A Border-based Coordination Language for Integrating Task and Data Parallelism[1]

Manuel Díaz, Bartolomé Rubio, Enrique Soler and José M. Troya
Dpto. Lenguajes y Ciencias de la Computación. Málaga University
29071 Málaga, SPAIN
E-mail: {mdr, tolo, esc, troya}@lcc.uma.es

This paper presents BCL, a Border-based Coordination Language focused on the solution of numerical applications. Our approach provides a simple parallelism model. Coordination and computational aspects are clearly separated. The former are established using the coordination language and the latter are coded using HPF (together with only a few extensions related to coordination). This way, we have a coordinator process that is in charge of both creating the different HPF tasks and establishing the communication and synchronization scheme among them. In the coordination part, processor and data layouts are also specified. Data distribution belonging to the different HPF tasks is known at the coordination level. This is the key for an efficient implementation of the communication among them. Besides that, our system implementation requires no change to the runtime support of the underlying HPF compiler. By means of some examples, the suitability and expressiveness of the language are shown. Some experimental results also demonstrate the efficiency of the model.

Key Words: coordination languages, high performance computing, numerical problems, task and data parallelism

1. INTRODUCTION

High Performance Fortran (HPF) [13] has emerged as a standard data parallel, high level programming language for parallel computing. However, a disadvantage of using a parallel language like HPF is that the user is constrained by the model of parallelism supported by the language. It is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data-parallel paradigm: pipelines of data parallel tasks [9], a common computation structure in image processing, signal processing or computer vision; irregular ap-

[1]This work was supported by the Spanish project CICYT TIC-99-0754-C03-03

plications [6]; multidisciplinary optimization problems like aircraft design[5], etc. For these applications, rather than having a single data-parallel program, it is more appropriate to subdivide the whole computation into several data-parallel pieces, where these run concurrently and co-operate, thus exploiting task parallelism.

Integration of task and data parallelism is currently an active area of research and several approaches have been proposed [12][11][14]. Integrating the two forms of parallelism cleanly and within a coherent programming model is difficult [1]. In general, compiler-based approaches are limited in terms of the forms of task parallelism structures they can support, and runtime solutions require that the programmer have to manage task parallelism at a lower level than data parallelism. The use of coordination models and languages to integrate task and data parallelism [5][18][15] is proving to be a good alternative, providing a high level mechanism and supporting different forms of task parallelism structures in a clear and elegant way. Coordination languages [4] are a class of programming languages that offer a solution to the problem of managing the interaction among concurrent programs. The purpose of a coordination model and the associated language is to provide a mean of integrating a number of possibly heterogeneous components in such a way that the collective set forms a single application that can execute on and take advantage of parallel and distributed systems.

BCL is a Border-based Coordination Language focused on the solution of numerical problems, especially those with an irregular surface that can be decomposed into regular, block structured domains. It has been successfully used on the solution of domain decomposition-based problems and multi-block codes [7]. Moreover, other kinds of problems with a communication pattern based on (sub)arrays interchange (2-D FFT, convolution, solution of PDEs by means of the red-black ordering algorithm, etc.) may be defined and solved in an easy and clear way [8].

Our approach is thought to be used by users such as engineers, mathematicians and physicists. Although their applications require a lot of computational power, in most cases this kind of users are not in the habit of programing with a parallel language. This fact has been taken into account in the design of BCL. Its two main objectives are, on the one hand, to provide an easy to learn and use high-level language, and, on the other hand, to offer an efficient approach in order to support the required applications. The first objective has been achieved by separating the coordination and computational aspects of the different tasks in which a problem is decomposed. This way, the programmer does not need to take into account the coordination details when he/she is programming the computational part, and vice versa. Moreover, the reusability of both parts is increased. In order to achieve the second objective, an appropriate task and data parallelism integration model is provided.

In this paper we describe BCL and the way it can be used to integrate task and data parallelism in a clear, elegant and efficient way. Computational tasks are coded in HPF. The fact that the syntax of BCL has an HPF style makes that both the coordination and the computational parts can be written using the same language, i.e., the application programmer does not need to learn different languages to describe different parts of the problem, in contrast with other approaches [14]. The coordinator process, besides of being in charge of creating the different tasks and establishing their coordination protocol, also specifies processor and data lay-

outs. The knowledge at the coordination level of data distribution belonging to the different HPF tasks is the key for an efficient implementation of the communication and synchronization among them. In BCL, unlike in other proposals [11][15], the inter-task communication schedule is established at compilation time. Moreover, our approach requires no change to the runtime support of the HPF compiler used. The implementation of the model, realized on top of the MPI communication layer and the public domain HPF compilation system ADAPTOR [3], has been evaluated by means of several examples showing a good performance. We also present some experimental results.

The rest of the paper is structured as follows. In section 1.1 some related work are sketched. Section 2 presents BCL. The use and expressiveness of the language is shown in section 3 by means of some examples. In section 4 some implementation issues and experimental results are mentioned. Finally, in section 5, some conclusions are sketched.

1.1. Related Work

In recent years, several proposals have addressed integration of task and data parallelism. We shall state a few of them and discuss the relative contributions of our approach.

The Fx model [20] expresses task parallelism by providing declaration directives to partition processors into subgroups and execution directives to assign computations to different subgroups (task regions). These task regions can be dynamically nested. The new standard HPF 2.0 [12] of the data parallel language HPF provides approved extensions for task parallelism, which allow nested task and data parallelism, following a similar model to that of Fx. These extensions allow the spawning of tasks but do not allow interaction like synchronization and communication among tasks during their execution and therefore might be too restrictive for certain kind of applications. However, some HPF 2.0 implementations overcome this limitation by means of a specific library **??**. Differently from these proposals, BCL does not need the adoption of new task parallel HPF constructs to express task parallelism. BCL is a coordination layer for HPF tasks which are separately compiled by an off-the-shelf HPF compiler that requires no change, while the task parallel coordination level is provided by the corresponding BCL library.

In HPF/MPI [11], the message-passing library MPI has been added to HPF. This definition of an HPF binding for MPI attempts to resolve the ambiguities appeared when a communication interface for sequential languages is invoked from a parallel one. In an HPF/MPI program, each task constitutes an independent HPF program in which one logical thread of control operates on arrays distributed across a statically defined set of processors. At the same time, each task is also one logical process in an MPI computation. In our opinion, the adoption of a message-passing paradigm to directly express HPF task parallelism is too low-level. Moreover, in our approach, the inter-task communication schedule is established at compilation time from the information provided at the coordination level related to the inter-domain connections and data distribution. In this case, expressiveness and good performance are our relative contributions.

KeLP [10] is one of the works that tries to separate the coordination from the computational aspects of a problem, although it is not a coordination language.

It is a C++ based software tool developed to ease the implementation of multi-level and mesh adaptive methods. It is a specific library that supports irregular and dynamic data structures. KeLP abstractions represent data decomposition and communication patterns. As a programming model it can be seen as coarse-grained data parallel model. The code of one application is typically written in two levels: high level KeLP code to control data structures and parallelism; and low level code written in C, C++ or Fortran to implement the numerical computations. In order to integrate data and task parallelism, KeLP-HPF [14] has been developed. This way, a high performance can be obtained in irregular, block structured problems, since two kinds of parallelism are being exploited: among blocks and within them. KeLP code is used to specify data distribution and inter-block communications and to invoke HPF concurrently on each block. This system does not imply any extension to the HPF language (KeLP is a C++ class library). However, it is necessary to modify both the HPF compiler and its runtime system (in order to allow for HPF code execution on dynamically defined processor subsets). Moreover, the programmer of scientific applications needs to know the object-oriented programming paradigm together with the data parallel language HPF.

Opus [5] is an object-oriented coordination language that has been developed to ease the implementation of scientific and engineering applications that are heterogeneous and multidisciplinary so that they do not fit for the data parallelism paradigm. In Opus, one can define classes of objects, called ShareD Abstractions (SDAs), using a syntax similar to that of HPF. SDAs can be computational servers or data deposits shared among different tasks. Data parallel tasks are dynamically started by creating instances of specific SDAs, while inter-task co-operation takes place by means of remote method invocations. Differently from Opus, our model is focused on the solution of numerical problems, starting and coordinating a statically fixed set of HPF tasks.

Another coordination language for mixed task and data parallel programs has been proposed in [18]. The model provides a framework for the complete derivation process in which a specification program is transformed into a coordination program. The former expresses possible execution orders between modules and describes the available degree of task parallelism. The latter describes how the available degree of parallelism is actually exploited for a specific parallel implementation. The result is a complete description of a parallel program that can be easily translated into a message-passing program. This proposal is more a specification approach than a programming approach. The programmer is responsible for specifying the available task parallelism, but the final decision whether the available task parallelism will be exploited and how the processors should be partitioned into groups is taken by the compiler. Moreover, it is not based on HPF. The final message-passing program is expressed in C with MPI.

Finally, $COLT_{HPF}$ [15] is a runtime support specifically designed for the coordination of concurrent and communicating HPF tasks. It provides suitable mechanisms for starting distinct HPF data-parallel tasks on disjoint groups of processors together with optimized primitives for inter-task communication. This communication is achieved by means of typed channels. The data distributed among the processors assigned to each task are communicated through these channels. Unlike in our approach, the inter-task communication schedule cannot be established

at compilation time. It is also implemented on top of MPI, but it requires small changes to the runtime support of the HPF compiler used, the public domain HPF compilation system ADAPTOR too. However, there is a new version realized on top of PVM [16] whose features allow to overcome this limitation.

2. THE COORDINATION LANGUAGE BCL

BCL is not a general purpose language, but it is focused on the solution of domain decomposition-based problems and multi-block codes. In addition, other kinds of problems with a communication pattern based on (sub)arrays interchange may be defined and solved in an easy and clear way.

Domain decomposition methods are successfully being used for the solution of linear and non-linear algebraic equations that arise upon the discretization of partial differential equations (PDEs) [19]. Figure 1 shows the solution scheme for a 2 domain parabolic problem.

Programming such applications is a difficult task because we have to take into account many different aspects, such as:

- The different numerical methods applied to each domain.
- The conditions imposed at the borders, the equations used to solve them and overlapping or non-overlapping techniques [17].
- The geometry of the problem, which may be complex and irregular.
- Possible integration of task and data parallelism. On the one hand, task parallelism is more appropriate for the communication among processes that solve each domain. On the other hand, the solution of each domain can be easier using a data parallel language (e.g. HPF) so that, very efficient programs can be obtained with relatively little effort.

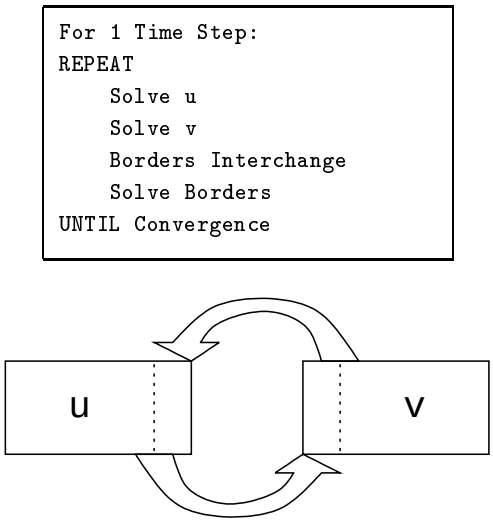

FIG. 1. A typical domain decomposition application scheme

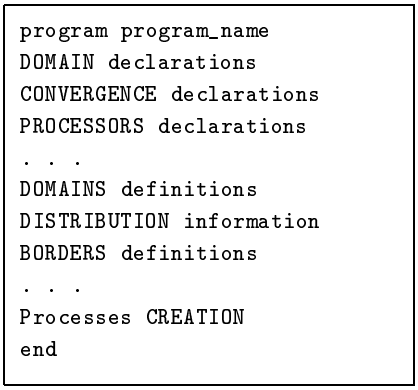

FIG. 2. A coordinator process scheme

2.1. A BCL Program Scheme

Figures 2 and 3 show a typical BCL program scheme. The coordinator process (Figure 2) is coded using BCL and is in charge of:

- Defining the different blocks or domains that form the problem. Each one will be solved by a worker process, i.e., by an HPF task.
- Specifying processor and data layouts.
- Establishing the coordination scheme among worker processes:
  - Defining the borders among domains.
  - Establishing the way these borders will be updated.
  - Specifying the possible convergence criteria.
- Creating the different worker processes.

On the other hand, worker processes (Figure 3) constitute the different HPF tasks that will solve the problem. They are declared as subroutines and receive as dummy arguments the domains and the convergence variables defined in the coordinator process. `GRID` variables are declared to store the data belonging to the corresponding domains. Local computations are achieved by means of standard HPF sentences while the communication and synchronization among worker processes are carried out through the BCL primitives `PUT_BORDERS`, `GET_BORDERS` and `CONVERGE`.

2.2. Coordinator process

A coordinator process declares `DOMAIN` variables as follows:

`DOMAIN`x`D u`

where $1 \leq x \leq 4$. A variable `u` of this type will consist of $2x$ numbers and represents a domain, i.e. a subset of Z^x, where x is the dimensionality of the problem. A domain definition is achieved by means of an assignment of Cartesian points. For the two-dimensional case the expression:

```
Subroutine subroutine_name (. . .)
DOMAIN declarations ! dummy args.
CONVERGENCE declarations ! dummy args.
GRID declarations
GRID distribution
GRID initialization
do while .not. converge
    . . .
    PUT_BORDERS
    . . .
    GET_BORDERS
    Local computation
    CONVERGENCE test
enddo
. . .
end
```

FIG. 3. A worker process scheme

```
u = (/1, 1, Nx, Ny/)
```

assigns to the variable `u` the region of the plane that extends from the point `(1,1)` to the point `(Nx,Ny)`. From the implementation point of view, a domain variable also stores the information related to its borders and the information needed from other(s) domain(s) (e.g. data distribution).

Different borders can be defined among the specified domains. For example, if `u`, `v` are declared `DOMAIN2D`:

```
u(Nx, 1, Nx, Ny) <- v(2, 1, 2, Ny)
```

indicates that the region of `u` delimited by points `(Nx,1)` and `(Nx,Ny)` will be updated by the values belonging to the region of `v` delimited by points `(2,1)` and `(2,Ny)`. The region sizes at both sides of the operator `<-` must be equal (although not their shapes). To apply a function at the right hand side of the operator `<-` in which several domains can be implied (or a region of them) is allowed. In order to solve some problems (e.g. PDE solution by means of the red-black ordering method) it is better to use several kinds of borders that are communicated in different phases of the algorithm. This way, a border definition can be optionally labeled with a number that indicates the connection type in order to distinguish kinds of borders (or to group them using the same number), so that complex patterns of communication can be expressed.

In order to achieve task and data parallelism integration, two directives have been included:

- Declaration of system processors is made in a similar way as in HPF. For example:

```
PROCESSORS p(4,4)
```

indicates a square arrangement of 16 processors. Unlike in HPF, when two or more PROCESSORS variables are declared in the same program it is understood that they refer to different subsets of processors.

- DISTRIBUTE is applied to DOMAIN variables. This instruction does not perform the distribution itself but indicates to the system the future distribution of the grid that is associated to the specified domain (see worker processes below). The distribution types correspond to those of HPF, for example:

```
DISTRIBUTE u (*,BLOCK) ONTO p
```

The knowledge of the data distribution at the coordination level is the key for an efficient implementation of the communication among HPF tasks. The coordinator process passes the distribution information to the workers in such a way that a process knows the distribution of its domain and the distribution of every domain with a border in common with its domain. So, it can be deduced which part of the border needs to be sent/received to/from which processors of other tasks. This is achieved at compilation time.

The coordinator process declares variables of CONVERGENCE type to allow the communication among worker processes in order to decide whether the convergence of a method has been reached or not. In general, this type is used to perform a reduction of a scalar number among the processes sharing c. For example:

```
CONVERGENCE c OF num
```

where num is the number of tasks that will share c.

The creation of worker processes is done by means of the CREATE instruction:

```
CREATE processName (u,c,...)  ON p
```

where processName is the name of the code segment that should be spawned as a new process in an asynchronous way, so that several processes can be executed in parallel. Variables u and c are of DOMAIN and CONVERGENCE types respectively. The process processName could receive, optionally, an arbitrary number of additional arguments of any type needed for the application. External-declared subroutines and functions could also be passed as arguments. The clause ON is used in order to indicate the HPF processors that will execute the indicated task.

2.3. Worker processes

In this case, when a worker process declares a CONVERGENCE dummy argument, the clause OF is not specified, since the worker processes do not need to know how many tasks are solving the problem. This way, the reusability of the workers is improved (coordination aspects are specified in the coordinator process).

The GRID attribute is used to declare a record with two fields, the data array and an associated domain (together with its borders). Therefore, the example:

```
REAL, GRID2D ::  g
```

declares a variable that contains a domain, g%DOMAIN, and an array of real numbers, g%DATA, which will be dynamically created when a value is assigned to the domain field. Note that this is an extension of our language since a dynamic array can not

be a field of a standard Fortran 90 record. On the other hand, the type of the array elements (which can be even user-defined) is specified inside the computational part, so that coordinator process reusability is increased.

A variable `g1` with `GRID` attribute can be assigned to another variable `g2` of the same type (`g2 = g1`) if they have the same domain size or if `g2` has no `DOMAIN` defined yet. In the latter case, the following steps would be automatically executed:

1. The copying of the `DOMAIN` field.
2. Dynamic creation of the field `g2%DATA` with enough space to store the data for its domain.
3. The copying of the data stored in the field `g1%DATA`.

The actual data distribution is carried out as in the following example:

```
!hpf$ distribute (*,BLOCK) ::  g
```

Note that this is a special kind of distribution since it produces the distribution of the field `DATA` and the replication of the field `DOMAIN`.

The data belonging to one process that are needed by another (as defined in the coordinator process), are sent by means of the instruction:

```
PUT_BORDERS (g)
```

where `g` is a variable with `GRID` attribute. This is an asynchronous operation.

In order to receive the data needed to update the borders associated to the domain belonging to a variable with `GRID` attribute, say `g`, the instruction:

```
GET_BORDERS (g)
```

is introduced. The process that calls this instruction will suspend its execution until the data needed to update all the borders associated to `g%DOMAIN` are received. If a function has been defined at the right hand side of the operator `<-`, it will be called. `PUT_BORDERS` and `GET_BORDERS` may optionally have a second argument, an integer number that represents the kind of border that is desired to be "sent" or "received".

Communication needed to determine whether the convergence criteria have been reached is achieved by means of the instruction:

```
CONVERGE (c, vble, procName)
```

where `c` is a `CONVERGENCE` variable, `vble` is a scalar variable of any type and `procName` is a subroutine name. This instruction produces a reduction of the scalar value used as second argument by means of the subroutine `procName`.

2.4. Additional aspects

In addition to the characteristics mentioned above, some other aspects have been added to BCL in order to improve language expressiveness.

In order to simplify some instructions, the use of a `DOMAIN` variable is allowed as an array index. There are also other primitives to manage domains and borders: `GROW`, `INTERSECTION`, `DECOMPOSE` and `ARGUMENTS`. The former is a function to increase or decrease the region of a domain received as dummy argument. The two following

```
 1) program example1
 2) DOMAIN2D u, v
 3) CONVERGENCE c OF 2
 4) PROCESSORS p1(4,4), p2(2,2)
 5) DISTRIBUTE u (BLOCK,BLOCK) ONTO p1
 6) DISTRIBUTE v (BLOCK,BLOCK) ONTO p2
 7) u = (/1,1,Nxu,Nyu/)
 8) v = (/1,1,Nxv,Nyv/)
 9) u (Nxu,Ny1,Nxu,Ny2) <- v (2,1,2,Nyv)
10) v(1,1,1,Nyv) <- u (Nxu-1,Ny1,Nxu-1,Ny2)
11) CREATE solve (u,c) ON p1
12) CREATE solve (v,c) ON p2
13) end
```

FIG. 4. The coordinator process for Jacobi's method

primitives are used to automatically define borders. The latter is a macro that expands the coordinates (separated by commas) that form a domain. This is useful when calling a Fortran subroutine which is to be reused. In the next section some examples of their use are sketched.

3. PROGRAMMING EXAMPLES

In this section, the expressiveness and suitability of the approach are shown by means of two simple examples.

3.1. Example 1. Laplace's Equation

Figure 4 shows the coordinator process for an irregular problem that solves Laplace's equation in two dimensions using Jacobi's finite differences method with 5 points. Although this it not the best method to solve this problem, its simplicity allows us to describe the language without having to take into account details of a more elaborate method.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad in\ \Omega \tag{1}$$

where u is a real function, Ω is the domain, a subset of R^2, and Dirichlet boundary conditions have been specified on $\partial\Omega$, the boundary of Ω:

$$u = g \quad in\ \partial\Omega \tag{2}$$

The domains in which the problem is divided are shown in Fig. 5 together with a possible data distribution and the border between domains. Dotted lines represent the distribution into each HPF task. Line 2 in the coordinator process is used to declare two variables of type `DOMAIN2D`, which represent the two-dimensional domains. These variables take their values in lines 7 and 8. These values represent

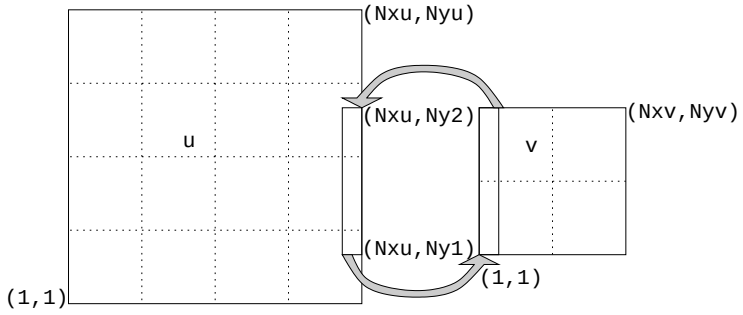

FIG. 5. Communication between two HPF tasks

Cartesian coordinates, i.e. the domain assigned in line 7 is a rectangle that cover the region from point `(1,1)` to `(Nxu, Nyu)`.

The border is defined by means of the operator `<-`. As it can be observed in the program, the border definition in line 9 causes that data from column `2` of domain `v` refresh part of the column `Nxu` of domain `u`. Symmetrically, the border definition in line 10, produces that data from column `1` of domain `v` are updated by part of the column `Nxu-1` of domain `u`.

Line 4 declares subsets of HPF processors where the worker processes are executed. The data distribution into HPF processors is declared by means of instructions 5 and 6. The actual data distribution is done inside the different HPF tasks.

A `CONVERGENCE` type variable is declared in line 3, which is passed as argument to the worker processes spawned by the coordinator. The clause `OF 2` indicates the number of HPF tasks that will take part in the convergence test. The worker processes receive this variable as a dummy argument.

Lines 11 and 12 spawn the worker processes in an asynchronous way so that both HPF tasks are executed in parallel.

The code for the worker processes is shown in Fig. 6. Lines 2 and 3 declare dummy arguments `u` and `c`, respectively, which are passed from the coordinator. The `GRID` attribute appears in line 4. This attribute is used to declare a record with two fields, the data array and an associated domain. Therefore, the variable `g` contains a domain, `g%DOMAIN`, and an array of double precision numbers, `g%DATA`, which will be dynamically created when a value is assigned to the domain field in line 6.

Line 5 produces the distribution of the field `DATA` and the replication of the field `DOMAIN` of both `g` and `g_old` variables.

Statement 9 produces the assignment of two variables with the `GRID` attribute. Since the first time the loop is executed `g_old` has not defined its domain yet, this instruction will involve the execution of the three automatic steps explained above. However, in the following iterations, only the copy of the values of field `g%DATA` to `g_old%DATA` will be carried out.

Lines 10 and 11 are the first in which communication is achieved. The instruction `PUT_BORDERS(g)` in line 10 causes that the data from `g%DATA` needed by the other task (see instructions 9 and 10 in the coordinator process) are sent. In order to

```
 1) subroutine solve (u,c)
 2) DOMAIN2D u
 3) CONVERGENCE c
 4) double precision, GRID2D :: g, g_old
 5) !hpf$ distribute (BLOCK,BLOCK) :: g, g_old
 6) g%DOMAIN = u
 7) call initGrid (g)
 8) do i=1, niters
 9)    g_old = g
10)    PUT_BORDERS (g)
11)    GET_BORDERS (g)
12)    call computeLocal(g,g_old)
13)    error = computeNorm (g,g_old)
14)    CONVERGE (c,error,maxim)
15)    Print *, "Max norm: ", error
16) enddo
17) end
```

FIG. 6. The worker process for Jacobi's method

receive the data needed to update the border associated to the domain belonging to g, the instruction GET_BORDERS(g) is used in line 11.

Local computation is accomplished by the subroutines called in lines 12 and 13 while the convergence method is tested in line 14. The instruction CONVERGE causes a communication between the two tasks that share the variable c. In this case, the maximum of the values of the variable error contained by each worker is finally assigned to the variable error. In general, this instruction is used when an application needs a reduction of a scalar value.

The rest of routines may be written in HPF without any BCL syntax so that they could be easily reused. However, it may be useful to employ the concept of domain to enhance the clarity of the subroutines initGrid, computeLocal and computeNorm. This has the drawback of modifying the subroutines introducing the type DOMAIN, the attribute GRID, etc. We think that offering both possibilities can be a good idea. Figure 7 shows all these subroutines.

The initGrid subroutine initializes the interior points of g to 0.0 and the boundary points to 1.0. A value can be assigned to the field g%DATA using only one statement following Fortran 90 syntax as shown in line 5 of subroutine initgrid. The variable interior (of type DOMAIN) will obtain in statement 6 the region of the domain of g but reduced in 1. Instruction 7 consists of an array indexing with a domain so that only the values of that region will receive the indicated expression (real value 0.0 in this case).

Note that in line 4 of subroutine computeLocal, the macro ARGUMENTS is being used. This macro is used when calling a subroutine written in standard HPF that is not desired to be modified.

```
 1) subroutine initGrid ( g )
 2) double precision, GRID2D ::  g
 3) !hpf$ inherit g
 4) DOMAIN2D interior
 5) g%DATA = 1.0
 6) interior = GROW (g%DOMAIN, -1)
 7) g%DATA (interior) = 0.0
 8) end

 1) subroutine computeLocal (g, g_old)
 2) double precision, GRID2D :: g, g_old
 3) !hpf$ inherit :: g, g_old
 4) call j5relax(g%DATA, g_old%DATA, ARGUMENTS (g%DOMAIN))
 5) end

 1) subroutine j5relax (a, a_old, ul0, uh0, ul1, uh1)
 2) integer ul0, uh0, ul1, uh1
 3) double precision,dimension(ul0:uh0, ul1:uh1) :: a, a_old
 4) !hpf$ inherit :: a, a_old
 5) integer i,j
 6) forall(j= ul1 + 1:uh1 -1,i= ul0 + 1:uh0 - 1)
 7)     a(i,j) = 1.0 / 4.0 * ( a_old (i-1,j) +      &
 8)             a_old (i+1,j) + a_old (i,j-1) + a_old (i,j+1))
 9) endforall
10) end

 1) double precision function computeNorm (g, g_old)
 2) double precision GRID2D :: g, g_old
 3) !hpf$ inherit g, g_old
 4) double precision r
 5) DOMAIN2D interior
 5) interior = GROW(g%DOMAIN, -1)
 7) r = MAXVAL (ABS( g%DATA(interior) - g_old%DATA (interior)))
 8) computeNorm = r
 9) end

 1) subrotine maxim (result, error, length)
 2) integer length, i
 3) double precision result, error (length)
 4) result = error (1)
 5) do i= 2, length
 6)   if (result < error (i) ) result = error (i)
 7) enddo
 8) end
```

FIG. 7. Computational subroutines

Subroutine `j5relax` is the one that performs the resolution of the equation, and so, the one with the most computational weight. It is completely written in HPF, so that it can be completely reused in other applications. Its dummy arguments correspond to the real arguments used in line 4 of subroutine `computeLocal`. Variables `a` and `a_old` correspond to `g%DATA` and `g_old%DATA`, respectively. The last four arguments correspond to those expanded by the macro `ARGUMENTS`.

Subroutine `computeNorm` also uses the indexation of the field `DATA` of a `GRID` by means of a `DOMAIN`.

Finally, the function `maxim` is the one passed to the instruction `CONVERGE` in line 14 of Fig. 6. This function is implicitly called. The argument `length` receives the number of processors defined in the coordinator process, in this case 2. This way, a reduction of the scalar variable `error` is performed.

3.1.1. Code reusability

In order to stress the way our approach achieves the code reusability, Fig. 8 shows another irregular problem that is solved by the program depicted in Fig. 9

The most relevant aspect of this example is that subroutine `solve` does not need to be modified, it is the same as in the example above. This is due to the separation that has been done between the definition of the domains (and their relations) and the computational part. Lines 15, 16 and 17 are instantiations of the same process for different domains. On the other hand, this coordinator process can also be reused in a different problem with the same geometry, as it has been carried out in the third application considered in section 4.

Note that, unlike in program `example1` of Fig. 2, domains `l`, `m`, and `r` have been defined taking into account their distribution in the plane. Here, the regions at both sides of the operator `<-` are the same. In these cases, our approach provides the notation `_` to be used at the right hand side. So, for example, line 11 could be written as follows:

```
l(Ncl,Nrm1 ,Ncl,Nrm2) <- m (_)
```

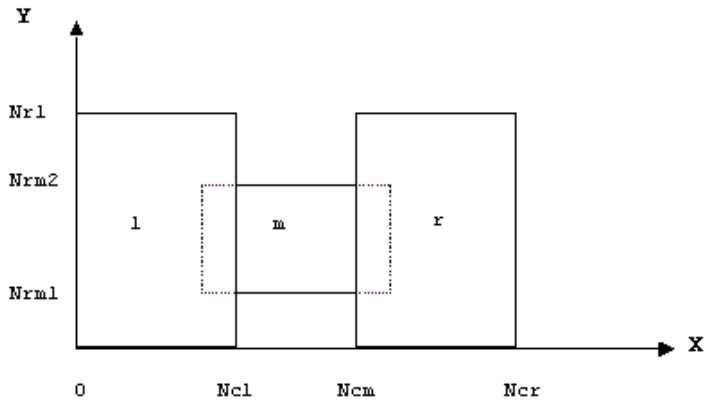

FIG. 8. Another irregular problem

```
 1) program example1_1
 2) DOMAIN2D l, m, r
 3) CONVERGENCE c OF 3
 4) PROCESSORS p1 (4,4), p2(2,2), p3(4,4)
 5) DISTRIBUTE l (BLOCK,BLOCK) ONTO p1
 6) DISTRIBUTE m (BLOCK,BLOCK) ONTO p2
 7) DISTRIBUTE r (BLOCK,BLOCK) ONTO p3
 8) l = (/0,0, Ncl, Nrl /)
 9) m = (/Ncl-1,Nrm1, Ncm+1, Nrm2 /)
10) r = (/Ncm,0, Ncr, Nrl /)
11) l(Ncl,Nrm1 ,Ncl,Nrm2) <- m(Ncl,Nrm1 ,Ncl,Nrm2)
12) m(Ncl-1,Nrm1 ,Ncl-1,Nrm2) <- l(Ncl-1,Nrm1 ,Ncl-1,Nrm2)
13) m(Ncm+1,Nrm1, Ncm+1,Nrm2) <- r(Ncm+1,Nrm1, Ncm+1,Nrm2)
14) r(Ncm,Nrm1, Ncm,Nrm2) <- m(Ncm,Nrm1, Ncm,Nrm2)
15) CREATE solve (l,c) ON p1
16) CREATE solve (m,c) ON p2
17) CREATE solve (r,c) ON p3
18) end
```

FIG. 9. Another coordinator process for Jacobi's method

Moreover, the borders can be implicitly obtained as a result of the intersection among domains, by means of the predefined subroutine `INTERSECTION`. For example, lines from 11 to 14 could be replaced by:

```
INTERSECTION(l,m)
INTERSECTION(m,r)
```

3.1.2. Functions to define borders

Figure 10 shows a T shape domain that has been decomposed into 3 overlapping subdomains, `l`, `m` and `r`. It can be observed that, in this case, there are regions of the domain that are common to the three subdomains. This does not suppose a problem for the definition of the solution with BCL (Figure 11).

Statements 13, 14, 17, 18, 21 and 22 in Fig. 11 are used to define the way the regions corresponding to domains at the left hand side of the operator `<-` will be updated when `GET_BORDERS` is called. So, line 14 defines that the rectangular region delimited by points `(Nmx1+1,Nym+1)` to `(Nmx2-1,Nom-1)` of the `GRID` having associated `l` as `DOMAIN`, say `g`, is updated by means of the result of computing the average of the values calculated for the GRIDs that have `l`, `m` and `r` as associated domains. A call to `GET_BORDERS(g)` by the worker process makes it to wait until the data of the GRIDs that contain `m` and `r` are ready, the average is calculated and then, `g%DATA` is updated. The function defined at the right hand side of the operator `<-` can also be a subroutine that will be implicitly called when executing the instruction `GET_BORDERS`. This is an elegant way of specifying more complex functions, such as imposing Neumann or Robin boundary conditions between subdomains [17].

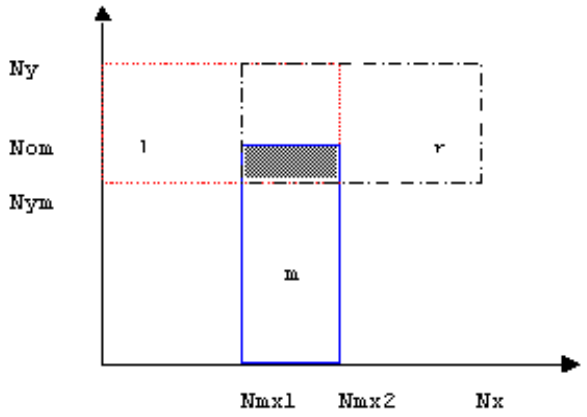

FIG. 10. Decomposition of a T shape domain

```
 1) program example1_2
 2) DOMAIN2D l ,m, r
 3) CONVERGENCE c OF 3
 4) PROCESSORS p1 (4), p2(4), p3(4)
 5) DISTRIBUTE l (*,BLOCK) ONTO p1
 6) DISTRIBUTE m (*,BLOCK) ONTO p2
 7) DISTRIBUTE r (*,BLOCK) ONTO p3
 8) l = (/0,Nym, Nmx2, Ny /)
 9) r = (/Nmx1,Nym, Nx, Ny /)
10) m = (/Nmx1,0, Nmx2, Nom /)
11) l(Nmx2, Nym + 1, Nmx2, Ny 1) <- r(_)
12) l(Nmx1+1,Nym,Nmx2-1,Nym) <- m(_)
13) l(Nmx1+1,Nom,Nxm2-1,Ny-1) <- (l(_) + r(_)) / 2.0
14) l(Nmx1+1,Nym+1,Nmx2-1,Nom-1) <- (l(_) + r(_) + m(_) ) / 3.0
15) r(Nmx1, Nym + 1, Nmx1, Ny 1) <- l(_)
16) r(Nmx1+1,Nym,Nmx2-1,Nym) <- m(_)
17) r(Nmx1+1,Nom,Nxm2-1,Ny-1) <- (l(_) + r(_)) / 2.0
18) r(Nmx1+1,Nym+1,Nmx2-1,Nom-1) <- (l(_) + r(_) + m(_) ) / 3.0
19) m(Nmx1, Nym + 1, Nmx1, Nom) <- l(_)
20) m(Nmx2, Nym + 1, Nmx2,Nom) <- r(_)
21) m(Nmx1+1,Nom,Nxm2-1,Nom) <- (l(_) + r(_)) / 2.0
22) m(Nmx1+1,Nym+1,Nmx2-1,Nom-1) <- (l(_) + r(_) + m(_) ) / 3.0
23) CREATE solve ( l, c )
24) CREATE solve ( r, c )
25) CREATE solve ( m, c )
26) end
```

FIG. 11. Coordinator process for the T shape domain

3.2. Example 2. 2-D Fast Fourier Transform

2-D FFT transform is probably the most widely used application to demonstrate the usefulness of exploiting a mixture of both task and data parallelism [11][15]. Given an N×N array of complex values, a 2-D FFT entails performing N independent 1-D FFTs on the columns of the input array, followed by N independent 1-D FFTs on its rows.

In order to increase the solution performance and scalability, a pipeline solution scheme is preferred as proved in [11] and [15]. Figure 12 shows the array distributions needed for that scheme. This mixed task and data parallelism scheme can be easily codified using BCL. Figure 13 shows the coordinator process, which simply declares the domain sizes and distributions, defines the border (in this case, the whole array) and creates both tasks. For this kind of problems there is no convergence criteria.

The worker processes are codified in Fig. 14. The stage 1 reads an input element, performs the 1-D transformations and calls `PUT_BORDERS(a)`. The stage 2 calls `GET_BORDERS(b)` to receive the array, performs the 1-D transformations and writes the result. The communication schedule is known by both tasks, so that a point to point communication between the different HPF processors can be carried out.

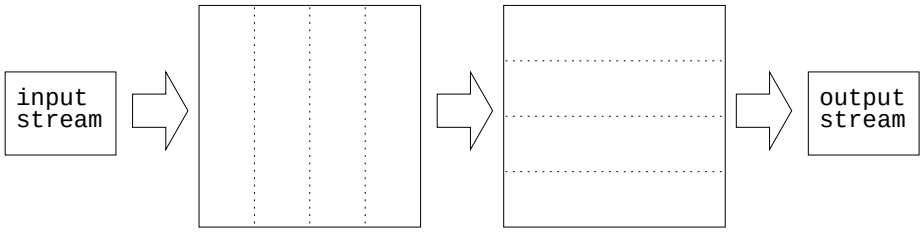

FIG. 12. Array distributions for the 2-D FFT problem

```
 1) program example2
 2) DOMAIN2D a, b
 3) PROCESSORS p1 (Np), p2(Np)
 4) DISTRIBUTE a (*,BLOCK) ONTO p1
 5) DISTRIBUTE b (BLOCK,*) ONTO p2
 6) a = (/1,1,N,N/)
 7) b = (/1,1,N,N/)
 8) b <- a
 9) CREATE stage1 (a) ON p1
10) CREATE stage2 (b) ON p2
11) end
```

FIG. 13. The coordinator process for the 2-D FFT problem

```
subroutine stage1 (d)                 subroutine stage2 (d)
DOMAIN2D d complex, GRID2D :: a       DOMAIN2D d complex, GRID2D :: b
!hpf$ distribute a(*,block)           !hpf$ distribute b(block,*)
a%DOMAIN = d                          b%DOMAIN = d
do i= 1, n_images                     do i= 1, n_images
 ! a new input stream element         GET_BORDERS (b)
 call read_stream (a%DATA)            !hpf$ independent
 !hpf$ independent                    do irow = 1, N
 do icol = 1, N                        call fftSlice(b%DATA(irow,:))
  call fftSlice(a%DATA(:,icol))       enddo
 enddo                                ! a new output stream element
 PUT_BORDERS (a)                      call write_stream (b%DATA)
enddo                                 enddo
end                                   end
```

FIG. 14. worker processes for the 2-D FFT problem

4. IMPLEMENTATION ISSUES AND RESULTS

In order to evaluate the performance of BCL, an implementation has been developed on a cluster of 4 DEC AlphaServer 4100 nodes interconnected by means of Memory Channel. Each node has 4 Alpha 22164 (300 MHz) processors sharing a 256 MB RAM memory. The operating system is Digital Unix V4.0D (Rev. 878). The implementation is based on source-to-source transformations together with the necessary libraries and it has been realized on top of the MPI communication layer and the public domain HPF compilation system ADAPTOR [3]. No change to the HPF compiler has been needed. Fig. 15 depicts the different phases to obtain the executable code. The BCL compiler translates the source code into an SPMD program that takes advantage of the `task_region` facility of HPF 2.0 so that the worker processes can be executed in different processor subsets. Communication among worker processes are achieved by means of calls to the BCL library (BCLLIB), which is implemented on top of MPI. The HPF program is compiled by the ADAPTOR compiler, which uses the library DALIB in order to manage the distributed arrays. Finally, the FORTRAN code generated by ADAPTOR is compiled by a standard FORTRAN compiler.

Several examples have been used to test it and the obtained preliminary results have successfully proved the efficiency of the model. Here, besides the results for the two problems explained above (Jacobi's method and 2-D FFT problem), a third application has been considered. It is a system of two non-linear reaction-diffusion equations solved by means of linearized implicit Θ-methods. The equations are the following:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + S(U) \tag{3}$$

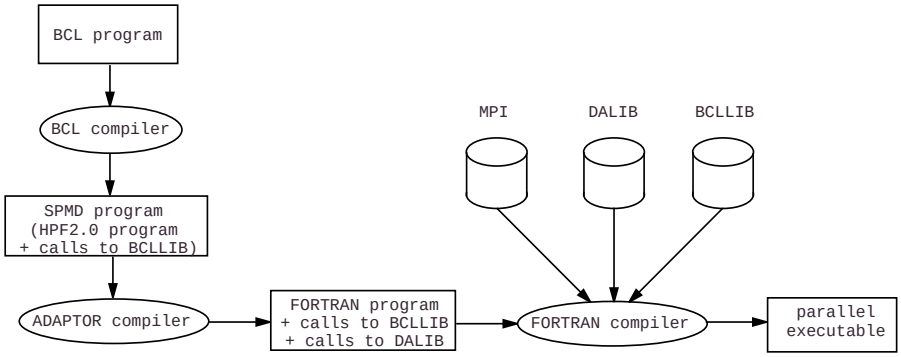

FIG. 15. Implementation scheme

$$U = \begin{pmatrix} u \\ v \end{pmatrix} \tag{4}$$

$$S(U) = \begin{pmatrix} -uv \\ uv - \lambda v \end{pmatrix} \tag{5}$$

where u and v are real functions defined in $\Omega \times [0, T]$ and λ is a constant.

The origins of this problem are both the propagation of spikes/pulses in biological systems and ignition and flame propagation through combustible mixtures (one-step reaction where u represents the fuel and v the temperature in simplified chemical kinetics). A detailed explanation of this problem and the employed numerical method can be found in [17]. The linearization yields a large system of linear algebraic equations solved by means of the BiCGstab algorithm [2]. The coordinator process for this problem is the one shown in Fig. 9, since this problem and the application explained in section 3.1.1 share the same geometry (Figure 8). Only the computational parts are different.

The following initial conditions have been considered (Figure 16): $u = 1$ and $v = 0$ at the boundaries where Dirichlet conditions have been imposed; $u(x, y, 0) = 1$ and $v(x, y, 0) = e^{-((x-x_{cent})^2+(y-y_{cent})^2)}$ for the left-most domain (`l`); $u(x, y, 0) = 1$ and $v(x, y, 0) = 0$ for the other two domains (`m` and `r`). The time step has been $\Delta t = 0.02$ and the calculations were performed until $t = 80$. Figure 17 shows the results at $t = 40$ and Fig. 18 depicts the final results at $t = 80$.

Table 1 compares the results obtained for Jacobi's method in HPF and in BCL considering a regular surface and 2, 4 and 8 domains with a 128×128 grid each one. The program has been executed for 20000 iterations. BCL offers a better performance than HPF due to the advantage of integrating task and data parallelism. When the number of processors is equal to the number of domains (only task parallelism is achieved) BCL has also shown better results. Only when there are more domains than available processors, BCL has shown less performance because of the context switch overhead among heavy processes.

Table 2 shows the execution time per input array for HPF and BCL implementations of the 2-D FFT application. Results are given for different problem sizes. Again, the performance of BCL is generally better. However, HPF performance is near BCL as the problem size becomes larger and the number of processors de-

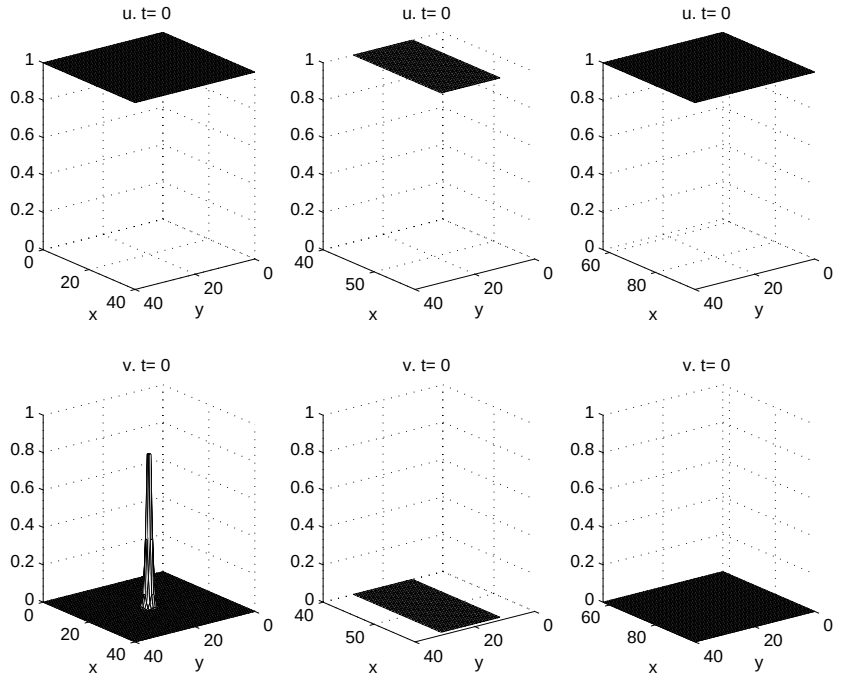

FIG. 16. Initial conditions for the three domains (`l`, `m`, `r`) of the non-linear reaction-diffusion problem. For each domain, variable u (up) represents fuel concentration and variable v (down) represents temperature

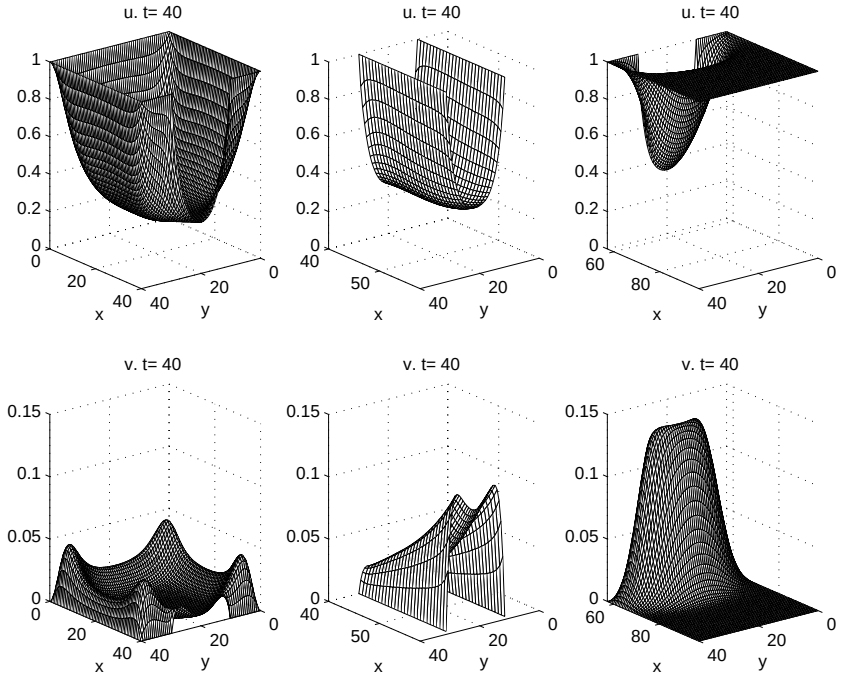

FIG. 17. Results at $t = 40$ for the non-linear reaction-diffusion problem

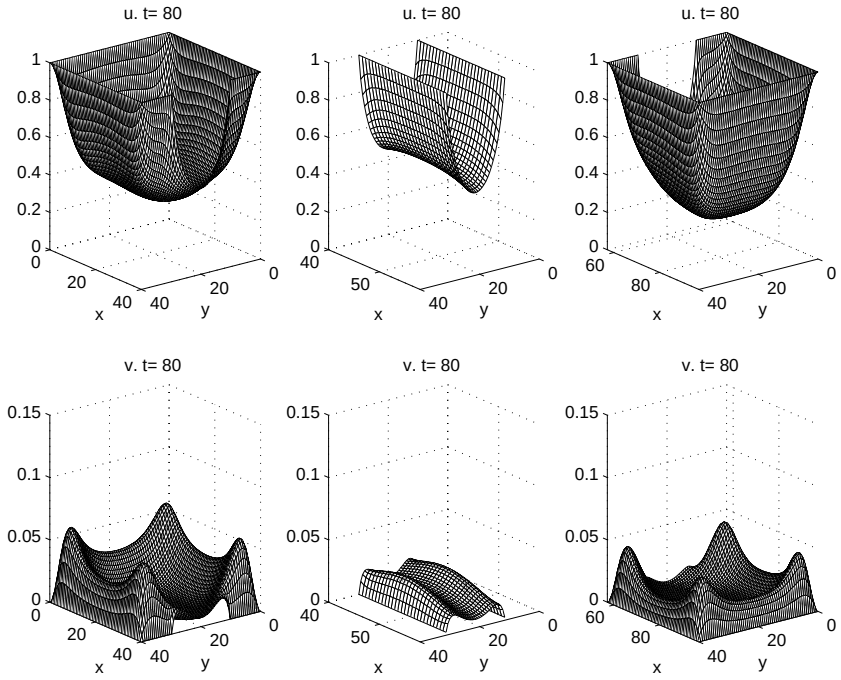

FIG. 18. Results at $t = 80$ for the non-linear reaction-diffusion problem

creases, as it also happens in other approaches [11]. In this situation HPF performance is quite good and so, the integration of task parallelism does not contribute so much.

Table 3 shows the results obtained for the non-linear reaction-diffusion equations. In this case, the surface is irregular (Figure 8), the number of (square) domains is fixed and different grid sizes have been considered (for example, in the first row, the grid size for `l` is 64×64, for `m` is 32×32 and for `r` is 64×64). For both HPF and BCL, 5, 9 and 16 processors have been considered. In the case of HPF, all the processors execute each domain. In the case of BCL, when 5 processors are used, 2 of them execute the domain `l`, 2 the domain `r` and 1 the domain `m`; for

TABLE 1
Computational time (in seconds) and HPF/BCL ratio for Jacobi's method

Domains	Sequential	HPF vs. BCL (ratio)		
		4 Processors	8 Processors	16 Processors
2	97.05	42.40/41.27 (1.03)	35.05/27.66 (1.27)	33.73/22.67 (1.49)
4	188.88	93.90/90.06 (1.04)	70.75/45.06 (1.57)	69.61/29.28 (2.38)
8	412.48	185.62/199.66 (0.93)	150.54/95.85 (1.57)	163.67/56.43 (2.90)

TABLE 2
Computational time (in milliseconds) and HPF/BCL ratio for the 2-D FFT problem

Array Size	Sequential	HPF vs. BCL (ratio)		
		4 Processors	8 Processors	16 Processors
32×32	1.507	0.947/0.595 (1.59)	0.987/0.475 (2.08)	1.601/0.921 (1.74)
64×64	5.165	2.189/1.995 (1.09)	1.778/1.082 (1.64)	2.003/1.095 (1.83)
128×128	20.536	7.238/7.010 (1.03)	5.056/4.081 (1.24)	4.565/2.905 (1.57)

TABLE 3
Computational time (in hours) and HPF/BCL ratio for the non-linear reaction-diffusion equations

Grid Sizes	Sequential	HPF vs. BCL (ratio)		
		5 Processors	9 Processors	16 Processors
64/32/64	0.21	0.28/0.16 (1.75)	0.31/0.14 (2.21)	0.29/0.13 (2.23)
128/64/128	2.07	1.34/1.05 (1.28)	1.16/0.67 (1.73)	1.05/0.54 (1.94)
256/128/256	21.12	11.14/11.88 (0.94)	8.88/7.14 (1.24)	6.87/4.31 (1.59)

9 processors, the mapping is 4/1/4 and for 16 processors it is 7/2/7. Note that when 5 processors are used, the one executing the domain `m` is idle most of the time since its total number of grid points is 1/4 of the other two domains. However, BCL offers a better performance than HPF except for a big problem where, as it is well-known, HPF shows better results. In all the other tested cases, BCL is better than HPF.

5. CONCLUSIONS

BCL, a Border-based Coordination Language, has been proposed for the solution of numerical problems, providing a simple, easy to use and learn parallelism model. The application programmer does not need to learn different languages to codify the whole problem solution. As coordination paradigm claims, the programmer can implement the different parts of his/her problem in an independent way. The glue needed to join these parts is a coordinator process, which is responsible for the definition of the domains, their borders, the functions employed to update these borders, processor and data layout, and the convergence criteria. This way, the coordinator code can be re-used to solve other problems with the same geometry, independently of the physics of the problem and the numerical methods employed.

On the other hand, the worker processes can also be re-used with independence of the geometry. The approach achieves the integration of task and data parallelism in a clear, elegant and efficient way. By means of some examples, we have shown the suitability and expressiveness of the language. The evaluation of the implementation has shown the efficiency of the model.

REFERENCES

1. Bal, H.E., Haines, M., "Approaches for Integrating Task and Data Parallelism", *IEEE Concurrency*, **6(3)**, 1998, pp. 74–84.
2. Barret, R., Berry, M., Chan, T.F., Demel, J., Donato, J., Dongarra, J., Eijhout, V., Pozo, R., Romine, C., van der Vorst, H., "Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods", *SIAM*, 1993.
3. Brandes, T., "ADAPTOR Programmer's Guide", *Technical documentation*, GMD-SCAI, Germany, 1999, `ftp://ftp.gmd.de/GMD/adaptor/docs/pguide.ps`.
4. Carriero, N., Gelernter, D., "Coordination Languages and their Significance", *Communications of the ACM*, **35(2)**, 1992, pp. 97–107.
5. Chapman, B., Haines, M., Mehrotra, P., Zima, H., Rosendale, J., "Opus: A Coordination Language for Multidisciplinary Applications", *Scientific Programming*, **6(2)**, 1997, pp. 345–362.
6. Chassin de Kergommeaux, J., Hatcher, P.J., Rauchwerger, L., (Eds.) "Parallel Computing for Irregular Applications", *Parallel Computing*, **26(13-14)**. 2000.
7. Díaz, M., Rubio, B., Soler, E., Troya, J.M., "BCL: A Border-based Coordination Language", *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, Las Vegas, Nevada, 2000, pp. 753–760.
8. Díaz, M., Rubio, B., Soler, E., Troya, J.M., "Integration of Task and Data Parallelism: A Coordination-Based Approach", *International Conference on High Performance Computing (HiPC'2000)*, LNCS vol. 1970, Springer-Verlag. Bangalore, India, 2000, pp. 173–182.
9. Dinda, P., Gross, T., O'Hallaron, D., Segall, E., Stichnoth, J., Subhlok, J., Webb, J., Yang, B., "The CMU task parallel program suite", *Technical Report CMU-CS-94-131*, School of Computer Science, Carnegie Mellon University, 1994.
10. Fink, S.J., Kohn, S.R., Baden, S.B., "Efficient Run-time Support for Irregular Block-Structured Applications", *J. of Parallel and Distributed Computing*, **50(1-2)**, 1998, pp. 61–82, doi:101006/jpdc.1998.1437.
11. Foster, I., Kohr, D., Krishnaiyer, R., Choudhary, A., "A library-based approach to task parallelism in a data-parallel language", *J. of Parallel and Distributed Computing*, **45(2)**, 1997, pp. 148–158, doi:101006/jpdc.1997.1367.
12. High Performance Fortran Forum, "High Performance Fortran Language Specification version 2.0", 1997.
13. Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M., "The High Performance Fortran Handbook". *MIT Press*, 1994.
14. Merlin, J.H., Baden, S. B., Fink, S. J. and Chapman, B. M., "Multiple data parallelism with HPF and KeLP", in Sloot, P., Bubak, M., Hertzberger, R. (eds.) HPCN'98. *Lecture Notes in Computer Science, Vol. 1401*, Springer-Verlag, 1998, pp. 828–839.
15. Orlando S., Perego, R., "$COLT_{HPF}$ A Run-Time Support for the High-Level Coordination of HPF Tasks", *Concurrency: Practice and experience*, **11(8)**, 1999, pp. 407–434.
16. Orlando S., Palmerini, P., Perego, R., "Mixed Data and Task Parallelism with HPF and PVM", *CLUSTER COMPUTING: The Journal of Networks, Software and Applications*, Baltzer Science Publishers, in press.
17. Ramos, J.I., Soler, E., "Domain Decomposition Techniques for Reaction Diffusion Equations in Two-Dimensional Regions with Re-entrant Corners", *Applied Mathematics and Computation*, **118(2-3)**, 2001, pp. 189–221.
18. Rauber, T., Rünger, G., "A Coordination Language for Mixed Task and Data Parallel Programs", *14th Annual ACM Symposium on Applied Computing (SAC'99), special Track on Coordination Models.* ACM Press, San Antonio, Texas, 1999, pp. 146–155.

19. Smith, B., Bjørstard, P., Gropp, W., "Domain Decomposition. Parallel Multilevel Methods for Elliptic P.D.E.'s", *Cambridge University Press*, 1996.

20. Subhlok, J., Yang, B., "A New Model for Integrated Nested Task and Data Parallel Programming", *6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'97)*, Las Vegas, Nevada, 1997, pp. 1–12.