Using MeDLey for the grid-decomposition methods

T. Es-sqalli, J. Guyard
RESEDAS Project
LORIA, Scientific Campus B.P. 239
54506 VANDŒUVRE-Les-NANCY CEDEX France

E. Dillon
Mathematics and Computing Technology
The Boeing Company P.O. Box 3707 MC 7L-20
Seattle, WA 98124-2207 U.S.A

Abstract ***Explicit parallelism relies on transmissions of messages between processes. However, as workstations are not intended to manage this kind of communications, it is necessary to use communication libraries, known as Message Passing. Currently, only MPI (Message Passing Interface) is still used, and its use became more complex. In order to solve this problem, a new language called MeDLey was developed, its purpose is to allow to the users an easier parallelism programming based on communications using Message Passing.***

In this paper, we will first overview the basics of the MeDLey syntax and semantics, before talking about the extension part of this language for the grid-decomposition methods.

Keywords: parallelism, Message Passing, communication libraries, grid-decomposition, MeDLey

1 The M*e*DL*ey* Language

The MeDLey project [2] [4] [3] is based on two statements :

- first, a lot of users are now implementing their parallel code using inter-task communications apparented to Message Passing ;
- secondly, a lot of new communication mechanisms are now available, leading a novice user to confusion.

Moreover, all users keep seeking efficiency, but the current development efforts (MPI for instance) are often guided by the need of portability, leading to drops of performances. So, on one hand such communication libraries try to provide more and more functionalities to allow fine tuning, but on the other hand, faced to this wide number of primitives, users may find it hard to get the best efficiency.

So, the first aim of MeDLey was to provide a unified notation to specify the communications within a distributed application. To reach this goal, this notation allows the definition of data exchanges with various semantics.

After this, the second aim of MeDLey was to guarantee efficient communications within such a distributed application based on MeDLey. That means a MeDLey compiler should be able to generate efficient communication primitives in a target language (currently C++) for some underlying implementation. Among them, MeDLey should be able to generate primitives on top of something of a portable library (MPI e.g.), but also for specific hardware environments (directly on top of *sockets*, or *AAL-4/5*, ...) to provide full efficiency.

In the following sections, we will first describe the syntax and semantics of the MeDLey notation. After this, we will talk about the extension part of this language for the grid-decomposition methods.

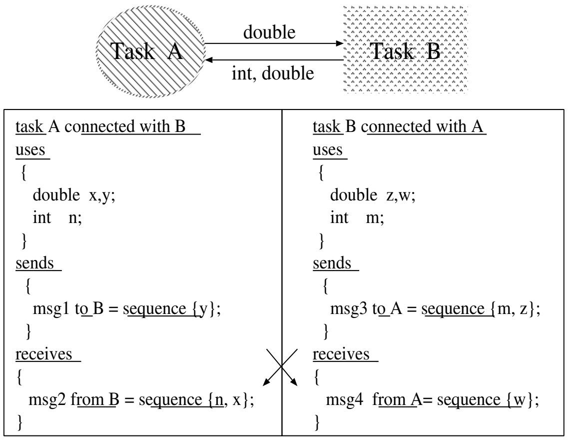

Figure 1: Example of a MeDLey specification

1.1 MeDLey's main features

So a MeDLey specification is the specification of the data and communication parts of a distributed application. That means this notation is mainly declarative, no control is provided.

In a MeDLey specification, a distributed application is split into tasks : each MeDLey module is the specification of a task. So, when specifying an application, the user must first define a set of tasks. After that, each task will be specified by giving the data it uses and the way it communicates with other tasks.

1.1.1 The task level

The application is the top-level entity in MeDLey. At the second level, an application is composed by a set of tasks.

A task specification in MeDLey mainly contains four parts as shows in figure 1.

The syntax remains very simple:

- The *connected with* section allows the user to map data coming from/towards tasks. This section is optional when the programming model of the application is *SPMD* (Single Program Multiple Data) and only needs one task definition. Of course, these data structures may also be used for computation purposes ;
- The *uses* section defines the data structures that will be used within the tasks to exchange data ;
- The *sends* part specifies the outgoing communications to other tasks using the previously declared data structures ;
- The *receives* section defines the incoming communications from other tasks into the data structures declared in the very first part.

1.2 Communication modes

The communication modes suggested for MeDLey are mainly inspired from message-passing paradigm programming. It is possible to indicate them with key words of this language when declaring messages in *sends* and *receives* blocks.

A send operation can be blocking or nonblocking. The call to a blocking send operation does not return until the message data has been safely stored away so that the sender is free to access and overwrite the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer. A nonblocking send can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver. On the other hand, a nonblocking send start call initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer.

Each of these two send modes could be synchronous or not. A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. An asynchronous send operation can be started whether or not a matching receive has been posted. If a send is executed and no matching receive is posted, MeDLey buffer the outgoing message, so as to allow the send call to complete.

On the other hand, there is only two receive operations : blocking and nonblocking. A call to receive blocking operation returns only after the receive buffer contains the newly received message. A nonblocking receive start call initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer.

1.3 Generating communication primitives

MeDLey specification can give birth to several implementations, when talking about communication primitives. First of all it could generate "Message Passing" like primitives, by generating MPI or PVM code : the main advantage would be that it ensures portability, thanks to MPI's or PVM's.

Secondly, MeDLey specifications could be used to generate more specific communication primitives, taking into account a particular communication layer (active messages, shared memory). This approach would of course forget portability to ensure best efficiency on a dedicated environment.

The experimentation part of the MeDLey language was carried out within the framework of a collaboration with the research laboratory in physics (LPMI) of the UHP-Nancy, France, and consisted in the use of a code of digital simulation used by the physicists [6] [5].

In this paper, we will present the extension part of this language proposed for the grid-decomposition methods, and the way approached here for parallelisation on parallel architectures with distributed memory.

2 Grid-decomposition methods

These methods fit parallel architectures based on distributed memory [1]. We split a domain (grid) into several sub-domains as many as the number of processes, and in each one, we perform calculations at the local level. The data on the borders are exchanged via communica-

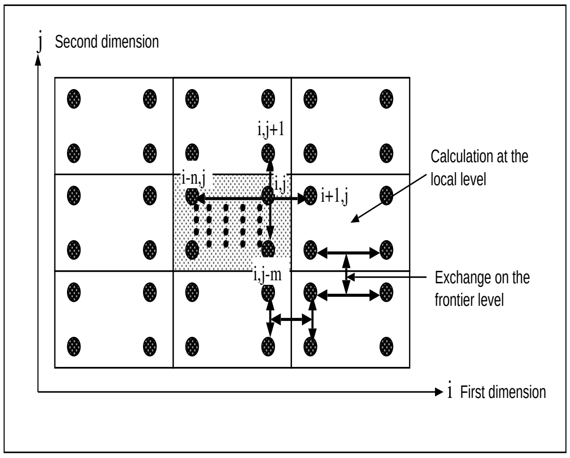

Figure 2: Grid-decomposition methods

tions by messages (see figure 2). The border size is much smaller than the size of the global domain.

A parallel version of the general algorithm, based on the principle of the grid-decomposition, is the following :

1. split the domain into as many of sub-domains as processes number at the execution time ;
2. reiterate :
   (a) exchange messages on borders ;
   (b) calculate.

2.1 MeDLey Extension

The use of MeDLey for the grid-decomposition methods is very interesting. There are indeed several communication libraries offering built-in functions which in particular make it possible to create a virtual grid of processes, to determine the neighbor processes, etc. Our extension is based on the MPI functions [7]. The syntax and the semantics of this extension were defined with the concern of being complete without being complex. This in order to allow people of different fields to use this unified notation while guaranteeing a certain level of performances.

The selected programming model is the SPMD (Single Program Multiple Dated) model. Only one code is applied on each sub-domains. There are many processes as sub-domains. For each sub-domain (or process), we need to know its neighbors. In the loop (see the iteration above), we will exchange the data with the neighbors on the borders and will compute inside each sub-domain.

Moreover, in the majority of applications using the principle of the grid-decomposition, the exchanges of the borders are performed with adjacent sub-domains. In such a situation, the user must first determine the neighbors of each sub-domain before launching the communication process. With our extension, MeDLey deals with the calculation of the neighbors of each sub-domain, the user only defines the content of the messages to be exchanged.

The adjacent neighbors of each sub-domain can be referred to by using the key word "Neighbor" followed by a combination to the following key words separated by underlined :

West or East : to determine the adjacent neighbors according to the first dimension;

North or South : to determine the adjacent neighbors according to the second dimension ;

Down or Up : to determine the adjacent neighbors according to the third dimension.

This technique makes it possible to specify only the adjacent neighbors. However, certain applications require exchanges with remoted neighbors, so we had to determine these neighbors. To reach this purpose, we propose two possibilities :

1. by using the key word "Neighbor" followed by indices presenting displacement in a number of steps in each dimension (see figure 3). The adjacent neighbors can be defined by this method. For example, Neighbor_West_North is equivalent to Neighbor[-1][1] ;

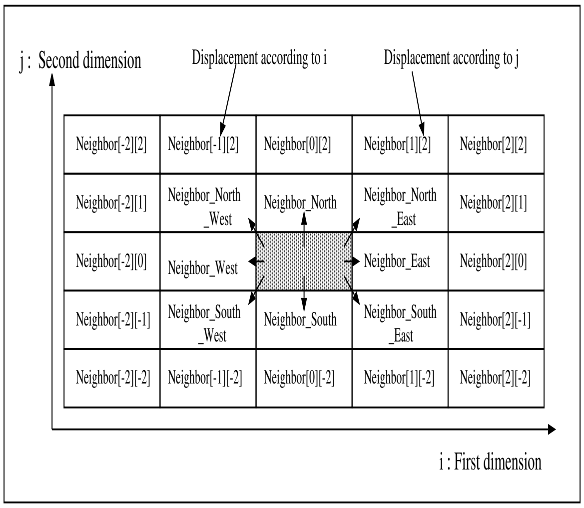

Figure 3: Specification of the neighbors in a bidimentionnel field

2. or the new MDL_Get_Neighbors function.

2.1.1 Suggested functions

In this paragraph, we briefly present the various functions suggested for the extension of MeDLey for the grid-decomposition methods.

MDL_Grid_Create : allows to build a grid of process and to determine the number of processes in each dimension ;

MDL_Grid_Rank : return the rank of the process associated with the local coordinates;

MDL_Get_Neighbors : allows to determine the neighbors of a process ;

MDL_Grid_Coords : return the coordinates of the process whose rank is specified in the call, (see figure 4) ;

MDL_Grid_Mycoords : it is an alternative of MDL_Grid_Mycoords which makes it possible to return the coordinates of the process which makes the call.

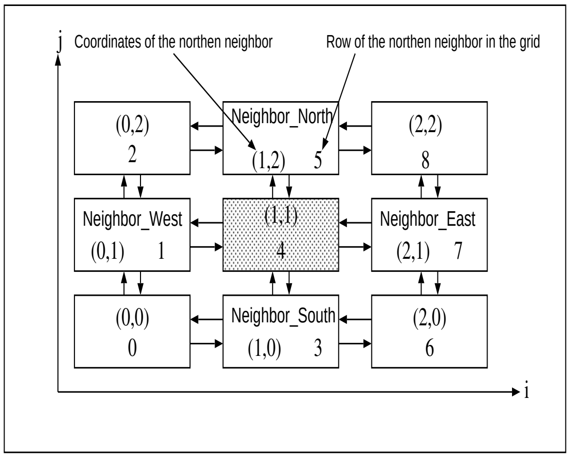

Figure 4: Correspondence between coordinates and the row of a process

2.1.2 Structure of a MeDLey specification

An example of the structure of a MeDLey specification using the grid-decomposition methods is the following :

```
Task  Node  connected  with grid(Ndim)
                       [of  Node]
  // Ndim : Number of dimensions,
  // of Node is optional

uses
 {
  // declaration of the data to be
  // used during communication
  // and computation

 int a,b;
 double c,d,e;
 }

sends
 {
  // declaration of messages to be
  // sent by using key words :
  //   to, sequence, Neighbor, East,
  //   West, North, South, Up, Down

 m1 to Neighbor_West=sequence{c,d,b};
 }
```

```
receives
 {
  // declaration of messages to be
  // received by using key words :
  //   from, sequence, Neighbor, East,
  //   West, North, South, Up, Down.
 m2 from Neighbor[2][3]=sequence{a,e};
 }
```

2.1.3 Generating communication primitives

The MeDLey compiler generates for each task a class in the target language (C++) which contains all the data to be sent and received as well as the communications methods. If we want to send the message *m1* of the task *Node*, described in the example below, we only need to perform a call, by using generated class for this task, to the following function: *MDL_SendTo_Node_m1()*.

3 Conclusion

Parallel programming is more complex than sequential case. Indeed, to write a parallel program, many tools are needed : language with explicit parallelism, tools of traces and visualisation, evaluation of performances, communication libraries, etc. One of the topics of the RESEDAS team, the new MeDLey language, is the component which allows the specification of the communications for distributed calculation.

In this paper, we have presented the extension part of this language related to the grid-decomposition methods. This programming model leads to a very natural parallelization and has the advantage of muching well the local memory use. In this framework, we have proposed structures and functions meeting the need of applications of these methods. Moreover, the use of the preset neighbors, proposed in this MeDLey extension, makes it possible to facilitate this programming model in fact that the user should worry only to define the contents of the messages to be exchanged.

References

[1] Brugeas (I.). – *Utilisation de MPI en décomposition de domaine.* – Rapport technique, IDRIS, 1996.

[2] Dillon (E.). – *MeDLey : User's guide.* – Rapport technique, CRIN-CNRS/INRIA-Lorraine, February 1997.

[3] Dillon (E.), Guyard (J.) et Wantz (G.). – Medley : An abstract approch to message passing. *PARA96 : Workshop on Applied Parallel Computing in Industrial Problems and Optimisation, Lynby, Denmark*, August 1996.

[4] Dillon (E.), Santos (C. Gamboa Dod) et Guyard (J.). – An environnement for quick design and efficient inplementation of message-passing applications. *HPCN Europe'97, Vienna, Austria*, 1997.

[5] Es-sqalli (T.), Dillon (E.), Bertrand (P.), Coulaud (O.), Sonnendrucker (E.) et Ghizzo (A.). – Parallelization of semi-lagrangian vlasov codes. *16th Conference on the Numerical Simulation of Plasmas, Santa-Barbara, Ca, USA*, February 1998.

[6] Es-sqalli (T.), Dillon (E.) et Guyard (J.). – Using medley to resolve the vlasov equation. *The 7th International Conference on High Performance Computing and Networking Europe, Amsterdam, The Netherlands*, April 1999.

[7] Forum (MPI). – *Extension to Message Passing Interface.* – NSF and ARPA, 1996.