BCL: A Border-based Coordination Language

M. Díaz, B. Rubio, E. Soler, J.M. Troya
Dpto. Lenguajes y Ciencias de la Computación
Málaga University
Málaga 29071, SPAIN

Abstract *In this paper we present BCL, a Border-based Coordination Language for the solution of numerical problems, especially those with an irregular surface that can be decomposed into regular, block structured domains. Using this language, the coordination issues of an application can be clearly separated from the numerical methods, increasing the reusability of both the computational and coordination codes. In the coordination part, the different blocks that form the global domain of the problem and the borders among those blocks are defined, including the way these borders are updated. All the coordination aspects are completely written in BCL. The computational part can be written in Fortran or HPF together with only a few BCL extensions. On the other hand, the integration of task and data parallelism is achieved, exploiting two levels of parallelism: among domains and within them. In addition, BCL implementation requires no change to the runtime system support of the HPF compiler used.*

Keywords: Domain Decomposition, Multiblock Problems, Irregular Problems, Coordination Languages, Task and Data Parallelism Integration.

1 Introduction

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 (PDE) [1]. 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:

```
For 1 Time Step:
REPEAT
    Solve u
    Solve v
    Borders Interchange
    Solve Borders
UNTIL Convergence
```

Figure 1: A typical domain decomposition application scheme

- 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 [2].
- The geometry of the problem 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 [3]) so that, very efficient programs can be obtained with relatively little effort.

Coordination languages [4] are a new 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 means 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. A number of interesting models and languages have been proposed [5][6][7] and applied to the parallelization of computation intensive sequential programs in the field of: simulation of fluids, dynamics systems, parallel and distributed simulation, computer graphics, etc.

On the other hand, several works related with programming languages that take advantage of both task and data parallelism have been done [8][9][10][11]. Integrating the two forms of parallelism cleanly and within a coherent programming model is difficult [12]. 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 [13][14][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.

In this paper we introduce BCL, a coordination language focused on the solution of domain decomposition problems. Moreover, other kinds of problems following a similar resolution scheme to that shown in figure 1 (e.g. multiblock codes) or with a communication pattern based on (sub)arrays interchange (2D FFT, Convolution , solution of PDE by means of the red-black ordering algorithm, etc.) may be defined and solved in an easy and clear way using BCL.

In a BCL program there will be a *coordinator* process and one or several *worker* processes. In the coordination process, the domains of the problem are defined together with their borders, the functions to update these borders and, possibly, the convergence criteria. This process is also responsible for the creation of the processes or tasks that will solve the domains (worker processes).

BCL syntax is based on Fortran 90/HPF. This way, both the coordination and the computational parts can be written in 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 [10]. In addition, an efficient program can be obtained integrating task parallelism among different domains and data parallelism applied to the solution of each domain. Since communications among the different tasks are limited to those specified in the coordination process, and the data distribution belonging to the different tasks is known at the coordination level, an efficient implementation can be achieved. In BCL, unlike in other proposals [9][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 rest of the paper is structured as follows. In the next section, the coordination language BCL is described. In section 3, some preliminary results are mentioned and, finally, in section 4, some conclusions are sketched.

2 The coordination language BCL

Figures 2 and 3 show a typical BCL program scheme. When a program written in BCL is initially executed, only one process (figure 2) is created. This process is responsible for defining the problem domains, the borders that exist among them and the variables needed for the convergence test. If task and data parallelism integration is considered, it will also

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

Figure 2: A coordinator process scheme

specify processor and data layout. Finally, it will spawn the processes (workers) that achieve the computations associated to each domain.

A scheme of a worker process is shown in figure 3. The worker processes are declared as a subroutine 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 Fortran / HPF sentences while the communication and synchronization among worker processes are carried out through the primitives `PUT_BORDERS`, `GET_BORDERS` and `CONVERGE`.

In [16] several examples show the suitability and expressiveness of the language. The language characteristics have been separated according to the kind of process that uses them:

2.1 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

```
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 subroutine subroutine_name
```

Figure 3: A worker process scheme

  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)`.

- 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)` to `(Nx,Ny)` will be updated by the values belonging to the region of `v` delimited by points `(2,1)` and `(2,Ny)`. 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 [16]. 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).

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

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

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

  indicates a square arrangement of 16 processors. 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 to which processors of other tasks. This is achieved at compilation time.

- The coordinator process declares variables of `CONVERGENCE` type for allowing 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 optional clause `ON` is used in order to indicate the HPF processors that will execute the indicated task.

2.2 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[1]. A GRID variable can be assigned to another variable of the same type if they have the same domain size or if the assigned variable has no DOMAIN defined yet. In this case, the following steps would be automatically executed:

1. Copying the DOMAIN field together with the information related to the borders associated to that domain.
2. Dynamic creation of the field DATA of the receiving variable with enough space to store the data for its domain.
3. Copying the data stored in the DATA field.

- 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.

2.3 Additional aspects

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

In order to simplify some instructions, the use of a DOMAIN variable as an array index is allowed. 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 primitives are used to automatically define borders. The latter is a macro that expands the coordinates (separated by commas) that form a domain. It is useful when calling a Fortran subroutine intended to be reused. A more detailed explanation together with some more examples of their utilization can be found in [16].

[1] this is an extension of our language since a dynamic array can not be a field of a standard Fortran 90 record

Table 1: Computational time (in seconds) for Jacobi's method

	HPF vs. BCL		
Dom.	4 Processors	8 Processors	16 Processors
2	40.98/35.26	45.14/25.09	65.22/27.62
4	83.82/54.31	79.99/39.64	106.91/32.15
8	163.95/179.59	147.74/61.36	190.23/47.07

Table 2: Computational time (in seconds) for the non-linear reaction-diffusion equations

	HPF vs. BCL		
Dom.	4 Processors	8 Processors	16 Processors
2	115.27/102.75	87.68/66.35	98.23/59.33
4	246.88/238.09	191.72/106.15	219.71/72.18
8	496.72/564.62	403.43/155.93	460.52/124.32

3 Preliminary results

A prototype has been developed in order to evaluate the performance of BCL. Several examples have been used to test it and the obtained preliminary results have successfully shown the efficiency of the model. Here, we have chosen two of them in order to illustrate the system performance. The first one is Laplace's equation solved by means of Jacobi's method. The second is a system of two non-linear reaction-diffusion equations solved by means of linearized implicit Θ-methods. A detailed explanation of the problem and the employed numerical method can be found in [2]. The linearization yields a large system of linear algebraic equations solved by means of the BiCGstab algorithm [17]. This algorithm is coded in Fortran 77 and has been reused by our system without any modification.

A cluster of 4 nodes DEC AlphaServer 4100 interconnected by means of Memory Channel has been used. Each node has 4 processors Alpha 22164 (300 MHz) sharing a 256 MB RAM memory. The operating system is Digital Unix V4.0D (Rev. 878) and Digital PVM [18] has been used to create and communicate HPF tasks. The implementation is based on source-to-source transformations together with the necessary libraries. No change to the runtime system support of the HPF compiler has been needed.

Table 1 compares the results obtained for Jacobi's method in HPF and in BCL considering 2, 4 and 8 domains with a 128×128 grid each. The program has been executed for 20000 iterations.

Table 2 shows the results obtained for the second problem for 2, 4, and 8 domains too. In this case, each one has a 64×64 grid.

In both examples, BCL offers a much better performance than HPF due to the advantage of integrating task and data parallelism. Only when there are more domains than available processors BCL has shown less performance because of the context change overhead among weight processes.

4 Conclusions

BCL, a Border-based Coordination Language has been proposed for the solution of domain decomposition-based numerical problems. Due to the characteristics of coordination languages, the programmer can implement the different parts of his/her problem in an independent way, so that, programs that solve subdomains may be composed to solve a more complex problem. The glue needed to join these programs is a coordinator process, which is responsible for the definition of the domains, their borders, the functions employed to update the borders 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. Since the domains are solved by processes that execute in parallel, an easy and clear parallelization model is provided. Furthermore, inside each domain data parallelism can be carried out, achieving an easy and elegant way of integrating task and data parallelism. The

evaluation of an initial prototype has shown the efficiency of the model.

References

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

[2] Ramos, J.I., Soler, E. Domain Decomposition Techniques for Reaction Diffusion Equations in Two-Dimensional Regions with Re-entrant Corners. To be published in *Applied Mathematics and Computation.*

[3] Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M. *The High Performance Fortran Handbook.* MIT Press, 1994.

[4] Carriero, N., Gelernter, D. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[5] Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[6] Banâtre, J-P., Le Métayer, D. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[7] Arbab, F., Herman, I., Spilling, P. An Overview of Manifold and its Implementation. *Concurrency: Practice and experience*, 5(1):23–70, 1993.

[8] High Performance Fortran Forum. *High Performance Fortran Language Specification version 2.0*, Jan 1997.

[9] 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):148–158, 1997.

[10] Merlin, J.H., Baden, S. B., Fink, S. J. and Chapman, B. M. Multiple data parallelism with HPF and KeLP. *Lecture Notes in Computer Science*, vol. 1401:828–839. Sloot, P., Bubak, M. and Hertzberger, R. (Eds.). Springer-Verlag, 1998 (HPCN'98, Amsterdam, April 1998).

[11] Hassen, S.B., Bal, H.E. Integrating Task and Data Parallelism Using Shared Objects. In *10th ACM International Conference on Supercomputing*, 317–324, Philadelphia, PA, May 1996.

[12] Bal, H.E., Haines, M. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

[13] Chapman, B., Haines, M., Mehrotra, P., Zima, H., Rosendale, J. Opus: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, 6(2), 1997.

[14] Rauber, T., Rünger, G. A Coordination Language for Mixed Task and Data Parallel Programs. In *13th Annual ACM Symposium on Applied Computing. Special Track on Coordination Models*, ACM Press 146–155, San Antonio, Texas, Feb 1999.

[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):407–434, 1999.

[16] Díaz, M., Rubio, B., Soler, E., Troya, J.M. Using Coordination for Solving Domain Decomposition-based Problems. *Technical Report LCC-ITI 99/14*, Departamento de Lenguajes y Ciencias de la Computación. University of Málaga, 1999.

[17] Barret, R., Berry, M., Chan, T.F., Demmel, 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.

[18] Digital PVM. User Guide. Digital Equipment Corporation. Maynard, Massachusetts, Oct 1997.