Using MeDLey to resolve the Vlasov equation

Tawfik Es-Sqalli, Eric Dillon, Jacques Guyard

Using MeDLey to resolve the Vlasov equation

T.Es-sqalli, E.Dillon, J. Guyard
E-mail: {sqalli,dillon,guyard}@loria.fr
RESEDAS Project

LORIA, Scientific Campus B.P. 239
54506 VANDOEUVRE-Lès-NANCY CEDEX France

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 experimentation part of this language.
This work is completed within team RESEDAS (Computer networks and Distributed systems) with the LORIA & INRIA Lorraine within the framework of a collaboration with the CCH (Lorraine Center of Competence in Modeling and Calculation with High Performance).

1 The M*e*DL*ey* Language

This section gives a quick overview of the MeDLey language.

1.1 Motivations

The MeDLey project [Dillon 96] [Dillon 97] 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.

In the following sections, we will describe the syntax and semantics of the MeDLey notation.

1.2 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.

The task level The application is the top-level entity in MeDLey. At the second level, an application is made of tasks. A task specification in MeDLey mainly contains three parts :

1. a part that declares the data structures used within the tasks to exchange data with other tasks. MeDLey compiler is intended to use this part to find an optimal memory layout (based on communications) ;
2. a part that specifies the outgoing communications to other tasks, by defining inter-actions out of the previously declared data structures ;
3. a part that defines the mapping of incoming communications from other tasks.

Before going any further, here is the skeleton of a MeDLey specification :

```
task t1 [ connected with t2,t5 ]
uses
sends {...}
receives {...}
```

The syntax remains very simple :

- the `uses` section defines the data structures used within the task `t1` ;
- the `sends` and `receives` sections define the content of the data exchanged.

The `connected with` section allows the user to map data coming from/towards tasks `t2` and `t5`. This section is optional when the programming model of the application is *SPMD* (Single Program Multiple Data) and only needs one task definition.

1.3 Data Structures Declaration

The data part ("uses") of a MeDLey specification allows the user to declare the data structures he will use within each task of its application.

Each data structure declaration is denoted by an identifier. Such a declaration follows the syntax below :

```
<datatype> <identifier>;
```

MeDLey provides a set of basic data types that can be combined into more complex ones [IOS 90]. We have chosen to use a notation close to IDL's, augmented with some conventions and constructors dedicated to high performance computing.

So, more precisely, all data types declared in a MeDLey specification are pseudo-abstract, i.e. they have a machine independent semantics.

As a result, when distributed applications are to be run across heterogeneous architectures, the data will be marshalled to guarantee their coherence. Here again, such a conversion between internal representations must be handled by a MeDLey compiler during communication primitives. Finally, if an application is to be run across heterogeneous architectures, it is the responsibility of the compiler to warn against "unmanageable" errors occurences (converting a 64 bits integer to a 32 bits integer, e.g.).

1.4 The communication definition

When the data structures have been defined within the MeDLey specification, it is possible to define the communication in which they will be involved.

The *communication* part of a task should specify three components :

1. the data exchanged ;
2. the source/destination of the communications ;
3. the semantics that should be used for each communication.

1.5 The MeDLey inter-actions

To summarise, a MeDLey communication is a data movement between tasks. This means we have two kinds of inter-actions :

- the outgoing inter-actions : when the data are picked up from the local memory to be drooped into the local memory of one or many other tasks ;
- the incoming inter-actions : when the data dropped into the local memory are coming from the local memory of one or many other tasks.

An inter-action is identified (within a task) by its name. In the `sends` statement, for example, MeDLey allows to specify the outgoing interactions. The syntax is given below :

`ia [to t1]=` *interaction-content-definition*`;`

This syntax means that the interaction identified by "ia" will be an outgoing communication **to** task t1. Its content is defined by the local data structures according to the *interaction-content-definition*.

MeDLey provides two levels of semantics of an inter-action. On the top level, we define the *synchronism level* of an inter-action :

- synchronous : when tasks involved in the communication all synchronise before completing their inter-actions ;
- asynchronous : when tasks locally[1] complete their inter-action.

The inter-action content definition The content of an inter-action is based on the data structures defined in the `uses` statement. Indeed, MeDLey provides constructors to built the content of an interaction by calling the identifiers of the data declared. So, if you want to send values of an *integer i*, a *float f*, and a *double d* during an inter-action, you may write :

```
m to t1 =sequence(i,f,d);
```

provided `i,f,d` have been declared in the `uses` section with the correct types.

1.6 A simple M*e*DL*ey* specification

This section gives an example of the MeDLey syntax.

```
TASK example [CONNECTED WITH example]
USES {
        // The variables are declared here
        FLOAT a,b;
        VECTOR<SHORT>[100] v;
        MATRIX<FLOAT>[100,100] m;
        CHAR ch1, ch2;
    }
SENDS
  ASYNC {
         // The message contents are defined here.
         m1 [TO example] = sequence ( a,ch1);
         m2 == sequence(m[][4]);
        }
RECEIVES
  SYNC  {
         // The received messages and their mapping.
         m3 [FROM example] = sequence(b,ch2);
         m4 [FROM example] = sequence(v);
        }
```

[1] locally, literally means that the completion of the interaction only depends on the local executing task. Of course, in practice, the compiler may only be able to provide a "best-effort" locality, because of resources availability, resulting in a pseudo-asynchronism.

In this example, we see that a MeDLey specification consists of three main parts for each task definition. The first one, denoted by `uses`, defines and declares the data structures and the identifiers that will be used within this task. These identifiers will be available for the programmer in the target language. The second part defines all out going inter-actions. These definitions are done by combining values of the defined identifiers. All inter-action definitions will be used to guide the compiler to find the best memory layout in the target language. Finally, the third part defines how to map the incoming data into the local variables.

1.7 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 [Team 94] 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, or even only socket layer). This approach would of course forget portability to ensure best efficiency on a dedicated environment.

2 Experimentation of M*e*DL*ey* in point-to-point communication

In this section, we study the part of the MeDLey language concerning point-to-point communications. Using this language, we will implement the "Vlasov" code already existing in a parallel version. This work was carried out within the framework of our collaboration [Es-sqalli 98] with the research laboratory in physics (LPMI) in the UHP.

2.1 The Vlasov code

General information Plasmas of thermonuclear fusions are the seat of non-linear instabilities of hydrodynamic and kinetic origin [Ghizzo 93], evolving on different scales of time.

A better comprehension of these phenomena can be brought by the numerical simulation. This one, located halfway between the theory and the experiment, makes it possible either to validate or not a theory, or to describe more precisely the concerned mechanisms.

The Eulerian Vlasov model is adapted to the study kinetic instabilities. It consists in solving the Vlasov equation which describes the evolution of the dynamics of particles. It is known as "Eulerian" in opposition to the "Lagrangian" model (code PIC, particle-in-cell) which solves equations of the movement.

Existing code A parallel version of the Vlasov code was already developed by physicists of LPMI, this version is based on the use of calls to the MPI communication library.

In the following paragraphs, we describe how to implement this code using the MeDLey language. Finally we will present our experimental results.

2.2 Implementation with MeDLey

The Vlasov code consists in a series of transformations applied to the starting matrix of distribution. The successive transformations make the values of this matrix converge until the difference between two iterations will be negligible.

The parallelisation of this code consists in distributing the starting matrix over the whole set of processors which will perform parallel and similar transformations on their local data. A gathering step is carried out after each series of transformations. This step is followed by a distribution if the stop condition is not satisfied.

An alternative parallelisation of this code consists in defining a task which deals with the distribution and gathering, and another task whose intances will perform in parallel the transformations on the matrix of distribution. In this case, the first task will be inactive during the transformation phase. To solve this problem, we can define only one task, a particular instance of this task deals of the distribution and the gathering. This instance together with the others of the same task will carry out in parallel the transformations on the matrix of distribution.

To use MeDLey in such a case, we set the interconnection model to the SPMD model. After that, we define the data used by this task, we finally specify the messages to be sent and received.

The task specification In this paragraph, we specify the communications required by the Vlasov task. The data used in communication phase are initially declared in the **use** part, followed by the **sends** part where we specify the messages to be sent, finally the **receives** part which contains the specification of messages to be received.

```
Task  Vlasov(i)  connected with setof (Vlasov)
  uses
  {
    // The variables are declared here
    vector<double>[NXPROC] Rhops;
    vector<double>[NXPROC] Work;
    vector<double>[NXPROC] Ekp;
    vector<double>[NXPROC] Xwork;
    matrix<double> [NXPROC,NVXPROC] fWork;
    vector<double>[N] Ef;
  }
```

```
sends
{
  // Structures sent in functions : engy, density
    Srhop = sequence(Rhop);
    SEkp = sequence(Ekp);
  // Structures sent in functions : transf, itransf
    Sfwork = sequence(fWork);
  // Structures sent in functions : transf, density
    Set to Vlasov = sequence(Ef);
}
receives
{
  // structures received in the engy function
    RWork = sequence(Work) matches SRhop;
    RXwork = sequence(Xwork) matches SEkp;
  // structures received in the transf function
    RfWork = sequence(fWork) matches SfWork;
  // structures received in the density function
    REf from Vlasov = sequence(Ef) matches SEf;
}
```

Experimental results In this paragraph, we give the various numerical results concerning the execution, using MeDLey, of the parallel version of the Vlasov code. These results were collected on a Power Challenge array (Silicon Graphics R10000) with 18 processors.

We carried out experiments on this code according to size (N) of the initial matrix of distribution, the number of processors (NPROC), and a version of the MeDLey implementation uses the communication primitives of MPI library (MDL (MPI)), and another version using a management primitives of shared memory (MDL (SHM)).

The various numerical results of this experimentation are given in table 1.

Observations

From the implementation point of view

- On the speed-up level :

  * Independently from the size of the distribution matrix, we get an average speed-up of 1.9 when the number of processors increases from 4 to 8, and of 1.8 in the case of passage of the number of processors from 8 to 16. These values show that by increasing the number of processors, we have a linear speed-up (see figure 1). So this code efficiently uses the computation capacities of the processors added each time.

tmax = 10 **N** = Size of matrix of distribution
dt = 0.25 **NPROC** = Number of processors
MDL (MPI) : version of the MeDLey implementation using the communication primitives of the MPI library
MDL (SHM) : version of the MeDLey implementation using the communication primitives of shared memory (systemV)
Computing time : execution time of the Vlasov code without calling communication primitives

Case 1 : NPROC = 4

N	256	512	1024	2048
MDL (MPI)	3.901	15.361	67.597	367.048
MDL (SHM)	4.048	15.952	68.401	377.062
Computing Time	3.751	14.668	63.398	349.664

Case 2 : NPROC = 8

N	256	512	1024	2048
MDL (MPI)	2.062	7.986	32.021	269.331
MDL (SHM)	2.173	8.225	33.932	211.876
Computing Time	1.964	7.506	30.175	175.304

Case 3 : NPROC = 16

N	256	512	1024	2048
MDL (MPI)	1.296	4.489	16.138	78.733
MDL (SHM)	?	?	?	?
Computing Time	1.095	4.385	15.084	66.805

Table 1. Execution time (in second) of the Vlasov code according to N, NPROC and the version of the MeDLey implementation on Power Challenge Silicon Graphics

* Apart from the number of processors, we reach an average speed-up of 3.8 when the size of the distribution matrix changes from 512 to 256, of 4.5 in the case (1024 to 512) and of 6 in the case (2040 to 1024). These results are illustrated by figure 2 which shows that the size of the matrix of distribution implies an exponential evolution of execution time.

– Considering the comparison of the execution time between the MDL(MPI) and the MDL(SHM) versions :
It is noticed that the execution time of the Vlasov code of the MDL(MPI) version constitutes 98% of the MDL(SHM) version. It's primarily due to the fact that among designers of MPI library there are specialists and system engineers who have a very thorough knowledge of the hardware. So the functions of this library allow a more efficient use of the available

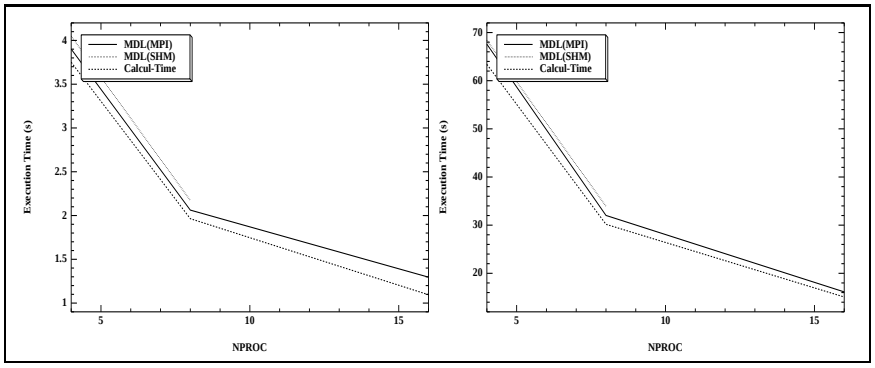

Fig. 1. Execution time of the Vlasov code according to NPROC and the MeDLey implementation version for N=256 (in the left) and N=1024 (in the right)

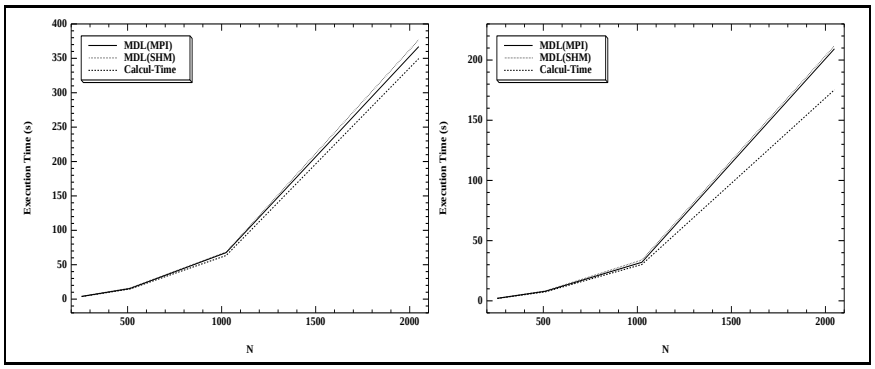

Fig. 2. Execution time of the Vlasov code according to size of matrix of distribution (N) and the MeDLey implementation version for NPROC=4 (in the left) and NPROC=8 (in the right)

hardware. Consequently, it is difficult for a tool in its first version (MDL (SHM)) to reach the performances of this library.

From the formalism point of view

- Facility of the use of MeDLey :
  The implementation of the Vlasov code by using the MeDLey language showed the facility of the use of this new tool. Indeed, after the declaration of the tasks, the MeDLey compiler generates for each task a class which contains the data sent and received as well as the methods necessary to initialise and terminate the communications. However, the user only calls these functions at the right place and time.

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.

The goal of this work consists in experimenting this language aiming to validate the existing concepts. It was carried out within the framework of a collaboration with the research laboratory in physics (LPMI) of the UHP, and consisted in the establishment of a code of digital simulation used by the physicists.

Our experimental part is partial, but a more wider experimental investigation is needed. In the future, we will interested to examine MeDLey with respect MPI or PVM implementation, and a sequential one in order to improve its communication performance.

References

[Dillon 96] E. Dillon, J. Guyard and G. Wantz. Medley : An abstract approch to message passing. *PARA96 : Workshop on Applied Parallel Computing in Industrial Problems and Optimisation, Lynby, Denmark*, August 1996.

[Dillon 97] E. Dillon. Medley : User's guide. Technical report, CRIN-CNRS/INRIA-Lorraine, February 1997.

[Es-sqalli 98] T. Es-sqalli, E. Dillon, P. Bertrand, O. Coulaud, E. Sonnendrucker and A. Ghizzo. Parallelization of semi-lagrangian vlasov codes. *16th Conference on the Numerical Simulation of Plasmas, Santa-Barbara, Ca, USA*, February 1998.

[Ghizzo 93] A. Ghizzo. *Application à la modélisation des interactions laser-plasmas et de l'équation gyrocinétique de Vlasov.* PhD thesis, UHP Nancy, LPMI, Juin 1993.

[IOS 90] IOS. *Specification of abstract syntax notation one (asn.1).* Information Processign Systems - Open System Interconnection, 1990.

[Team 94] MPI Team. *PVM 3 users's guide and reference manual.* ORNL/TM-12187, Oak Ridge National Laboratory, Septembre 1994.