Integration of Task and Data Parallelism: A Coordination-Based Approach[⋆]

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

Dpto. Lenguajes y Ciencias de la Computación. Málaga University
29071 Málaga, SPAIN
{mdr, tolo, esc, troya}@lcc.uma.es
http://www.lcc.uma.es

Abstract. This paper shows a new way of integrating task and data parallelism by means of a coordination language. 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. The knowledge of data distribution belonging to the different HPF tasks at the coordination level is the key for an efficient implementation of the communication among them. Besides that, our system implementation requires no change to the runtime system support of the HPF compiler used. We also present some experimental results that show the efficiency of the model.

1 Introduction

High Performance Fortran (HPF) [1] 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 [2], a common computation structure in image processing, signal processing or computer vision; multi-block codes containing irregularly structured regular meshes [3]; multidisciplinary optimization problems like aircraft design[4]. 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 [5][6][7]. Integrating the two forms of parallelism cleanly and within a coherent programming model is difficult [8].

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

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 [4][9][10] 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 [11] 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 [12][13] 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. 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.

In this paper we describe the way BCL 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 a Fortran 90 / 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 [7]. 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 of data distribution belonging to the different HPF tasks at the coordination level is the key for an efficient implementation of the communication and synchronization among them. In BCL, unlike in other proposals [6][10], 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 evaluation of an initial prototype has shown the efficiency of the model. We also present some experimental results.

The rest of the paper is structured as follows. In Sect. 2, by means of some examples, the use of BCL to integrate task and data parallelism is shown. In Sect. 3, some preliminary results are mentioned. Finally, in Sect. 4, some conclusions are sketched.

2 Integrating Task and Data Parallelism Using BCL

Using BCL, the computational and coordination aspects are clearly separated, as the coordination paradigm proclaims. In our approach, an application consists of a coordinator process and several worker processes. The following code shows the scheme of both a coordinator process (at the left hand side) and a worker process (at the right hand side).

```
program program_name            Subroutine subroutine_name (. . .)
DOMAIN declarations             DOMAIN declarations ! dummy
CONVERGENCE declarations        CONVERGENCE declarations ! dummy
PROCESSORS declarations         GRID declarations
. . .                           GRID distribution
DISTRIBUTION information        GRID initialization
DOMAINS definitions             do while .not. converge
BORDERS definitions                 . . .
. . .                               PUT_BORDERS
Processes CREATION                  . . .
end                                 GET_BORDERS
                                    Local computation
                                    CONVERGENCE test
                                enddo
                                . . .
                                end subroutine subroutine_name
```

The coordinator process 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 constitute the different HPF tasks that will solve the problem. Local computations are achieved by means of HPF sentences while the communication and synchronization among worker processes are carried out through some incorporated BCL primitives.

The different primitives and the way BCL is used are shown in the next sections by means of two examples. The explanation is self contained, i.e., no previous knowledge of BCL is required.

2.1 Example 1. Laplace's Equation

The following program 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.

$$\Delta u = 0 \ \ in \ \Omega \tag{1}$$

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:

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

```
 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
```

The domains in which the problem is divided are shown in Fig. 1 together with a possible data distribution and the border between domains. Dot 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. In general, the dimension ranges from 1 to 4. 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 cover the region from point `(1,1)` to `(Nxu, Nyu)`. 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).

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 refreshed by part of the column `Nxu-1` of domain `u`.

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). The language provides useful primitives in order to ease (or even automatically establish) the definition of domains and their (possibly

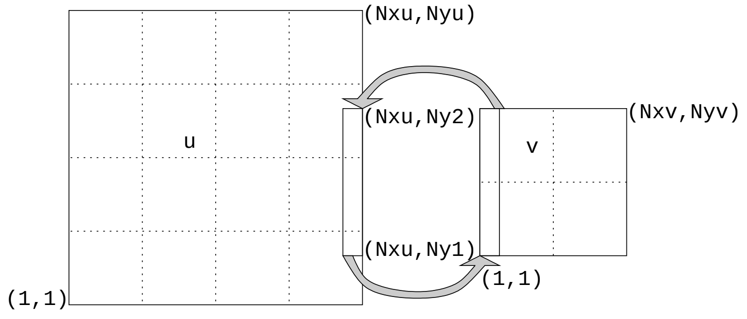

Fig. 1. Communication between two HPF tasks

overlapping) borders (e.g. intersection, shift, decompose, grow). The region sizes at both sides of the operator <- must be equal (although not their shapes). Optionally, a function can be used at the right hand side of the operator that can take as arguments different domains [12].

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. The knowledge of the future data distribution at the coordination level allows a direct communication schedule, i.e., each HPF processor knows which part of its domain has to be sent to each processor of other tasks.

A CONVERGENCE type variable is declared in line 3, which is passed as an 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 criteria. The worker processes receive this variable as a dummy argument. However, when the type of the dummy argument is declared, the clause OF is not specified, as the worker processes do not need to know how many processes are solving the problem. This way, the reusability of the workers is improved (coordination aspects are specified in the coordinator process).

Lines 11 and 12 spawn the worker processes in an asynchronous way so that both HPF tasks are executed in parallel. The code for worker processes is shown in the following program:

```
 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) g_old%DOMAIN = u
 8) call initGrid (g)
 9) do i=1, niters
10)    g_old = g
11)    PUT_BORDERS (g)
12)    GET_BORDERS (g)
13)    call computeLocal (g,g_old)
14)    error = computeNorm (g,g_old)
15)    CONVERGE (c,error,maxim)
16)    Print *, "Max norm: ", error
17) enddo
18) end subroutine solve
```

Lines 2 and 3 declare dummy arguments u and c, 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. This is an extension of our language since a dynamic array can not be a field of a standard Fortran 90 record.

Note that line 5 is a special kind of distribution since it produces the distribution of the field `DATA` and the replication of the field `DOMAIN`.

Statement 10 produces the assignment of two variables with `GRID` attribute. Since `g_old` has its domain already defined, this instruction will just produce a copy of the values of field `g%DATA` to `g_old%DATA`. In general, a variable with `GRID` attribute 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, before copying the data stored in the `DATA` field, a dynamic allocation of the field `DATA` of the receiving variable is carried out.

Lines 11 and 12 are the first where communication is achieved. The instruction `PUT_BORDERS(g)` in line 11 causes that the data from `g%DATA` needed by the other task (see instructions 9 and 10 in the coordinator process) are sent. This is an asynchronous operation. In order to receive the data needed to update the border associated to the domain belonging to `g`, the instruction `GET_BORDERS(g)` is used in line 12. The worker process will suspend its execution until the data needed to update its border are received.

In this example, there is only one border for each domain. In general, if several borders are defined for a domain, `PUT_BORDERS` and `GET_BORDERS` will affect all of them. However, both instructions may optionally have a second argument, an integer number that represents the kind of border that is desired to be "sent" or "received".

Local computation is accomplished by the subroutines called in lines 13 and 14 while the convergence method is tested in line 15. The instruction `CONVERGE` causes a communication between the two tasks that share the variable `c`. In general, this instruction is used when an application needs a reduction of a scalar value.

In order to stress the way our approach achieves the code reusability, Fig. 2 shows another irregular problem that is solved by the following program:

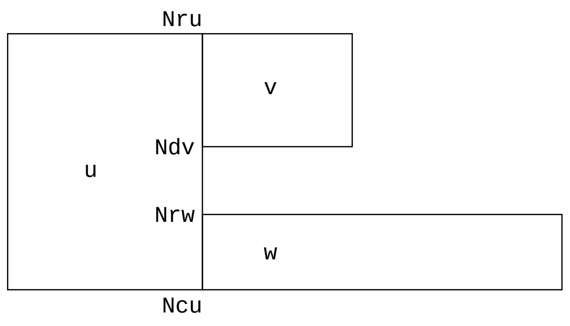

Fig. 2. Another irregular problem

```
 1) program example1_bis
 2) DOMAIN2D u, v, w
 3) CONVERGENCE c OF 3
 4) PROCESSORS p1 (4,4), p2(2,2), p3(2,2)
 5) DISTRIBUTE u (BLOCK,BLOCK) ONTO p1
 6) DISTRIBUTE v (BLOCK,BLOCK) ONTO p2
 7) DISTRIBUTE w (BLOCK,BLOCK) ONTO p3
 8) u = (/1,1,Ncu,Nru/)
 9) v = (/1,1,Ncv,Nrv/)
10) w = (/1,1,Ncw,Nrw/)
11) u (Ncu,1,Ncu,Nrw) <- w (2,1,2,Nrw)
12) u (Ncu,Ndv,Ncu,Nru) <- v (2,1,2,Nrv)
13) v (1,1,1,Nrv) <- u (Ncu-1,Ndv,Ncu-1,Nru)
14) w (1,1,1,Nrw) <- u (Ncu-1,1,Ncu-1,Nrw)
15) CREATE solve (u,c) ON p1
16) CREATE solve (v,c) ON p2
17) CREATE solve (w,c) ON p3
18) end
```

The most relevant aspect of this example is that subroutine `solve` does not need to be modified, it is the same one than in the example before. 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.

2.2 Example 2. 2-D Fast Fourier Transform

2-D FFT transform is probably the application most widely used to demonstrate the usefulness of exploiting a mixture of both task and data parallelism [6][10]. 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.

```
program example2
DOMAIN2D a, b
PROCESSORS p1 (Np), p2(Np)
DISTRIBUTE a (*,BLOCK) ONTO p1
DISTRIBUTE b (BLOCK,*) ONTO p2
a = (/1,1,N,N/)
b = (/1,1,N,N/)
a <- b
CREATE stage1 (a) ON p1
CREATE stage2 (b) ON p2
end
```

In order to increase the solution performance and scalability, a pipeline solution scheme is preferred as proved in [6] and [10]. This mixed task and data parallelism scheme can be easily codified using BCL. The code above 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 coded as follows. 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.

```
subroutine stage1 (d)            subroutine stage2 (d)
DOMAIN2D d                       DOMAIN2D d
complex, GRID2D :: a             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
```

3 Preliminary Results

In order to evaluate the performance of BCL, a prototype has been developed. Several examples have been used to test it and the obtained preliminary results have successfully proved the efficiency of the model [13]. Here, we show the results for the two problems explained above.

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). 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 [14]. No change to 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 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

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)

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 change overhead among weight 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 decreases, as it also happens in other approaches [6]. In this situation HPF performance is quite good and so, the integration of task parallelism does not contribute so much.

4 Conclusions

BCL, a Border-based Coordination Language, has been used for the integration of task and data parallelism. By means of some examples, we have shown the suitability and expressiveness of the language. The clear separation of computational and coordination aspects increases the code reusability. This way, the coordinator code can be re-used to solve other problems with the same geom-

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)	1.601/1.092 (1.47)
64×64	5.165	2.189/1.995 (1.09)	1.778/1.238 (1.44)	2.003/1.095 (1.83)
128×128	20.536	7.238/7.010 (1.03)	5.056/4.665 (1.08)	4.565/3.647 (1.25)

etry, 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 evaluation of an initial prototype by means of some examples has proved the efficiency of the model. Two of them have been presented in this paper.

References

1. Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M.: The High Performance Fortran Handbook. MIT Press (1994)
2. 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)
3. Agrawal, G., Sussman, A., Saltz, J.: An integrated runtime and compile-time approach for parallelizing structured and block structured applications. IEEE Transactions on Parallel and Distributed Systems, **6(7)** (1995) 747–754
4. Chapman, B., Haines, M., Mehrotra, P., Zima, H., Rosendale, J.: Opus: A Coordination Language for Multidisciplinary Applications. Scientific Programming, **6(2)** (1997)345–362
5. High Performance Fortran Forum: High Performance Fortran Language Specification version 2.0 (1997)
6. 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)148–158
7. 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) 828–839
8. Bal, H.E., Haines, M.: Approaches for Integrating Task and Data Parallelism. IEEE Concurrency, **6(3)** (1998) 74–84
9. Rauber, T., Rnger, 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) 146–155
10. 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) 407–434
11. Carriero, N., Gelernter, D.: Coordination Languages and their Significance. Communications of the ACM, **35(2)** (1992) 97–107
12. 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 Computacion. University of Málaga, (1999) `http://www.lcc.uma.es/~tolo/publications.html`
13. 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) 753–760
14. Brandes, T.: ADAPTOR Programmer's Guide (Version 7.0). Technical documentation, GMD-SCAI, Germany. (1999) `ftp://ftp.gmd.de/GMD/adaptor/docs/pguide.ps`