ELSEVIER

Information Processing Letters 70 (1999) 197–204

Information Processing Letters

www.elsevier.com/locate/ipl

Two design patterns for data-parallel computation based on master-slave model

Kuo-Chan Huang [a,1], Feng-Jian Wang [b,*], Jyun-Hwei Tsai [a,2]

[a] *National Center for High-Performance Computing, Taiwan, ROC*

[b] *Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, ROC*

Received 18 March 1998; received in revised form 5 January 1999
Communicated by F.Y.L. Chin

Abstract

This paper presents two design patterns useful for parallel computations of master-slave model. These patterns are concerned with task management and parallel and distributed data structures. They can be used to help addressing the issues of data partition and mapping, dynamic task allocation and management in parallel programming with the benefit of less programming efforts and better program structures. The patterns are described in object-oriented notation, accompanied with illustrative examples in C++. We also provide our experience in applying these patterns to two scientific simulation programs simulating Ising model and plasma physics respectively. Since master-slave model is a widely used parallel programming paradigm, the design patterns presented in this paper have large potential application in parallel computations. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Software design and implementation; Parallel processing; Design pattern

1. Introduction

Parallel programming is a necessary and complicated task to make use of computation power of parallel computers. A classical, but largely unrealized, goal in the parallel programming community is that users write pure serial programs, and the compiler synthesizes automatically the good parallel programs based on analysis of the serial programs. Although very interesting results were obtained in this direction, they all are far from a practical use yet. On the other hand several existing parallel languages, such as FORTRAN D, CM FORTRAN, C*, High Performance FORTRAN (HPF), etc. [10], provide functional abstractions for specifying partitioning and distribution strategies for static and regular data structures such as array. However, functional abstractions are insufficient for expressing partitioning and distribution strategies for irregular or dynamic data structures [13]. Currently, for these irregular and dynamic applications basic parallel programming tools are message-passing libraries, among which PVM [6] and MPI [7] are the most popular.

Message-passing based parallel programming is a kind of programming at assembler level. Although it is possible to write highly efficient parallel programs in message-passing languages, most users find such programming to be incredibly tedious [11]. A more serious problem is that current parallel programming

* Corresponding author. Email: fjwang@csie.nctu.edu.tw.
[1] Email: c00kch00@nchc.gov.tw.
[2] Email: c00jht00@nchc.gov.tw.

0020-0190/99/$ – see front matter © 1999 Elsevier Science B.V. All rights reserved.
PII: S0020-0190(99)00057-5

practice intermixes the specification of parallel control structure, such as data partition, mapping, and task management, with the code specifying the real computation stuff. The intermix complicates the software structure and thus reduces the reusability and maintainability of the code constructed.

Some researchers have recognized this issue and begun to study programming language constructs to resolve it [11]. However, it will take a long time before these constructs appear in daily programming languages such as C++ or FORTRAN. Before the appearance, programmers of parallel programs need other kinds of tools handy to approach this issue. This paper extends our previous work [4] and presents two design patterns to address the issue. The proposed patterns in this paper are not tied to any specific programming languages, thus provide a portable design-level solution. These patterns can be easily implemented in widely available object-oriented programming languages, e.g., C++.

2. Design pattern

OOD has proven to be quite popular in practice, and sophisticated OOD methodologies offer significant leverage for designing software [2], including ease of decomposing a system into its constituent elements and partitioning system functionality and responsibility among those elements. However, it is not well suited for describing complex interactions between groups of objects. Likewise, individual objects can often be reused in other implementations, while capturing and reusing common design idioms, involving multiple objects, can be difficult [3]. Design patterns are important because they fill design gaps that objects handle poorly. Patterns help to express design in terms of the relationships between the parts of a system and the rules to transform those relationships. Patterns have given us a vocabulary to talk about structures larger than modules, procedures or objects [12]. Many of these structures are not new; but even though some of them are decades old, they are seldom explicit. Patterns bring these structures long known to expert practitioners to the daily programmer. They help programmers gain competence, and sometimes excellence using microarchitectures such as those published in [5]. The complexity of parallel programming lies on the interaction among modules in the computation. Since design pattern is good at describing the interactions among objects, it is thus a natural and good supporting methodology for parallel programming.

Recently, design patterns have been applied to successfully solve issues in developing concurrent or distributed software systems [14–18]. Most of these works resolve issues such as event handling, job dispatching, connection management, etc. This paper applies software design patterns to parallel processing, which is similar to distributed/concurrent computing conceptually but with different focus. The patterns in this paper focus on computation topology and task management as well as data distribution and exchange among tasks.

3. The computing space pattern

Parallel computation can be viewed as a group of processes cooperating to achieve a common goal or obtain a global result in a faster way. The structuring of the group of processes becomes a major constituent for the complexity and efficiency of parallel programs. Current practice scatters the code for the group management around the entire program. This makes the used group structure and parallel processing techniques unclear and not easy to understand; it unavoidably leads to trouble in program maintenance. It is even worse when the computation exploits nested, multiple parallelisms, or dynamic group structure. The *computing space* pattern is developed to address the above issue and consists of two kinds of objects: the topology object and neighbor objects. The topology and neighbor objects reside on master and slave processes, one for each, respectively. The relationships between these two kinds of objects are depicted in Fig. 1.

The topology object maintains the process group structure. The topology object itself contains two layers, as shown in Fig. 3. The upper layer describes the logical structure of the process group specified by the parallel algorithm, including the number of slave processes, the interconnection among the slave processes, and the information about the programs executed by the slave processes. The lower layer maintains the information about the physical computing environment, including the number of processor nodes,

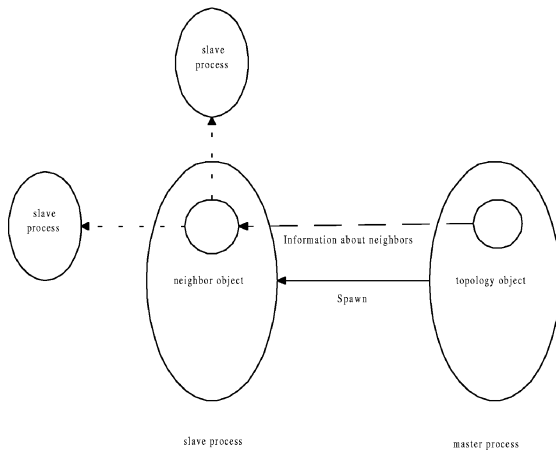

Fig. 1. Topology and neighbor objects.

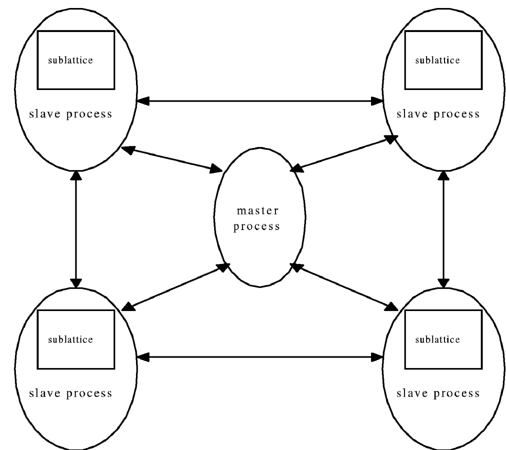

Fig. 2. Parallel simulation of Ising model.

processor speed and workload, networking structure, communication link properties such as latency and bandwidth. The information is crucial for adaptive parallel systems to perform dynamic processor allocation and dynamically adjust the data partition and distribution methods. A method in the topology object is responsible for efficiently mapping the logical process group structure onto the physical computing environment.

The topology object provides an interface allowing a master process to create certain parallel computation structure according to the group structure information in it. The group structure is determined by the parallel algorithm used. A slave process, after spawned by the master process, first instantiates a neighbor object which records the communication structure among it and its neighbors according to information received from the topology object. Through the *computing space* pattern, the computation and communication structure of a parallel computation is gathered into software modules in a form which can be easily accessed and managed dynamically. This leads to a clearer and more understandable software structure for maintenance. Moreover, through dynamic manipulation of the topology and neighbor objects the pattern especially benefits adaptive parallel computation which needs to dynamically reconfigure its computation and communication structure for better performance. In the following, we use the parallel simulation of Ising model [8] as an example to illustrate how to build and apply the *computing space* pattern. In parallel simulation of two dimensional Ising model using master-slave approach, the lattice of spins is divided into a set of sublattices each of which is processed by a slave process, as shown in Fig. 2. The master process collects data from slave processes at the end of each iteration and then computes the macroscopic information. In this case, C++ and PVM [6] library together are used to implement the *computing space* pattern. We defined two classes for topology and neighbor objects, respectively. The interface of these two classes is shown in Table 1.

Class `Topology` uses a data structure, `Mesh`, to store the mesh group structure. The member function, `Spawn()`, uses the facilities provided by PVM to spawn the slave processes according to the group structure and pass them the information of the neighborhood. In the constructor of class `Neighbor`, it receives the process Ids of its up, down, left, right neighbors and the master process from the master process. These process Ids are then used by the slave process for the communication through PVM in the following computation.

4. Group communication pattern

In data-parallel computation, distribution adds a new design dimension requiring a designer to take into account the partitioning and distribution strategies with respect to distributed data structures which themselves are kept in processes running on differ-

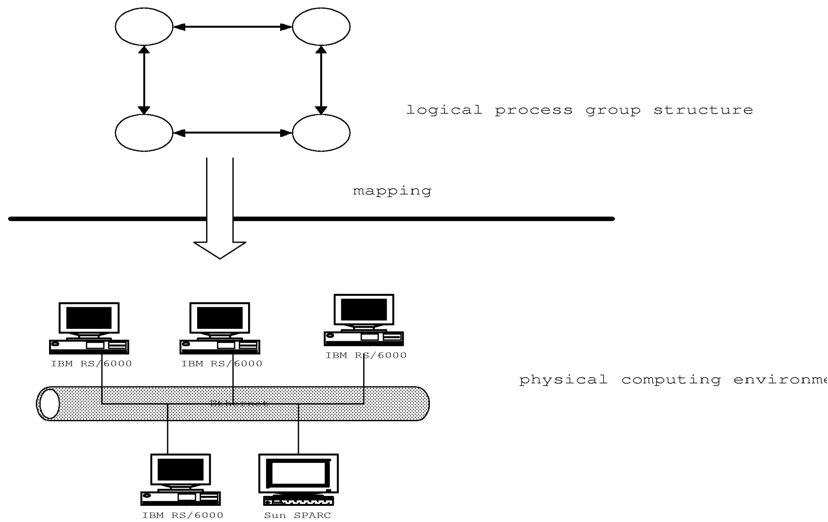

Fig. 3. Layered structure of the topology object.

Table 1
Interfaces of class `Topology` and `Neighbor`

```
Class Topology
{
public:
  Topology(Mesh);
  void Spawn();
  ...
private:
  Mesh slave_topology;
}
```

```
class Neighbor
{
public:
  Neighbor(int up, int down, int left,
           int right, int master);
  int Left();
  int Right();
  int Up();
  int Down();
  int Master();
  ...
private:
  int left, right, up, down, master;
  ...
};
```

ent processors. Group communication is a recognized technique for managing the complexity of distributed systems [1]. It can be used to simplify implementation of the partitioning and distribution strategies by enabling simultaneous addressing of all members of a group. The partitioning and distribution strategies may be reused with different applications that use the same data and task structures.

The *group communication* pattern in this section addresses the issue of partitioning and distribution strategies in data-parallel computation based on the master-slave model in cooperation with the *computing space* pattern described in the previous section. A common feature of the master-slave model in parallel computation is that the master process broadcasts global information or distributes data to slave processes and collect computation results from the slave processes. The *group communication* pattern consists of two kinds of objects: the global and local objects. The global object resides on the master process and the local objects on the slave processes. Conceptually, the global object and the collection of local objects represent the

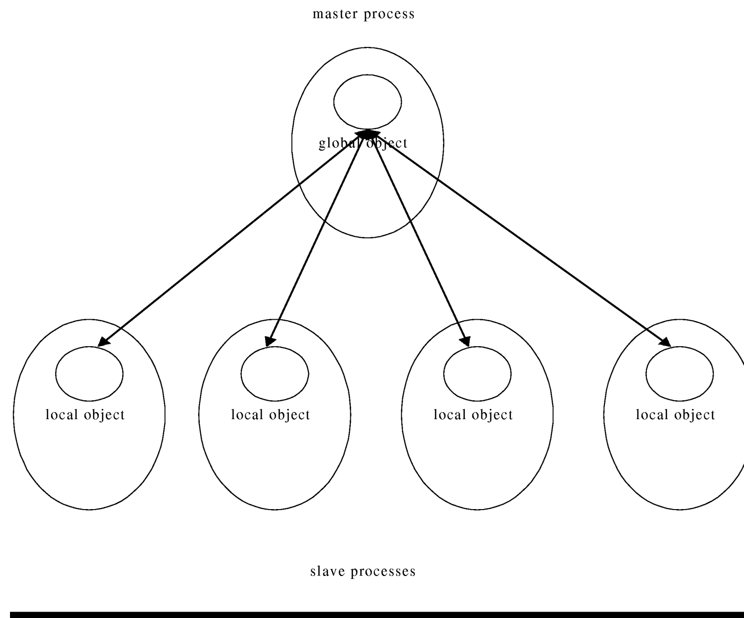

Fig. 4. Communication structure between global and local objects.

same shared data, but used in the master and slave processes, respectively. Technically, the data sharing is done by keeping two copies of the shared data on the global and the collection of local objects, respectively. Replication is used to improve the efficiency of local accesses to the shared data. The consistency between the two copies of data is guaranteed by proper updating of global and local objects through communication among processes at specific points of computation. The communication details are encapsulated in the global and local objects, however, when to update data for maintaining consistency is determined by the program context in the master and slave programs. The communication structure among the global and local objects is depicted in Fig. 4. The global object sets up its relationship with the local objects according to the group structure information which is stored in the topology object of the *computing space* pattern.

The usage of the shared data in the master-slave model usually has some simple and clear fixed patterns. Therefore, the data consistency problem in this model is less complex and easier to solve than in general distributed computing scenarios. The following are three common usage patterns of shared data in this model:

- *Global data broadcasting*. The master process needs to broadcast global information or data stored in the global object to the local objects in the slave processes at perhaps the beginning of each computation iteration. Here, each local object gets the same whole copy of the global object.
- *Computation result collection*. The master process has to collect the computation results from the slave processes for further processing or display at the end of each iteration of computation. Here, the results in all local objects are put together into the global object to form a whole picture of the computation results.
- *Data scattering and gathering*. The master process at the beginning of each iteration distributes initial data in the global object to the slave processes. Each slave process gets a part of the initial data and stores it in the local object to perform the computation. At the end of the iteration, the slave process has updated the initial data in the local object according to the computation performed. The master process

Table 2
Global data broadcasting

```
Class Global
  {
  public:
    Global(Topology*);
    void Broadcast();
    Grid grids;  //The global information
         about the fields in plasma reactor
    ...
  private:
    //information about the
      corresponding local objects
    ...
  };

  class Local
  {
  public:
    Local();
    Receive();
    Grid grids;  //The global information
               received from global object
    ...
  private:
    //information about the
      corresponding global object
    ...
  };
```

Table 3
Computation result collection

```
class Global
  {
  public:
    Global(Topology*);
    Collect();
    Field fields; //the collected whole
                          computation result
    ...
  private:
    //information about the
      corresponding local objects
    ...
  };

  class Local
  {
  public:
    Local();
    Send();
    Field fields; //computation result
                       in each slave process
    ...
  private:
    //information about the
      corresponding global object
    ...
  };
```

then gathers the updated data from slave processes into the global object to get complete information.

In the above three shared data usage patterns, the accesses to the shared data from the master and the slave processes have fixed sequence and would not interferes each other. Therefore, the complex multiple concurrent read and write accesses problem does not occurs here. The following are two examples of how to build and apply the distributed object pattern. The two examples concern the *global data broadcasting* and *computation result collection* usage patterns which appear in our parallel implementation of plasma simulation [9]. The implementation is based on C++ and the PVM [6] library. Table 2 shows the interface definitions of the global and local objects for the global data broadcasting usage pattern.

The global object's constructor, `Global(Topology*)`, needs a topology object to help initialize the relationship with local objects. The local object's constructor, `Local()`, sets up connection with the global object for following communication. The methods `Broadcast()` and `Receive()` perform the data updating stuff through the help of PVM to maintain the consistency of shared data. In this case, the shared global information, `Grid`, is broadcasted from the global object to a set of local objects. Table 3 shows the interface definitions of the global and local objects for the computation result collection usage pattern. The computation result collection is achieved through the cooperation of the two member functions, `Collect()` and `Send()`. The computation result, `Field`, in each local object is just a part of the whole result. The other parts of interface definitions are the same as in Table 2.

5. Performance evaluation

While with better software structure and other benefits, software design pattern usually introduces an additional layer of encapsulation, which may lead to performance degradation through the overheads of addi-

Table 4
Performance of the parallel plasma simulation

	sequential	2 nodes	4 nodes	6 nodes	8 nodes	10 nodes	12 nodes	14 nodes
time (sec)	2828.82	1427.81	734.47	493.45	382.41	312.22	275.15	234.20
speedup	1	1.98	3.85	5.73	7.39	9.06	10.28	12.07

Table 5
Overhead evaluation of the two design patterns

	computing space pattern	group communication pattern
overhead (sec)	0.03	0.14

tional procedure-call and object initialization. However, previous experience [14] shows that with careful design, performance loss can be kept to a minimum. This section evaluates the additional overheads when applying the two design patterns proposed in this paper. The evaluation is based on a parallel plasma simulation [9]. Table 4 shows the runtime performance of the parallel simulation on IBM SP2 with IBM PVMe.

Different implementations of the parallel simulation, with or without the two design patterns, are compared to evaluate the possible performance loss. Table 5 shows the result of the comparison. Compared to the total execution time in Table 4, the overhead incurred by the design patterns is negligible. Moreover, the additional procedure-call overhead can be further minimized by the *inline* technique in programming languages such as C++.

6. Conclusions

This paper presents two useful design patterns concerning group structuring and communication for parallel computation of master-slave model. The two patterns work in a collaborative fashion. The topology object in the *computing space* pattern stores the group structure information of the parallel processes which the *group communication* pattern use to set up the relationship between the global and local objects. Applying these two patterns has a number of significant benefits. First, it promotes design-level reuse: routine solution to the data partition and distribution problem with well-understood properties can be reapplied to new applications with confidence. Second, it can lead to significant code reuse: often the invariant aspects of process group structure lend themselves to shared implementations. Third, it is easier for others to understand the organization of a parallel computation because conventionalized structures are used. Fourth, use of standardized group structure supports interoperability. Our experience, when applying these two patterns to two real applications: parallel simulations of Ising model and plasma respectively, indicates that they do provide help as explained previously. As indicated by the evaluation, the runtime overhead of these patterns due to the higher-level object-oriented structuring is negligible in comparison with the total execution time.

References

[1] M.F. Kaashoek, A.S. Tanebaum, Group communication in the amoeba distributed operating system, in: Proc. 11th International Conference on Distributed Computer Systems, Arlington, VA, May 1991, pp. 222–230.

[2] J. Rumbaugh et al., Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[3] R.T. Monroe, A. Kompanek, R. Melton, D. Garlan, Architectural styles, design patterns, and objects, IEEE Software 14 (1) (1997) 43–52.

[4] K.C. Huang, P.C. Wu, F.J. Wang, Developing high-performance scientific applications in distributed computing environments, in: Proc. 5th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, 1995, pp. 308–312.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.

[7] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, MA, 1994.

[8] K. Huang, Statistical Mechanics, John Wiley & Sons, Inc., New York, 1963.

[9] K.C. Huang, J.H. Tsai, F.J. Wang, Parallel computation of loosely synchronous system with time-increasing data set, in: Proc. 2nd Symposium on Computer and Communication Technology, 1996, pp. 50–56.

[10] T.G. Lewis, H. El-Rewini, Introduction to Parallel Computing, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[11] A.L. Lastovetsky, mpC: a multi-paradigm programming language for massively parallel computers, ACM SIGPLAN Notices 31 (2) (1996) 13–20.

[12] J.O. Coplien, Idioms and patterns as architectural literature, IEEE Software 14 (1) (1997) 36–42.

[13] R. Panwar, G. Agha, Partitioning and distribution strategies as first class objects, Technical Report available at http://www-osl.cs.uiuc.edu/Papers/Parallel.html.

[14] R.K. Keller, J. Tessier, G.V. Bochmann, A pattern system for network management interfaces, ACM Comm. 41 (9) (1998) 86–93.

[15] D.C. Schmidt, Acceptor-connector—An object creational pattern for connecting and initializing communication services, in: Proc. European Pattern Language of Programs Conference, July 10–14, 1996.

[16] I. Pyarali, T. Harrison, D.C. Schmidt, T.D. Jordan, Proactor—An object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events, in: Proc. 4th Annual Pattern Languages of Programming Conference, Allerton Park, IL, September 2–5, 1997.

[17] R.G. Lavender, D.C. Schmidt, Active object—An object behavioral pattern for concurrent programming, in: Proc. 2nd Pattern Languages of Programs Conference, Monticello, IL, September 6–8, 1995.

[18] S. Baker, Design patterns for network development, UNIX Review (1997) 33–37.