Experimentation and Extension of a Specification Language of the Communications for Distributed Computation

Tawfik Es-Sqalli, Eric Dillon, Jacques Guyard

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Experimentation and extension of a specification language of the communications for distributed computation

Tawfik Es-sqalli, Eric Dillon, Jacques Guyard
{sqalli,dillon,guyard}@loria.fr

Projet RESEDAS

No 3455

Juillet 1998

THÈME 1

ISSN 0249-6399

Experimentation and extension of a specification language of the communications for distributed computation

Tawfik Es-sqalli, Eric Dillon, Jacques Guyard
{sqalli,dillon,guyard}@loria.fr
Projet RESEDAS

Thème 1 — Outils logiciels pour les télécommunications et les systèmes distribués
Projet RESEDAS

Rapport de recherche n ° 3455 — Juillet 1998 — 36 pages

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, which ensure data exchanges between parallel tasks of a same application. 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 the users an easier parallelism programming based on communications using Message Passing.

MeDLey is an attempt to provide an abstract language to specify the communications of a distributed application independently from any current underlying communication layer. Based on this specification, a MeDLey compiler will generate several levels of implementation for all communication primitives : a first implementation based on PVM or MPI code, and a second one, more specific, more efficient, directly implemented on top of a network layer, or a more specific communication layer (active messages, shared memory, e.g.) to bypass the overhead introduced by message passing libraries.

As for any data-processing tool, the experimentation of the MeDLey language was of obvious need, therefore the topic of this work whose aim initially is to experiment this language on real example, then to propose some extensions which prove to be necessary in the applications based on communications by Message Passing. This work is completed within team RESEDAS (Computer networks and Distributed systems) with the CRIN-CNRS & INRIA Lorraine within the framework of a collaboration with the Center Charles Hermite (Lorraine Center of Competence in Modeling and Calculation with High Performance).

In this paper, we will first overview the basics of the MeDLey syntax and semantics, before talking about the experimentation and extension parts of this language.

Key-words: parallelism, Message Passing, communication libraries, experimentation, extension, MeDLey, language

(Résumé : tsvp)

CCH

Unité de recherche INRIA Lorraine
Technopôle de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY (France)
Téléphone : 03 83 59 30 30 - International : +33 3 3 83 59 30 30
Télécopie : 03 83 27 83 19 - International : +33 3 83 27 83 19
Antenne de Metz, technopôle de Metz 2000, 4 rue Marconi, 55070 METZ

Expérimentation et extension d'un langage de spécification des communications pour le calcul distribué

Résumé :

Le parallélisme explicite repose souvent sur le principe de la transmission des messages entre processus. Cependant, comme les stations de travail ne sont pas prévues à l'origine pour gérer ce type de communication, il est nécessaire d'utiliser des bibliothèques de communication, dites par passage de messages, qui assurent les échanges de données entre les tâches parallèles d'une même application. Actuellement, ces bibliothèques de communication sont de plus en plus nombreuses, et leur utilisation est devenue de plus en plus complexe et fastidieuse. Afin de pallier ce problème, un nouveau langage appelé MeDLey, a été développé au sein de l'équipe RESEDAS, dont le but est de permettre aux utilisateurs une programmation plus aisée du parallélisme reposant sur le principe de la communication par passage de message.

MeDLey est une tentative de fournir un langage abstrait pour spécifier les communications d'une application distribuée indépendamment de la couche de communication utilisée. Basé sur cette spécification, le compilateur de MeDLey génére plusieurs niveaux d'implantation pour tous les primitives de communication : une première implantation basée sur le code de PVM ou de MPI, et une deuxième, plus spécifique, plus efficace, directement mise en application sur une couche réseau, ou une couche plus spécifique de communication (messages actifs, mémoire partagée, par exemple).

Comme pour tout outil informatique, l'expérimentation du langage MeDLey était de nécessité évidente, d'où le thème de ce travail qui vise en premier lieu, à expérimenter ce langage sur des exemples réels, ensuite à proposer quelques extensions qui s'avèrent nécessaires dans les applications reposant sur le principe de la communication par passage de message. Ce travail est réalisé au sein du groupe RESEDAS (Réseaux d'ordinateurs et systèmes distribués) au CRIN-CNRS & INRIA Lorraine dans le cadre d'une collaboration avec le Centre Charles Hermite (Centre Lorrain de Compétence en Modélisation et Calcul à Haute Performance).

Ce rapport est construit de la façon suivante : le premier chapitre présente les notins de base du langage MeDLey. Les autres chapitres constituent la partie expérimentation et extension de ce langage.

Mots-clé : parallélisme, communication par passage de messages, bibliothèques de communication, expérimentation, extension, MeDLey, langage

Chapter 1

The MeDLey Language

This section gives a quick overview of the MeDLey language.

1.1 Motivations

The MeDLey project 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, the last section will consider the development of a MeDLey compiler.

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.

1.2.1 The application level

MeDLey allows two kinds of distributed applications :

1. *opened applications*
2. *closed applications*

The first case is dedicated to the design of applications that may cooperate with another. For example, this case could be used when specifying client-server applications, where the server would be a distributed application that would accept connexions from clients, for instance. This case could also be used when specifying an application that, once started, could occasionally accept new tasks for tuning, monitoring or visualisation purposes.

The second case could apply more on something like jobs started in batch, the kind of jobs that have to compute until they reach the expected result or precision without any external action.

1.2.2 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 that will be used within the tasks to exchange data.
2. A part that specifies the outgoing communications to other tasks.
3. A part that defines the mapping of incoming communications from other tasks.

In the first part, the user must declare all the data structures that will be involved in communications with other tasks. In fact, a MeDLey compiler is intended to use this part to find an optimal memory layout (based on communications). Of course, these data structures may also be used for computation purposes.

The two other parts are dedicated to communications. Firstly, to specify the outgoing communications, by defining inter-actions out of the previously declared data structures, and secondly to define the mapping of incoming communications into the data structures declared in the very first part.

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 (as well as the `receive` section) is optional when the programming model of the application is *SPMD* and only needs one task definition.

Finally, since there are two kinds of applications, MeDLey provides two kinds of tasks, depending on the way they are included in an application :

- *internal* tasks : they are tasks that were started by another task of the application. This is the default kind of tasks when using "closed applications".

- *foreign* tasks : they are tasks that were born independently, i.e. not started by any other task of the application. Such a task can only be used within the specification of an "opened application".

Of course, from the specification point of view, defining a task as a "foreign" or "internal" task makes absolutely no difference, but from the implementation point of view, different communication primitives will have to be generated (see section 1.6.1 for details).

The next sections give details about the semantics of the three main parts of a task specification.

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.

In order to improve raw performances during communications, the memory layout of all data structures has to be optimal. Within this context, "optimal" means a memory layout that allows minimum efforts during communications (and so maximum efficiency). This will be the main job of a MeDLey compiler.

So, as a consequence all data structures involved in communications must be declared within this part, so that all communications will be built thanks to these data declaration. More clearly, a communication in a MeDLey specification can only be defined out of the data declarations made in this part.

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. 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. But when implementing and running an application specified with MeDLey, each datatype is mapped to a native data type via the appropriate language mapping. Indeed, those data types may be used for computations, and so, may need efficiency.

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" error 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.

With MeDLey, a communication may have different semantics, based on the way the data will be picked up at the sender's side and dropped at the receiver's side. More clearly, this means we cannot say that communications in MeDLey are only similar to Message Passing (e.g., since the various

semantics allowed by MeDLey do not always imply the Message Passing properties). Anyway, each communication involves a content that is moved from the sender's side to the receiver's. MeDLey only ensures that the data will be moved from one task to another according to some properties (see section 1.4.1 for more details), but it is not responsible for the way the data are actually transmitted. Here again, the implementation is left to the compiler's charge, in order to guarantee the best efficiency.

In the rest of the paper, all communications will be denoted as *inter-actions* between tasks. Such an inter-action will be completely defined by giving the three previous components.

1.4.1 The MeDLey inter-actions

To summarize, a MeDLey communication is a data movement between tasks. From the task point of view, a communication implies to two inter-actions : one on the sender's side and another on the receiver's side.

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 dropped 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.

As a consequence, if two tasks communicate with each other, they must both have matching inter-actions, i.e. the first one must have an *outgoing* inter-action matching an *incoming* inter-action of the other task.

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.*

As we saw, a MeDLey inter-action is defined by three components. The two first are source/destination and content. We will now focus on the last one, to define the semantics of an inter-action.

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

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

Finally, a MeDLey communication is well defined if there exist matching inter-actions. So, the inter-action matching is defined by :

1. the content,
2. the source/destination of the inter-actions,
3. the level of synchronism.

More clearly, assuming we have two inter-actions :

- i_1 : defined within task t_1 as an outgoing inter-action which destination is task t_2.

[1] locally, literally means that the completion of the interaction only depends on the local executing task.

- i_2 : defined within task t_2 as an incoming inter-action which source is task t_1.

We will say that i_1 and i_2 can match if they have the same content (i.e. if the data types of the data involved in the communication are the same on both sides) and the same level of synchronism.

1.4.2 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.

More clearly, when you define the content of a MeDLey inter-action, you do not mean "*I'm sending a integer value*" but "*I'm sending the value of n, which is an integer*". After that, depending on the semantics of the inter-action, the value of n will be buffered or won't be.

Symmetrically, on the receiver's side, when defining an incoming inter-action, you do not only mean that "*an incoming integer value may arrive*", but that "*the integer value of n will arrive*". Here again, depending on the semantics of the inter-action, the incoming value will be buffered on arrival, or won't be.

This particular semantics has been chosen in order to provide all information to a MeDLey compiler to allow full optimizations during transfers. Thanks to this, during all inter-actions, a compiler knows the address of the involved value, and so can avoid buffering (according to the inter-action's semantics). Moreover, since all inter-actions are defined out of the data declarations, it is possible to produce a memory layout of data declaration based on the definition of the inter-actions contents : the "message-based" memory layout.

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

This section gives an example of the MeDLey syntax.

```
//
//  Simple Specification with MeDLey
//
TASK example [CONNECTED WITH example]
USES {
     // The variables are declared here
     FLOAT a,b,c;
     SHORT s;
     VECTOR<SHORT>[50] v;
     INT aa,bb,cc;
     MATRIX<FLOAT>[100,100] m;
     CHAR ch;
     }
SENDS

  SYNC {
          // The message contents are defined here.
          m1 [TO example] = SEQUENCE ( a,s,ch );
```

```
          // by default all "=" operators will be considered as
          // "==" operators, provided that the message content is
          // contiguous.

          // We extract a vector from m
          m2 == VECTOR FROM m[][4];
       }
  ASYNC {
          // We extract values from m3, no data coherence.
          m3 = SET FROM m[n][2n+3] WITH (n=0,10);

          // A matrix sent by rows
          m4 == MATRIX m [BY ROWS];

          // a more complex message
          m5 = SEQUENCE (m1, aa, m4);
        }
RECEIVES

  SYNC  {
          // We define here the received  messages and their mapping
          //  into the local variables.
          // (Optional field in the SPMD case)
          m1 [FROM example];
          m2 [FROM example];
        }
  ASYNC {
          m3 [FROM example];
          m4 [FROM example] == MATRIX m [BY COLUMNS];
         // to transpose the matrix.
        }
```

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 (optional in the SPMD case) defines how to map the incoming data into the local variables.

1.6 Towards a MeDLey compiler

First of all, MeDLey is intended to be an abstract language designed to specify the "communications" part of a distributed application. Thanks to this specification, a MeDLey compiler should now be able to generate a set of efficient communication primitives in a target language.

Of course, a MeDLey specification does not include any kind of control operations, so that the algorithm of the application should still be written directly in the target language. Anyway, the use of MeDLey primitives ensures efficiency during all the communication parts of the application.

More clearly, the task of a MeDLey compiler stands in two points :

1. derivating an optimal layout in memory based on inter-actions definitions, for all data declarations.

2. generating efficient communication primitives, specific to the parallel application.

The next sections deal with these main issues. We will present the way the data layout could be handled by a MeDLey compiler, before talking about the generation of the communication primitives. But first of all, we will talk about some interesting features that should be included in a MeDLey compiler.

From now on, we will suppose we have specified the parallel tasks of a parallel application with MeDLey.

1.6.1 Compiling MeDLey specifications

Compiling MeDLey specification means feeding a MeDLey compiler with MeDLey specifications to get both an optimal memory layout for each task and communication primitives into a target language (currently C or C++).

Two approaches may be taken to compile a MeDLey specification :

- a distributed (or only separated) compilation (not yet implemented) : each task specification is compiled independently.

- a centralized compilation : the tasks are compiled together, i.e. a whole application can only be compiled.

The "distributed" approach allows to compile each task independently (as done in C, when first generating ".o" files before any final linking). This approach has some consequences, mainly when talking about *opened applications*. As C-language provides "include" files, a MeDLey compiler may need the set of "public" available inter-actions from an opened application to compile the client (to be able to perform semantics checkings, e.g. : "internal" and "foreign" tasks should help in this way). So, when specifying an opened application (i.e. including foreign tasks), all inter-actions coming-from/going-to a foreign task will be marked as public inter-actions. Moreover, all "internal" tasks connected with a "foreign" task (i.e. including inter-actions coming-from/going-to a foreign task) may need special primitives to *listen to* and *accept* the arrival of foreign task into the community of the opened application.

But if such a "distributed" compilation could be supported for all cases, it would lead to a drop of performances in some cases. More clearly, when compiling "closed applications", it could be very interesting to tune the implementation, using global policies rather than local policies only, for each tasks.

So, both kinds of compilations should be available : the distributed one to compile opened applications (client-server for instance), and the centralized one to compile closed applications, in order to get full efficiency.

Finally, if C/C++ is used as a target language, a MeDLey compiler should mainly generate two things per task :

1. a kind of "include" file, where all data declarations of the "uses" part of the specification, would have been re-ordered to make an optimal memory layout. This file could be directly included from the C/C++ program of the corresponding task.

2. a C/C++ library (.hh+.C file, e.g.), containing all communications primitives (plus some other mandatory primitives). This file should be compiled and linked to the program of the corresponding task.

Those two things are detailed in the following sections.

1.6.2 Derivating an optimal data layout in memory

Derivating an optimal memory layout means taking all data declarations, taking into account the way they will be used within inter-actions to re-order them so that all data are contiguous in memory.

This is possible because ANSI C does not make any choice for the implementation of identifiers, i.e. if two C-variables are successively declared in a C/C++ source code, they will be contiguous in memory after compilation.

Of course, some problems may arise because of alignment and heterogeneity. On the first case, most architectures often add padding bytes to guarantee alignment rules, so that even if two data structures are identical, they can be represented with a different number of bytes. Consider the following example :

```
struct foo
{
   char c;
    double d;
} aFooStruct;
int anotherInt;
int anInt;
```

if an inter-action is to send the sequence (anInt, aFooStruct)[2], without alignment rules, we may send :

```
  4 bytes :   anInt,
  1 byte  :   aFooStruct.c,
  8 bytes :   aFooStruct.d
= 13 bytes
```

But because of alignment rules, the structure in memory is more likely to be something of :

```
  4 bytes :   anInt,
  1 byte  :   aFooStcuct.c,
  3 bytes :   padding,
  8 bytes :   aFooStruct.d
= 16 bytes
```

or maybe,

```
  4 bytes :   anInt,
  4 bytes :   padding,
  1 byte  :   aFooStruct.c,
  3 bytes :   padding,
  8 bytes :   aFooStruct.d
= 20 bytes
```

In fact, the problem is that a structure has alignment of the most aligned member. If it is not the first member, extra padding bytes will be added. That means a MeDLey compiler has to make a choice :

- if the communication is to be sent through a slow network, bandwidth may be critical, and so, sending extra padding bytes would make no sense : in this case packing (buffering) seems better.

[2] This illustrates the work of a MeDLey compiler : since anInt and aFooStruct are to send sent together, they should be contiguous in memory.

- Conversely, if the communication is to send over a high speed network (mainly talk about high bandwidth), or through shared memory implementation, a canonical order using natural alignment would make sense, provided that are the same in all communicating tasks, which may require global policies for task compilation[3].

Finally, a MeDLey compiler should use two main policies :

1. local policies : to re-arrange the order of the data declarations according to the inter-action definitions.
2. global policies : to perform fine tuning between tasks, taking into account the architecture of the runtime environment (alignment rules, heterogeneous architectures, ...)

1.6.3 Generating communication primitives

Here again, a 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, or even only socket layer). This approach would of course forget portability to ensure best efficiency on a dedicated environment.

Both approaches are detailed below.

Generating message passing implementations

The very first aim of MeDLey was to simplify the use of all kinds of communication libraries without sacrifying efficiency.

But currently Message Passing libraries are more and more sophisticated and so error prone. Using MeDLey could avoid all kinds of packing/unpacking operations with PVM or MPI_types construction, by leaving it all on the compiler's charge.

Moreover, this "Message Passing" level of implementation should provide a way to immediately use and debug an application, since a lot of tuning and trace-tools are available with both MPI and PVM. But as a consequence, the use of such "standardized " interfaces implies a certain amount of overhead due to their generic approach.

This leads to the second level of implementation detailed below.

Towards specific implementations

Such Message Passing libraries (MPI or PVM, e.g.) often use specific communication layers available on most MPP or Workstations, so the second level of implementation generated by a MeDLey compiler could bypass the library level to directly use these underlying layers.

Indeed, since all communications will be statically specified in the MeDLey specification, it will be possible to generate communication primitives dedicated to each particular application, to take advantage of the underlying architecture.

For example, you can imagine that a MeDLey specification is to be implemented on a network of Workstations using ATM. Of course, PVM already exists on AAL4/5, but if a MeDLey compiler generates explicit communications primitive directly on top of AAL4/5, the overhead introduced by PVM's portability could be avoided.

More clearly, this second level of implementation is the "raw" implementation of a distributed application.

[3] that's the reason why a MeDLey compilation cannot always be fully distributed

Finally, both levels of implementation lead to a three-step development process of a distributed application with MeDLey :

- firstly, identify the communications within the parallel application in a MeDLey specification,
- then generate the corresponding communication primitives into PVM-like code, in order to tune the computing algorithm, with all debugging tools available,
- and finally generate the specific communication primitive dedicated to the hardware environment, re-link with the application to get an efficient implementation.

So, without ensuring that the parallel algorithm will be really efficient, using MeDLey should provide an efficient implementation of all communications occuring between the cooperating tasks.

Chapter 2

Experimentation of MeDLey in point-to-point communication

In this chapter, 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 in FORTRAN. This work was carried out within the framework of our collaboration with the research laboratory in physics (LPMI) in the UHP.

2.1 The Vlasov code

2.1.1 General information

Plasmas of thermonuclear fusions are the seat of nonlinear 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 model of Vlasov 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.

2.1.2 Existing code

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

In the following paragraphs, we briefly present transformations and improvements made to the initial code. Thereafter, we describe how to implement this code using the MeDLey language. Finally we will present our experimental results.

2.2 Preliminary steps

Two steps were carried out on the parallel version of the Vlasov code :

- first, the translation from FORTRAN into C ;
- second, the optimization.

2.2.1 Translation step

The initial Vlasov code was written in FORTRAN. However, the only target languages available with the MeDLey compiler were C and C++. Therefore, the first step of this experimentation was a rewriting phase which consisted in translating the Fortran program into C.

2.2.2 Optimization step

The use of parallelism can improve the speed up execution of algorithms. But before parallelizing a code, and for a more effective use of this technique, the sequential execution time of this code must be optimal.

Many methods can be applied to optimize the execution time of codes. We give here a list of some techniques which have been used to optimize the initial version of the Vlasov code :

Elimination of common under-expressions : frequently, the address of data in a table is evaluated several times. It is possible to preserve this address in order to avoid its computation.

Elimination of useless code : certain parts of code are generated but cannot be reached. It is not necessary to preserve them.

Displacement of code : the calculations carried out in a loop which are independent of this one can be computed out of this loop.

Induction variables : calculations on induction variables can be transformed in order to use less expensive operations (in time).

Reduction of force : as well as the induction variables, it is possible to transform the arithmetic expressions into less time consuming ones.

Direct transformations of loops : various types of transformations can be classified according to the modification type which they perform. Usually, we consider transformations of execution order, transformations related to memory access, and some other transformations like fusion of two or several loops into a single loop, and bursting of a loop in several.

Thanks to these techniques, we optimized the Vlasov code execution time by a factor of 15%.

2.3 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 parallelization 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 parallelization 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 (Single Program Multiple Data) model. After that, we define the data used by this task, we finally specify the messages to be sent and received.

2.3.1 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 **uses** 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
  {
       // Data used in engu.c, density.c
    vector<double>[NXPROC] Rhops;
    vector<double>[NXPROC] Work;
       // Data used in engu.c
    vector<double>[NXPROC] Ekp;
    vector<double>[NXPROC] Xwork;
       // Data used in transf.c, itransf.c
    matrix<double> [NXPROC,NVXPROC] fWork;
       // Data used in density.c
    vector<double>[N] Ef;
  }
  sends
  {
    // Structures sent in functions : engy, density
      Srhop = sequenceRhop;
      SEkp = sequenceEkp;
    // Structures sent in functions : transf, itransf
      Sfwork = sequencefWork;
    // Structures sent in functions : transf, density
      Set to Vlasov = sequenceEf;
  }
  receives
  {
    // structures received in the engy function
      RWork = sequenceWorkmatches SRhop;
      RXwork = sequenceXwork matches SEkp;
    // structures received in the transf function
      RfWork = sequencefWork matches SfWork;
    // structures received in the density function
      REf from Vlasov = sequenceEf matches SEf;
  }
```

2.3.2 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 the communication primitives of shared memory (MDL (SHM)).

The various numerical results of this experimentation are given in table 2.1.

tmax = 10 **N** = Size of initial 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 (system V)
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 2.1: Execution time (in second) of the Vlasov code according to N, NPROC and the version of the MeDLey implementation on Power Challenge Silicon Graphics

2.3.3 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 2.1). So this code efficiently uses the computation capacities of the processors added each time.

  * 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 2.2 which shows that the size of the matrix of distribution implies an exponential evolution of execution time.

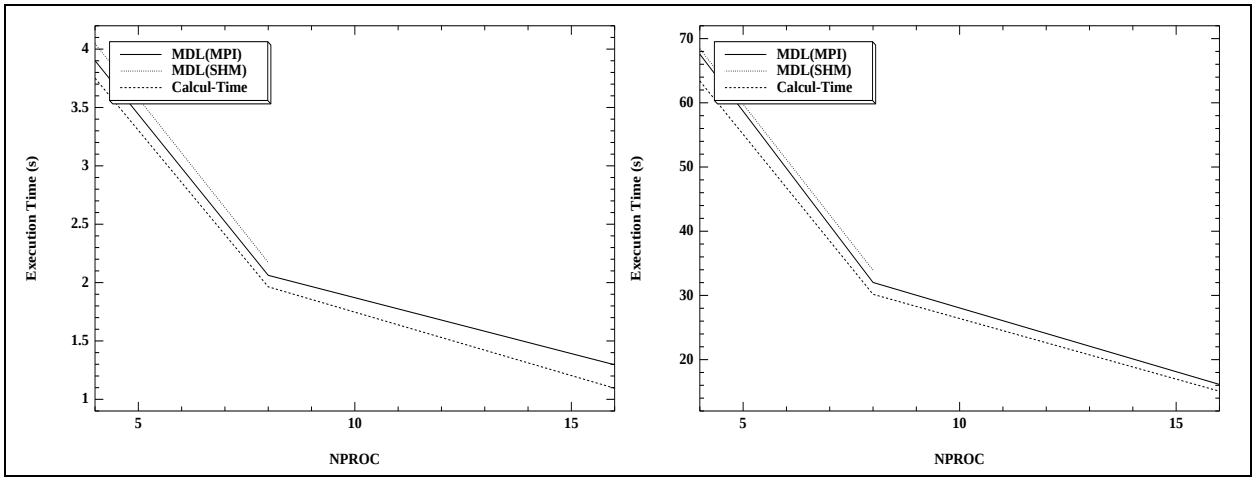

Figure 2.1: Execution time of the Vlasov code according to NPROC and the MeDLey implementation version for N = 256 (in the left) and N = 1024 (in the right)

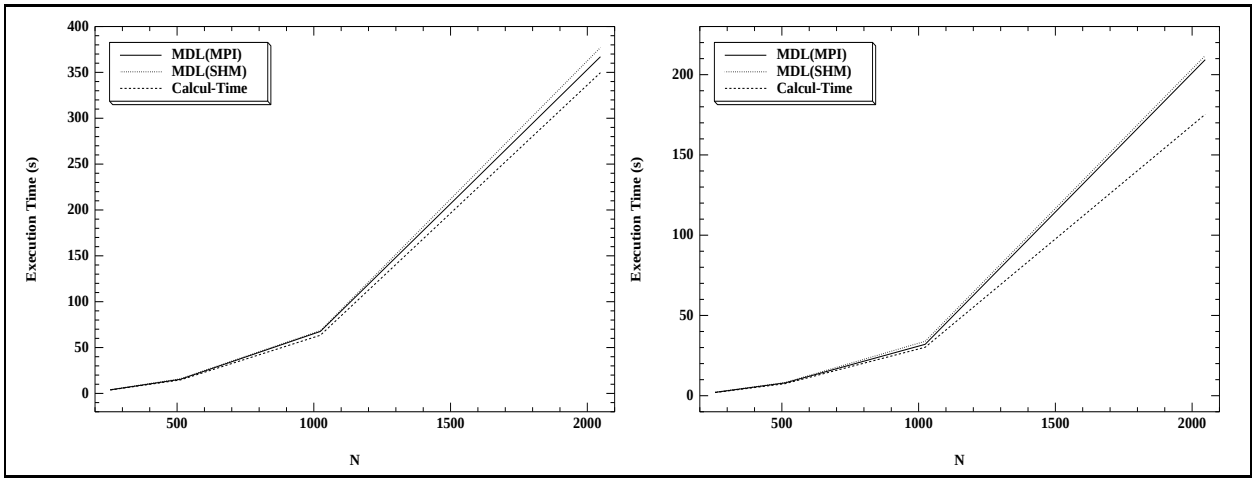

Figure 2.2: Execution time of the Vlasov code according to size of matrix of distribution (N) and the MeDLey implementation version for NPROC = 4 (in the left) and NPROC = 8 (in the right)

- 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 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

- Amalgam in the definition of sending/reception modes :
  The MeDLey language considers that the synchronous sending or reception is a blocking process which blocks the task until the complete send (respectively reception of the message), whereas the asynchronous send and reception are not blocking.

  The amalgam in this definition consists in the fact that a send can be blocking or not blocking, and each of these two modes can be synchronous or asynchronous. Concerning the reception, it is always asynchronous, and it can be blocking or not blocking. This definition is that used in the majority of the communication libraries including the standard MPI.
- 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 initialize and terminate the communications. However, the user only calls these functions at the right place and time.

2.4 Conclusion

Initial Vlasov code was scalable. With MeDLey, we could keep this characteristic by obtaining a significant acceleration factor. It is in fact a proof of the best use of the processor resources. Moreover, the MeDLey approach can be easily used and makes it particularly possible to simplify the calls to the communication primitives.

Conversely, this experimentation enabled us to discover the flaws of this new formalism such as the implementation version with shared memory which seems to need optimizing and correcting especially if a great number of processors is used.

Chapter 3

Extension of MeDLey in collective communications

3.1 Introduction

Contrary to the point-to-point operations which allow two processes to exchange data, collective operations implement a communication shared by more than two processes. These operations can be classified in three categories :

- data transfer operations ;
- synchronization operations ;
- collective calculations.

In the following paragraphs, we will present features offered by MeDLey in each one of these categories. We describe then the extensions of MeDLey suggested in this field. Finally, we give an example of application where the new functions are used.

3.2 Collective operations of data transfer

This first category gathers the operations which transfer data between several processes. These operations are : broadcast or one to all, scattering, gathering and transposition (all to all or multi-scattering). In the following, we will present by schemas the principle of these operations (see figure 3.1).

3.2.1 Existing functions

MeDLey provides two collective functions of data transfer [Dillon 97] :

- Diffusion operation makes it possible to diffuse a message to the whole set of task instances. Here is an example : `m1 to sun = sequence(.....);`

- The multi-diffusion operation makes it possible to diffuse a message over a subset of the instances of a task which must be specified at the time of the declaration as for the following example : `m1 to sun{1,3,..) = sequence(.....);`

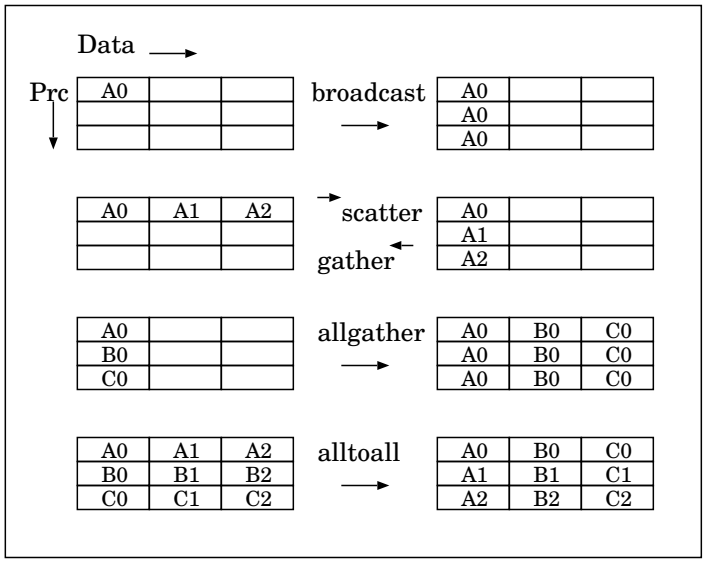

Figure 3.1: The collective operations of data transfer in group of three processes

3.2.2 Suggested extension

Structure of the task

Our extension is based on the functions of the MPI communication library [Forum 96]. The selected programming model is the SPMD model (Single Program Multiple Data). The various operations are performed between the instances of the same task.

To give a more particular aspect to these operations, we propose that the declaration of their messages be done in a new block called **collective**. Each set of operations of the same type must be declared in a sub-block corresponding to this type :

```
Task example [CONNECTED WITH example]
uses

// declaration of the data to be used during the communication and calculation

sends

// declaration of the messages to be sent

receives

// declaration of the messages to be received

collective // declaration of messages corresponding to the collective operations
{
Bcast    // for broadcast operation
    {
         // declaration of messages to be diffused
    }
    // declarations of other used operations : Scatter, Gather, Alltoall...
}
```

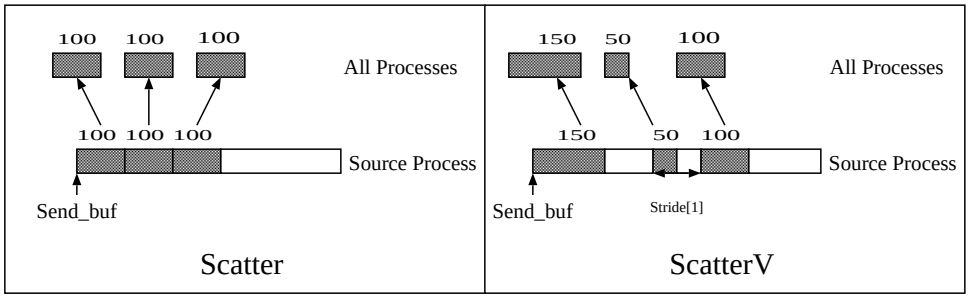

Figure 3.2: Difference between Scatter and ScatterV

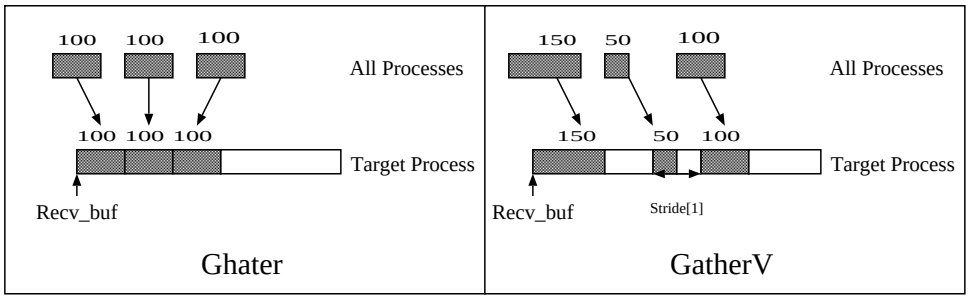

Figure 3.3: Difference between Gather and GatherV

After the phase of specification, MeDLey generates for each message a function of name MDL followed by the type of its operation and the name of this message. For an example, MeDLey generates for a message of the name "mes" declared in a Bcast sub-block a function of the name MDL_Bcast_mes(..).

Suggested operations

In this paragraph, we briefly present the various operations suggested for the extension of MeDLey in collective communications.

Bcast : allows to diffuse a message from a source instance towards the set of task instances. An alternative to this operation is **Mcast** which makes it possible to diffuse a message on a subset of task instances.

Scatter : allows to split data coming from the instance source out of N segments of a same size, N being the number of the instances at the execution time, the i^{th} segment is sent with the i^{th} instance. ScatterV is an alternative to Scatter where segments to be distributed can have different sizes (see figure 3.2).

Gather : it is the opposite operation of Scatter, it makes it possible to gather segments of the same size, each one coming from an instance of the task, and to put them in the reception buffer of the destination instance (root). These segments will be concatent according to the ranks of their source instances. An alternative to this operation is GatherV which makes it possible to gather segments of different sizes and to put them in different displacements compared to the beginning of the reception buffer (see figure 3.3).

Allgather : it is an alternative to the Gather function where all the instances of the task receive the result of the gathering, the segment of data sent by the i^{th} instance will be received by all instances and will be placed in the i^{th} block in the reception buffer. An extension of this operation is Alltoall ; any instance sends distinct data to each other. The j^{th} block sent by the i^{th} instance will be received by the instance of rank J and will be placed in the i^{th} block.

3.3 Synchronization operations

This second category gathers the operations which make it possible to synchronize processes. The **barrier**, already defined in the MeDLey language is the basic synchronization function and allows to block all instances of the same task until each of these instances reach this operation.

The use of this function is done by the call of MDL_BARRIER. In this extension we propose a new alternative of this function, called MDL_SBARRIER, which makes it possible to block only one subset of task instances. The ranks of these instances must be supplied in argument during the call to this function.

In the following, an example of use of this new function :

```
# include "node.h"       // the task ``node'' as an example
# include "nodeMDL.h"    // the class presenting the task ``node''
struct node *vdata;
int  *S_ens;              // contains the ranks of the instances which must be blocked
.........
Vdata.MDL_SBARRIER(S_ens);// for example, if we work in a master/slave context and
//we want to synchronize the slaves instances, we must specify theirs ranks in S_ens
```

3.4 Collective calculations

3.4.1 General informations

This last category includes the functions intended to perform calculations on data provided by all the processes to obtain a single result.

Each process performs this reduction function by supplying a value. The reduction then applies an operator to this set of values to obtain only one value result which is intended for one or the set of sources processes. In general, the reduction function can handle vectors directly where the reduction is made by elements of the same rank.

3.4.2 Extension suggested

Currently, the MeDLey language does not provide any function for collective calculation. To make up for this gap, we propose two new operations : **Reduce** and **Allreduce**. As well as for the collective operations of data transfer, each set of reduction messages using one of these two operations must be declared in a sub-block of collective.

The syntax and the semantics of these two operations are the following :

- **Reduce** : allows to calculate the result of an operation having as arguments the data provided by all the instances of task. This result will be made available on only one instance destination called root. This operation makes it possible to use several operations among which we can quote maximum (MAX), minimum (MIN) or sums (SUM).
- **Allreduce** : it is an extension of the Reduce operation where the computation is made available to all instances.

3.5 Application : the N-body problem

3.5.1 General information

Many problems of numerical simulation require the computation of the interaction force between a great number of particles or objects. If the force between the particles is completely described by the

addition of the forces between all the pairs of the particles, and the forces between each pair react along the distance between its particles ; such a problem is called the N-body problem [Gropp 94]. In the following paragraphs, we suppose that the interaction force between two particles is not symmetrical, that means that the calculation of the interaction force between two particles requires the calculation of this force in both directions.

As we will see, this problem can be efficiently solved in a multiprocessor environment. That relies on the distribution of the particles over the various processes. Each of these processes is occupied on the one hand by calculating the interaction force between the particles it has, and on the other hand by calculating those held by the other processes ; this makes it possible to distribute calculation efficiently.

3.5.2 Implementation with MeDLey

To implement the N-body code, we need to know, if, in the beginning, the particles are distributed to the same number between the various processes or not. In the following, we will propose solutions for the second case.

Moreover, the calculation of the interaction force between the set of particles is done in two steps. During the first one, each process calculates the interaction force between its particles (calculation 1). The second step consists, for each process, in calculating the interaction force between its particles and those of other processes (calculation 2). That requires, after the first step, to exchange the particles between the various processes.

After the second step of calculation, the various processes exchange the results of their calculations in order to be able to calculate the global interaction force. So the resolution of the N-body problem can be done in four phases :

- Phase 1 : calculation 1 ;
- Phase 2 : communication (exchange of the particles between the processes) ;
- Phase 3 : calculation 2 ;
- Phase 4 : communication (calculation of the sum of the interaction forces).

Let us recall that our goal by using the problem of N-body is not to detail calculations of the interaction force between the various particles, but to specify the communication messages needed during the resolution of this problem while using the new functions of MeDLey in collective communication. For this reason, we will particularly be interested in the second and the fourth phase.

First of all, it is necessary to declare the structure of a particle. The declaration of such a structure with MeDLey is done with the syntax used in C language.

Afterwards, we suppose that the structure of a particle, named Particle, is already defined. Each process uses an array of Particle of size MAX_PARTICLES, which is necessary to include in the block **uses** of the N-body task :

```
task N-body
uses
  {
     Particle     particles[MAX_PARTICLES];
     .....
  }
....
```

After the first calculation (phase 1), the various instances exchange the particles between them. That consists in gathering all the particles in the buffer particles of instance 0 (root). Then to diffuse

the set of these particles over all instances. The gathering step can be done in two ways according to whether the various instances have the same number of particles or not. If the number of the particles is the same on all instances, we use the Gather operation,we start by declaring its message :

```
task N-body
.......
collective
  {
   Gather(0) // we can write only Gather() since 0 are the default value
    {
        m1 = sequence {particles, particles[..]};
    }
  }
```

That will enable us thereafter to call the MDL_Gather function :

```
......
N-body *Vdata;
int   count;  // number of particles per instance
.......
MDL_Gather_m1(Vdata, , particles[0, (count-1)]);
........
```

Otherwise, if the number of the particles is not the same on all instances, we use the GatherV alternative of the Gather operation. But the root instance must as a preliminary know the number of particles available to each instance. That can be done by using the Gather operation. Both operations require the use of some parameters which we must declare in the **uses** block :

```
uses
{
......
int  *counts;  // contains the number of particles per instance
int  *displs;  // displacements relating to the beginning of the buffer of the root instance
int    count;  // the number of particles in the collective instance
}
.......
collective
{
  Gather(0) // we can write Gather() since 0 are the default value
   {
       m1 = sequence{counts,count};
   }
  GatherV(0) // we can write GatherV() since 0 are the default value
   {
       m2 = sequence{particles, particles, counts, displs};
   }
}
```

the use of these messages is done as follows :

```
......
```

```
N-body *Vdata;
........
int size;   // a number of the instances of N-body
size = Vdata.MDL_Sise();
counts = (int *)malloc(size * sizeof(int));
displs  = (int *)malloc(size * sizeof(int);
MDL_Gather_m1(Vdata);
if (Vdata.MDL_Myrang == 0)
  {
      displs[0]=0;
      for (i=1; i<size;i++)
         displs[i] = displs[i-1] + counts[i-1];
  }
MDL_GatherV_m2(Vdata);
........
```

Therefore, the instance of rank 0 has all particles, it remains to diffuse these particles to the other instances, that can be done by using the broadcast operation. Initially, we start by specifying the messages corresponding to this operation in the block collective :

```
Bcast(0)
{
        m3 = particles;
}
```

that makes it possible thereafter to call the MDL_Bcast function :

```
......
N-body *Vdata;
.....
MDL_Bcast_m3(Vdata);
```

The second calculation step consists, for each instance, in calculating the interaction force between its starting particles and those of which it has just been an owner. In our example, we used the buffer particles as emission buffer during the call to the MDL_Gather function, and at the same time as reception one during the call to the MDL_Bcast function. If all instances had at the beginning the same number of particles, there is no problem, and each instance can refer its old block of particles which corresponds to its rank during the execution. Otherwise, if the number of particles was not the same, all instances, except the root instance, cannot refer their starting particles since the contents of the buffer particles are crushed after the call of MDL_Bcast. A possible solution consists in using the MDL_Allgather function instead of the MDL_Gather function for the calculation of the table counts which contains the number of particles in each instance.

Let us notice that an alternative would consist in using the MDL_Allgather function instead of the MDL_gather function followed by MDL_Bcast. The MDL_AllgatherV function will be used if the number of particles at the beginning is not the same one for all the instances.

After the first communication phase, each instance has all the particles which makes it possible thereafter to begin the second calculation step (phase 3). At the end of this step, we calculate the sum of the interaction forces calculated by each instance. For all that, we use the reduce operation while starting by specifying his message :

```
uses
```

```
{
  ......
  double force;          // the interaction force calculated by each instance
  double force_total;  // the sum of the interaction forces
}
.......
collective
{
  ...
  reduce(0)
  {
     m5 = {force, force_total,1,MDL_SUM};
  }
}
```

the use of this message is done as following :

```
.......
N-body *Vdata;
........
MDL_Reduce_m5(Vdata);
..........
```

3.6 Conclusion

The various functions suggested in this chapter make it possible to meet the needs to perform a collective communication in a SPMD model, where the various operations performed between the instances of the same task. However, it remains to define the functions making it possible to various instances of several tasks to perform this type of communication.

Moreover, the base of the various functions suggested in this chapter is made up only by calls to some functions of MPI library. If we manage to facilitate the use of these functions, which is among the goals of the MeDLey language, we are not sure, at this time, to guarantee an optimal execution time of these new functions. However, each collective communication function can implement by using point-to-point communications. That means that if we manage to efficiently optimize the execution time of these operations, which is among the major assets of the MeDLey language, we can considerably improve the time of the collective operations basing oneself on calls of the point-to-point communication functions. The use of such a solution requires taking in to account some problems that can occur at the execution time like deadlock.

Chapter 4

Extension of MeDLey for the grid-decomposition methods

4.1 Introduction

The parallelization of codes using grid-decomposition methods gains a very great interest. After a short reminder of the principle of these methods, we will present the extension of MeDLey proposed in this domain and the way approached here for parallelization on parallel architectures with distributed memory. Finally, we give a sample program using these new functionalities.

4.2 Grid-decomposition methods

These methods fit parallel architectures based on distributed memory [Brugeas 96]. 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 communications by messages (see figure 4.1). 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 at the execution time ;
2. reiterate :

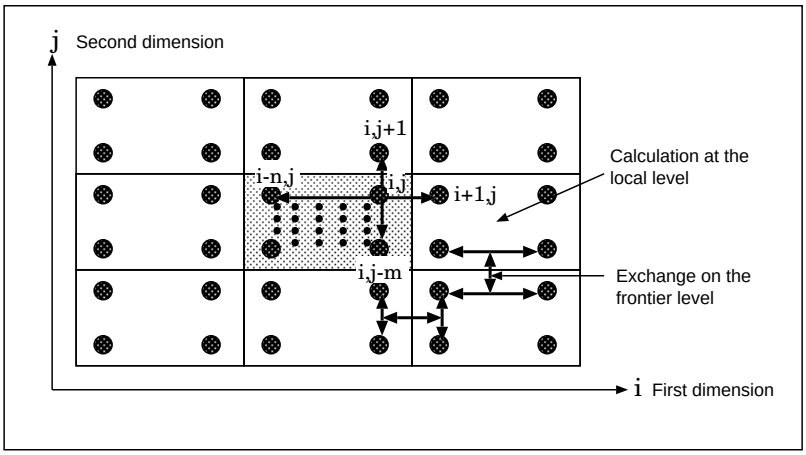

Figure 4.1: grid-decomposition methods

(a) exchange messages on borders ;

(b) calculate.

4.3 Extension of MeDLey

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 [Forum 96]. 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.

4.4 Structure of a MeDLey specification

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 4.2). The adjacent neighbors can be defined by this method. For example, Neighbor_West_North is equivalent to Neighbor[-1][1] ;
2. or by using the new MDL_Get_Neighbors function.

The structure of a MeDLey specification proposed for the grid-decomposition methods is the following :

```
Task  A  connected  with grid(Ndim) [ of  A  ] // Ndim = Number of dimensions,
                                                // by default = 2,
                                                // of A is optional
uses
 {
   ..... // declaration of the data to be used during the communication and computation
```

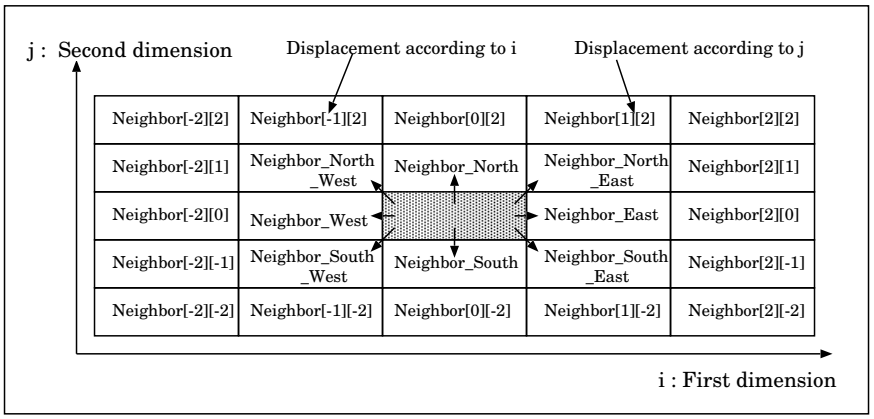

Figure 4.2: Specification of the neighbors in a bidimentionnel field

```
 }
sends
 {
   ..... // declaration of messages to be sent by using key words :
         //to, sequence, Neighbor, East, West, North, South, Up, Down
 }
receives
 {
   ..... // declaration of messages to be received by using key words :
         //from, sequence, Neighbor, East, West, North, South, Up, Down.
 }
```

4.4.1 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 ;

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.

4.5 Example of application

4.5.1 The Poisson problem

The Poisson problem is a differential equation expressed by the two following equations [Gropp 94] :

$$\begin{cases} \nabla_u^2 = f(x,y) & \text{in the interior.} \\ U(x,y) = g(x,y) & \text{in the terminals (margins).} \end{cases}$$

The goal of this problem is to find an approximation of U using F and G. To simplify our discussion, we choose as domain (U) a square of rank N+2 (N is the size of F). For this reason, we define a grid constituted by a set of the items (X_i, Y_j) such as :

$X_i = i/N + 1, i = 0, 1, ..., N + 1$. And $Y_j = j/N + 1, j = 0, 1, ..., N + 1$.

We use $U_{i,j}$ like abbreviation of U_{X_i,X_j}. The value 1/N+1 is often used : it will be noted h. We start by initializing U, then we give an approximation of U in each point of the grid by reiterating the following calculation :

$$U_{i,j}^{K+1} = 1/4 * (U_{i-1,j}^{K} + U_{i,j+1}^{K} + U_{i,j-1}^{K} + U_{i+1,j}^{K} - h * h * f_{i,j})$$

This process, named Jacobi iteration, is repeated until we reach a solution or well going beyond of a maximum number of iterations without reaching a solution.

4.5.2 Resolution with MeDLey

The resolution of this problem with MeDLey consists in dividing the domain into as many tasks as the number of process. Each task will on the one hand calculate the values of $U_{i,j}$ in the sub-domain in which it performs its computations, and on the other hand handle the exchange of borders with the close tasks.

Structure of the task node

For our example, we take 2 as number of dimensions. The structure of the task node (presenting a sub-domain) will be the following :

```
Task node(i)  connected with grid(2)// (2) is optional
uses
  {
   matrix<double>[N+2,N+2] U;
  }
sends
  {
   NeighborNorthS  =  sequence{U[..][..]} ;
   NeighborSouthS  =  sequence{U[..][..]} ;
   NeighborEastS   =  sequence{U[..][..]} ;
   NeighborWestS   =  sequence{U[..][..]} ;
  }
receives
  {
   NeighborNorthR  = sequence{U[..][..]} ;
   NeighborSouthR  = sequence{U[..][..]} ;
   NeighborEastR   = sequence{U[..][..]} ;
   NeighborWestR   = sequence{U[..][..]} ;
  }
```

4.5.3 Notations used

The domain is split in the direction of I and of J. We place the last points in the last sub-domains. Each process is located by its local coordinates and its rank in the grid. The exchange of the borders is done with the adjacent processes (see figure 4.3).

The used notations are the following :

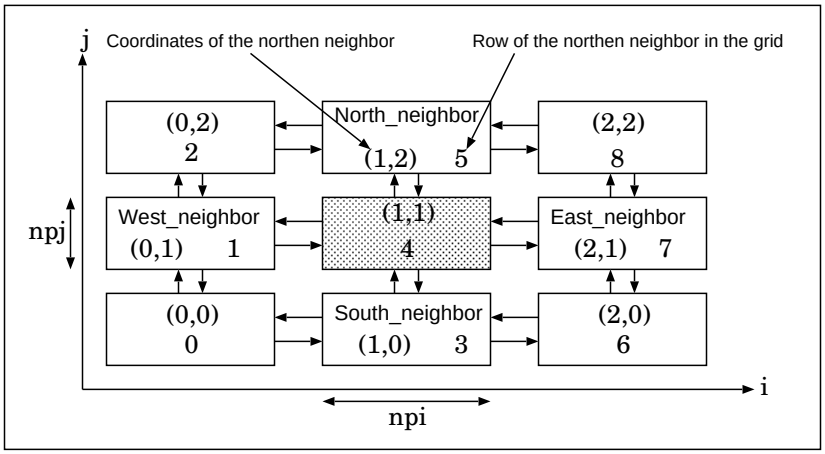

Figure 4.3: Exchange of borders between Neighbors processes

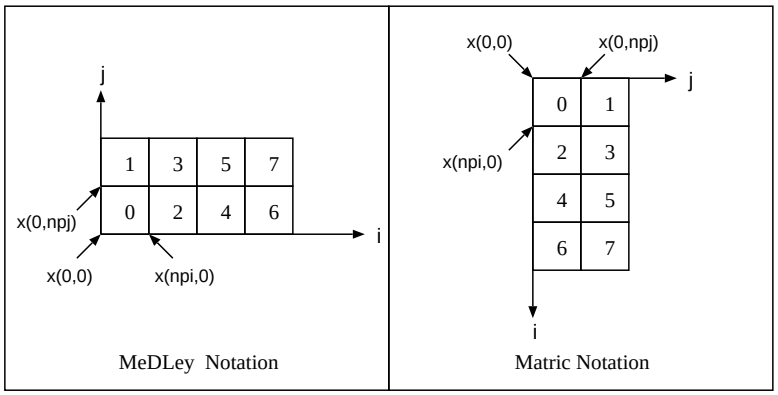

Figure 4.4: Difference between matric notation and that used in MeDLey

```
int Ntask;    // the number of tasks (intuitively the task instances).
int dims[2 ]; // the number of sub-domains in each dimension.
int npi,npj;  // the number of points of the sub-domain according to axes of i and j.
int INFi, SUPi, INFj, SUPj // terminals of each sub-domaine
int Max_nbr_iteration      // maximum number of the iterations
doubles Unew[N+2][N+2 ];   //contains the new values of U after each iteration.
```

In C, the two-dimensional arrays are stored in a contiguous way in memory according to the lines. With MeDLey, the same technique will be used except that it is necessary to make a rotation of the reference mark compared to that used in C (see figure 4.4).

Main Program

The schema of the main program used for the resolution of the Poisson problem using the new functionalities of MeDLey of grid-decomposition is the following :

```
# include "node.h"        // The task ‘‘node’’
# include "nodeMDL.hh"    // The class presenting the task  ‘‘node’’
struct node *vdata;
Initialisation;
Creation_grille;
```

```
Calcul_taille_Sdomaine;
Do
    Vdata->U = Unew;
    Iteration_Jacobi;
    Calculation_difference;
    Exchange_ border;
While ( ++nbr_iteratin < Max_nbr_iteration )  && ( DIF > Min_dif )
         // Test of convergence according to the value of dif
```

Description of the procedures

In this paragraph, we present the various procedures which we used in the schema of the main program :

Initialization procedure : allows to initialize variables used such as U, F...

Creation_gride procedure : allows the creation of the grid, it is done by calling the following function : MDL_Grid_Create(dims) ;

Calculation_Size_Sdomaine procedure : allows to determine the terminals of each sub-domain. We present in the following the way of calculating terminals according to the first dimension. The calculation of terminals according to the second dimension uses the same technique ;

```
MDL_Grid_Mycoords(Vdata,coords);
if ( Neighbor_East != MDL_PROC_NULL)
 {
    npi    =  N/dims[0];
    INFi = coords[0] * npi + 1;
    SUPi  = INFi + npi - 1;
 }
else
 {
    npi    = N - ( dim[0] - 1 ) * ( N/dim[0] );
    INFi = N - npi + 1;
    SUPi  = N;
 }
```

Iteration_Jacobi Procedure : it is composed from the following loop :

```
  for ( int i = INFi ; i <= SUPi ; i++)
   for(int j=INFj; j<= SUPj ; j++)
       Unew[i][j] =  0.25 * ( Vdata->U[i-1][j] + Vdata->U[i][j+1] +
                     Vdata-> U[i][j-1] + Vdata->U[i+1][j] - h*h*f[i][j]);
```

Calculation_difference procedure : it makes it possible to calculate the sum of the differences between U and UNEW of the set of the processes. Following the extension suggested to the precede chapter, we must start by declaring a message of reduction in the collective block :

```
collective
{
  Allreduce // the result of computation will be retourned to all instances
   {
```

```
        Sum_dif = sequence{dif,DIF ,1,MDL_SUM};
             // dif : difference per instance
             // DIF : the result of the sum of dif
    }
}
```

Which makes it possible to use the following function : MDL_Allreduce_Sum_dif();

Exchange_border procedure : during each iteration, each process calculates new values of $U_{i,j}$ for $i \in [INF_i, SUP_i]$ and $j \in [INF_j, SUP_j]$. Thus, each process uses the block U[INFi,SUPi][INFj,SUPj] to perform it's local calculations. Thus, each task uses during its calculations the values not belonging to its own block such as U_{INF_i-1}, U_{INF_i,SUP_i+1}. Where the need after each iteration to communicate these values by exchange at the borders between the close tasks.

We give an example of communication with the North neighbor. The principle is the same one to communicate with the South, West and East neighbors. These exchanges differ only on the level of the contents of the messages (borders) to exchange.

```
if ( Neighbor_North != MDL_PROC_NULL)
{
  MDL_SendTO_node_NeighborNorthS(Vdata, U[INFi, SUPi][SUPj],Neighbor_North);
  MDL_RecvFrom_node_NeighborNorthR(Vdata, U[INFi,SUPi][SUPj+1], Neighbor_North);
}
```

4.6 Conclusion

The programming method based on grid-decomposition leads to a very natural parallelization and has the advantage of muching well the local memory use. The example presented in this chapter is a case with two dimensions, nevertheless the new functions of MeDLey can be used in the case of a problem with tree dimensions.

The communication with a process of rank MDL_PROC_NULL also makes it possible to abstain from the conditional tests to know if the process exists or not, and consequently allows a better legibility of the program.

Moreover, the use of the preset neighbors makes it possible to facilitate the programming. The user should worry only to define the contents of the messages to be exchanged. However, as shown in the example dealt with in this chapter, the user must take into account the difference between the matric notation and that of MeDLey by presenting the domains..

Conclusion

Parallel programming is more complex than sequential cases. Indeed, to write a parallel program, many tools are needed : language with explicit parallelism, tools of traces and visualization, evaluation of performances, communication libraries, etc. One of the topics of the RESEDAS team, the new MeD-Ley 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 and to propose evolutions and extensions of its environment. The experimentation part was carried out within the framework of a collaboration with the research laboratory in physics (LPMI) of the UHP, and consisted in the use of a code of digital simulation used by the physicists.

The extension part of this language was intended to enrich the initial environment, to make it possible to describe common usual problems encountered in the parallelism domain. The first extension related to the collective communications ; we proposed a new approach and new functions allowing to specify and carry out this type of communication. The second one related to the methods of decomposition of field ; we have completed the definition of the mode of interconnection existing, then proposed functions meeting the need of applications of these methods.

Bibliography

[Brugeas 96] I. Brugeas. Utilisation de mpi en décomposition de domaine. Technical report, IDRIS, 1996.

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

[Forum 96] MPI Forum. *Extension to Message Passing Interface*. NSF and ARPA, 1996.

[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.

[Gropp 94] W. Gropp, E. Lusk and A. Skjellum. *Using MPI : Portable Programming with the Message Passing Interface*. Mass-Ins-Tech, 1994.

Unit´e de recherche INRIA Lorraine, Technop^ole de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rh^one-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399