cluster of Internet or intranet connected workstations. Object migration is performed by system transparently to the application and without relying on operating system[3]. Thus parallel software design can benefit greatly from introducing distributed object and mobile object.

· Object Group: Object-Group can be used for addressing a set of objects which share some common characteristics. For instance, some objects might want to be informed whenever a certain event occurs. An object-group holding these objects can be created and a multicast can be sent to the group whenever the event occurs. Object-Groups encapsulate an internal, possibly replicated state and make it accessible through a set of well-defined methods.

· Load Balancing: Large-Scale, high-performance parallel machines generally consist of many nodes (often SMP nodes), which coordinate in a loosely synchronous fashion. In order to utilize better the available resources, it is important to avoid overloading some nodes while leaving the other idle. The software support system should facilitate the implementation of the dynamic load-balancing strategies. An overloaded site should forward some objects for other sites with less workload.

· Fault tolerance: User object failure and site failure cannot be neglected in the distributed system. In our scheme, availability and fault tolerance are increased by using an object group to implement the services. Each object can fail independently from the other objects in the group. The service then retains operational as long as at least one of the group members is operational.

We propose a distributed object based framework for parallel computations to provide an efficient, robust and integrated environment for parallel computing on communicating processors. Section 1 devises a four-layer architecture for parallel computing, each layer is discussed in detail. Section 2 focuses on some of the functions and services in the architecture. Then Section 3 shows the experimental results. Finally the conclusion is given.

1 Framework

1.1 Architecture

The framework includes four layers: application layer, group-object layer, mobile object layer and low-level messaging layer (Fig.1). The functionality of application layer is to enable the activation, monitoring and termination of contact between a user and a computing application, the components of this layer must be installed into the user hosts. The group-object layer enables application objects to communicate transparently and the objects completely responsible for their modification, protection, execution and communication, so the difficulty of designing object communication and reliability is relieved. The mobile object layer enables the forwarding of mobile objects, allows applications to be able to pick up and move from one site to another in which the decisions on when and where to move are made autonomously. The low-level messaging layer provides a set of interfaces for the distributed object system, in particular, the MPI (message passing interface) is adopted while designing and implementing my platform, other system such as PVM and Express can be accommodated in this layer too.

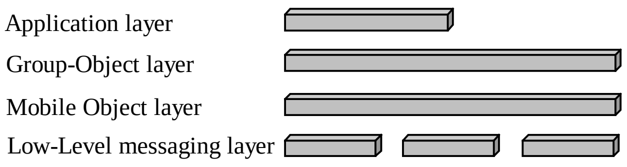


Fig.1 System architecture

1.2 Application layer

1.2.1 Distributed objects and object group

The central issues in exploitation of distributed computation are what resource should be distributed, whose activities should be carried out in cooperative way and how such activities should interact with one another. In designing a software system that exploits modularity and distribution, these issues boil down to the problem of how the system should be decomposed into components that can be activated cooperatively and what function should be given to each component to enhance reliability. To maintain system modularity and transparency, decomposition should be natural and modular. To achieve reliability, information objects should be replicated and works cooperatively to maintain the consistency.

In our system, a communication object is self-contained and provided with a unified communication interface. The objects are grouped and they cooperate by passing messages. It intends to guarantee properties of message ordering, dynamic group membership and fault tolerance.

1.2.2 The “divide and be conquered” model

In the traditional model, the originator (the node which initiates the computation) identifies a set of sites on the network as being available to it. It then initiates the computation by sending data (and possibly code) to compute on. The originator is in charge of collecting results from other nodes when they become available. It can then combine these results in order to accomplish the task it set out to perform. In a sense, the originator “pulls” computing power from other members. There is a need for a predefined pool of computational servers, from which the originator can draw other members for a particular computation[4].

The model we advocate in this paper eliminates the burden of managing a pool of participators. At first the originator advertises his need for computational power along with the data set to compute on. Any node that sees this advertisement can choose to join the group as a worker for this particular task, after joining the group it becomes a member of the group and will receive the work load it should compute on, which is transferred as mobile object. The created group can be kept instead of being destroyed after finishing computation; it can perform another computation next time. One could say that the workers “push” their computing power to the originator or that the workers “shop around” on the net, looking for suitable tasks to compute on. We call this model “divide and be conquered”, since a large computation is broken up into small subtasks and the initiative for processing these subtasks is coming from the workers.

This approach scales to a large number of hosts involved in a computation, since the originator needs no a priori knowledge of the workers that will be involved. In particular, the originator does not (need to) know which node is going to act when and for how long as a worker for his task. The hassle of maintaining a dedicated pool of workers disappears.

If the number of workers becomes large, the number of requests to the originator increases accordingly. In order to eliminate this potential bottleneck, the originator can create a number of sub-groups. The result is a tree of originator s where only the leaves communicate with workers directly. After the workers' results are combined on a sub-originator, the result of this combination process is sent to the parent originator until the top of the tree is reached where the final result becomes available.

1.2.3 Parallel object-oriented language (POOL)

The language we uses can be considered an extension of C++ for distributed programming and has been implemented in C++. In our implementation, the MPI procedures are used to perform object distribution and message exchange on a network of heterogeneous computers. POOL's syntax and semantics is close to that of C++, a few new operations were added, and a limited set of C++ operations ("<<", ">>", "+=", "-=", "&", "|", etc.) were specialized to cope with the management of objects. Some notes about language syntax, semantics and implementation, as well as how distribution and communication operators have been implemented on the basic of MPI procedures, are described in the following sections.

1.3 Group-Object layer

This layer aims at providing methodology for construction and management of communicating object groups. Communication details implemented by underlying group protocol are thus hidden from users, application objects can communicate with each other transparently and the difficulty of designing object communication and reliability is relieved

1.3.1 Objects and groups

An object group is denoted as $G=\{O_1,O_2,\ldots,O_n\}$ where O_i is the individual objects which communicate by message passing. For each site, there is a Communication Manager (CM) executing on it, which is to maintain consistent membership for the groups and to transmit object messages reliably. Each group has a value version to represent the configuration status of this group, for an object group G with id Gid, its version records the times of the group reconfiguration. For a communication object O_i in group G_i, it can also be identified with (G_i,*seq_num*), where *sep_num* is the object's member number in group G_i.

1.3.2 Object management

Any application wishes to create a group of n objects invoking function CreateGroup which returns a group id Gid. In the function, CM takes the Gid and registers it as an entry to the Group Name Server (GNS), GNS stores the group id Gid, group's version and group's membership-list (ML). Particularly, group version (initially 0) records the times of the group that have been reconfigured since it is created; membership_list (ML) constitutes of object ids.

A GNS can be view as a table (Table 1).

Table 1 The items in group name server (GNS)

Gid	Version	Membership-List
……	……	$\{O_1,O_2,\ldots,O_n\}$

We can remove an object from a group G_1 and add it to the second group G_2 through the forms:

G_1–=O_1

G_2+=O_2

Suppose object O_i joins group G, the G's membership-list is changed by ML:=ML+$\{O_i\}$ and the group version is increased by version=version+1. On the other hand, if O_i leaves G, the corresponding operations become ML=ML–$\{O_i\}$ and version:=version+1.

We further define the union, the difference and the intersection of the two groups G_1 and G_2 through the forms:

$G_{union}=G_1+G_2$

$G_{diff}=G_1=G_2$

$G_{inter}=G_1\&G_2$

1.3.3 Message transmission

Objects communicate with each other through asynchronous messages. Each message can contain a set of elements composed of data and arrays belonging to a subset of C++ elementary data types. To be sent, the data of a message are encapsulated into an object called message (msg).

The data of a message can be managed through C++ input/output operators and the methods offered by the message object. Data and arrays of data are inserted into a message through the overloading of C++ input operator, i.e., “<<”. For example, the form:

msg << 20;

inserts the integer 20 in msg.

Data and arrays of data are got from a message and assigned to an output variable through the overloading of C++ output operator (i.e., “>>”). For example, if *var*1 is an integer variable, then the form

msg >> *var*1

gets the first integer from *msg* and assigns it to *var*1.

If an object O_i sends a message outmsg to another object O_j, it can use the form:

O_i >> *outmsg*;

And that object receives it through the form:

O_j << *inmsg*;

Suppose object *O* wants to send message m to group *G*. It transmits the message to the local CM with *m.Gid*= *G.Gid*. The CM checks *Gid* in the GNS to see if ML contains any remote objects. If there are remote objects on other sites, the CM forwards m to other sites. If *Gid* only contains some local objects, the CM simply passes m to the members without transmitting it through network.

If CM receives m from an local object *Oid*, it checks *m.Gid* entry in GNS, if m is the expected message by the group and it is delivered to the local objects in *G*, then CM accepts the message to forwards it to the corresponding object.

1.4 Mobile object layer

The mobile object layer allows programs to be able to pick up and move from one site to another in which the decisions on when and where to move are made autonomously. This layer provides the tools to build distributed, mobile data structures, which consist of a number of mobile objects held together with mobile pointers.

1.4.1 Directory

The directories store the locations of objects and help to determine the current location of the mobile object, this operation needs a lookup in the directory. Each site maintains a directory that helps to find the location of a mobile object, in which each directory entry contains the site number of the current best guess of the object’s location. We construct a directed graph for each node, where each node holds a list of mobile pointers to other nodes. For the sake of high efficiency, the directories are updated lazily, allowing some local directories to be out of date.

An object sends a message to a mobile object via the address indicated by its local directory, which may be incorrect due to the movement of objects. If the location turns out to be incorrect, then the layer forwards the message towards the real location, and sends an update message back to the site where the source object locates. After the message gets forwarded, if a site receives an update for an object, all further messages from the site to the object will go directly to the object’s new location. In this way, updates to a site’s local directory occur only when one of the site’s messages misses the correct location of the target object. When the user moves an object from some source site to some target site, only the source and the target sites are aware of the change, other sites' directories are updated lazily. This lazy-update may avoid the need to broadcast updates to all sites each time an object moves, as a result the cost of moving an object is reduced.

1.4.2 Object migration

To migrate an object in the mobile object layer, the code must uninstall the object from the original site, sends the object data along with the moving information that the layer uses to track the object's state to another site, and then installs the object on the new site.

Mobile object layer provides the mobile pointer to implement object migration. A mobile pointer consists of the number of the home node where the object was originally allocated and an index number that is unique on that site. A (home node, index number) pair forms a name for the mobile object that is unique over the whole system. The user calls CreateMobilePtr by passing in a pointer to the local object to apply a new mobile pointer.

To move an object, user first calls UninstallObject to update the site's directory entry to reflect the mobile object's next location. Then the object's data is moved to the new site using MPI_Send. Finally the object is installed on the new site by calling InstallObject.

FreeMobileObject, called with the object's mobile pointer and a user handler as parameters, is used to free a mobile object. The layer invokes the user handler on the site where the object resides, so that the handler can free the object data. Then the layer sets this site's directory entry for the mobile object to point back to mobile pointer's home node, which can later reuse the mobile pointer for a new object when the user calls NewMobilePtr.

1.5 Low-Level messaging level

The mobile object layer augments a low-level messaging layer such as MPI, PVM, DMCS[5], Active Messages[6] and LAPI[7]. Moreover, the upper layer still has complete and direct access to the underlying communication substrate. This is essential if the application is to obtain maximal performance.

We use the MPI to design and implement the platform, the basic communication functions are:

MPI_SEND(void*buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MPI_RECV(void*buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status*status)

To implement object group communication, the following collective communication functions are used:

MPI_BARRIER(MPI_Comm comm)

MPI_BCAST(void*buffer, int count, MPI_Datatype datatype,int root, MPI_Comm comm)

MPI_GATHER(void*sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, MPI_Datatype recvtype, int root, MPI_Comm comm)

The modes of point-to-point communication we use include, (1) standard mode--blocked send MPI_SEND and blocked MPI_RECV, unblocked send MPI_ISEND and unblocked MPI_IRECV; (2) buffered mode--blocked send MPI_BSEND and unblocked send MPI_IBECV; (3) synchronous mode--blocked synchronous send MPI_SEND and unblocked synchronous send MPI_ISSEND; (4) ready mode--blocked ready send MPI_RSEND and unblocked ready send MPI_IRSEND.

2 Functions

2.1 Load balancing

To implement load balancing, a number of heuristics have been used to determine which objects should be transferred. Typical goals include minimizing communication costs, minimizing the amount of data migrated, minimizing the number of neighboring processors, optimizing the shapes of the sub-domains, or combinations of these goals[8]. Many load-balancing algorithms use a version of the gain criteria form the algorithm by Kernighan and Lin (KL)[9] to select objects to transfer[10]. In these algorithms, for each of a processor's object, the gain of transferring the objects to another processor is computed. In some variants of the KL algorithm only objects on sub-domain boundaries are examined for transfer.

In our scheme, we use local improvement method to implement load balancing. Local methods are incremental, they move objects only within a small group of sites. One iteration of a local balance can reduce a single heavily loaded site's work load significantly, a small number of iterations may be all that is needed to reduce imbalance to an acceptable level. Moreover, local methods can also be executed synchronously, sites can initiate load balancing when they become idle, requesting work as they need it. Two load-balancing methods are used in our works:

(1) Subgroup method. When a site becomes overload, it can choose to replace one object with heavy work load by an object-group (Fig.2). The applications which use the object distribution need not to be modify, and load sharing can thus be increased step by step. The way of creating a subgroup is just similar to that of creating a new group, which has been discussed in the object-group layer.

(2) Work-Steal method. An overloaded site should forward some objects for other sites with less workload. Each CM maintains a counter of the amount of work-load that is currently waiting to be processed, and consults a threshold of work to determine when work should be requested from other sites. When the work load falls below the threshold, the CM requests an object to be forwarded to it from other sites. By associating a mobile pointer with each mobile object, messages sent to migrated objects will be forwarded by the mobile object layer to the object's new locations (Fig.3). One object A in site moves to site B, the migration is initiated by the Load Balancing Model and managed by the CM on respective site, the CM on site A forwards the object to the CM on site B through the mobile object layer.

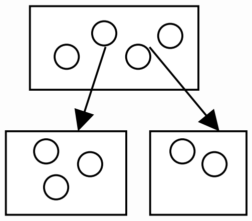


Fig.2 SubGroup method

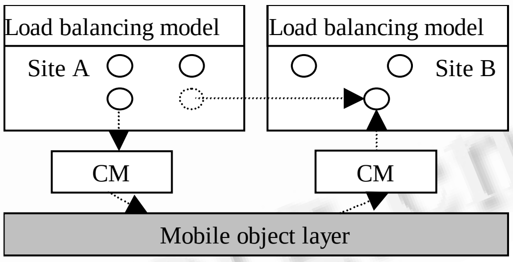


Fig.3 Object migration between sites

2.2 Fault tolerance

A troop of mobile objects is exposed to many resources of failures. Processors can crash, communications can fail, and processes may fail due to buggy user code. As more applications are built on top of mobile objects, support for the detection and recovery from such failures will becomes correspondingly more important.

In our paper, the fault-tolerant mechanism is implemented in the object-group layer, and this mechanism aims to tolerate:

· User object failure. The failed object will not respond to CM when CM commands the object to receive a message. To cope with this failure, it is the responsibility of the CM to delete the object from the group entry as well as its membership list. To maintain consistence of the membership in all the sites, a 2-phase protocol is performed. The objects invoked in the same process as the object will also be deleted from the corresponding groups.

·Originator failure. This failure will block the group management. To tolerate such fault, the object-group layer generates two active objects, Originator and Originator-twin. The Originator-twin keeps monitoring the activity of Originator by receiving the messages from Originator and acknowleging the Originator regularly. Originator-twin is thus a backup for Originator. Once the Originator stops, Originator-twin takes over the failed Originator position and handles all the tasks of group management and message passing. In the mean time, Originator-twin (now it becomes Originator) notifies the layer to generate another active Originator-twin objects. This failure-replacement process is hidden from users. If the Originator-twin fails, Originator does the same to inform the layer to create new Originator-twin.

· Site failure stop: In fact, group membership management will report the site failure to the object-group layer, then the layer eliminates the CM on the failed site out of the CM group and changes related group MLs to eliminate the objects from the site. We omitted the details here.

2.3 Performance guarantee

In the application layer, the “divide and be conquered” model we advocate eliminates the burden of managing a pool of participators. The approach scales to a large number of hosts involved in a computation, since the originator needs no a priori knowledge of the workers that will be involved. In particular, the originator does not (need to) know which site is going to act when and for how long as a worker for his task. The hassle of maintaining a dedicated pool of workers disappears.

In the mobile object layer, the directories are updated lazily, allowing some local directories to be out of date, updates to a site’s local directory occur only when one of the site’s messages misses the correct location of the target object. This avoids the need to broadcast updates to all sites each time an object moves and minimize the cost of moving an object.

3 Related Work

A usual requirement for parallel software systems is the possibility to distribute them on a heterogeneous network of computers. Software tools like PVM and MPI allow programmer to partition a program into pieces which may then execute in parallel, synchronizing and exchanging data on a heterogeneous network of computers. There are some frameworks supporting the composition of parallel components such as LSA[11], POOMA[12], PARDIS[13] and so on.

However, an object-oriented methodology seems to be the most appealing way of coping with the problem of heterogeneity because it allows both rapid prototyping and the implementation of complex system somehow, disregarding the actual architecture on which application are intended to be run.

Distributed OO systems have been extensively investigated. Examples include Amber[14], Network Objects[15], PRESTO[16]. PRESTO achieves inter-object communication and synchronization through shared memory. Network Objects provide support for distributed objects at the programming level and reliable distributed services. Amber aims at using objects to obtain performance improvements on a multiprocessor. None of the systems have ever addressed the issues of direct inter-object communication transparency and object group communication protocols for distributed fault-tolerant OO applications. Mentat[17] and CHOICES[18] aim to build distributed languages and system starting from C++. They offer constructs for parallel execution, parallel objects, message-passing and shared-memory synchronization methods, however, they do not introduce new methods for object coordination and composition that might simply the definition of software system.

In Chaos++[19], global objects are owned by a single processor and all other processors with data-dependencies to a global object possess shadow copies. Our system does not use shadow objects because of the complexity of the code required to maintain consistency between the original and the shadow objects. Instead, we rely on an efficient message-forwarding mechanism to locate and fetch data from mobile objects through mobile pointers. Emerald[20] and Amber[14] are comprehensive, object-oriented, high-level languages that support object mobility. However, these systems are designed to make mobile pointers look nearly identical to local pointers, implementing this transparency is very difficult.

A parallel runtime substrate (mobile object layer) is presented in Ref.[21], it supports data or object mobility and automatic message forwarding, a global name space for message passing and distributed directories is implemented to assist in the translation of logical to physical addresses. Our framework aims to provide an integrated environment for parallel computation, the mobile object layer doesn't use global name space to implement the object migration. Whereas user decide to migrate objects (the way in Ref. [21]), in our scheme the object migration is often initiated by the load balancing module.

4 Experiments and Analysis

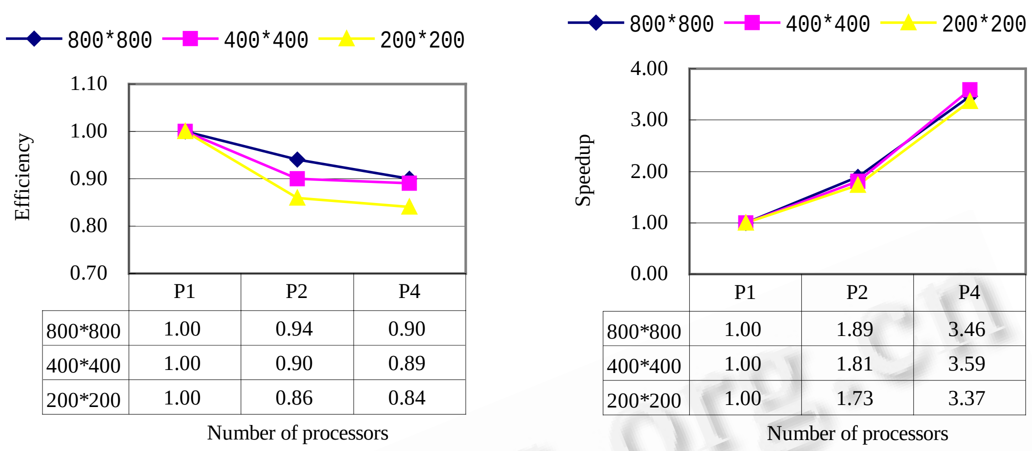
Before the computation process is launched, the whole computation task resides on the originator which plays a role of group leader and communication coordinator. The originator advertises his need for computational power via the network. Any node that sees this advertisement can choose to join the group as a worker for this particular task. After joining the group a node will receive the work load it should compute on, which is transferred as mobile object. We use Master/Slave mode to perform computation inside the group. Master/Salve mode is a common and wide-used parallel computation mode. It can be understood as: the client application is a task with large volume which is managed by an object (originator), and the originator-twin is created and copy the information and data on originator, then the members inside the group cooperate with each other to perform the computation. Our experiments use this computation mode to get good encapsulation and reusability.

In fact, after the user submits the computation task, all manage work including task distribution, task allocation, task scheduling, along with load balancing and fault tolerance, are managed by the system automatically. Group management model and object migration model are in charge of the trivial detail of task scheduling, load balancing and fault tolerance. The application layer is visible to the user, who can choose to use the functions provided by the object-group layer, the lower two layers (mobile object layer and low-level messaging layer) are hidden and transparent to the software developer.

4.1 Mandelbrot algorithm

Mandelbrot is a famous number set in fractal theory. Mandelbrot set is generated by nonlinear mapping $x \leftarrow x^2 + \mu$. We display M set using the color of point set in windows. Because the computation of each Zk depends on only single point, the computation of different points can be done in parallel. We test the speedup of Mandelbrot algorithm in a network of workstation (SGI indy and SUN sparc 20) connected by 10Mbps Ethernet. The maximum iterative number is set to 1024, the pictures of M set are displayed in windows with 800*800, 400*400, 200*200 size respectively.

The results are indicated in Fig.4. Results show that the efficiency while using two processors is higher than the efficiency while using four processors, the main reason is that while using two processors the task allocation is more balancing. The speedup will increase while the data volume increases, this can be ascribed to the reason that the overhead of sending large message over Ethernet is approximately equal to that of sending small message over Ethernet.



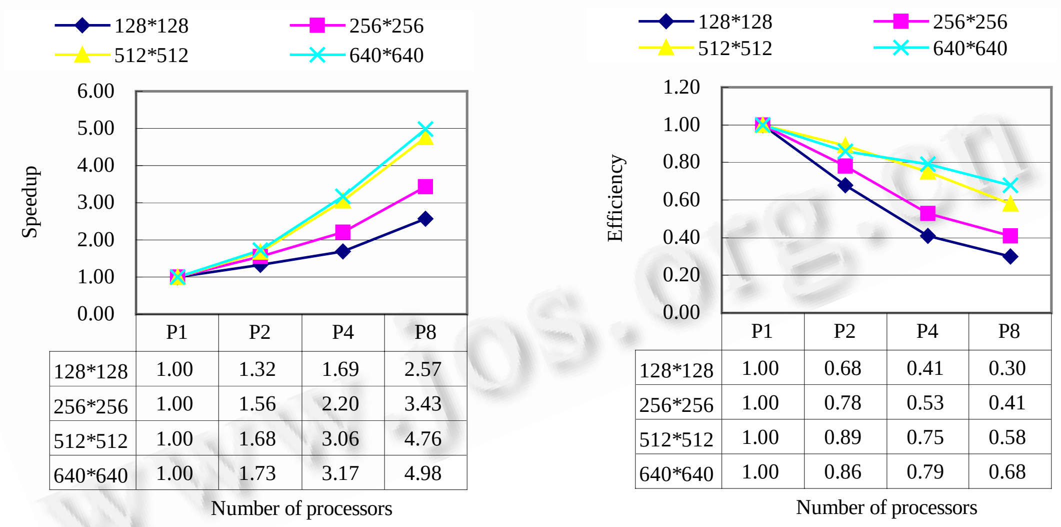
	P1	P2	P4
800*800	1.00	0.94	0.90
400*400	1.00	0.90	0.89
200*200	1.00	0.86	0.84

	P1	P2	P4
800*800	1.00	1.89	3.46
400*400	1.00	1.81	3.59
200*200	1.00	1.73	3.37

Fig.4 Speedup and efficiency of Mandelbrot algorithm

4.2 Matrix multiplication

Matrix multiplication needs heavy computation and large volume of communication, hence it is suitable for experiences on parallel computing. We use the following partition algorithm $C=[A^T_1,A^T_2][B_1,B_2,\dots,B_{block\_num}]$, whose communication cost is $(p/4+3)*n_2*comm\ cost$. Figure 5 shows the speedup and efficiency of 128*128 matrix multiplication on NOW.



	P1	P2	P4	P8
128*128	1.00	1.32	1.69	2.57
256*256	1.00	1.56	2.20	3.43
512*512	1.00	1.68	3.06	4.76
640*640	1.00	1.73	3.17	4.98

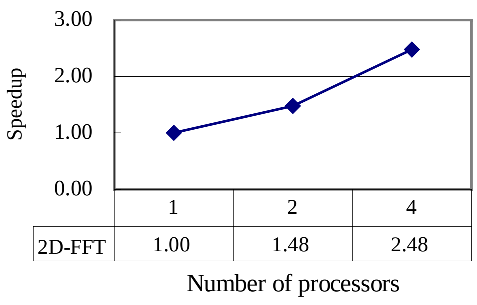
	P1	P2	P4	P8
128*128	1.00	0.68	0.41	0.30
256*256	1.00	0.78	0.53	0.41
512*512	1.00	0.89	0.75	0.58
640*640	1.00	0.86	0.79	0.68

Fig.5 Speedup and efficiency of matrix multiplication algorithm

The main reason causing the reduction of processing efficiency while the number of processing increases is that the communication overhead increases faster.

4.3 D-FFT algorithm

We construct a 2D-FFT algorithm to be tested in my system: the originator dispatched data to workers line by line; then the workers compute the FFT value of each line in parallel and sends back result to the originator; and then the originator transposes the result matrix and dispatches the transposed matrix to all workers line by line; and



	1	2	4
2D-FFT	1.00	1.48	2.48

Fig.6 Speedup of 2D-FFT alogrithm

then the workers compute FFT value and send back result to the originator again; finally the originator get the final result. Figure 6 shows the comparison of the speedup and efficiency of the 2D-FFT algorithm on a network connected by 4 SGI Indy workstations.

The efficiency and speedup gained from our scheme verifies the efficiency of our mechanisms. The extra overhead includes schedule algorithm overhead, communication overhead, object management overhead (esp. object group management overhead), dynamic load balancing management overhead and fault tolerance management overhead. The scheduling and communication overhead is the intrinsic cost of the system, while other types of overhead will decrease with the decrease of problem scale and number of processors. In fact, the intrinsic cost is the main cause of total extra overhead, whereas the increase extent of other overheads is not very obvious and hence good scalability of our scheme is guaranteed.

5 Conclusion

Parallel systems and applications have been developed over the last years, however, because of the lack of suitable supporting environment of software development, designing and implementing parallel software is very difficult and coding is often out of control. In many systems the problems of load balancing and fault tolerance are not taken into consideration. And the parallel software is hard to be reused. So we present a distributed object based framework for parallel computation, which can provide the management and control structures necessary to build and capture the complete lifecycle of parallel software system. We describe it in an attempt to have it serve as an efficient model for how to structure distributed object based parallel application. The services in the framework include naming, load balancing and fault tolerance. We envision the services will include many of the services one finds in environments like DCE and CORBA or other novel services.

References:

[1] Tan, S.T., Wang, T.N., Zhao, Y.F., *et al*. A constrained finite element method for modeling cloth deformation. Visual Computer, 1999,15(2):90~99.

[2] MPI forum. Message-Passing interface standard 1.0 and 2.0. http://www.mcs.anl.gov/mpi.index.html, 1997.

[3] Geist, G.A., Beguelin, A., Dongarra, J., *et al*. PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Networked Parallel Computing. Massachusetts: The MIT Press, 1994.

[4] Dag, J., Keith, M., Kare, L. An approach towards an agent computing environment. In: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Workshop on Middleware. Austin, Texas: IEEE Computer Society Press, 1999. 78~83.

[5] Laurent, B. A Java framework for massively distributed sumbolic computing. Mathematics and Computers in Simulation, 1999,49(3):151~160.

[6] Chrisochoides, N., Pingali, K., Kodukula, I. Data movement and control substrate for parallel scientific computing. Lecture Notes in Computer Science (LNCS), 1199. Berlin: Springer-Verlag, 1997. 256~256.

[7] Chang, C., Czajkowski, G., Hawblitzel, C., *et al*. Low-Latency communication on the IBM RISC system/6000 SP. In: ACM/IEEE Supercomputing'96. Pittsburgh, PA, 1996. http://wwwipd.ira.uka.de/parastation.

[8] DiNicola, P., Gildea, K., Govindaraju, R., *et al*. LAPI architecture definition: low level API draft. IBM Confidential Report, 1996.

[9] Bruce, H., Karen, D. Dynamic load balancing in computational mechanics. Computer Methods in Applied Mechanics and Engineering, 2000,184:484~500.

[10] Kernighan, B., Lin, S. An efficient hueristic procedure for partitioning graphs. Bell System Technical J.29, 1970. 291~307.

[11] Xu, C., Lau, F., Diekmann, R. Decentralized remapping of data parallel applications in distributed memory multiprocessors. Technical Report tr-rsfb-96-021, Department of Computer Science, University of Paderborn, Paderborn, Germany, 1996.

[12] Gannon, D., Bramley, R., *et al*. Developing component architectures for distributed scientific problem solving. IEEE Computational Science & Engineering, 1998,5(2):50~63.

[13] Reynders, J.V., Hinker, P.J. *et al*. POOMA: a framework for scientific simulation on parallel architectures. 2002. http://www.acl.lang.gov/pooma/documentation.

[14] Keahet, K., Gannon, D. PARDIS: a parallel approach to CORBA. Technical Report, 1997. ftp://ftp.cs.indiana.edu/pub/techreports/TR475.ps.Z.

[15] Chase, J.S., Amador, F.G., Lazowska, E.D., *et al*. The amber system: parallel programming on a network of multiprocessors. In: Proceedings of the 12th ACM Symposium on Operating Systems Principles. ACM Press, 1989. 147~158.

[16] Birrell, A., Nelson, G., Owicki, S., *et al*. Network objects. Software-Practice and Experience, 1995,25(S4):87~130.

[17] Bershad, B.N., Lazowska, E.D., Levy, H.M. PRESTO: A system for object-oriented parallel programming. Software-Practice and Experience, 1988,18(8):713~732.

[18] Grimshaw, A.S. Easy-to-Use object oriented parallel programming with Mentat. Technical Report CS-92-32, Department of Computer Science, University of Virginia, Charlottesville, 1992.

[19] Campbell, R., Islam, N. Research Directions in Concurrent Object-Oriented Programming. Cambridge, MA: The MIT Press, 1993. 393~451.

[20] Chang, C., Sussman, A., Saltz, J., *et al*. Chao++, Parallel Programming Using C++. Cambridge, MA: MIT Press, 1998.

[21] Jul, E,. Levy, H., Hutchison, N., *et al*. Fine-Grained mobility in the emerald system. TOCS, 1988,6(1):109~133.

[22] Chrisochoides, N., Barker, K., Nave, D. *et al*. Mobile object layer: a runtime substrate for parallel adaptive and irregular computations. Advances in Engineering Software, 2000,31:621~637.

基于分布式对象的并行计算框架

李国东, 张德富

(南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

摘要: 在为工作站机群构造并行软件的过程中,计算特征和组成特征非常重要.但是,由于缺乏有效的支撑环境,当今的分布式并行计算软件系统效率低下,这在计算特征方面尤为明显.提出一个基于分布式对象的并行计算框架,目的在于保证高效的并行计算开发,提供封装和复用并行程序的机制,并保证系统的动态平衡和容错性.框架是4层模型,包括对象组层和移动对象层.实验结果证明了方案的有效性.

关键词: 分布式对象;移动对象;并行计算;工作站机群;框架
中图法分类号: TP311 文献标识码: A