Coupling DSM-based Parallel Applications

Yvon Jégou

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Coupling DSM-based Parallel Applications

Yvon JEGOU

N°4060

Décembre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4060--FR+ENG

Coupling DSM-based Parallel Applications

Yvon JEGOU

Thème 1 — Réseaux et systèmes
Projet Paris

Rapport de recherche n°4060 — Décembre 2000 — 19 pages

Abstract: When coupling applications running on distributed memory architectures or clusters, the coupling library must adapt to the distribution of the data in the memory of each computation node. The library must be prepared to redistribute the data when the coupled applications use different data mappings or when the number of processors of the two architectures are different. Mome is a user-level software DSM which allows programs running on a distributed memory architecture or cluster to create segments and to share data objects through memory mapping. The segments of the DSM form a simple linear address space where all shared objects of applications are allocated. The Mome coupling library accesses the data through mappings of the DSM segments on the memories of the communication threads. The parallel communication threads are distributed on the computation nodes and exploit the communication capacity of each processor. The data are moved directly between the DSM segments and the transfers do not rely on any knowledge on the application use of these segments.

Key-words: DSM, memory mapping, code coupling, parallel communication.

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Couplage d'applications parallèles construites sur une mémoire répartie partagée

Résumé : Pour coupler des applications s'exécutant sur des architectures à mémoires distribuées, les bibliothèques de couplage doivent s'adapter à la distribution des données dans les mémoires de chaque nœud de calcul. Une telle bibliothèque doit être capable de réaliser une redistribution de ces données lorsque les deux applications utilisent des répartitions différentes dans leurs mémoires ou lorsque le nombre de processeurs des deux architectures diffère. Mome est une mémoire répartie partagée (MRP) fonctionnant dans l'environnement utilisateur et qui permet à des programmes s'exécutant sur une architecture à mémoire distribuée, ou une grappe de stations de travail, de créer des segments partagés et de partager des objets par projection des ces segments dans leur espace d'adressage. Les segments de Mome offrent un espace d'adressage simple et linéaire où tous les objets partagés de l'application parallèle sont alloués. La bibliothèque de couplage de Mome accède aux données au travers de projections de segments de la MRP dans l'espace des processus de communication, indépendamment de la manière dont les calculs sont distribués dans les applications parallèles. Les processus de communication sont répartis sur l'ensemble des nœuds de calcul et exploitent les capacités de communication de chaque processeur. Les données sont déplacées directement dans les segments des MRP et leur accès ne nécessite aucune connaissance sur l'utilisation des segments par les applications.

Mots-clé : mémoire répartie partagée, projection en mémoire, couplage de codes, flots de communication parallèles

1 Introduction

There is a growing need to build large high performance simulation codes from more basic and well-understood software components. Assembling all such components to form a single program is not always possible. First of all, all these programs do not necessarily run efficiently on the same hardware architectures. For instance, the software may request for some specific hardware which is not present on all platforms. This is often the case when interactively visualizing the results of the simulation of a physical phenomenon: the best parallel platform for the mesh computation is probably not the visualization platform. Through the availability of powerful communication networks, it is now possible to run each program on the best platform.

Component models such as CORBA [4], or Java Beans [3] have been developed and allow to connect codes running on distinct platforms. However, the distributed nature of high performance parallel architectures has not been considered in these models. Extensions to these models such as the Parallel CORBA Objects [5] have been proposed and consider the distribution of the object elements on the processor memories.

To efficiently couple parallel simulation codes, it is necessary to consider the specificities of the target architectures, the parallel programming models and also the distribution of the data in the memories. High performance architectures are often parallel computers with communication capability distributed on the processors. Each processor has a limited communication capacity and all processors must participate to data exchanges if high throughput is expected. This problem is not specific to code coupling and must also be tackled through parallel I/O. Using parallel programming models, each processor is usually aware of only a small part of the program data, the data handled by the application on this processor. Communicating with the external world may necessitate expensive redistributions of the data before (or after) the transfers if the internal distribution is not considered before scheduling the transfer. Moreover, using simple communication patterns where all data transit through the same communication link, and then assembling the whole data structure in the memory of a single processor might be impossible. The last problem comes from the nature of simulation coupling. In many cases, only parts of the data structures need to be exchanged. For instance, in atmosphere-ocean coupling, only the ocean surface interacts with the atmosphere model. The coupling

procedure is not restricted to a simple copy of a data structure from a program to another program: the data must be extracted from the numerical model of some application and inserted in the data structures of the other application. Moreover it is possible that the same quantity is not modeled in the same way in the two models. In this case, it is sometimes necessary to introduce some intermediate "transfer" representation, for instance through interpolation of mesh values.

The COCOLIB [2] library enable the coupling of industrial simulation codes on parallel computers. This library recognizes different mesh modeling of the same quantities and allows for coupling of non-matching meshes. The COCOLIB library considers the presence of the MPI communication library for the internal parallel program communication and relies on this library for its own communication needs through the uses of inter-communicators. So the efficiency of the data exchanges depends on the implementation of MPI library and on the presence of a parallel inter-communicator.

PAWS [1] is a software infrastructure for use in connecting separate applications within a component-like model. Applications connect through the PAWS controller and the PAWS API uses the Nexus communication library for the data transfers. PAWS allows for dynamic coupling of applications during their execution, exploits parallel connections between the applications and accept different parallel layout strategies for shared data structures.

Mome is a relaxed consistency DSM which allows programs running on clusters or distributed memory architectures to create shared segments and to map these segments on their local memories, and so, to share data. The presence of a DSM avoids the burden of locating the elements when accessing the distributed objects in the memories of each processor. In this paper, we show that through such a DSM, it is also possible to organize parallel data transfers between the coupled architectures without being restricted by the complexity of the data layouts used by the parallel applications. The next section presents the coupling environment we are considering and section 3 outlines the main features of the Mome DSM. Section 4 presents the application interface of our coupling library and section 5 describes its implementation. Some preliminary performance results are presented in section 6.

2 Coupling Environment

In order to couple distributed applications, the Mome coupling library needs to contact some kind of controller in charge of coordinating the applications. Such a controller keeps track of the registered applications and of their data as well as some information on the way coupling requests can be transmitted. This coordination service is similar to the service offered by the PAWS *controller* or by CORBA Oject Request Brokers. During the course of its execution, a Mome application can register the name – as well as some parameters – of the data structures it makes available to other applications. Some form of URL (the hostname and a port number in our implementation) where coupling requests are to be sent is associated to each registered data. A client application can query the controller with a server and data name. If successful, it receives the corresponding URL. In our simplified implementation of the controller, the role if this controller is limited to transmitting connection URLs to the clients.

2.1 Coupling Strategy

The Mome coupling library defines a few basic data transfer protocols:

- *asynchronous:* each time the client requests it, the current version of the coupled data is transfered.
- *periodic:* a new transfer is initiated every p time-steps.

In order to allow some overlap between computation and communication on the server side as well as on the client side, all transfers must be encapsulated between two requests to the library. On the server side:

- `MNewVersion(handle)`: a new version of the coupled data is made available. A transfer is initiated for all clients requesting a copy for this time-step.
- `MWaitSend(handle)`: the server application waits until all pending transfers on the object are terminated.

On the client side:

- `MNextVersion(handle)`: the client is ready to receive the next version.

- `MWaitReceive(handle)`: the client waits until a copy of the data has been received.

These requests are similar to the asynchronous data transfer requests of MPI. A similar strategy is also available in COCOLIB.

2.2 Parallel Transfers

On distributed memory parallel architectures or clusters, the coupling library must consider the data distribution on the processors of each parallel application and must take into account the fact that, in general, each processor has a limited communication capacity.

The decomposition of the parallel data introduces many problems in the coupling library:

- it is possible that a processor can make access to only parts of the distributed data.
- it is possible that, because different data distribution are used on the two sides, the data managed by some processor must be distributed on different client processors.
- also the same data distribution strategy is used on the two sides, the client and the server might be running on different numbers of processors.

If a data transfer involves non local data – the data distribution on the processors is different from the data distribution for the transfers–, the processors must exchange or redistribute the data before the transfer on the sender side or after the transfer on the client side.

Moreover, when the communication capacity of each processor is limited, the bottlenecks on the communication layer must be tracked.

- the communication load must be balanced on all communication links,
- communication (destination) conflicts must be avoided through optimized schedules.

2.3 Object Coupling

The coupling needs of parallel applications can range from direct sharing of data to more complex exchanges of coupling quantities defined by the numerical simulations. In the simplest form, the data transfer is a simple copy of the data object. The form corresponds to object sharing implemented in PAWS. In the more complex form, the coupled applications must first agree on the structure of the data to be transfered – dimensions, type of elements. The server must also specify how this data is to be extracted from its own data. Data extraction can be limited to fetching sub-arrays, for instance. But more complex routines involving the interpolation of coupling values have also been imagined when the applications consider different grid or mesh representations for the same data, for instance in COCOLIB.

3 Mome DSM

Mome is a relaxed consistency multiple writers DSM running in user-space. Mome targets the execution of loop-level parallelism using an SPMD execution model. The associated runtime library integrates synchronization, reduction and communication routines. Runtime systems for HPF and OpenMP are currently under development.

Mome interacts with the parallel applications through segments. Each process of a parallel application can map a local data structure on a Mome segment. A mapping request on a DSM segment is very similar to a Posix `mmap()` request on a file, where the file descriptor is replaced by a segment number. The DSM is in charge of managing the consistency of all memory pages mapped on a same segment page, whether these mapping belong to the same computation node or to different nodes. Mome uses a relaxed memory consistency model: modifications to shared data by some processor are propagated to other processors only on request from the application, for instance after synchronization barriers. This strategy avoids expensive inter-processor communications during parallel and independent computation phases.

The Mome DSM is thread-safe, the threads can independently request segment mappings, and groups of threads can synchronize on independent barriers.

3.1 Coupling Mome-based codes

Using the Mome DSM, a parallel application can allocate the shared data objects in DSM segments and then each process of the application program maps the whole object or only the parts it wishes to access on its address space. These mappings can be modified dynamically following the process needs, and even in the case where all mappings have been removed, the segment data are remanent.

A DSM reference (segment number, offset and length) supply a uniform reference to a shared data object which does not depend of the way each processor can read or write the data. If some thread wishes to make access to some part of a DSM segments, it needs only to map this part of the segment in its memory space, independently on the processor in charge of running this thread. The Mome coupling library is based on this feature : it considers the DSM segments as a uniform and shared address space. The communication threads are distributed on the computation node and each thread maps in its address space the sub-segment containing the data it is in charge off.

4 Application Interface

The application interface of the Mome coupling library defines six major entry-points : `MRegisterObject`, `MNewVersion` and `MWaitSend` for the server-side, `MFindObject`, `MGetNext` and `MWaitReceive` for the client-side.

- `MRegisterObject`: this library routine exports an application object. The parameters include the application and object names (for the naming service), a description of the exported object (its type), the localization of the object in a Mome segment, its availability, and some extra parameter concerning transfer protocols.
- `MFindObject`: this routine is similar to the previous one. The application and object names serve to locate the object. The type defines the expected type. The access parameters define where, in a DSM segment, the elements of the object must be stored. The client can specify how the transfers are initiated (server or client initiative) as well as the periodicity and the number of copies it is waiting for.

- `MNewVersion`: the server application calls this routine each time a new version of the object is available. This call initiates transfers if one or more clients requested a copy of this version.
- `MWaitSend`: this routine checks that the server can safely modify the object.
- `MGetNext`: the client is ready to receive the next available copy of the object.
- `MWaitReceive`: this routine checks that a copy of the object has been received and can be safely accessed.

All these routines are collective in our SPMD execution model and synchronize the processors. In the current implementation, the object type describes a multi-dimensional array, the number of dimensions, the number of elements on these dimensions and the element type. On the client side, it is possible to first request the object type in order be able to fetch objects of unknown size.

The localization parameter contains a Mome segment number, a base offset and one address stride for each dimension. The array elements need not be contiguous inside the segment. Sub-arrays can be extracted from or inserted inside larger shared structures.

The transfers can be initiated on server initiative. In this case, at connection-time, the server knows for which time-step it must start the transfer from the period parameter of each client. When the transfer is initiated on client initiative, the client must first call the `MGetNext` routine before the transfer is started on the server side.

The extra parameters give some more information such as the endianness of the requester or the connection strategy. It is currently possible to specify how the object is used by the application (for instance that a two dimensions array is block distributed column-wise) and how to distribute the object among the communication threads.

5 Implementation

5.1 Strategy

Our implementation targets the connection of clusters of PC through a Gigabit Ethernet link. Each processor of the cluster has limited throughput using FastEthernet 100Mbits links. In order to get efficient transfers, the following rules must be enforced.

rule 1: a maximum number of processors participate to the transfers,

rule 2: destination conflicts must be avoided,

rule 3: the transfer load must be balanced between the processors,

rule 4: consider data locality of applications.

Because our coupling library is DSM-based, all processors have access to the whole shared space and can participate to the transfers. The transfer patterns can be organized independently of the access patterns of the application. Rule 1 can be applied without any need for data redistribution. Destination conflicts occur when two or more processors try to simultaneously send data to the same processors. Destination conflicts can limit the whole throughput when they delay sending other messages to conflict-free destinations. Avoiding destination conflicts can necessitate a global analysis of all communication requests in order to organize a conflict-free schedule. Rule 3 and 4 may be incompatible. For instance, it can happen that only a 2D data section of a three-dimensional mesh is involved in the coupling and that this section is managed by only few processors of the application. Applying rule 3 necessitates data redistribution on the processors while the application of rule 4 restricts the data transfers to few processors. In the Mome coupling library, the data transfers are organized as follow:

1. **communication channels**: Each channel is in charge of the transfers on a subset of the coupled object. The number of channels and the allocation of the data to each channel can be controlled by both applications. The number of processors on both sides, the data access patterns can also be considered.

2. **sender threads**: a sender thread is forked for each channel. The sender threads are distributed on the server processors.

3. **receiver threads**: a receiver thread is associated to each channel. The receiver threads are distributed on the client processors.

There is a server thread for each receiver thread. This point-to-point organization of the communications using independent threads limits the performance reduction in case of destination conflicts – with two or more receive threads on the same processor. When generating the channels, the library considers the number of processors on both sides. The applications can also steer channel generation using the data distribution parameters. From the data distribution patterns, it is possible to roughly evaluate the performance of different channel organizations. For each possible organization, it is possible to evaluate the cost of the transfers, the cost of extracting the data from the server memory and the cost of inserting it into the client memory.

5.2 Coupling Sequence

The applications are coupled through the following sequence:

1. the server application exports the object. This call is treated by the master server processor. The other processors wait on a barrier. The call specifies the object name, its type, the addressing parameters of the exported object in the DSM space, and some protocol specification.

2. the coupling library opens a listening socket and forks a connection thread on the master processor.

3. the connection thread waits for connection requests on the listening socket.

4. the server registers the exported object and the listening URL in the naming service.

5. depending on the specified connection protocol, the server application waits for the connections (synchronous connections) or resumes execution (dynamic coupling).

6. the client application calls the `MFindObject` coupling routine with the specification of the object name, the object type, the addressing parameters of

the imported object in the DSM address space, possible data distribution, and some protocol request. The call is handled by the client master processor, the other processors wait on a synchronization barrier.

7. the client interface requests the object from the naming service, and receives the server URL.

8. the client coupling library send a coupling request to the server listening thread. The request specifies the expected object type, some architectural informations such as the number of processors, their endianness, the data distribution and coupling parameters.

9. the listening thread computes the number of communication channels depending on the number of processors, the distribution specifications and the protocol options on both sides. The performance of various combinations of channel organizations can be evaluated.

10. the listening thread creates a channel table mapped on a DSM segment. This table contains the object type, its mapping in the DSM space and the definition of each channel. Each channel specification defines the corresponding array section and specifies the processing node in charge of the channel.

11. the coupling library synchronizes on all processors. Each processor scans the channel table (in the DSM shared space) and forks a sender thread for each allocated channel. It is possible to have no, one or many threads on the same processing node.

12. each sender thread opens a connection socket, updates the channel table and waits for a connection request from a receiver thread.

13. the listening thread extracts the access parameters and the URLs from the channel table and generates a channel list. The virtual object type and the channel list are sent back to the client application.

14. the client library validates the object type and builds a channel table in the DSM space from the channel list. The channel table contains the object type,

the addressing parameters of the imported object in the DSM space, the representation (endianness) of the data, and, for each channel element, the corresponding subarray specification and a processing node number. The distribution of the channels on the processors is the responsibility of the client library.

15. each client processor scans the channel table and forks one receive thread for each allocated channel.
16. each receive thread maps its section of the exported object into its address space.
17. each receive thread connects to the URL specified by its channel element.
18. the connections are established, the call to the coupling library returns on the client-side.

5.3 Data Transfers

Depending on the selected protocol, a data transfer is initiated on server initiative, on client request or when both applications agree. For periodic transfers, the server application calls the coupling library for each time-step and the library checks if a transfer must be initiated for each coupled client for this time-step.

5.4 Object distribution

Many parameters such as the number of processors on both sides, the data access patterns as well as the relative performance of the DSM and of the communication layer can be considered during the computation of the number of communication channel and the distribution of the exported elements to the channels. The best performance is expected when all communication channels are exploited (all processors communicate), when the communication load is balanced on the processors and when the memory accesses by the applications and by the communication layer exhibit good spatial locality. In the general case, however, all constraints are difficult too meet simultaneously, due to different numbers of processors for the applications or due to incompatible access patterns, for instance. Moreover it frequently

happens that only a subsection (the surface of a 3D object for instance) of a parallel object involved in the computation is specified in the coupling request. In this case, it might be difficult to characterize the data access patterns for such a section. But, thanks to the presence of the DSM, the connection procedure is always the same, independently of the number of processors or of the access patterns. The current implementation of the Mome coupling library considers an aligned-transfer mode and a free-transfer mode. The aligned transfer mode can be selected when the two applications specify block distributed access patterns. This mode intersects the two distribution specifications and creates a communication channel for each intersection: the data transferred by a communication channel are accessed by the same processor on the server side and on the client side. Using this mode, the communication threads use the distribution patterns of the applications but the communication load of the transfer channels can be different. In the free-transfer mode, the number of communication channels depends only on the number of processors on each side. Currently, the library considers the following rules for the free-transfer mode:

- the number of channels is a multiple of the smallest number of processors,
- the number of channels is less or equal than the largest number of processors,
- the number of channels is limited to three times the smallest number of processors.

For instance consider coupling eight processors on the server side and six processors on the client-side. The coupled vector (2048 elements) is block distributed on the two sides.

The free transfer mode in Fig. 1 uses six channels. For the client-side, on each computation node, the communication threads access the same data elements as the application: locality is maximal. This is not the case on the server side: only six processors participate to the transfers. The transfer threads generate page-faults when accessing the data. The aligned transfer mode in Fig. 2 uses 13 channels. But each of these channels accesses the same data as the local application.

channel	section	sender	receiver
0	0 : 341	0	0
1	342 : 683	2	1
2	684 : 1025	3	2
3	1026 : 1367	4	3
4	1368 : 1709	6	4
5	1710 : 2047	7	5

Figure 1: Free transfer mode

channel	section	sender	receiver
0	0 : 255	0	0
1	256 : 341	1	0
2	342 : 511	1	1
3	512 : 683	2	1
4	684 : 767	2	2
5	768 : 1023	3	2
6	1024 : 1025	4	2
7	1026 : 1279	4	3
8	1280 : 1367	5	3
9	1368 : 1535	5	4
10	1536 : 1709	6	4
11	1710 : 1791	6	5
12	1792 : 2047	7	5

Figure 2: Aligned transfer mode

6 Performance

We consider a simple application, a finite difference discretization of Helmholtz equation. During each time-step, the main Fortran program computes a new 2048×2048 double precision solution array. This main program is running on a cluster of 8 dual-processor 450Mhz Pentium II connected through SCI. The distributed arrays are mapped on a segment of Mome DSM. The client program computes the

error between the solution array and the exact solution. The client is running on a 6 dual-processor 500Mhz Pentium III cluster connected through Gigabit Ethernet. All processors of the two clusters are connected to the same Ethernet switch through 100Mbits FastEthernet links. The coupling of the client to the server is dynamic. In the experiments, the client requested 20 transfers and the requested time-step period is 5.

Before a first connection request has been received, the main program executes a new iteration every 0.18 seconds. For each transfer, the data is first copied into a local buffer and then this data is transmitted using a socket write. The first step is necessary because, in general, the sub-array is not contiguously stored in the memory and so, cannot be output directly. Moreover, when the two architectures have different endianness, or when the data type for the transfer is different from the data types in the applications, data and type conversions are applied during this copy phase. The data copy must be terminated before the server application can modify the shared array. During this copy the transfer threads and the application share the computation resources – processors and memory system – of the local node. Once the data has been extracted from the application memory, computation and communication can proceeded in parallel. Figure 3 shows an execution trace of the server application. During the traced run, the server received the coupling request during time-step 17. The sender threads initiate the transfers as soon as the connection has been established with the client receive threads. The slow-down due to the second transfer during time-step 22 is smaller than average but the third transfer generates a long delay at time-step 27. This comportment comes from the initialization phase of the client program: once the connection with the server is established, the client must allocate its data structures before integrating the received data from the reception buffers. As long as these buffers cannot be reused, the receive threads don't allow any transfer and delay the communication of step 22 data. When the server wants to modify the shared array in time-step 27, it must wait until the previous communication is finished and the data copied in the output buffer. After this time-step, the server, the communication layer and the client application form a pipeline. As shown Figure 3, the slow-down of the server is limited to two iterations, and the presence of the code coupling does not seem to affect the performance for the other iterations. Note however that the cluster nodes are dual processor nodes: the Fortran application program rarely conflicts with the library threads for computational resources.