OMP: A One-sided Message Passing Programming Model for P2HP*

Hai Jin, Fei Luo, Qin Zhang, Xiaofei Liao, and Hao Zhang

Cluster and Grid Computing Laboratory,
Huazhong University of Science and Technology, Wuhan, 430074, China
{hjin, luofeiren, qzhang, xfliao, haozhang}@hust.edu.cn

Abstract. P2HP is a *Peer-to-Peer* (P2P)-based *high performance distributed computing* (HPDC) platform. Since the programming model is critical to HPDC systems, this paper focuses on the design and implementation of OMP, a *One-sided Message Passing* programming model for P2HP, which presents a general way to design the programming model for HPDC systems and shows that P2HP is a practical and efficient HPDC platform. OMP provides a one-sided message-passing communication library. And based on it, a software development kit with rich APIs for user applications is designed and implemented. The performance of OMP on P2HP is evaluated through an application benchmark of sequence alignment.

1 Introduction

P2P-based HPDC systems have powerful computing and storage capability to resolve large-scale scientific computing problems. For their unprecedented scale, many basic issues have to be reinvestigated, such as programming models, performance models, and class of applications or algorithms suitable to this architecture [1]. The programming model attacks the problem at the level of the model that acts as an interface between software and hardware issues, and it is one of the main approaches to drive the development of HPDC systems. P2HP is one of such systems, which is exploited by our research group. This paper focuses on the design and implementation of the OMP programming model for P2HP.

With well-defined *Application Programming Interfaces* (API), a programming model is for users to write applications, which are used in various parallel platforms [2]. It depends on the communication environment which is an abstract machine providing certain operations to the programming level above and requires implementations for each of these operations on all of the architectures below.

Shared-memory and message passing are the two common communication models in parallel programming [3]. In the shared-memory programming model,

* This paper is supported by National Science Foundation of China under grant 60433040, China CNGI project under grant CNGI-04-12-2A and CNGI-04-12-1D.

threads are employed to communicate by storing to and loading a shared location in the address space. It simplifies programming by hiding communication details such as control over communication and computation costs. For the limitation of the scalability, this model is not commonly used in wide area environments.

For the message passing paradigm, processes communicate with each other by sending and receiving messages mutually. It simplifies writing applications involving a large number of nodes by providing a simple flat name space with which nodes can communicate with each other [4]. It is hard to be utilized in dynamic environments for the nature of its simple name space.

One-sided communication combines some of the strengths of shared-memory and message passing together, while currently it is still utilized in cluster computing environments. One-sided communications in MPI defines communication routines that can be completed by a single process. These include shared-memory operations (put/get) and remote accumulate operations [5].

The most widely used programming models for distributed computation will be socket, RPC [6], and Java RMI (remote method invocations) [7]. RPC and RMI communication raise the level of communication abstraction, and they do not adaptively coordinate to the customized environment of P2HP. The programming model of P2HP, OMP, is just based on the socket communication primitives.

OMP consists of a communication library and a software development kit. In the communication library, OMP provides simple one-sided message passing communication primitives. Based on the communication library, the SDK presents a software development methodology for user applications, which releases the burden of the programmer. An application instance in bioinformatics, sequence alignment, is programmed in OMP and run in P2HP, which shows that OMP is easy to program, architecture-independent, and easy to understand.

The contribution of this paper is as follows. First, a general way to design and implementation of a programming model for P2P-based HPDC platforms is presented. Currently, there are no universal programming models for such systems, while they are platform-specific. Associated with the programming environments, the communication library and APIs of OMP are designed and implemented. Then, an Internet-wide one-sided message passing protocol is introduced. Unlike the regular send/receive communication with matching operations by sender and receiver in traditional message passing programming models, a task accesses remote data without a matching communication call. Finally, approaches to parallelize applications whose parallelism is in the task-level are shown in this paper. Divided tasks of applications are programmed with OMP as work units which are performed in the distributed environment of P2HP.

The rest of the paper is organized as follows. The programming environment is shown in Section 2. Section 3 and Section 4 detail the design and implementation of OMP, respectively. In Section 5, a benchmark is programmed with this model to evaluate the performance. Related works are presented in Section 6 and conclusions are drawn in Section 7.

2 The Programming Environment of P2HP

Based on P2P networks, P2HP is a testbed for high performance distributed computing, which supports applications without data-dependence between tasks. As shown in Figure 1, P2HP is composed of *Monitor Group* (MG), *Dispatcher Group* (DG), *Worker Group* (WG), a Datapool and User. Each MG, DG or WG comprises multi-nodes which are voluntarily self-organized in the Internet.

MG is responsible for monitoring the whole system, such as the submission of new jobs from users, the disposal for the joining and leaving of dispatchers and workers in the system. DG schedules its workers to execute work units and monitors their working states. Voluntary nodes in the Internet make up of WGs, and each WG is managed by some DGs, in which workers execute subtasks of applications. The User is an interface for users to submit applications to the system, and all data related to the applications' subtasks is stored in the Datapool.

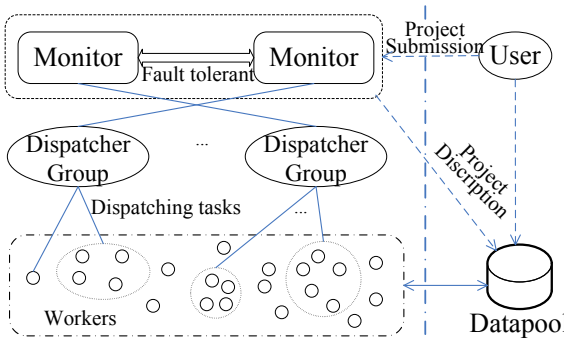

Fig. 1. Architecture of P2HP

The parallelism of applications in P2HP is in the task level. Applications in P2HP are divided into many small work units, and all the divided units are distinguished as the main task and the subtasks. Each task finishes some computing job for the application. One of them participates in the control of the tasks' execution flow, and it is defined as the main task, while the others are defined as subtasks. There is only one main task and many subtasks for an application. The job of the main task includes the creation of new subtasks, the control of subtasks' execution, and the collection of the final result of the application.

To execute an application in its distributed environment, P2HP works as follows. After programming the application according to the programming model, OMP, the resulting data is transferred into the Datapool, and a project file (*JobDesc*) is generated. The file is submitted to a monitor in MG and batches of subtasks are then redirected to a dispatcher by the monitor when the main task begins. The dispatcher which accepts subtasks allocates them to the attached

workers. The workers get their related data from the Datapool and perform them, and results are transferred to the Datapool, which are collected and further disposed by the main task to present the application's final result.

3 Design of OMP

With regard to its programming environment, the programming model of P2HP, OMP, has been deliberately kept simple with minimal conceptual overhead. As shown in Figure 2, OMP consists of a *communication library* (ComLib) and a *software development kit* (SDK) for users. The ComLib is maintained by the runtime of P2HP, and is used for the communication between tasks and the Datapool. The SDK is the interface between the project and the runtime, and is responsible for providing programming interfaces for the main task (*MainAPI*) and subtasks (*SubAPI*) of the application.

Subtasks are independent of each other, so there is no communication among them. All the data related to subtasks are stored in the Datapool, and tasks exchange data with the Datapool during their execution. Therefore, the ComLib has been designed for the subtasks' communication with the Datapool. It consists of a *Transformation Protocol* to transfer data, and a *Storage Management* to manage the local storage of the transferred data. *Transformation Protocol* and *Storage Management* are combined with the storage subsystem and network I/O subsystem of the platform.

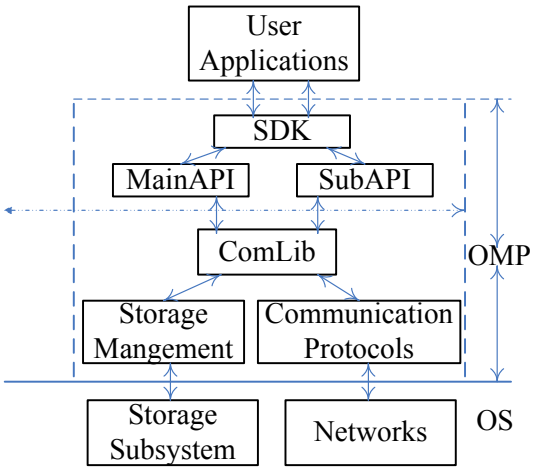

Fig. 2. Architecture of OMP

In the SDK, the APIs in SubAPI collaborate with the runtime of P2HP to perform subtasks, and the APIs in MainAPI control the workflow of the application in P2HP. These APIs of the SDK pack the user requests into a

message, which implements a one-sided message passing model. When the clients with a task need correlative data, they only locally trigger off a message like sending or receiving. Because tasks are maintained by the system runtime of P2HP, the corresponding system call is resolved by the Datapool. The requested data is transferred by the ComLib between the Datapool and clients.

4 Implementation of OMP

We have implemented the OMP of P2HP in Java which is critical to the platform-independence characteristic of P2HP. In the rest of this paper, we will describe the basic mechanisms used to implement the ComLib and SDK of OMP.

4.1 Implementation of ComLib

The communication between the worker and the Datapool passes through three processes, such as *Construct Channel*, *Request Transaction*, and *Close Channel*. First, the communication procedure begins with Construct Channel, which constructs a pipeline between the client (a worker) and the server (Datapool). Then the process of Request Transaction is triggered off to transfer the requested data files between them. Finally, after they finish disposing all the requests, the process of Close Chunnel is turned on to close the data channel. All the requested data is transferred by the Transformation Protocol and locally managed by Storage Management.

Transformation Protocol. A transformation protocol [8] to transfer requested data files has been implemented in OMP. As shown in Figure 3, the transformation process for data between the client and the server is expressed by the transition of a finite state machine of the defined states, including three transformation states and two synchronization states, which are depicted in Table 1.

Both the client and the server start the transformation from the IDLE state. After sending the request message for data, the client enters the state of `SYN_RECV`. Receiving the message, the server sends back the response (SEND/SYN), and enters the state of `SYN_SENT`. If an error occurs, both sides become IDLE, or they initialize the pipeline and enter the state of `BEGIN_TRANS`. One of them sends the data file, and the other receives it.

Having finished transferring a file and been synchronized with the message of SEND/SYN, the server enters the state of `SYN_SENT`, and the client enters the `SYN_RECV`. If an error occurs, the transfer fails and both of the client and the server become IDLE and prepare the next transaction for the next request message. If the transfer is successful, both of them are `FINISH_ONCE`. Then they check whether it is the end of the transaction.

If there are other data files to be transferred, both of them enter the state of `BEGIN_TRANS` and begin to transfer the next data file, or the server enters the

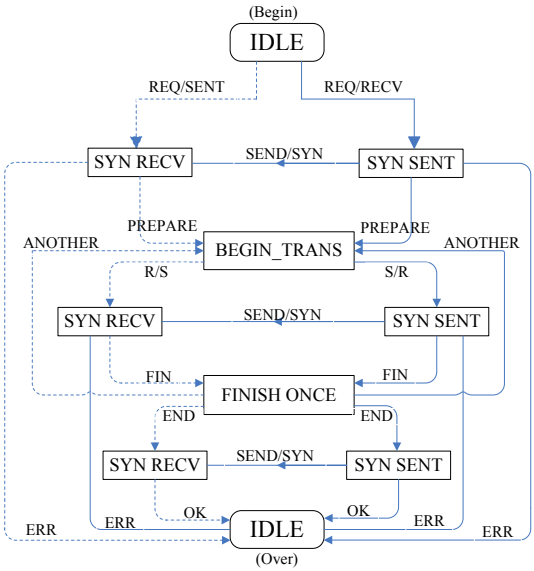

Fig. 3. States Transition during Data's Transformation. The real line represents the server's state transition, while the dashed represents clients' state transition.

`SYN_SENT`, and the client enters the `SYN_RECV`. Confirmed through the synchronization with the message of SEND/SYN, both of them again become IDLE, and prepare for the transaction of the next request message. The procedure is iterated until all data files have been transferred, or an error occurs during the transaction.

Table 1. States' Description during Data's Transformation

Classification	State	Description
Transformation	IDLE	There is no data being transferred.
	`BEGIN_TRANS`	Begin to transfer data and prepare for it.
	`FINISH_ONCE`	One data file has been transferred.
Synchronization	`SYN_RECV`	A synchronization message has been received.
	`SYN_SENT`	A synchronization message has been sent.

Storage Management. In the workplace of the Datapool, the data is accessed in the form of files, and all files of a subtask are assembled in a directory.

To transfer a subtask's data, its files are formatted as an I/O stream (Figure 4) which consists of multiple file streams (AFileStream) and an end token (EndTrans) in the end (Figure 4-a). Each AFileStream comprises a beginning token

(BeginFile), a prefix of the stream (Prefix), and the file (File), shown in Figure 4-b. The Prefix is the relative directory (Directory) of the file, which ends with a token (EndDir). The byte stream is the content (Content) of the file, and is surrounded with the size of the file (FileSize) and an end token (EndFile), as shown in Figure 4-c.

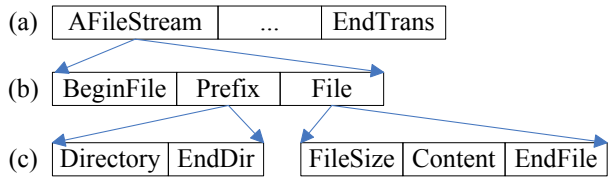

Fig. 4. I/O Stream Format

The directory management of the Datapool is combined with a light-weight database, the Berkeley DB, which is an embedded database that records several fields related to the tasks' data, such as the user ID, the task ID, the registering time. The database is also to record and dynamically update the information of tasks' states.

Services are provided by the database for the management of the Datapool, and they give universal interfaces to the Datapool's requests from clients. With it turned on, the process of the usage for these services is shown as follows:

a) Apply a service for the task with a service name and other parameters, such as the user ID, the task ID;

b) Obtain the information of a task, including its states and the location of its data, which are used by the data transmission sub-module;

c) Return the result of the service to the database to indicate whether it is correct or not, and the database updates the task's states for another request.

4.2 Implementation of the SDK

Based on the communication library, *ComLib*, the MainAPI for the main task and the SubAPI for subtasks are implemented in this section.

MainAPI. The main task controls the execution of the application, including its starting, running and ending. Setting up the job description file (*JobDesc*) provided by the user after the project is submitted, the main task is started to initiate a session with the monitor. MainAPI includes functions such as *newSubTask*, *getTaskStatus*, *updateTaskData*, and *getResult*, which are depicted as follows.

void newSubTask (int n, Job jobId, Linklist subtasklist): The processing procedure of the project is controlled by the main task which launches the

running of subtasks via calling this interface. One or some (n) new subtasks are applied from the monitor by the main task. The *Job* is the abbreviative form of JobDesc, while the *subtasklist* stores the parameters of the applied subtasks. The performance of the platform will be maximized only when it is not overloaded. The programmer tunes the parallel processing speed according to the system environment by applying subtasks in batches through the interface. The monitor accepts the requisition and redirects it to one of its registered dispatchers with the least load. Then the dispatcher schedules some usable workers and dispatches the subtasks to them.

int getTaskStatus (SubTask task, String jobId): Inquire the status of a subtask of the project with the appointed ID of *jobId*. The *SubTask* gives a description of the special one. The working state of the project is monitored by the Monitor. During the parallel processing of subtasks, the main task checks the status of all the subtasks one by one through this interface. Only when all the applied subtasks have been finished, the project can be further processed by the main task. Programmers can take fault-tolerant mechanism in the application level, or the main task will be aborted if an error happens.

String getResult (SubTask task, Job jobId, String filepath): The filename of the result of the appointed subtask (*task*) in the project (*jobId*) is returned by calling the interface, where the *filepath* is the path to store it. When all the subtasks have been finished, the project will be further disposed by the main task. It collects the results of all the subtasks one by one via this interface, and then checks the termination condition of the project. If the project is finished, it gets the final result of the project, or it generates more subtasks by the *updateTaskData* interface described below and continues the parallel processing.

int updateTaskData (String filePath, Job jobId, int nums): Update the data of the subtasks for a project in the Datapool. The programs of subtasks are the same, while the parameter data are different. They are all accessed in files. If the data change, new subtasks are generated. When the project is an undetermined problem, a program with a data file can be a number of subtasks. After the old subtasks have be finished, new subtasks have to been created to fit the terminating condition of the project. The main task dynamically creates new subtasks for undetermined projects via this interface, where the *filePath* is the path of the new data file, the *jobId* is the appointed project being solved, the *nums* is the number of the new generated subtasks with this data. An error happens if a nonzero value returns, or the parallel processing is continued.

SubAPI. Subtasks are programmed by defining the class of *usersubtask* which extends the *TaskRun* class in SubAPI. It is the juncture of the runtime and the user subtask. Collaborating with the runtime of P2HP, the worker performs dispatched subtasks. First, it gets the program file and data file from the Datapool through the ComLib and puts them in the workplace. Then it starts the subtask by scheduling the *TaskRun* class, where the running environment of the

subtask is set, such as the program file and data file path. In the *run* function of *TaskRun*, the running environment is called and the subprogram is started with the set environment. Having been finished, the subtask sends its result to the Datapool, and the running state will be reported to the Monitor by the runtime.

To set the running environment for the subprogram and perform its execution, the interfaces in SubAPI are provided for the class of usersubtask, which are depicted in the following.

String getSubTaskId (): The ID of the dispatched subtask is presented by calling this interface, which can be used to regulate the control to the subtask.

String getDataFile (), String getProgramPath (): Get the parameter file and path for the subprogram, respectively, which are used to set the running environment.

String getResultPath (): Get the path to store the result file of the subtask, where the subtask sends its result after finished.

void setTaskState (int state): The state of the subtask is set by calling the API after its running. The execution can be successful or failed, which is used for the system runtime to monitor the running state of subtasks.

boolean sendResult (String originalpath, String resultname): Once the subtask has been finished, its result will be sent to the Datapool by the subtask via this interface, where the *orignalpath* and *resultname* is the path and filename of the result, respectively. The returned value indicates whether the operation is successful or not. Programmers can add fault-tolerant mechanism in the subtask according to the returned boolean value.

With the OMP programming model of P2HP, a project is programmed as a main task and many subtasks with corresponding APIs described above. Combining with the runtime of P2HP, the classification of the main tasks and subtasks decreases the programming difficulty.

5 Benchmark

We have implemented the proposed design of OMP in P2HP, and applications can be run in it with the programming model. In this section, we will describe a benchmark application in bioinformatics to evaluate the performance. The experiment is conducted in a network with about 50 normal PCs, which are connected by 3 100M Ethernet switches. It is supposed that each of them has the interest to be a worker of P2HP, and three of them are configured as the Monitor, Dispatcher, and Datapool.

Bioinformatics uses computation to advance the scientific understanding of living systems, in which the most pressing tasks involve the analysis of sequence information. *Computational Biology* (CB) is the name given to this process, which applies a problem of the Sequence Alignment in P2HP with its programming mode. Currently the program of Clustal W [8] is popularly used, and we parallelize the original Clustal W program with OMP. The evaluation is based

on the performance of the parallelized program to that of the original Clustal W program.

5.1 Evaluation

Fifty small sequences are selected from NCBI (*National Center for Biotechnology Information*), which is one of sequence databases in the world. They are aligned by the serial Clustal W in a PC. The pairwise (PW) calculation represents the first stage of the algorithm, followed by the construction of a guide three (GT), and then the progressive alignments are continued according to the GT. Through the pairwise calculation in 3 PCs whose *Linpack* performance values are 228889Kflops, 343333Kflops and 429167Kflops, the phylogenetic tree of these sequences attained; averagely it costs about 399.32 seconds.

Then the pairwise calculation of Clustal W is parallelized with the SDK of OMP and run in P2HP. Programmed with the APIs of MainAPI and SubAPI presented above, the main task and 130 subtasks are executed in different PCs. First the main task reads the local sequence file and does some pretreatment. Then it starts a session with the monitor (BeginSession), and applies new subtasks (newSubTask) in batches which are redirected to a load-least dispatcher. The dispatcher schedules workers to execute these subtasks.

Subtasks are started by calling the programmed class (usersubtask) during the runtime of the workers in P2HP. During their execution for these subtasks, the workers get the data relevant to the subtasks from the Datapool, and send their running states and results (sendResult) to it. The main task monitors the subtasks' executions (GetTaskStatus), collects (GetTaskResult) and disposes of all the results of the subtasks. Finally, the phylogenetic tree is formed the same as the one from the serial program above.

The parallelized Clustal W is conducted while workers are idle in the night. With the rise of the number of workers, the time for the execution of all subtasks decreases, as shown in Figure 5, where the average time for the serial program is also presented to contrast with the parallel one. But if the number of workers is too small, the time for the parallel is more than that for the serial, which means that the parallel income of a small quantity of workers can not afford the cost of distributed communication.

The benchmark demonstrates that P2HP is an efficient and practical HPDC platform, and with the help of OMP, the performance of P2HP increases with the rise of the workers' number. To mine the most parallelism, the number of usable workers should be more and match that of subtasks.

6 Related Work

There have been many programming models for globally distributed computing environments. OmniRPC [10][11] is a RPC-based programming environment for cluster and grid computing. OmniRPC allocates calls dynamically on appropriate remote computers. ARMI [7] is a communication library which is an

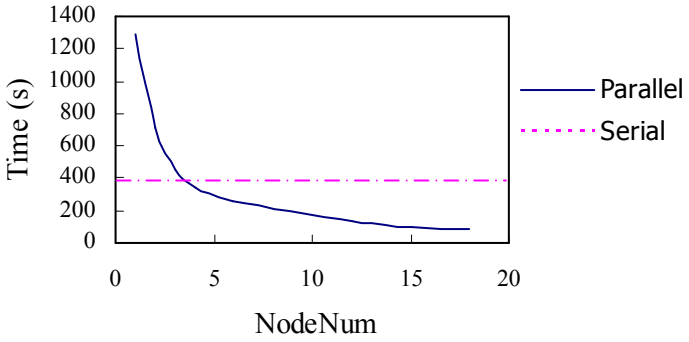

Fig. 5. Performance with the increase of the workers' number

implementation of the RMI protocol. It provides a framework for expressing fine-grain parallelism and mapping it to a particular machine using shared-memory and message passing library calls.

Global computing has been popular for the success and huge aggregate computing power of such systems as SETI@home [12], Folding@home [13]. They focus on solving one scientific or commercial problem, which makes them not a universal platform without a programming model. For example, SETI@home is a just special system for massively distributed computing to search extraterrestrial intelligence.

Provided with programming models, XtremWeb [1] and BOINC [14] are global distributed computing systems using volunteered computer resources. The programming model of XtremWeb utilizes the concept of RPC and has been implemented by P2P-RPC [6], a *remote procedure call* (RPC) API based on two interfaces of low-level and XWrpc API. Supporting applications that have large computation requirements, storage requirements, or both, and low data/compute ratio, BOINC provides APIs which are a set of C++ functions. A number of @home public-resource computing projects are now using BOINC, such as SETI@home, Predictor@home [15].

However, XtremWeb and BOINC do not tailor their APIs according to different requirement of the main task and subtask. The main task is the hinge of the processing flow, while the subtasks are executed redundantly over the global resources. Correspondingly their scheduling and programming APIs are different. They leave the burden of differentiating them to the programmers by combining the two classes of API together.

Customizing simple programming APIs for the main task and subtask, the programming model of P2HP is based on message passing mechanism. Currently, the most novel message passing libraries in distributed computing systems are MPI [16][17] and PVM [18], which are mainly used in cluster computing environments. A light-weight Java message-passing library with a deliberately simple programming interface is presented in [19], which allows developing distributed algorithms in a message passing model.

7 Conclusion

P2HP is a P2P-based testbed for high performance distributed computing, which is exploited to support a large range of applications. The parallelism of applications in P2HP is in the task level, and P2HP provides users with a message-passing programming environment. Based on the basic communication mechanism in P2HP, which is an Internet-wide one-sided message-passing between entries that are linked by channels, the programming model provides users with APIs for the main task and other subtasks, respectively. The application of sequence alignment in bioinformatics is programmed in OMP and run in P2HP, which shows that OMP is an easy, architecture-independent, and performance-guaranteed programming model.

To enhance the performance of P2HP, Datapool is going to be distributed into the system. ComLib of OMP will be self-adaptive to the system environment through the runtime, while the SDK will keep stable for users. Furthermore, to support more applications with data-dependence, ComLib of P2HP will support the communication between workers, and correspondingly, the SDK will be enriched with more one-sided message passing interfaces.

References

1. Fedak, G., Germain, G., Neri, V., Cappello, F.: XtremWeb: A Generic Global Computing System. In: Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, Brisbane, Australia, 2001. 582-587
2. Sievert, O., Casanova, H.: Policies for Swapping MPI Processes. In: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), Washington, DC, USA, 2003
3. Skillicorn, D. B., Talia, D.: Models and Languages for Parallel Computation. ACM Computing Surveys (CSUR), Vol. 30, No. 2 (1998) 123-169
4. Taura, K., Kaneda, K.: Phoenix: a Parallel Programming Model for Accommodating Dynamically Joining / Leaving Resources. In: Proceedings of the ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03), San Diego, CA, 2003. 216-229
5. Message Passing Interface Forum: MPI-2.0: Extensions to the Message-Passing Interface. Technical report (1997)
6. Djilali, S.: P2P-RPC: Programming Scientific Applications on Peer-to-Peer Systems with Remote Procedure Call. In: Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), Tokyo, Japan, 2003. 406-413
7. Saunders, S., Rauchwerger, L.: ARMI: An Adaptive, Platform Independent Communication Library. In: Proceedings of ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03), San Diego, CA, 2003. 230-241
8. Jin, H., Luo, F., Zhang, Q., Zhang, H.: An Efficient Data Transfer Protocol for P2P-Based High Performance Computing. Journal of Computer Research and Development, Vol. 43 (2006) (In Chinese)

9. Blanchette, M., Kwong, S., Tompa, M.: An Empirical Comparison of Tools for Phylogenetic Footprinting. In: Proceedings of the Third IEEE Symposium on Bioinformatics and Bioengineering, Bethesda, MD, 2003. 69-78
10. Sato, M., Boku, T., Takahashi, D.: OmniRPC: A Grid RPC System for Parallel Programming in Cluster and Grid Environment. In: Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), Tokyo, Japan, 2003. 206-213
11. Boku, T., Onuma, K., Sato, M., Nakajima, Y., Takahashi, D.: Grid Environment for Computational Astrophysics Driven by GRAPE-6 with HMCS-G and OmniRPC. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), Denver, CO, 2005. 176a
12. Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: An Experiment in Public-Resource Computing. Communications of the ACM, Vol. 45, No. 11 (2002) 56-61
13. Pande, V. S., Baker, I., Chapman, J., Elmer, S. P., Khaliq, S., Larson, S. M., Rhee, Y. M., Shirts, M. R., Snow, C. D., Sorin, E. J., Zagrovic, B.: Atomistic Protein Folding Simulations on the Submillisecond Time Scale Using Worldwide Distributed Computing. Biopolymers, Vol. 68 (2003) 91-109
14. Anderson, D. P.: BOINC: A System for Public-Resource Computing and Storage. In: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, Pittsburgh, USA, 2004. 4-10
15. Taufer, M., An, C., Kerstens, A., Brooks, C. L.: Predictor@Home: A "Protein Structure Prediction Supercomputer" Based on Public-Resource Computing. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), Denver, CO, 2005
16. Lusk, E.: MPI-2: Standards beyond the Message-Passing Model. In: Proceedings of 3rd Working Conference on Massively Parallel Programming Models, London, 1997. 43-49
17. Liu, J. X., Jiang, W., Wychoff, P., Panda, D. K., Ashton, D., Buntinas, D., Gropp, W., Toonen, B.: Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe, New Mexico, USA, 2004. 16
18. Gropp, W., Lusk, E.: Goals Guiding Design: PVM and MPI. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02) Chicago, IL, USA, 2002. 257-265
19. Schreiner, W.: A Java Toolkit for Teaching Distributed Algorithms. In: Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education, Aarhus, Denmark, 2002. 111-115