DIP: a Pattern-based Approach for Task and Data Parallelism Integration

Manuel Díaz, Bartolomé Rubio, Enrique Soler, José M. Troya
Dpt. Lenguajes y Ciencias de la Computación
University of Málaga
Campus de Teatinos, 29071-Málaga. SPAIN
Tel: +34 95 2131394
Email: (mdr, tolo, esc, troya@lcc.uma.es)

Keywords
Coordination patterns; task and data parallelism integration; pattern reusability.

ABSTRACT
This paper introduces the DIP (Domain Interaction Patterns) language, which provides a new way of integrating task and data parallelism. Coordination skeletons or patterns are used to express task parallelism among a collection of data parallel HPF tasks. Patterns specify the interaction among domains involved in the application along with the processor and data layouts. The use of domains, i.e. regions together with some interaction information such as borders, improves pattern reusability. The knowledge at the coordination level of data distribution belonging to the different HPF tasks is the key for an efficient implementation of the communication among them. Besides that, our system implementation requires no change to the runtime system support of the HPF compiler used.

1. INTRODUCTION
High Performance Fortran (HPF) [4] has emerged as a standard data parallel, high level programming language for parallel computing. However, a disadvantage of using a 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 applications cannot be efficiently implemented following a pure data-parallel paradigm: pipelines of data parallel tasks, a common computation structure in image processing, signal processing and computer vision; multi-block codes containing irregularly structured regular meshes; multidisciplinary optimization problems like aircraft design. 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 [1]. Integrating the two forms of parallelism cleanly and within a coherent programming model is difficult. 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 [2] to integrate task and data parallelism is proving to be a good alternative [3][5][6], providing a high level mechanism and supporting different forms of task parallelism structures in a clear and elegant way.

In this paper we present DIP, a high level coordination language to express task parallelism among a collection of data parallel HPF tasks, which interact according to static and predictable patterns[*]. DIP allows an application to be organized as a combination of common skeletons, such as multi-blocking or pipelining. Skeletons specify the interaction among domains involved in the application along with the mapping of processors and data distribution. On the one hand, the use of domains, which are regions together with some interaction information such as borders, make the language suitable for 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, Narrowband Tracking Radar, etc.) may be defined and solved in an easy and clear way. The use of domains avoids that some computational aspects involved in the application, such as data types, have to appear at the coordination level, as it occurs in other approaches [5][6]. This improves pattern reusability.

On the other hand, 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 DIP, unlike in other proposals [5], the inter-task communication

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

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. Section 2 introduces DIP. In section 3 some conclusions are sketched.

2. THE DIP LANGUAGE

DIP is a high level coordination language which allows the definition of a network of cooperating *HPF tasks*, where each task is assigned to a disjoint set of processors. Tasks interact according to static and predictable patterns and can be composed using predefined structures, called *patterns* or *skeletons*, in a declarative way. We have initially established two patterns in DIP. The *multi-block* pattern is focussed on the solution of multi-block and domain decomposition-based problems, which conform an important kind of problems in the high performance computing area. The other skeleton provided by DIP is the *pipeline* pattern, which pipelines sequences of tasks in a primitive way.

DIP is based on the use of *domains*, i.e. regions together with some interaction information that will allow efficient inter-task coordination. HPF tasks receive the domains they need and use them to establish the necessary variables for computation. Local computations are achieved by means of HPF sentences while the communication and synchronization among tasks are carried out through some incorporated DIP primitives.

2.1 The MULTIBLOCK Pattern

Domain decomposition methods are successfully being used for the solution of linear and non-linear algebraic equations that arise upon the discretization of partial differential equations (PDEs). 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; overlapping or non-overlapping techniques; the geometry of the problem, which may be complex and irregular and, finally, possible integration of task and data parallelism.

In order to express this kind of problems in an easy, elegant and declarative way, the MULTIBLOCK pattern has been defined. Program 1 shows the general scheme of this pattern.

```
MULTIBLOCK pattern_name  domain definitions
  task_1(domain_1:(data distribution))  processor layout
  task_2(domain_2:(data distribution))  processor layout
  .....
  task_m(domain_m:(data distribution))  processor layout
WITH BORDERS
  border definitions
END
```

Program 1. The MULTIBLOCK pattern.

A *domain definition* is achieved by means of an assignment of Cartesian points, i.e. the region of the domain is established. For example, for the two-dimensional case the expression u/1,1,Nx,Ny/ assigns to the domain u the region of the plane that extends from the point (1,1) to the point (Nx,Ny). In general, a region will consist of 2x numbers, where x is the dimensionality of the problem ($1 \le x \le 4$).

Different *borders* can be defined among the specified domains. For example, if u and v are two-dimensional domains defined as u/1,1,Nxu,Nyu/ and v/1,1,Nxv,Nyv/, the expression u(Nxu,1,Nxu,Nyu) $\leftarrow$ v(2,1,2,Nyv) indicates that the zone of u delimited by points (Nxu,1) and (Nxu,Nyu) will be updated by the values belonging to the zone of v delimited by points (2,1) and (2,Nyv). To apply a function at the right hand side of the operator $\leftarrow$ in which several domains can be implied is allowed. In order to solve some problems (e.g. PDE solution by means of the red-black ordering method), it is better to use several kinds of borders that are communicated in different phases of the algorithm. This way, a border definition can be optionally labeled with a number that indicates the connection type in order to distinguish kinds of borders (or to group them using the same number).

In the *task call* specification, the name of the domain to be solved by the task and *data distribution* are specified. The *processor layout* is also indicated. The distribution types correspond to those of HPF. This declaration does not perform any data distribution but indicates the future distribution of data associated to the specified domain (see section 2.3). The knowledge of data distribution at the coordination level is the key for an efficient implementation of the communication among HPF tasks. A task knows the distribution of its domain and the distribution of every domain with a border in common with its domain by means of the information declared in the pattern. So, it can be deduced which part of the border needs to be sent to which processor of other task. This is achieved at compilation time.

2.2 The PIPE Pattern

The PIPE pattern pipelines sequences of HPF tasks. Figure 2 depicts the structure of the general n-stage pipeline corresponding to the PIPE pattern shown in Program 2.

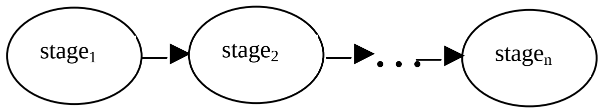

Figure 1. Structure of the n-stage pipeline.

```
PIPE pattern_name [input/output domains]
   stage_1
   stage_2
   .....
   stage_n
END
```

Program 2. The PIPE pattern.

PIPE patterns can be nested. Except for the outer one, i.e. the main PIPE, a list of *input/output domain definitions* appear after the pattern name. One of the predefined words IN, OUT, INOUT must precede each domain definition. For example, the expression INOUT a/1,1,N,N/ specifies a domain a that is considered both an input domain and an output domain for the pipeline. Besides these input/output domains, other “internal” domains can be used. They are defined the first time their name appear.

A *stage* of the pipeline can be one of the following:

- A *pipeline call*, i.e. the name of a nested PIPE pattern together with the domains it needs.
- A *task call*, which has a similar form to task call specification in the MULTIBLOCK pattern. In this case, the task can receive several domains.
- A REPLICATE directive, which is used to replicate a non-scalable stage. This improves the pipeline throughput as different data sets can be computed in parallel on different sets of processors.

2.3 Computational Tasks

A task achieves local computations by means of HPF sentences while the communication and synchronization among tasks are carried out through some incorporated DIP primitives. A new type and a new attribute have also been included. Program 3 shows the general scheme of a task.

```
subroutine task_name(list_of_domains)
  domain declarations
  grid declarations
  grid distribution
  grid initialization
  body code
end subroutine
```

Program 3. A general scheme of a task.

By means of the type DOMAINxD (1≤x≤4), a task declares variables for the received domains. For example, the expression DOMAIN2D u declares the two-dimensional domain variable u.

We use the attribute GRIDxD to declare a record with two fields, the data array and the associated domain. 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.

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

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

The actual data distribution is achieved as in the following example: !hpf$ distribute (*,BLOCK) :: g. Note that this is a special kind of distribution since it carries out the distribution of the field g%DATA and the replication of the field g%DOMAIN.

The body code consists of local computations and communication and synchronization aspects. The data belonging to one task that are needed by another (as defined in the corresponding coordination pattern), are sent by means of the instruction PUT_DATA(g) where g is a variable with GRID attribute. This is an asynchronous operation. On the other hand, in order to receive the data needed by a task, the instruction GET_DATA(g) is introduced. The task that calls this instruction will suspend its execution until the required data are received. PUT_DATA(g) and GET_DATA(g) instructions may optionally have a second argument, an integer number that represents the kind of border that is desired to be "sent" or "received" (see MULTIBLOCK pattern).

When the MULTIBLOCK pattern is used, it is necessary to establish certain communication among tasks in order to determine whether the convergence criteria have been reached. The instruction CONVERGE(g,vble,ProcName) is in charge of this aspect. 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.

The information needed by the three previous instructions to carry out an efficient and transparent communication and synchronization among tasks is stored in g%DOMAIN, as it was declared in the corresponding coordination pattern.

3. CONCLUSIONS

We have presented DIP, a Domain Interaction Pattern-based high level coordination language. The main advantage of this approach is to supply programmers with a concise, pattern-based, high level declarative way to describe the interaction of their HPF tasks. By means of predefined skeletons, the programmer can express task parallelism among a collection of data parallel HPF tasks, so that task and data parallelism integration is achieved. The use of domains and the establishment of data and processor layouts at the coordination level allow pattern reusability and efficient implementations, respectively.

4. REFERENCES

[1] Bal, H.E., Haines, M. Approaches for Integrating Task and Data Parallelism. IEEE Concurrency 6, 3, 74-84, 1998.

[2] Carriero, N., Gelernter, D. Coordination Languages and their Significance. Communications of the ACM 35, 2, 97-107, 1992.

[3] Díaz, M., Rubio, B., Soler, E., Troya, J.M. Integration of Task and Data Parallelism: A Coordination-based Approach. To appear in Proceedings of HiPC'00 (Bangalore, India, December 2000).

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

[5] Orlando S., Palmerini, P., Perego, R. Coordinating HPF Programs to Mix Task and Data Parallelism. In Proceedings of SAC'00 (Vila Olmo, Como, Italy, March 2000), ACM Press, 240-247.

[6] Rauber, T., Rünger, G. A Coordination Language for Mixed Task and Data Parallel Programs. In Proceedings of SAC'99 (San Antonio, Texas, February 1999), ACM Press, 146-155.