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

Manuel Díaz, Bartolomé Rubio, Enrique Soler, and José M. Troya

Departamento Lenguajes y Ciencias de la Computación, Málaga University, 29071 Málaga, Spain
E-mail: mdr@lcc.uma.es, tolo@lcc.uma.es, esc@lcc.uma.es, troya@lcc.uma.es

Received February 9, 2001; revised October 15, 2001; accepted November 19, 2001

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 such as 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 applications [6]; multidisciplinary optimization problems like aircraft design [5], etc. For these applications, rather than having a single data-parallel program, it is

1 This work was supported by the Spanish project CICYT TIC-99-0754-C03-03.
more appropriate to subdivide the whole computation into several data-parallel pieces, where these run concurrently and cooperate, thus exploiting task parallelism.

Integration of task and data parallelism is currently an active area of research and several approaches have been proposed [11, 12, 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 the programmer 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, 15, 18] 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.

Our approach is focused on the solution of numerical problems, especially those with an irregular surface that can be decomposed into regular, block structured domains. As the major communication among different domains is carried out through their borders, we have named it BCL (border-based coordination language). It has been successfully used on the solution of domain decomposition-based problems and multiblock 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].

BCL 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 user is not in the habit of programming 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 or 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 how 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 it so 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 layouts.
The knowledge at the coordination level of the 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 intertask 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 present some experimental results.

The rest of the paper is structured as follows. In Section 1.1 some related works are sketched. Section 2 presents BCL. The use and expressiveness of the language are 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 kinds of applications. However, some HPF 2.0 implementations overcome this limitation by means of a specific library [3]. Different 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 that 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 intertask communication schedule is established at compilation time. This is done using the information provided by the instructions at the coordination level related to the interdomain connections and data distribution.
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 a 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 interblock 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 intertask cooperation takes place by means of remote method invocations. Different 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 [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 intertask communication. This commu-
nication is achieved by means of typed channels. The data distributed among the pro-
cessors assigned to each task are communicated through these channels. Unlike in
our approach, the intertask 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 this limitation to be overcome.

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 multiblock 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 nonlinear 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 nonoverlapping techniques [17].
- The geometry of the problem, which may be complex and irregular.

\[
\text{For 1 Time Step: REPEAT}
\text{Solve } u
\text{Solve } v
\text{Borders Interchange}
\text{Solve Borders}
\text{UNTIL Convergence}
\]

FIG. 1. A typical domain decomposition application scheme.
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).

### 2.1. A BCL Program Scheme

Figures 2 and 3 show a typical BCL program scheme. The coordinator process (Fig. 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.

```plaintext
program program_name
DOMAIN declarations
CONVERGENCE declarations
PROCESSORS declarations
....
DOMAINS definitions
DISTRIBUTION information
BORDERS definitions
....
Processes CREATION
end
```

**FIG. 2.** A coordinator process scheme.

```plaintext
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.
• 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 (Fig. 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

\[
\text{DOMAIN}xD \ u,
\]

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

\[
u = (/1, 1, Nx, Ny/)
\]

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 \(\text{DOMAIN2D}\)

\[
u(Nx, 1, Nx, Ny) <\rightarrow v(2, 1, 2, Ny)
\]

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 \(<\rightarrow\) must be equal (although not their shapes). It is allowed to apply a function at the right hand side of the operator \(<\rightarrow\) in which several domains can be implied (or a region of them). 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,

\[
\text{PROCESSORS p}(4,4)
\]

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.

— DISTRIBUT 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,

\[
\text{DISTRIBUT } u(\ast, \text{BLOCK}) \text{ ONTO } p.
\]

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,

\[
\text{CONVERGENCE } c \text{ OF num},
\]

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

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

\[
\text{CREATE processName (u,c,...) ON p},
\]

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

\[ \text{REAL, GRID2D :: g} \]

declares a variable that contains a domain, \( g\%\text{DOMAIN} \), and an array of real numbers, \( g\%\text{DATA} \), 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 cannot 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 \( g_1 \) with GRID attribute can be assigned to another variable \( g_2 \) of the same type (\( g_2 = g_1 \)) if they have the same domain size or if \( g_2 \) has no DOMAIN defined yet. In the latter case, the following steps would be automatically executed:

1. The copying of the \text{DOMAIN} field.
2. Dynamic creation of the field \( g_2\%\text{DATA} \) with enough space to store the data for its domain.
3. The copying of the data stored in the field \( g_1\%\text{DATA} \).

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

\[ !\text{hpf}\$\text{distribute}(*,\text{BLOCK}) :: g. \]

Note that this is a special kind of distribution since it produces the distribution of the field \text{DATA} and the replication of the field \text{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

\[ \text{PUT\_BORDERS} (g), \]

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

\[ \text{GET\_BORDERS} (g) \]

is introduced. The process that calls this instruction will suspend its execution until the data needed to update all the borders associated to \( g\%\text{DOMAIN} \) are received. If a function has been defined at the right hand side of the operator \(<-\), it will be called. \text{PUT\_BORDERS} and \text{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

\[
\text{CONVERGE} (c, \text{vble}, \text{procName}),
\]

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

### 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 \(\text{DOMAIN}\) variable is allowed as an array index. There are also other primitives to manage domains and borders: \(\text{GROW}\), \(\text{INTERSECTION}\), \(\text{DECOMPOSE}\), and \(\text{ARGUMENTS}\). The former is a function to increase or decrease the region of a domain received as dummy argument. The two following 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

```plaintext
1) program example1
2) DOMAIN2D u, v
3) CONVERGENCE c 0F 2
4) PROCESSORS p(4,4), p2(2,2)
5) DISTRIBUTE u (BLOCK,BLOCK) ONTO p1
6) DISTRIBUTE v (BLOCK,BLOCK) ONTO p2
7) u = (/1,i,Nxu,Nyu/)
8) v = (/1,i,Nxv,Nvy/)
9) u(Nxu,Nyi,Nxu,Ny2) <- v(2,1,2,Nyv)
10) v(1,1,1,Nyv) <- u(Nxu-1,Nyi,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.
```
five points. Although this is 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.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{in } \Omega, \quad (1)$$

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

$$u = g \quad \text{in } \partial \Omega. \quad (2)$$

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 Cartesian coordinates; i.e., the domain assigned in line 7 is a rectangle that covers the region from point $(1,1)$ to $(Nx_u, Ny_u)$.

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 $v$ to refresh part of the column $Nx_u$ 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 $Nx_u-1$ 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.

FIG. 5. Communication between two HPF tasks.
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, \( g\%\text{DOMAIN} \), and an array of double precision numbers, \( g\%\text{DATA} \), 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 \( \text{DATA} \) and the replication of the field \( \text{DOMAIN} \) 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 \( g\%\text{DATA} \) to \( g\_old\%\text{DATA} \) will be carried out.

Lines 10 and 11 are the first in which communication is achieved. The instruction \( \text{PUT\_BORDERS}(g) \) in line 10 causes the data from \( g\%\text{DATA} \) needed by the other task (see instructions 9 and 10 in the coordinator process) to be sent. In order to receive the data needed to update the border associated to the domain belonging to \( g \), the instruction \( \text{GET\_BORDERS}(g) \) 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 \( \text{CONVERGE}(c, \text{error}, \text{maxim}) \) causes a communication between the two tasks that share the variable \( c \). In this case, the maximum of the values of the variable \( \text{error} \) contained by each worker is finally assigned to the variable \( \text{error} \). In general, this instruction is used when an application needs a reduction of a scalar value.
subroutine initGrid ( g )
   double precision, GRID2D :: g
   !hpf$ inherit g
   DOMAIN2D interior
   g%DATA = 1.0
   interior = GROW (g%DOMAIN, -1)
   g%DATA (interior) = 0.0
end

subroutine computeLocal (g, g_old)
   double precision, GRID2D :: g, g_old
   !hpf$ inherit :: g, g_old
   call jsrelax(g%DATA, g_old%DATA, ARGUMENTS (g%DOMAIN))
end

subroutine jsrelax (a, a_old, u10, uh0, ul1, uh1)
   integer u10, uh0, ul1, uh1
   double precision, dimension(u10:uh0, ul1:uh1) :: a, a_old
   !hpf$ inherit :: a, a_old
   integer i, j
   forall (j = ul1 + 1:uh1 -1, i = u10 + 1:uh0 - 1)
      a(i,j) = 1.0 / 4.0 * ( a_old (i-1,j) + &
                           a_old (i+1,j) + a_old (i,j-1) + a_old (i,j+1))
   endforall
end

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

subroutine maxim (result, error, length)
   integer length, i
   double precision result, error (length)
   result = error (1)
   do i = 2, length
      if (result < error (i)) result = error (i)
   enddo
end

FIG. 7. Computational subroutines.
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 $g\%\text{DATA}$ 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 does not need to be modified.

Subroutine j5relax is the one that performs the resolution of the equation, and thus 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 $g\%\text{DATA}$ and $g\_old\%\text{DATA}$, respectively. The last four arguments correspond to those expanded by the macro ARGUMENTS.

Subroutine computeNorm also uses the indexing 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.

![FIG. 8. Another irregular problem.](image-url)
FIG. 9. Another coordinator process for Jacobi's method.

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. 4, 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:

\[ l(Nc1, Nrm1, Nc1, Nrm2) <- m(_, \_). \]

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 11 to 14 could be replaced by:

\[
\text{INTERSECTION}(1,m) \\
\text{INTERSECTION}(m,r)
\]
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, Nnx2, Ny /)
9) r = (/Nnx1,Nym, Nx, Ny /)
10) m = (/Nnx1,0, Nnx2, Nmx /)
11) l(Nmx2, Nym + 1, Nmx2, Ny 1) <- r(_)
12) l(Nmx1+1, Nym, Nmx2-1, Nym) <- m(_)
13) l(Nmx1+1, Nym, Nmx2-1, 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, Nym, Nmx2-1, 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, Nmx2-1, 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.
3.1.2. Functions to Define Borders

Figure 10 shows a T shape domain that has been decomposed into three 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 (Fig. 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 \( \leftarrow \) will be updated when \texttt{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 \texttt{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 \texttt{GET\_BORDERS}(\( g \)) by the worker process makes it wait until the data of the GRIDs that contain \( m \) and \( r \) are ready, the average is calculated, and then \texttt{g\%DATA} is updated. The function defined at the right hand side of the operator \( \leftarrow \) can also be a subroutine that will be implicitly called when executing the instruction \texttt{GET\_BORDERS}. This is an elegant way of specifying more complex functions, such as imposing Neumann or Robin boundary conditions between subdomains [17].

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 \( N \times N \) 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 problem there is no convergence criteria.

![FIG. 12. Array distributions for the 2-D FFT problem.](image-url)
The worker processes are codified in Fig. 14. The stage 1 reads an input element, performs the 1-D transformations, and calls \texttt{PUT\_BORDERS(a)}. The stage 2 calls \texttt{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.

4. IMPLEMENTATION ISSUES AND RESULTS

In order to evaluate the performance of BCL, an implementation has been developed on a cluster of four DEC AlphaServer 4100 nodes interconnected by means of Memory Channel. Each node has four Alpha 21264 (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

![Program text](image)

![Worker processes](image)
communication layer and the public domain HPF compilation system ADAPTOR [3]. No change to the HPF compiler has been needed. Figure 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 (BCLIB), 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), the simple and well-known ping-pong benchmark [11] and a fourth application have been considered.

The benchmark consists of two tasks that exchange an array that is distributed by columns on the first one and by rows on the second. This is repeated for a predetermined number of iterations. It has been coded in both BCL and ADAPTOR (using its characteristics to integrate task and data parallelism) in order to measure the possible overhead introduced by our system. The intertask communication schedule in the ADAPTOR version is established only once at the beginning of the program execution.

The last chosen application is a system of two nonlinear reaction–diffusion equations solved by means of linearized implicit $\Theta$-methods. The equations are

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + S(U)$$  \hspace{1cm} (3)

$$U = \begin{pmatrix} u \\ v \end{pmatrix}$$  \hspace{1cm} (4)

$$S(U) = \begin{pmatrix} -uv \\ uw - \lambda v \end{pmatrix},$$  \hspace{1cm} (5)

where $u$ and $v$ are real functions defined in $\Omega \times [0, T]$ and $\lambda$ 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 (Fig. 8). Only the computational parts are different.

The following initial conditions have been considered (Fig. 16): $u = 1$ and $v = 0$ at the boundaries where Dirichlet conditions have been imposed; $u(x, y, 0) = 1$ and $v(x, y, 0) = e^{-((x-x_{\text{cen}})^2+(y-y_{\text{cen}})^2)}$ 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 $t = 80$.

Tables 1 to 4 show the results for the four applications. The results obtained for the ping-pong benchmark are shown in Table 1. Different array sizes have been considered. The number of iterations has been 500 and the results (in milliseconds) are given per iteration. ADAPTOR is better when only two processors are used, as
it is not necessary to establish the intertask communication scheme and there is no integration of task and data parallelism. When more processors are used, BCL has shown better performance. This is also achieved even for two processors when small arrays are used. So, we can appreciate that BCL provides more expressiveness without losing performance. Although BCL implementation is based on

**TABLE 1**

Computational Time (in Milliseconds) and ADAPTOR/BCL Ratio for Ping-Pong Benchmark

<table>
<thead>
<tr>
<th>Array size</th>
<th>2 Processors</th>
<th>4 Processors</th>
<th>8 Processors</th>
<th>16 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>64×64</td>
<td>1.26/1.22</td>
<td>9.11/8.08</td>
<td>1.21/0.77</td>
<td>1.87/1.14</td>
</tr>
<tr>
<td></td>
<td>(1.03)</td>
<td>(1.13)</td>
<td>(1.58)</td>
<td>(1.64)</td>
</tr>
<tr>
<td>128×128</td>
<td>4.47/5.17</td>
<td>3.61/3.16</td>
<td>3.13/2.54</td>
<td>3.93/3.24</td>
</tr>
<tr>
<td></td>
<td>(0.86)</td>
<td>(1.14)</td>
<td>(1.23)</td>
<td>(1.21)</td>
</tr>
<tr>
<td>256×256</td>
<td>17.3/20.5</td>
<td>13.0/12.5</td>
<td>12.0/10.4</td>
<td>13.5/13.2</td>
</tr>
<tr>
<td></td>
<td>(0.84)</td>
<td>(1.04)</td>
<td>(1.15)</td>
<td>(1.02)</td>
</tr>
<tr>
<td>512×512</td>
<td>110/140</td>
<td>792/521</td>
<td>479/478</td>
<td>575/503</td>
</tr>
<tr>
<td></td>
<td>(0.79)</td>
<td>(1.52)</td>
<td>(1.00)</td>
<td>(1.14)</td>
</tr>
</tbody>
</table>
FIG. 18. Results at $t = 80$ for the nonlinear reaction-diffusion problem.

ADAPTOR, we can obtain better results because the intertask communication is not carried out using ADAPTOR primitives, but directly using MPI and the information given at the coordination level.

Table 2 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 $128 \times 128$ grid in each

<table>
<thead>
<tr>
<th>TABLE 2</th>
<th>Computational Time (in Seconds) and HPF/BCL Ratio for Jacobi’s Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domains</td>
<td>Sequential</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>97.05</td>
</tr>
<tr>
<td>4</td>
<td>188.88</td>
</tr>
<tr>
<td>8</td>
<td>412.48</td>
</tr>
</tbody>
</table>
Table 3 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 decreases, as also happens in other approaches [11]. In this situation HPF performance is quite good and so the integration of task parallelism does not contribute as much.

Table 4 shows the results obtained for the nonlinear reaction-diffusion equations. In this case, the surface is irregular (Fig. 8), the number of (square) domains is one. The program has been executed for 20,000 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 has BCL shown less performance because of the context switch overhead among heavy processes.

### Table 3

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Sequential</th>
<th>HPF vs BCL (ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 Processors</td>
<td>8 Processors</td>
</tr>
<tr>
<td>32 × 32</td>
<td>1.507</td>
<td>0.947/0.595 (1.59)</td>
</tr>
<tr>
<td>64 × 64</td>
<td>5.165</td>
<td>2.189/1.995 (1.09)</td>
</tr>
<tr>
<td>128 × 128</td>
<td>20.536</td>
<td>7.238/7.010 (1.03)</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>Grid Sizes</th>
<th>Sequential</th>
<th>HPF vs BCL (ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 Processors</td>
<td>9 Processors</td>
</tr>
<tr>
<td>64/32/64</td>
<td>0.21</td>
<td>0.28/0.16 (1.75)</td>
</tr>
<tr>
<td>128/64/128</td>
<td>2.07</td>
<td>1.34/1.05 (1.28)</td>
</tr>
<tr>
<td>256/128/256</td>
<td>21.12</td>
<td>11.14/11.88 (0.94)</td>
</tr>
</tbody>
</table>
fixed, and different grid sizes have been considered (for example, in the first row, the grid size for \( l \) is \( 64 \times 64 \), for \( m \) is \( 32 \times 32 \), and for \( r \) is \( 64 \times 64 \)). 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 five processors are used, two of them execute the domain \( l \), two the domain \( r \), and one the domain \( m \); for nine processors, the mapping is 4/1/4 and for 16 processors it is 7/2/7. Note that when five 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 or 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 reused to solve other problems with the same geometry, independent of the physics of the problem and the numerical methods employed. On the other hand, the worker processes can also be reused 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


MANUELDÍAZ received his MS and PhD in computer science from the University of Málaga in 1990 and 1995, respectively. From 1990 to 1995 he was an assistant professor at the Department of Lenguajes y Ciencias de la Computación of the University of Málaga. Since 1995 he has been an associate professor in the same department. He has worked in the areas of distributed and parallel programming and in real-time systems, especially in the areas of software engineering for this kind of systems. He has been member of the Software Engineering Group since its foundation. He has published several papers in international refereed journals and representative congresses in the area.

BARTOLOMÉRUBIO received his MS and PhD in computer science from the University of Málaga in 1990 and 1998, respectively. From 1991 to 2000 he was an assistant professor at the Department of Lenguajes y Ciencias de la Computación of the University of Málaga. Since 2001 he has been an associate professor in the same department. He has worked in the areas of distributed and parallel programming and coordination models and languages. He has also been a member of the Software Engineering Group since its foundation. He has published several papers in international refereed journals and representative congresses in the mentioned areas.

ENRIQUE SOLEReceived his MS and PhD in computer science from the University of Málaga in 1990 and 2001, respectively. From 1995 to 2001 he was an assistant professor at the Department of
Lenguajes y Ciencias de la Computación of the University of Málaga. Since 2001 he has been an associate professor in the same department. He has worked in the areas of distributed and parallel programming, domain decomposition techniques, and coordination models and languages. He has been member of the Software Engineering Group since 1995. He has published several papers in international refereed journals and representative congresses in the mentioned areas.

JOSÉ M. TROYA received his MS and PhD from the University Complutense of Madrid in 1975 and 1980 respectively. From 1980 to 1988 he was an associate professor in that University, and since 1988 he has been a full professor at the Department of Lenguajes y Ciencias de la Computación of the University of Málaga. He has worked on parallel algorithms for optimization problems and on parallel programming and software engineering for distributed systems. He has been the head of the Software Engineering Group since its foundation in 1990. He has directed 11 PhD theses and published more than 15 papers in international refereed journals.