The distributed data protocol within OOPAR

Philippe R. B. Devloo[1], Edimar C. Rylo[2], and Tiago L. D. Forti[3]

[1] FEC UNICAMP, Campinas - SP, Brazil,
phil@fec.unicamp.br
[2] ecrylo@fec.unicamp.br
[3] fortiago@fec.unicamp.br

Abstract. This contribution describes a distributed data protocol as implemented in the object oriented programming environment OOPAR. OOPAR develops a paradigm for the development of parallel algorithms based on the concepts of tasks which act on distributed data. The task objects which implement a part of the parallel program depends on data objects with specific version to operate. The dependency of the tasks with respect to the data are registered by a data manager which takes actions to satisfy the tasks request. The procedure implemented by the data manager resembles a distributed data access protocol and is documented in the paper. Its main features are : -the administration is performed in a distributed manner (there is no "central processor" -objects are transmitted between processors only if needed -several processors can access an object simultaneously if they requested read access -the administration guarantees the consistency of access states if the requests are consistent -tasks which request access to objects with version older than the actual version of the object are automatically canceled.

1 Introduction

Parallel scientific computing is generally associated with the use of communication libraries such MPI or PVM [5, 1]. These libraries offer the user a API[4] for transmitting data between processors. The user, however, is still responsible for organizing his program in such a way that each data transmitted on one processor is received by a corresponding call on the other processor in a synchronous manner. This imposes a very strong synchronism on the coding structure and makes the maintenance of parallel software very difficult.

OOPAR [2, 4] is an object oriented environment whose objective is to propose a programming paradigm for parallel computing which is structured and extensible. A parallel program should be able to run unmodified on both shared and distributed memory computers. Using OOPAR, a parallel program is structured as a sequence of tasks which act on a set of distributed data. The sequence of tasks is synchronized by the version of the data which the task needs in order to execute.

[4] Application Programmer Interface

Many other approaches are proposed for applying the object oriented programming philosophy to parallel computing [6, 3]. The authors believe the OOPAR is significantly different in scope and generality.

2 General Concept of OOPAR

2.1 Serialization and data distribution

Using the concept of serialization, any data type can be transmitted between processors. Data which has been transmitted between processors cannot be referenced again within a traditional programming language such as C++.

In order to reference a data object on a pool of processors a unique Id needs to be associated with each referenced data object. The data manager within OOPAR associates a unique id with each data object submitted to it.

2.2 Tasks

Data is transformed by processes. Processes in object oriented concepts are associated with methods executed on data objects. Methods, however cannot be transmitted between processors as data objects. Within OOPAR, tasks are associated with parts of the parallel algorithm which needs to be executed. By associating objects with pieces of executable code, these same objects can be executed on any processor in a flexible way (as long as the data on which they operate is available).

2.3 Data dependency of tasks

Tasks depend on the availability of data with an appropriate version in order to execute. A task manager, which is instantiated on each processor, administers tasks submitted to it. When a task is submitted to the task manager, its data dependency (object id, version and data access type) is submitted to the data manager.

Tasks can request access to data by read access, write ace's or version access. When a task has read access to a data object, it will not modify the data or its version. When a task has write access to a data object, it can modify the object and its version. When a task has access to the version of a data object, it can only modify its version.

2.4 Data manager

The data manager undertakes the necessary steps to make the data available to the submitted tasks. The protocol which documents these steps is the object of this contribution.

The OOPAR programming paradigm can be associated with a data flow programming paradigm. Within the data flow paradigm, the execution of a program can be modeled by a sequence of processes which transform data.

3 Concept of distributed data and data version

Data objects are generally referred to by pointers. Pointers contain the memory location where the data can be found within the memory space of the process. One of the difficulties in parallel programming is that the traditional memory concept does not apply to distributed memory machines, because each process has its own memory allocation.

Within OOPAR, a unique identifier is associated with all objects which need to be referenced over the pool of processors. This identifier is assigned by the data manager and is composed of the id of the processor together with a unique identifier.

Data is transformed by tasks. In order to keep track of the state of the data, a version is associated with each data object. A version is stored as a stack of object versions with cardinalities. The cardinality indicates the maximum number a version can attain at a certain level. Figure (1) illustrates the concept of versions and cardinalities. The version object has a maximum level and at each level has a version and a cardinality associated.

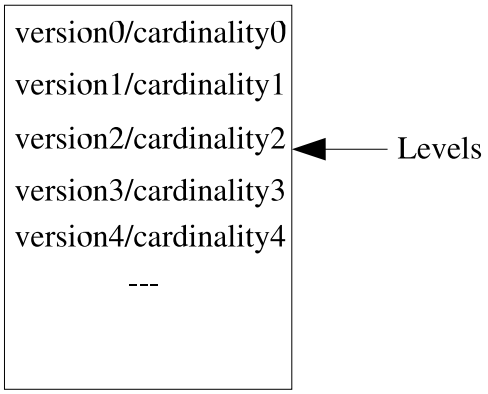

Fig. 1. Versions and cardinalities

Version object implement the following interface:

- IncrementVersion() : will increment the version of the highest level. If this version is then equal to the cardinality, the level is decreased and this level is incremented
- IncrementLevel(int cardinality) : will increment the stack of version/cardinality objects and associate the parameter with the cardinality of the highest level
- DecreaseLevel() : decreases the level and increments the version.
- Comparison operators

The stack of version/cardinality allows a fine control of the version of data objects. A fixed number of iterations on an object by a task can be programmed

by making the task dependent on an arbitrary version of the data object of a given level. Once the object has been incremented this fixed number of times, its level will be automatically decreased and the task canceled. Figure (2) illustrates a task which depends on any version at level 3 and which will be executed recurrently until the version of level 3 reaches niter. Then the version will be $\{1/-1, 2/5, 5/-1\}$. As this version will be greater than the version requested by the task, the task will be canceled.

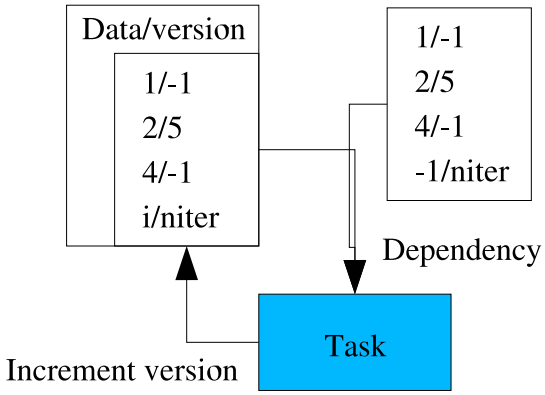

Fig. 2. Task iteration on an object

4 Data access requests

Tasks are objects which will access and/or transform data objects. They represent a part of the execution of the code. Tasks declare their data dependencies through a list of data dependency objects. A data dependency consists of:

- Object Id: which uniquely identifies the object within the pool of processors
- Data version: the version the data object needs to have to be compatible
- Data access state: a task can request to access the data for reading, writing of version modifying

The task can execute only if the data is available with the appropriate version and data access. The decision process for giving access to the data is implemented by the data manager and collaborating classes.

If a task needs to access a data object which is owned by a different processor, then an access request will be sent by the data manager to the owning processor. Ownership of data is discussed in the following section.

5 Object ownership

One of the concepts within OOPAR is that data is owned and administered by one processor. This means that at any given time, a processor can be associated with a data object.

Ownership of a data object can change during execution. Only the owning processor can grant write access to a task object (which filed such a request). Therefore, in order to grant write access to a data object on a given processor, the ownership of that data object first needs to be transferred to the processor of the requesting task.

The ownership of a data object is traced by the processor which previously owned the data. As the processor which originally received the submitted data object can be inferred by the id of the object, the ownership of any data object can be traced. The ownership information is updated dynamically by the response of the owning processor (once it has been found).

6 Access priority rules

OOPAR defines three types of access state: read access, write access, version access. When a task has read access, it will read the data without modifying it. Several tasks may have read access to a same data object simultaneously. When a task has write access to a data object, it will modify the data. No other task will be granted any access state to the data object. When a task has version access to a data object, it can modify the version of the object but has no access to the data itself. A version access request will not transfer the data to the requesting processor nor will it transfer its ownership. Version access state is meant to give control to tasks which are meant to trigger the execution or cancellation of other tasks.

Several tasks may request access to a single data object simultaneously. Such requests may be conflicting, e.g. one task may request read access while another task requests write access. OOPAR prioritizes requests in the following way:

- read access has higher priority than other access requests. While there is a task with a compatible read access request, tasks which request other access states will not be granted access
- version access has priority over write access
- access states can not be modified if a task is locked on the data. Once a task which is granted an access request and is executing, the request cannot be revoked anymore

The access priority rules are crucial in the decision process of the data manager which compatible should be granted next. It is up to the user to submit tasks with compatible access requests. If the access requests are inconsistent, the environment will lock. Through the data logging mechanism, it is usually fairly easy to back trace which task sequence generated an inconsistent state. Algorithms to perform auto diagnosis of the environment are feasible and will be developed in future work.

7 Data states

During the processing of access requests, the data object will assume several transition states during which other access requests may or may not be granted. The correct definition of these states determines the consistency of the environment.

- No state: the default transition state of the object; no state means the absence of a transition state
- Is accessing: a task is executing using the data. During this state only compatible states may be granted. If a task is accessing the data with read access, other tasks/processors may be granted read access.
- Cancel read access: the owning processor asked to cancel the access state. This state occurs if the processor has read access to the data but is not owner of the data. The owning task send a request to cancel the read access. The read access cancellation was not honored because the task is executing using the data object.
- Request delete: a delete object request was issued for the object. This state will remain as long as executing tasks are using the object and as long as processors with read or version access have not sent an access cancellation confirmation.
- Suspend read access: when a task/processor is granted a version access request, the access of other processors is suspended until the task with version access terminates. During a suspend read access state, no task is granted access to the object
- Read access: any number of tasks and/or processors can have read access to a data object simultaneously. The owning processor keeps track of the list of processors which have read access to the data. This configures the read access state

8 Request response protocol

The request response protocol is the decision tree which is followed to decide whether an access request can be granted or not. This decision tree is influenced by the state of the object, the type of request submitted and whether the processor on which the request was made is owner of the data.

After each data state modification the method *VerifyAccessRequests* is called. This method verifies which access requests can be granted and which access requests will cause the requesting task to be canceled. Access requests are submitted to data objects through three channels:

- A task submits an access request
- A request is filed from a different processor through a request task
- A request is filed form a different processor owner of the data object

Figure (3) shows the possible flow of requests. The type of requests depend on the originator: A task only asks for straight data access requests. A data manager which doesn't own the data may file data access requests and/or transmit confirmation messages. A data manager which owns the data (e.g. request1) may grant or revoke access states.

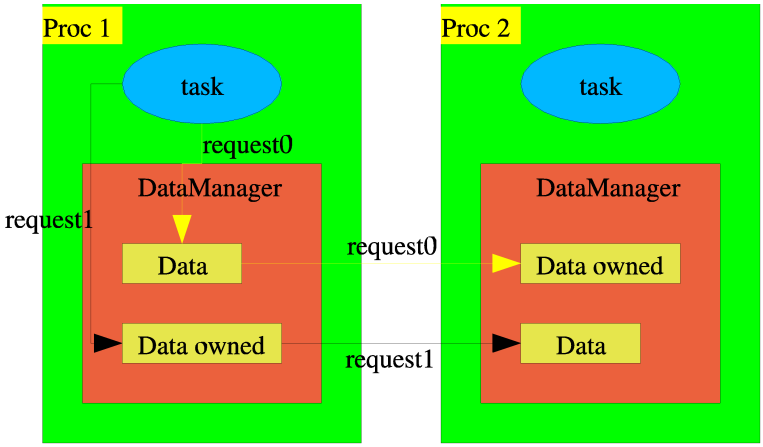

Fig. 3. Requests filed between components

8.1 Procedure within *VerifyAccessRequests()*

VerifyAccessRequests() will verify all access requests which are filed at the data object and grant a compatible request according to the request priorities and the state of current data access. Verifying access requests occurs in two stages: first the method verifies if the data needs to be put in a transition state (i.e. cancelread access, suspend read access) or whether write access needs to be revoked (a read access request has higher priority than a write access request); then the access requests are examined one by one to verify whether they can be granted or not.

Preliminary verification

- If the data object has write access granted and a compatible read access request, then write access will be revoked: a read access request has priority over a write access request. This is a exceptional situation: if the write task would have been put into execution, then the write access would not have been canceled
- If the data object has version access request and no read access requests, then initiate the process of suspending the read access state of all processors (the data will be put in suspendreadaccess state)

– If the data object has write access requests and no other access requests, then initiate the process of canceling the read access state of all processors (the data will be put in cancelreadaccess state)

Granting access requests For access requests regarding tasks of the current processor, the task manager is notified directly through the method *NotifyAccessGranted*. This doesn't mean the task is put into execution: a task is put into execution only if all access requests have been satisfied.

For access requests regarding other processors, a message is sent to the requesting processor:

– If the access request is read access, read access is granted to the processor. If their is a compatible write access request for the data object, a cancel read access request is filed subsequently
– If the access request is write access (and can be granted), the ownership of the data is transferred to the processor.

8.2 Cancel Read Access protocol

A cancel read access is performed in two steps: first a message is sent to all accessing processors, requesting read access cancellation; second the accessing processors send a cancel read access confirmation message. Between these steps the data is put in a cancel read access transition. Figure (4) illustrates the two steps.

The cancel read access confirmation is sent by the not owning processors only if there are no read access requests. The data is deleted from the processor before sending the cancel read access confirmation message.

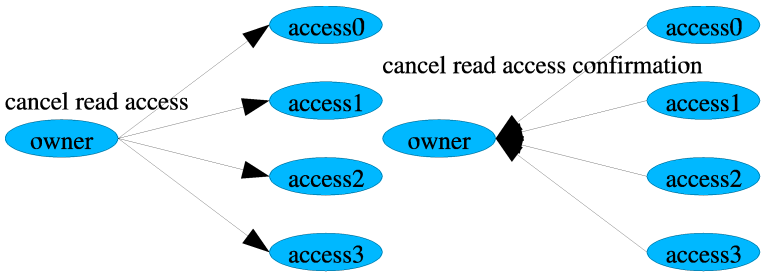

Fig. 4. Cancel read access protocol

8.3 Suspend read access protocol

The suspend read access protocol is identical to the cancel read access protocol, except that the data is not deleted from the not owning processor. The data is put in suspend read access state which read access is suspended.

8.4 Transfer ownership

The transfer ownership protocol occurs in two steps between the currently owning processor and the processor to which the data is transferred. First a message transfer ownership is filed, then a transfer ownership confirmation closes the cycle. The data from the original processor is deleted after the confirmation message is received.

9 Possible improvements

The protocol for data access within OOPAR has been implemented and tested on parallel scientific codes. The protocol is sound in that the authors believe all cases have been covered. No formal proof of consistency has been presented though.

The consistency of the requests is still the responsibility of the user. The access request protocol will still block if inconsistent requests are filed. It would be possible to detect such locks and take corrective action. This not implemented.

The protocol is not robust with respect to communication failures. It is assumed that each message sent will be received by the target processor. The authors believe that fail safe mechanisms can be built in, introducing the notion of timeouts and data mirroring. This is not implemented yet.

10 Conclusions

The kernel of OOPAR is the distributed data access protocol. This protocol is responsible to follow up on data access requests submitted by the tasks. A request is filed at the data object and should not be repeated.

The protocol favors the read access state, allowing the object to be present in different processors simultaneously. Tasks which request data access state for obsolete versions will be canceled by the Data manager. This allows for task management within the OOPAR environment. Tasks which are recurrent can be canceled by putting their dependent data a version higher than requested.

The data access consistency is enforced by the data manager. Inconsistent requests are logged by a logging environment for debugging purposes.

The protocol processing is governed by a method VerifyAccessRequests which can called repeatedly. The implementation is still complex, which indicates that further studies are warranted. However the protocol has been tested on a complex parallelization of a CFD software.

Further improvements have been recognized and can be implemented without changing the interface of OOPAR. This indicates that the concept of tasks and data is sound and leads to a workable, robust concept for parallel computing.

References

1. A. Geist Et. Al. *PVM Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Programming*. Cambridge : Massachussets Institute of Technology, 1994.
2. P.R.B. Devloo, F.A.M. Menezes, and E.C. Silva. OOPAR : The development of an environment for parallel computing using the object oriented programming philosophy. In B.H.V. Topping, editor, *Advances in Computational Structures Technology*, pages 151–156, 10 Saxe-Coburg Place, Edinburgh, EH3 5BR, UK, 1996. CIVIL-COMP Press.
3. P. Lu G. V. Wilson. *Parallel Programming Using C++*. MIT Press, 1996.
4. Gustavo C. Longhin and Philippe R. B. Devloo. Parallelization of a scientific code using oopar. In *IBERIAN LATIN-AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING*, volume XXIV. ABMEC, 2003.
5. A. Skjellum W. Gropp, E. Lusk. *Using MPI : Portable Parallel Programming with the Message Passing Interface*. Cambridge : Massachusetts Institute of Technology, 1996.
6. P. Verbaeten W. Joosen, S. Bijnens. Object parallelism in XENOOPS. In *Proceedings of the First Annual Object Oriented Numerics Conference*, 1993.