Domain Interaction Patterns to Coordinate HPF Tasks ⋆

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

Dpto. Lenguajes y Ciencias de la Computación. Málaga University. 29071 Málaga, SPAIN

Abstract

This paper describes DIP (Domain Interaction Patterns), a pattern-based, high level coordination language, which provides a new way of integrating task and data parallelism. Coordination 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, improves pattern reusability. Data distribution belonging to the different HPF tasks is known at the coordination level. This is the key for both computational code reutilization and an efficient implementation of the communication among tasks. Besides that, our system implementation requires no change to the runtime system support of the HPF compiler used. In addition, a set of different implementation templates are provided in order to ease the programmer task. The suitability, expressiveness and efficiency of the language are shown by means of some examples.

Key words: Coordination patterns, task and data parallelism integration, pattern and computational code reusability

1 Introduction

There has been a tendency in the parallel programming paradigm to ignore high level issues, particularly in the area of programming model and language design. This may be reasonable since performance is the main purpose of parallel programming. Currently, an important effort is being carried out in order to apply structured programming principles to parallel programming.

⋆ This work was supported by the Spanish project CICYT TIC-99-1083-C02-01

* tolo@lcc.uma.es

This is justified by the experience, as real parallel programs rarely consist of random collections of processes interacting in an unpredictable way, but these interactions are usually well structured and fit a set of patterns. In this sense, in [22] a methodology for structured development of parallel software is proposed. This methodology is based on the establishment of a fixed set of patterns or constructors of parallelism, which are the only way to express the parallel structure of the program.

In the same way, the coordination paradigm [8] provides parallel computing with a high level way of facing the development of parallel software. It is based on the separation of coordination and computational aspects. Both coordination and computation are considered the two orthogonal axes that span software space. Coordination models and languages offer a good solution to the problem of managing the interactions among concurrent processes. 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.

A research area that can take advantage of both structured parallel programming and coordination models and languages is the integration of task and data parallelism. This area tries to overcome the constrain that arises when using a pure data parallel language such as High Performance Fortran (HPF) [19]. It is widely accepted that many important parallel applications cannot be efficiently implemented following a pure data parallel paradigm: pipelines of data parallel tasks [15], a common computation structure in image processing, signal processing or computer vision; irregular applications [10]; multidisciplinary optimization problems like aircraft design [9], etc. For these applications, rather than having a single data parallel program, it is more appropriate to subdivide the whole computation into several data parallel pieces, where these run concurrently and co-operate, thus exploiting task parallelism. Although several approaches have been proposed [18][16][27], integrating the two forms of parallelism cleanly and within a coherent programming model is difficult [5]. 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 languages and structured parallel programming is proving to be a good alternative, providing high level mechanisms and supporting different forms of task parallelism structures in a clear and elegant way [24][4][11].

In this paper we describe DIP (Domain Interaction Patterns), a new approach to integrate task and data parallelism using patterns. DIP is 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.

Patterns specify the interaction among domains involved in the application, the relationship between domains and tasks, and the mapping of processors and data distribution. In order to ease the programmer task, a set of different implementation templates are provided. This way, the programmer uses a higher level of abstraction to manage communication and synchronization aspects, and some other low level details can also be avoided in the computational part.

On the one hand, the use of domains (regions together with some interaction information such as borders) eases the solution of numerical problems [13], especially those with an irregular surface that can be decomposed into regular, block structured domains. In this paper, we prove how DIP can be successfully used on the solution of domain decomposition-based problems and multi-block codes. Moreover, we show how other kinds of problems that take advantage of integrating task and data parallelism and have a communication pattern based on (sub)array interchange may also be defined and solved in an easy and clear way. The use of domains also 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 [24][11]. This improves pattern reusability.

On the other hand, the knowledge at the coordination level of data distribution belonging to the different HPF tasks is the key for both computational code reutilization and an efficient implementation of the communication and synchronization among them. In DIP, unlike in other proposals [16][11], 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. In this paper, we also present some implementation issues of a developed initial prototype and analize the efficiency of the model by means of some experimental results.

The rest of the paper is structured as follows. Section 1.1 discusses related work. DIP is described in section 2. In section 3, the model expressiveness and suitability to integrate task and data parallelism are demonstrated by means of some examples. Section 4 discusses the implementation issues and preliminary results and, finally, in section 5, some conclusions are sketched.

1.1 Related work

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

The Fx model [27] 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 data parallel standard HPF 2.0 [18] provides some 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, may be too restrictive for certain application classes. Differently from these proposals, DIP does not need the adoption of new task parallel HPF constructs to express task parallelism. DIP 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 DIP library.

In HPF/MPI [16], the message-passing library MPI has been added to HPF. This definition of an HPF binding for MPI attempts to resolve the ambiguities appeared when a communication interface for sequential languages is invoked from a parallel one. In an HPF/MPI program, each task constitutes an independent HPF program in which one logical thread of control operates on arrays distributed across a statically defined set of processors. At the same time, each task is also one logical process in an MPI computation. In our opinion, the adoption of a message-passing paradigm to directly express task parallelism is too low-level. Moreover, in our approach, the inter-task communication schedule is established at compilation time from the information provided at the coordination level related to the inter-domain connections and data distribution. In this case, expressiveness and good performance are our relative contributions.

Opus [9] is an object-oriented coordination language that has been developed to ease the implementation of scientific and engineering applications that are heterogeneous and multidisciplinary so that they do not fit for the data parallelism paradigm. In Opus, one can define classes of objects, called ShareD Abstractions (SDAs), using a syntax similar to that of HPF. SDAs can be computational servers or data deposits shared among different tasks. Data parallel tasks are dynamically started by creating instances of specific SDAs, while inter-task co-operation takes place by means of remote method invocations. Differently from Opus, our model is focused on the solution of numerical problems, starting and coordinating a statically fixed set of HPF tasks.

The SkIE [4] environment is also focused on heterogeneous and multidisciplinary applications and uses another coordination language (SkIECL) but, in this case, a pattern-based approach is adopted. The basic idea of this integrated heterogeneous environment is to allow the rapid prototyping and development of complex applications on several platforms. This work evolves from P3L [3], a structured parallel programming language that embeds a set of parallel constructs expressing basic parallel paradigms into C. In SkIE, differ-

ent sequential languages for computational tasks have been considered (e.g., C, C++, F77, F90, Java, ...). The user can also encapsulate parallel code using MPI and specialized libraries. In our approach, computational tasks are thought to be HPF tasks instead and, as indicated before, we are focused on numerical problems.

Another work that evolves from P3L is taskHPF [11]. It is possibly the closest proposal to DIP. It is also a high level coordination language to define the interaction patterns among HPF tasks in a declarative way. Considered applications are also structured as ensembles of independent data parallel HPF modules, which interact according to static and predictable patterns. taskHPF provides a pipeline pattern and directives which help the programmer in balancing the pipelined stages: the `ON PROCESSORS` directive fixes the number of processors assigned to an HPF task and the `REPLICATE` directive can be used to replicate non-scalable stages. Patterns can be composed together to build complex structures in a declarative way. Our approach also has a pipeline pattern with similar directives. However, the differences are substancial: a) we work with domains, without considering data types at the coordination level, improving pattern reusability; b) our pattern provides information about the future data distribution together with the processor layout, which allows scheduling the inter-task communication pattern at compilation time. On the other hand, DIP provides a multi-block pattern that makes the language suitable for the solution of domain decomposition-based problems and multi-block codes. Both proposals provide implementation templates. However, DIP improves computational code reutilization as the data distribution is specified in the pattern instead of the task.

The implementation of taskHPF is based on $COLT_{HPF}$ [21], a runtime support specifically designed for the coordination of concurrent and communicating HPF tasks. It is implemented on top of MPI (there is a new version using PVM) and requires small changes to the runtime support of the HPF compiler used. DIP implementation is based on BCL [13][14], 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. BCL is also implemented on top of the MPI communication layer, but no change to the HPF compiler has been needed.

Another coordination language for mixed task and data parallel programs has been proposed in [24]. 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 among 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 easily be 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, it is worthy of remark some others skeletal coordination approaches with goals quite different from those of DIP. In [12], Activity Graphs are defined to provide an intermediate layer for the process of skeletal program compilation, serving as a common, language-independent target notation for the translation from purely skeletal code, and as the source notation for the specific phase of base language code generation. In the former role, they also provide a precise operational semantics for the skeletal layer. In [25], the way the Network Of Tasks model is used to built programs is described . This model is a extremely powerful program composition technique which is both semantically clean and transparent about performance. Software developers can reliably predict the performance of their programs from the knowledge of the performance of the component nodes and the visible graph structure.

2 The DIP coordination 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*, in a declarative way. We have established two patterns in DIP. The *multi-block* pattern is focused 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 pattern provided by DIP is the *pipeline* pattern, which pipelines sequences of tasks in a primitive way. Both should be considered control parallel patterns, following the classification established in the literature [2]. They model parallel computations whose parallel activities come from the computation of different data items. In addition, the pipeline pattern is a stream parallel pattern.

Local computations are achieved by means of HPF sentences while the communication and synchronization among tasks are carried out through some predefined DIP primitives.

The programmer can use a higher level of abstraction to manage communication and synchronization aspects, by means of implementation templates. They also improve computational code reusability. We have established a set of

them in order to solve the kind of problems we have dealt with. The approach allows the programmer to define new templates for his/her applications.

2.1 Domains

DIP is based on the use of *domains*, i.e. regions together with some information established at the coordination level (pattern specification) that will allow efficient inter-task coordination: data distribution, interaction among domains and mapping of processors.

A domain definition is achieved by means of an assignment of Cartesian points, i.e. the region of the domain is established. The expression `u/1,1,Nxu,Nyu/` assigns to the two-dimensional domain `u` the region of the plane that extends from the point `(1,1)` to the point `(Nxu,Nyu)`. In general, a region will consist of $2x$ numbers, where x is the problem dimensionality ($1 \leq x \leq 4$). A domain does not have any associated data type, which improves pattern reusability.

The data distribution types correspond to those of HPF. For example, the expression `u:(*,BLOCK)` declares a distribution by columns for the domain `u`. This declaration does not actually perform any data distribution but only indicates the future distribution of the data that will be bound to the specified domain.

Another information related to domains is the interaction information among domains (borders). For example, in Figure 1, the zone of `u` delimited by points `(Nxu,Ny1)` and `(Nxu,Ny2)` will be updated by the values belonging to the zone of `v` delimited by points `(2,1)` and `(2,Nyv)`. The syntax for specifying borders depends on the kind of pattern.

In addition to data distribution and borders, the relationship between domains and HPF tasks and the processor layout where the tasks are going to be executed are also indicated. Each task is solved by a disjoint set of processors.

The knowledge at the coordination level of all this information related to domains is the key for both computational code reutilization when different data distributions are needed and an efficient implementation of the communication among HPF tasks. The inter-task communication scheme is established at compilation time.

At the computational part, i.e. inside an HPF task, variables for the received domains are declared by means of the type `DOMAIN`x`D` ($1 \leq x \leq 4$). For example, the expression `DOMAIN2D d` declares the two-dimensional domain variable `d`. We use the attribute `GRID(d)` to declare array variables associated to the domain `d`. The expression `double precision,GRID(d)::g` declares a two-

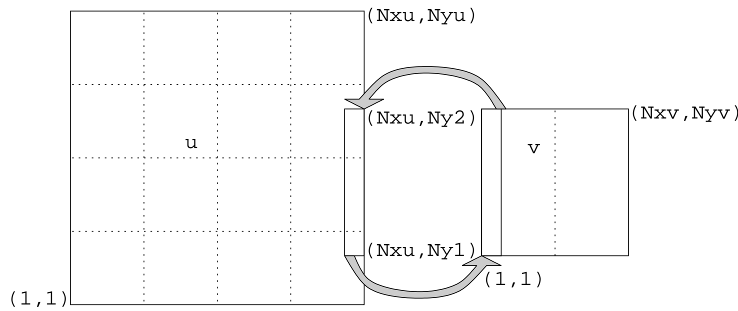

Fig. 1. Communication between two HPF tasks

dimensional array of real numbers, which are dynamically created with the size of the domain region. On the other hand, the actual data distribution is achieved by means of the corresponding HPF directive.

2.2 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) [26]. Programming such applications is a difficult task because we have to take into account many different aspects, such as: the physics of each domain; the different numerical methods applied; 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, the possible integration of task and data parallelism.

In order to express this kind of problems in an easy, elegant and declarative way, the control parallel `MULTIBLOCK` pattern has been defined. It controls the task parallelism among a collection of data parallel activities. The following code shows the general scheme of this pattern:

```
MULTIBLOCK pattern_name domain definitions

 task1(domain1:(data distribution)) processor layout
 task2(domain2:(data distribution)) processor layout
 .....
 taskm(domainm:(data distribution)) processor layout

WITH BORDERS

 border definitions

END
```

The following code shows the `MULTIBLOCK` pattern for the example depicted in Figure 1:

```
MULTIBLOCK example  u/1,1,Nxu,Nyu/, v/1,1,Nxv,Nyv/
  solve(u:(BLOCK,BLOCK)) ON PROCS(4,4)
  solve(v:(*,BLOCK)) ON PROCS(2)
WITH BORDERS
  u(Nxu,Ny1,Nxu,Ny2) <- v(2,1,2,Nyv)
  v(1,1,1,Nyv) <- u(Nxu-1,Ny1,Nxu-1,Ny2)
END
```

The operator `<-` indicates that the region specified at the left hand side will be updated by the data of the region at the right hand side. Optionally, a function that can take as arguments different domains can be used at the right hand side of the operator. 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).

The directive `ON PROCS()` establishes the processor layout. In the example above, the first task is executed on a 4×4 mesh of processors while the second one, on an array of 2 processors.

In Figure 1, dotted lines represent data distribution, and it shows a `(BLOCK, BLOCK)` distribution for `u` and a `(*,BLOCK)` for `v`. 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, the part of the border that needs to be sent to which processor of other task can be deduced.

2.3 The PIPE pattern

The stream parallel `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 the following code:

```
PIPE pattern_name domain definitions
  stage1
  stage2
  .....
  stagen
END
```

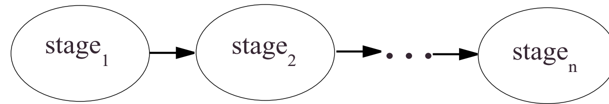

Fig. 2. Structure of the n-stage pipeline

Each stage in the pipeline consumes and produces a data stream, except the first and the last stages that only produces and consumes, respectively. The data stream consists of a number of elements. This number, i.e. the stream length, may or may not be statically known.

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

- A task call, which has a similar form to the task call specification in the `MULTIBLOCK` pattern.
- A pipeline call, i.e. the name of a nested `PIPE` pattern together with the domains it needs.
- A `REPLICATE` directive (also called FARM in the literature) that is used to replicate a non-scalable stage, which can be a task call or a pipeline call. This improves the pipeline throughput as different data sets can be computed in parallel on different sets of processors.

We use the following example to explain the different characteristics of our `PIPE` pattern:

```
PIPE nested_pipe f/1,1,N,N/
  task2(f:(*,BLOCK)) ON PROCS(4)
  task3(f:(BLOCK,*)) ON PROCS(4)
END
PIPE main_pipe d/1,1,N,N/, e/1,1,M,M/
  task1(d:(*,BLOCK)) ON PROCS(2)
  nested_pipe(d)
  task4(d:(BLOCK,*),e:(BLOCK,BLOCK)) ON PROCS(4)
  REPLICATE (2) task5(e:(BLOCK,*)) ON PROCS(2)
  task6(e:(BLOCK,*)) ON PROCS(2)
END
```

Figure 3 shows the graphical scheme of the previous example. The domains involved in each pipe are defined after the pipe name. In our example, the main pipe uses two domains, `d` and `e` and the nested one only requires a domain `f`. Unlike in the `MULTIBLOCK` pattern, the communication between two pipe stages is specified by using the same domain as argument in both stages (the border is the full domain). The stage where this domain first appears is considered the one that generates the data associated to the domain. If the same domain appears in several stages, it means that the intermediate stages,

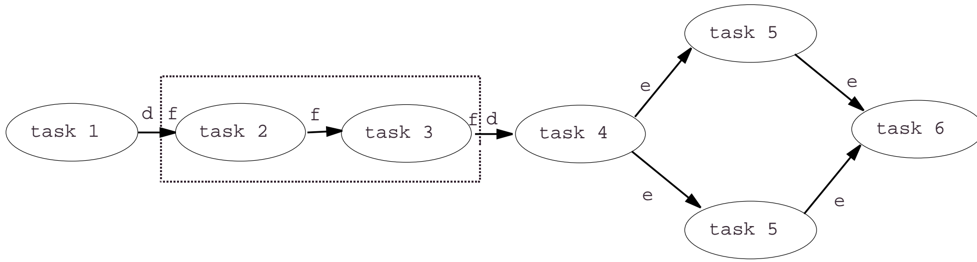

Fig. 3. Structure of the n-stage pipeline

i.e. those different from the first and the last, not only receive data from the previous stage but also send them (usually after some computation) to the following stage. In the example above, `task4` generates the data associated to `e` and sends them to `task5`, which receives them and, after some computation, sends them to `task6`. The data distribution of a domain must be specified in every stage it appears.

In the example above, `task5` is replicated in such a way that two instances of this task are executed on two processors each. This way, `task4` and `task6` must carry out some special work due to the presence of the replicate structure. Thus, `task4` must dispatch the different elements of its output stream to the two instances of `task5`, and `task6` has to collect the elements received from both instances. Different dispatch methods can be carried out, e.g. round-robin, on demand, etc.

Note that `PIPE` patterns allow us a high level description of an application structure. A programmer can completely change this structure by only modifying this concise, simple and high level description, so that several alternative implementations can be tested and evaluated. Unlike in other proposals [11], the absence of input/output directives improves the reutilization of simple, nested and replicated stages.

Finally, recent results [2] demonstrated how arbitrary compositions of stream parallel patterns, including pipelines and farms, can always be rewritten into an equivalent "normal form" pattern composition as farms of sequential compositions, delivering a service time which is equal or even better to the service time of the original pattern composition. These results only consider sequential code tasks/stages. We think that in our case, where tasks are data parallel activities, the first part of these results can be held, i.e., we can obtain a normal form pattern composition as a farm (REPLICATE) of data parallel code composition. However, preliminary experiments carried out on different applications do not predict good results in order to obtain a similar or better service time. Anyway, we are working on these transformation/rewriting aspects.

A task achieves local computations by means of HPF sentences while the communication and synchronization among tasks are carried out through some incorporated primitives. A new type (`DOMAIN`x`D` $(1 \leq x \leq 4)$) and a new attribute (`GRID`) have also been included. The following code 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
```

For example, in the following task:

```
subroutine solve(d)
    DOMAIN2D d
    double precision,GRID(d)::g,g_old
    !hpf$ distribute(BLOCK,BLOCK)::g,g_old
    call initGrid(g)
    do i=1,niters
        g_old = g
        UPDATE_DATA(g)
        call computeLocal(g,g_old)
        error = computeNorm(g,g_old)
        REDUCE(error,maxim)
        Print *, Max norm: , error
    enddo
end subroutine
```

the expression `DOMAIN2D d` declares the two-dimensional domain variable `d`, and the expression `double precision,GRID(d)::g,g_old` declares 2D arrays of real numbers, which are dynamically created with the size of the domain region.

The task body of the example consists of a loop where besides local computations, communication and synchronization aspects are carried out by means of two primitives: `UPDATE_DATA(g)` and `REDUCE`. The data belonging to one task that are needed by another (as defined in the corresponding coordination pattern) are interchanged by means of the instruction `UPDATE_DATA(g)` where `g` is a variable with `GRID` attribute. This instruction actually calls sequentially `PUT_DATA(g)` and `GET_DATA(g)`, two other primitives provided by the model

in order to send and receive data, respectively. The use of these two instructions separately can take advantage of both the implementation of complex communication patterns and the overlapping of communication and computation, which is the key for the efficiency of some applications. `PUT_DATA(g)`, `GET_DATA(g)` and `UPDATE_DATA(g)` instructions may optionally have a second argument, an integer number that represents the kind of border that is desired to be "sent" and/or "received" (see the `MULTIBLOCK` pattern).

When the `MULTIBLOCK` pattern is used, it may be necessary to establish certain communication among tasks in order to determine whether the convergence criteria of a method have been reached. The instruction `REDUCE(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 first argument by means of the subroutine `ProcName`.

2.4.1 Implementation templates

Besides this direct way of codifying the computational tasks, we provide a set of different implementation templates in order to ease the programmer task. In this way, the programmer uses a higher level of abstraction to manage communication and synchronization aspects, and some other low level details can also be avoided.

The implementation templates are known, efficient and parametric ways of implementing patterns on parallel architectures [3] [22]. In our case, a template is a code skeleton of an HPF task cooperating with other tasks according to a fixed interaction pattern. In order to obtain the actual implementation of the tasks, the templates must be instantiated by including programmer-provided code. Templates are parameterized by means of arguments to receive this code. In addition to Fortran predefined types, we have established two new argument types: HPF_DECL and HPF_CODE for variable declarations and HPF code, respectively. The system compiler is in charge of automatically carrying out this instatiation. The template argument types aid the compiler to detect possible errors in the programmer specification.

The next two sections describe the different templates we have initially established. A programmer can add new templates appropriated to the kinds of applications he/she is dealing with.

Multiblock templates

We have defined two implementation templates in order to generate HPF tasks for the `MULTIBLOCK` pattern: the `Elliptic` and the `Parabolic` templates. Both fit the problem of solving PDEs by means of finite difference methods using domain-decomposition, which has been used in order to evaluate our approach.

Elliptic equations, such as Laplace's or Poisson's, are generally associated with equilibrium or steady-state problems. On the other hand, problems involving time as one independent variable lead usually to parabolic equations. Obviously, these templates are not the only ones that can be defined for this kind of problems and more complex templates can be established, for example, those that take into account multigrid, red-black ordering, etc.

Figure 4 shows the DIP specification of the `MULTIBLOCK` pattern together with a task "called" from it that solves a parabolic equation. The programmer will fill the different necessary code sections of the task using HPF code. The figure also depicts the instantiation of the corresponding template. The template

Fig. 4. The Parabolic template and its instantiation from DIP specification

```
TEMPLATE Elliptic(HPF_DECL decl;
                  HPF_CODE init,preupdate,postupdate,results;
                  LOGICAL convergence)
  decl
  init
  do while (.not. convergence)
    preupdate
    UPDATE_DATA(grid)
    postupdate
  enddo
  results
end
```

Fig. 5. The Elliptic template

we show is organized as two nested loops. The outermost is used to evolve in the time variable, the innermost iterates until the convergence conditions among domains are achieved. Inside this convergence loop, the borders among tasks are updated by means of the `UPDATE_DATA` instruction, which receives as argument the grid declared in the task specification and associated to the task domain. The instantiation is carried out by the DIP compiler. The compiler adds some necessary information to the programmer variable declarations and initializations before instantiating the template. From a domain definition, its distribution specified in the `MULTIBLOCK` pattern and the variables declared in the task specification, the compiler generates:

- The domain dimensionality.
- The dimensionality of the grid associated to the domain.
- The distribution of the grid.
- The dynamic allocation of the grid.

The `termination`, `convergence`, `preconverge`, `postconverge`, `preupdate`, `postupdate` and `results` programmer code sections are directly used for the template instantiation.

Note that if the same task is called from the `MULTIBLOCK` pattern but using different data distributions, the task specification will be the same and the compiler will generate different instances from the template. This way, the computational code reusability is improved. On the other hand, a template may be independent of the problem dimensionality (1D, 2D, etc.). The templates for the `PIPE` pattern take advantage of both characteristics as well.

The `Elliptic` template we have defined only requires a loop since, in this case, no time variable is taken into account. In this way, the programmer does not have to provide the `termination`, `preconverge` and `postconverge` code sections. Figure 5 shows this `Elliptic` template. The instantiation process is similar to that shown in Figure 4.

The PIPE pattern establishes a chain of data-flow stages, which consume and produce an input and output data stream, respectively. Thus, the implementation template of a generic stage has to be organized as a loop that receives elements from the input stream, executes some code and sends the resulting data to the output stream.

We have considered two different cases to deal with the stream length. The first one assumes that all the stages know the stream length by means of the number of iterations established by the programmer. In the second one, the stages do not know it, so that a distributed termination protocol has been established associating a termination mark with each element of the stream.

Figure 6 shows the implementation template where the first approach has been considered. DIP specification for the PIPE pattern and for a task "called" from

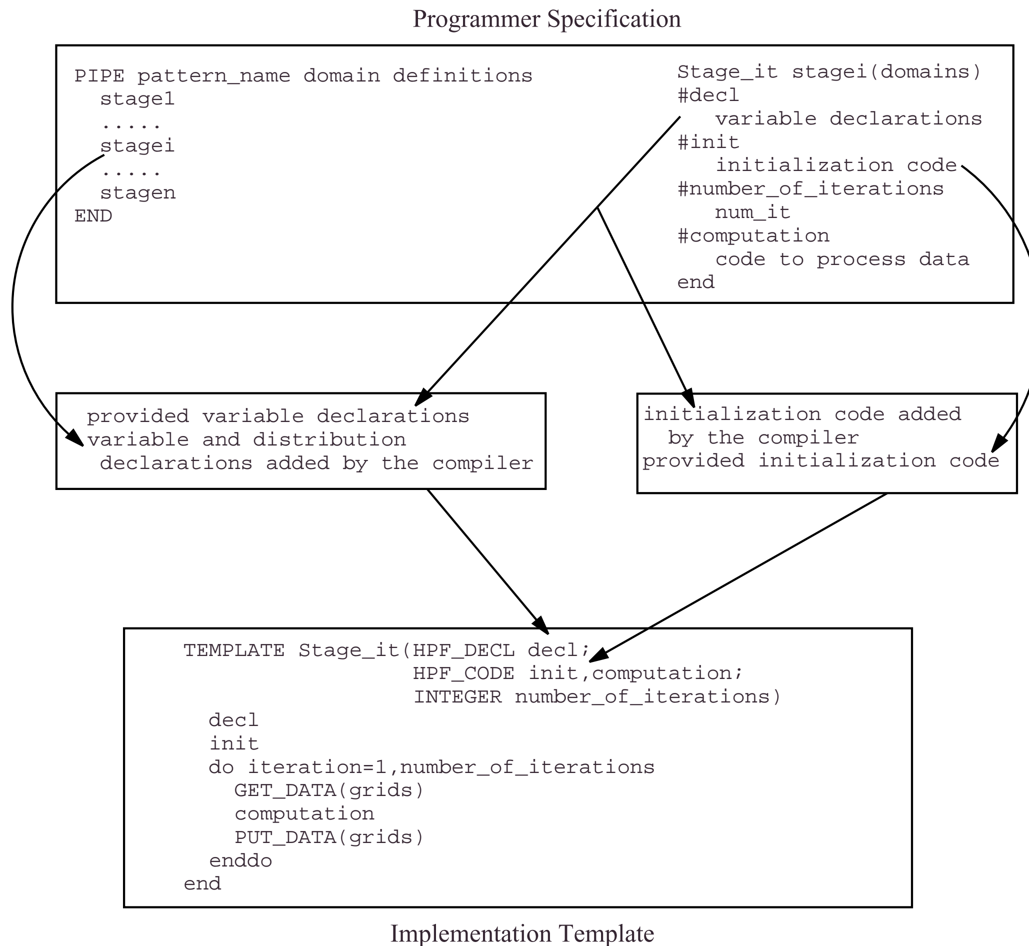

Fig. 6. The Stage template and its instantiation from DIP specification. The stream length is known

```
TEMPLATE Stage_mark(HPF_DECL decl;
                    HPF_CODE init,computation)
  decl
  init
  <receive mark>
  do while (.not. <end_of_stream>)
    GET_DATA(grids)
    computation
    <send mark>
    PUT_DATA(grids)
    <receive mark>
  enddo
  <send end_of_stream>
end
```

Fig. 7. The Stage template when the stream length is not known

it are also depicted together with the instantiation process. Note that we do not need special templates for the first and last stages of the pipeline, since the compiler will deduce how many `PUT_DATA` and `GET_DATA` instructions have to be introduced in the instantiation process from the information described in the pattern and task specification. For example, the stage that generates the data associated to a domain will not have any `GET_DATA` instruction for these data, meanwhile the last stage consuming them will not have any `PUT_DATA` for them.

Figure 7 shows the implementation template where the stream length is not known by the stages. In this case, the task specification does not need the `number_of_iterations` section. The loop is repeated until the `end_of_stream` mark is received. However, with this approach, we need a special template for the first stage of the pipeline, which is in charge of generating the mark. A logical expression must be provided in the task specification in order to control the loop of this first stage.

Finally, when the `REPLICATE` directive is used, the round-robin policy has been considered. This way, when the number of iterations is provided, the compiler automatically decides how many iterations have to be done by each instance of the replicated stage. On the other hand, when the stream length is unknown, the compiler introduces special code for the stages before (emitter) and after (collector) the replicated stage. Thus, when the emitter sends the `end_of_stream` mark at the end of its execution, it must actually send as many marks as instances in the replicated stage. In the same way, the collector must receive the `end_of_stream` marks from all the instances before sending its own mark.

3 Programming examples

3.1 *Example 1. Reaction-diffusion problem*

The first example we have chosen is a system of two non-linear reaction-diffusion equations solved by means of linearized implicit Θ-methods. The origins of this problem are both the propagation of spikes/pulses in biological systems and ignition and flame propagation through combustible mixtures, i.e. one-step reaction where a variable (u in the equations below) represents the fuel and another variable (v) the temperature in simplified chemical kinetic. A detailed explanation of this problem and the employed numerical method can be found in [23]. The linearization yields a large system of linear algebraic equations solved by means of the BiCGstab algorithm [6]. The equations are:

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

$$U = \begin{pmatrix} u \\ v \end{pmatrix}, S(U) = \begin{pmatrix} -uv \\ uv - \lambda v \end{pmatrix} \tag{2}$$

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

Figure 8 shows an irregular geometry of the problem that has been decomposed into three domains (`left`,`center`,`right`). Dotted lines represent data distributions. The domains have been defined taking into account their distribution in the plane. Thus, in the definition of the borders, 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. The `MULTIBLOCK` pattern for this problem is the following:

```
MULTIBLOCK reaction_diffusion   left/0,0,Nxl,Nyl/,
                                center/Nxl-1,Nyc1,Nxc+1,Nyc2/,
                                right/Nxc,0,Nxr,Nyl/
  solve(left:(BLOCK,BLOCK)) ON PROCS(2,2)
  solve(center:(*,BLOCK)) ON PROCS(2)
  solve(right:(BLOCK,BLOCK)) ON PROCS(2,2)
WITH BORDERS
  left(Nxl,Nyc1 ,Nxl,Nyc2) <- center(_)
  center(Nxl-1,Nyc1 ,Nxl-1,Nyc2) <- left(_)
  center(Nxc+1,Nyc1, Nxc+1,Nyc2) <- right(_)
  right(Nxc,Nyc1, Nxc,Nyc2) <- center(_)
END
```

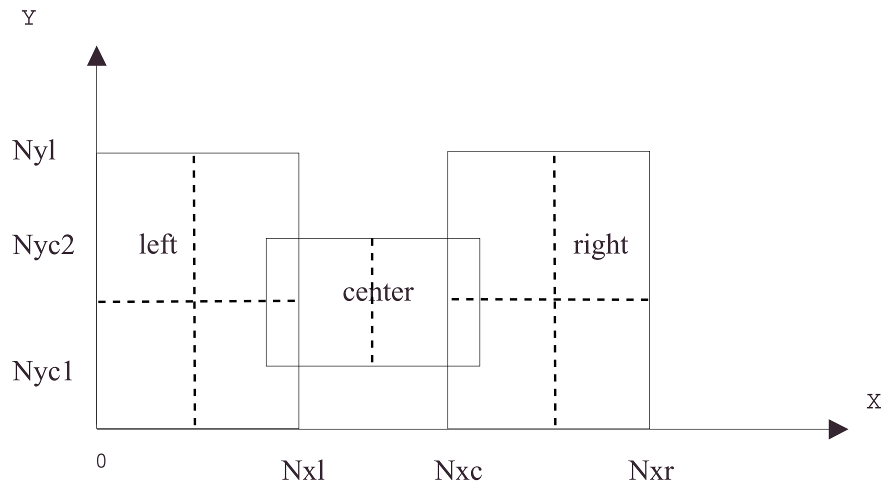

Fig. 8. An irregular geometry.

The DIP specification of the task `solve` is the following.

```
Parabolic solve(d)
#decl
  use BiCG
  real, parameter :: tottime = 80., tstep = 0.2, toler = 1.0e-10
  type vble
     double precision  u,v
  end
  type (vble), GRID(d) :: dg,g,rhs,dg_old
  type (vble), dim(6), GRID(d) :: ab
  real time
  double precision  error
  logical conver
#init
  call initialize(g)
  time = 0.
  dg = 0.
#termination
  time > tottime
#preconverge
  time = time + tstep
  call set_values(ab,rhs,g)
  conver = .false.
#convergence
  conver
#preupdate
  call modify_rhs(rhs,dg)
  call save_borders(dg,dg_old)
```

```
  call BiCG(ab,rhs,dg)
#postupdate
  error = compute_error(dg,dg_old)
  REDUCE(error,maxim)
  if (error < toler) conver = .true.
#postconverge
  g = g + dg
#results
  call show_results(g)
end
```

The grid g in the code above represents the U variable in the equations 1-3. Since this variable consists of two variables u and v, the grid element type is a record with these two variables as fields. The linearization of the problem yields to a system of linear equations where the unknowns are stored in the dg grid and represents ΔU. The matrix of the algebraic linear system is stored in the ab grid. Since this is a sparse matrix, only 6 diagonals have to be taken into account. The right hand side of the linear system is stored in the rhs grid. The last grid, dg_old, is used to compute the local error. When there are more than one grid associated to the domain, the firstly declared (here dg) is the one used to carry out the communication of the domain borders.

From the Parabolic template it can be deduced that the computation evolves until the time variable reaches the tottime value. For each time step, before the convergence loop, the values of ab and rhs are set as a function of g. Inside the convergence loop, before the border communications, first the rhs values corresponding to the domain borders are modified, then the dg values for these borders are saved on dg_old, and finally, the system of linear equations is solved. After the border communications, the local error at the border points is computed and its reduction among all the tasks is performed.

For this problem, the compiler will generate two different instances from the template, one for the (BLOCK,BLOCK) distribution established in the pattern to solve the domains left and right, and another for the (*,BLOCK) distribution specified for the domain center.

Finally, note that the task specification above is reused for three different domains and different distributions. In addition, it could be reused for a different geometry of the global problem. On the other hand, the specified pattern is independent of the computational aspects inside the tasks. This way, it can be reused to solve another different problem with the same geometry.

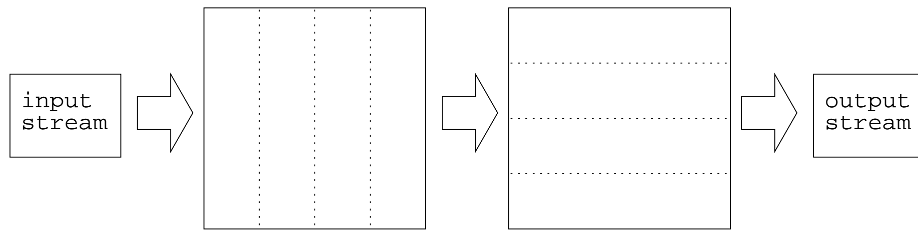

Fig. 9. Array distributions for 2-D FFT.

3.2 Example 2. 2-D Fast Fourier Transform

2-D FFT is probably the most widely used application to demonstrate the usefulness of exploiting a mixture of both task and data parallelism [16][11]. Given an N×N array of complex values, a 2-D FFT entails performing N independent 1-D FFTs on the columns of the input array, followed by N independent 1-D FFTs on its rows. In order to increase the solution performance and scalability, a pipeline solution scheme is preferred as proved in [16] and [11]. Figure 9 shows the array distributions needed for that scheme.

This mixed task and data parallelism scheme can be easily codified using DIP. The following code shows the `PIPE` pattern. A domain `d/1,1,N,N/` is defined for representing the application array at the coordination level. Again, data distribution and processor layout are indicated in the task call specification.

```
PIPE FFT2D d/1,1,N,N/
  cfft(d:(*,BLOCK)) ON PROCS(4)
  rfft(d:(BLOCK,*)) ON PROCS(4)
END
```

The DIP specification for the two stages is the following:

```
Stage_it cfft(d)
#decl
  integer, parameter :: nimages = 500
  complex, GRID(d) :: a
#init
#number_of_iterations
  nimages
#computation
  call read(a)
  call colfft(a)
end

Stage_it rfft(d)
#decl
```

```
  integer, parameter :: nimages = 500
  complex, GRID(d) :: b
#init
#number_of_iterations
  nimages
#computation
  call rowfft(b)
  call write(b)
end
```

The stage `cfft` reads an input element, performs the 1-D column transformations and implicitly calls `PUT_DATA(a)`. The stage `rfft` receives the array by means of the `GET_DATA(b)` instruction, performs the 1-D row transformations and writes the result. In the Stage_it template instantiation phase, the compiler establishes that the stage `cfft` sends the data and the stage `rfft` receives them. Thus, the first one does not need any `GET_DATA(a)` instruction but it needs the `PUT_DATA(a)` primitive after the computation. On the other hand, the second stage only needs the `GET_DATA(b)` instruction before the computation.

An alternative solution that uses nested `PIPE` patterns is given in the following code. The tasks `cfft` and `rfft` shown above must be modified since read/write operations are now carried out by `Input` and `Output` stages.

```
PIPE FFT2D d/1,1,N,N/
  cfft(d:(*,BLOCK)) ON PROCS(4)
  rfft(d:(BLOCK,*)) ON PROCS(4)
END

PIPE Alternative_Solution d/1,1,N,N/
  Input(d:(*,*)) ON PROCS(1)
  FFT2D(d)
  Output(d:(*,*)) ON PROCS(1)
END
```

This solution shows one of the advantages of using nested pipes, since it establishes a more structured scheme, providing a more useful `PIPE FFT2D` pattern, which can be reused in more complex applications, such as Convolution [16] and FFT-Hist [27].

3.3 Example 3. NPB-FT

The third example has been taken from the NAS Parallel Benchmark [20]. It is the Fourier Transform (FT) to solve a 3-D diffusion equation for an unknown

$u(x, y, z, t)$:

$$\frac{du}{dt} = \alpha \nabla^2 u \tag{3}$$

After applying a FT to each side of this equation, the solution is given by the function:

$$f(x, y, z, t) = e^{-4\alpha\pi^2|z|^2 t} v(x, y, z, t = 0) \tag{4}$$

where $v(t = 0)$ is the Discrete Fourier Transform of $u(t = 0)$.

The FT benchmark uses this formula to compute the solution to the diffusion equation for a number of time steps. An HPF implementation is described in [17]. As in the previous example, a pipeline scheme is preferred [1] in order to improve the solution performance by means of the overlapping of communication and computation.

In addition, our solution takes advantage of another level of parallelism as the time step iterations are independent since the result of one iteration is not used for the next one. We achieve this by means of the `REPLICATE` directive.

The `PIPE` pattern for this problem is the following:

```
PIPE NPB_FT d/1,1,1,Nx,Ny,Nz/
  init_evolve(d:(*,BLOCK,*)) ON PROCS(P1)
  FFT_Z(d:(*,BLOCK,*)) ON PROCS(P2)
  REPLICATE (R) FFT_XY(d:(*,*,BLOCK)) ON PROCS(P3)
END
```

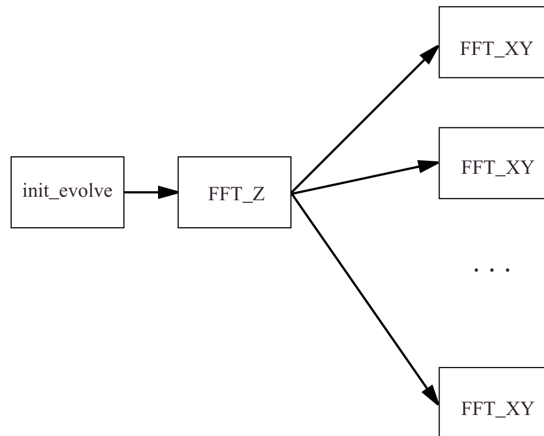

Fig. 10. Computation scheme for the NPB-FT problem.

The computation scheme is shown in Figure 10. The first stage `init_evolve` calculates the initial condition and its Fourier transform. Then, it generates the transformed solutions for different values of `t`. The inverse Fourier transform has to be applied to these transformed solutions in order to solve the problem. This is carried out through two stages. The first one, `FFT_Z`, performs 1-D FFTs along the Z axis. It sends the results to the following stage, `FFT_XY`, which calculates the 1-D FFTs along the X and Y axes and finally checks the results. This last stage has been replicated as mentioned before. The different instances receive the data following a round robin strategy. The following code shows the programmer specification for these three stages. For the sake of simplicity some details have been omitted.

```
Stage_it init_evolve(d)
#decl
  complex, GRID(d) :: w,v,u
  !hpf$ distribute (*,*,BLOCK) :: u
#init
  call init_cond(u)
  call fft3d(u,v)
#number_of_iterations
  T
#computation
  call evolve(v,w,iteration)
end

Stage_it FFT_Z(d)
#decl
  complex, GRID(d) :: w
#init
#number_of_iterations
  T
#computation
  call inverse_fft3d_z(w)
end

Stage_it FFT_XY(d)
#decl
  complex, GRID(d) :: w
#init
#number_of_iterations
  T
#computation
  call inverse_fft3d_xy(w)
  call checksum(w)
end
```

Note that in the `init_evolve` stage, the distribution of the grid `u` is explicitly declared, as it requires a different distribution from that specified for the domain `d` in the pattern.

4 Implementation issues and results

In order to evaluate the performance of DIP, a prototype has been developed on a cluster of 4 DEC AlphaServer 4100 nodes interconnected by means of Memory Channel. 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 [7]. No change to the runtime support of the HPF compiler has been needed. Figure 11 depicts the different phases to obtain the executable code. The DIP compiler translates DIP code into BCL code. As mentioned in section 1.1, BCL is a previous work and it has inspired different aspects of DIP, mainly the `MULTIBLOCK` pattern and the way of specifying computational tasks.

If it is required, the DIP compiler will use the implementation templates. In a BCL program there are one coordinator process and one or several worker

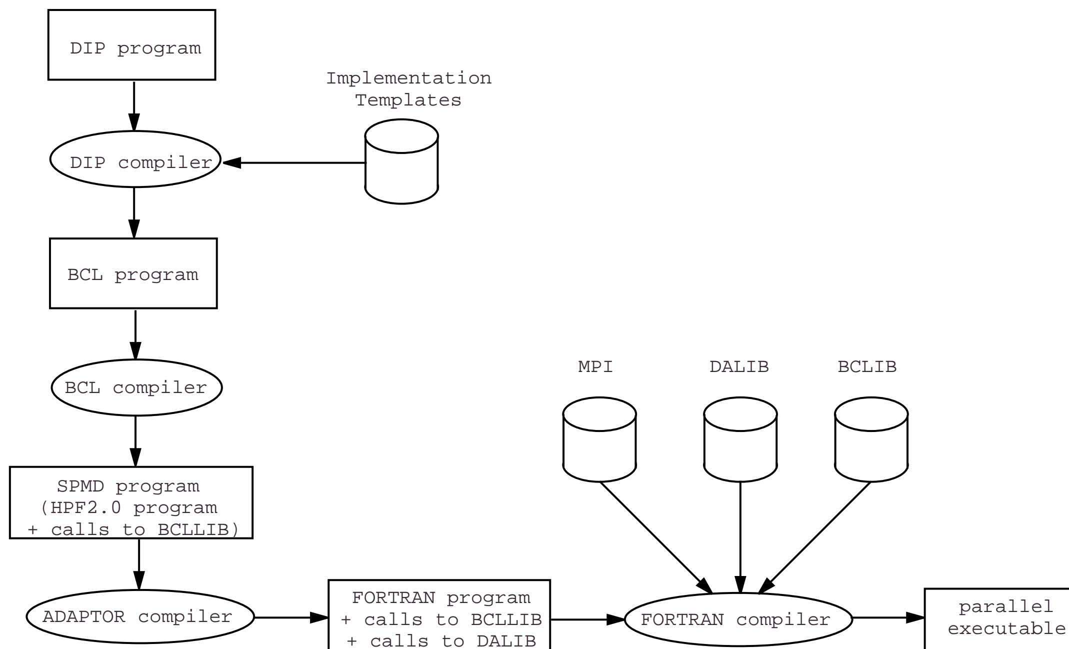

Fig. 11. Implementation scheme

processes. The coordinator process is in charge of establishing all the coordination aspects and creating the worker processes. A worker process is the computational task. The BCL compiler translates the code generated by the DIP compiler into an SPMD program that takes advantage of the `task_region` HPF 2.0 facility, so that the worker processes can be executed on 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 the prototype and the obtained preliminary results have successfully proved the efficiency of the proposal. Here, we show the results for the three examples presented in section 3 and we compare them with respect to HPF.

Table 1 shows the results obtained for the non-linear reaction-diffusion equations. Different grid sizes have been considered (for example, in the first row, the grid size for `left` is 64×64, for `center` is 32×32 and for `right` is 64×64). For both HPF and DIP implementations, 5, 9 and 16 processors have been considered. In the case of HPF, all the processors execute each domain, so a great number of messages is needed to solve each domain. In the case of DIP, when 5 processors are used, 2 of them execute the domain `left`, 2 the domain `right` and 1 the domain `center`; for 9 processors, the mapping is 4/1/4 and for 16 processors it is 7/2/7. These processor layout reduce the number of necessary messages to solve each domain, which implies a better performance of DIP with respect to HPF. Actually, this is a known consequence of integrating task and data parallelism to solve irregular problems [10].

Note that when 5 processors are used, the one executing the domain `center` is idle most of the time since its total number of grid points is 1/4 of the other two domains. However, in this case, DIP also offers a better performance than HPF except when the problem size becomes larger.

On the other hand, the maximum speedup obtained for this problem is 5 using 16 processors, while HPF got 3.

Table 2 shows the execution time per input array for HPF and DIP implementations of the 2-D FFT application. Results are given for different problem sizes. Again, the performance of DIP is generally better due to the pipeline solution scheme used, which increase both the solution performance and the scalability [16][11].

However, HPF performance is near DIP as the problem size becomes larger and the number of processors decreases, as it also happens in other approaches

Table 1
Computational time (in hours) and HPF/DIP ratio for the non-linear reaction-diffusion equations

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

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

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

[16]. In this situation HPF performance is quite good and so, the integration of task parallelism does not contribute so much.

In this application, DIP obtained a maximum speedup of 7 using 16 processors, while HPF got 4.5.

Finally, tables 3 and 4 show the results obtained for the NPB-FT problem considering different number of instances in the replicated stage (R) and data sizes. Since each stage is executed on at least one processor, the version with R=4 can not be executed on 4 processors. We can observe in table 3 that for small data sizes, DIP is better than HPF. As in the previous examples, the better results are generally for the greater number of processors. On the other hand, as the number of instances grows, the performance improves except in the case where the replication avoids the integration of task and data paral-

Table 3
Computational time (in milliseconds) and HPF/DIP ratio for the NPB-FT problem (array size 32x32x32)

Processors	Sequential	HPF vs. DIP (ratio)		
		DIP R=1	DIP R=2	DIP R=4
4	94.7	37.5/35.4 (1.06)	37.5/35.3 (1.06)	
8	94.7	33.4/23.8 (1.40)	33.4/20.7 (1.61)	33.4/22.6 (1.48)
16	94.7	43.5/37.3 (1.17)	43.5/19.8 (2.20)	43.5/15.0 (2.90)

Table 4
Computational time (in milliseconds) and HPF/DIP ratio for the NPB-FT problem (array size 64x64x64)

Processors	Sequential	HPF vs. DIP (ratio)		
		DIP R=1	DIP R=2	DIP R=4
4	999	339/388 (0.87)	339/480 (0.70)	
8	999	239/207 (1.15)	239/211 (1.13)	239/278 (0.86)
16	999	213/118 (1.80)	213/113 (1.88)	213/148 (1.44)

lelism. For example, with 8 processors and R=4, the replicated stages do not take advantage of data parallelism.

As the data size grows (table 4), HPF performance is better that DIP in the cases where few processors are used or only task parallelism is carried out.

Here, the maximum speedup obtained is nearly 9 for DIP and 5 for HPF, using 16 processors.

5 Conclusions

A new pattern-based approach to integrate task and data parallelism has been proposed. The main advantage of this approach is to supply programmers with a concise, pattern-based, high level declarative way to describe the interaction among HPF tasks. The use of domains and the establishment of data and processor layouts at the coordination level allow pattern reusability and efficient implementations, respectively. Patterns allow a high level description of an application structure and the programmer can modify it by means of simple changes in the patterns. The approach also provides the programmer with implementation templates, so that a higher level of abstraction to manage communication and synchronization aspects and computational code reusability can be achieved. By means of some examples we have shown the expressiveness and suitability of the model. The evaluation of a prototype has also shown the efficiency of the approach.

References

[1] Agarwal, R.C., Gustavson, F.G., Zubair, M., An Efficient Parallel Algorithm for the 3-d FFT NAS Parallel Benchmark, *Proceedings of SHPCC'94* (1994) 129–133.

[2] Aldinucci,M., Danelutto, M., Stream parallel skeleton optimization, *Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems* (MIT, Boston, USA, 1999) 955–962.

[3] Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M., P^{3}L: A Structured High Level Programming Language and its Structured Support, *Concurrency: Practice and Experience*, **7** (1995) 225–255.

[4] Bacci, B., Danelutto, M., Pelagatti, S., Vanneschi, M., SkIE: A Heterogeneous Environment for HPC Applications, *Parallel Computing*, **25(13-14)** (1999) 1827–1852.

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

[6] Barret, R., Berry, M., Chan, T.F., Demel, J., Donato, J., Dongarra, J., Eijhout, V., Pozo, R., Romine, C., van der Vorst, H., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM, 1993).

[7] Brandes, T., ADAPTOR Programmer's Guide, *Technical documentation*, GMD-SCAI, Germany (1999).

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

[9] Chapman, B., Haines, M., Mehrotra, P., Zima, H., Rosendale, J., Opus: A Coordination Language for Multidisciplinary Applications, *Scientific Programming*, **6(2)** (1997) 345–362.

[10] Chassin de Kergommeaux, J., Hatcher, P.J., Rauchwerger, L., eds., Parallel Computing for Irregular Applications, *Parallel Computing*, **26(13-14)** (2000).

[11] Ciarpaglini, S., Folchi, L., Orlando, S., Pelagatti, S., Perego, R., Integrating Task and Data Parallelism with taskHPF, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)* (CSREA Press, Las Vegas, Nevada, 2000) 2485–2492.

[12] Cole, M., Zavanella, A., Activity Graphs: A Model-Independent Intermediate Layer for Skeletal Coordination, *Proceedings of the 15th Annual ACM Symposium on Applied Computing (SAC'00), Special Track on Coordination Models* (ACM Press, Villa Olmo, Como, Italy, 2000) 255–261.

[13] Díaz, M., Rubio, B., Soler, E., Troya, J.M., BCL: A Border-based Coordination Language, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)* (CSREA Press, Las Vegas, Nevada, 2000) 753–760.

[14] Díaz, M., Rubio, B., Soler, E., Troya, J.M., Integration of Task and Data Parallelism: A Coordination-based Approach, *Proceedings of the International Conference on High Performance Computing (HiPC'2000)* (Bangalore, India, 2000), volume 1970 of *LNCS*, Springer-Verlag 173–182.

[15] 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).

[16] 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.

[17] Frumkin, M., Jin, H., Yan, J., Implementation of NAS Parallel Benchmark in High Performance Fortran, *Technical Report NAS-98-009*, NASA Ames Research Center. Moffett Field, California (1998).

[18] High Performance Fortran Forum, High Performance Fortran Language Specification version 2.0 (1997).

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

[20] Numerical Aerodynamic Simulation. NAS Parallel Benchmark, NPB version 2.3. NASA Ames Research Center. Moffett Field, California (2000).

[21] 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.

[22] Pelagatti, S., *Structured Development of Parallel Programs* (Taylor&Francis, London, 1997).

[23] Ramos, J.I., Soler, E., Domain Decomposition Techniques for Reaction Diffusion Equations in Two-Dimensional Regions with Re-entrant Corners, *Applied Mathematics and Computation*, **118(2-3)** (2001) 189–221.

[24] Rauber, T., Rünger, G., A Coordination Language for Mixed Task and Data Parallel Programs, *Proceedings of the 14th Annual ACM Symposium on Applied Computing (SAC'99), Special Track on Coordination Models* (ACM Press, San Antonio, Texas, 1999) 146–155.

[25] Skillicorn, D., Pelagatti, S., Building Programs in the Network Of Tasks Model, *Proceedings of the 15th Annual ACM Symposium on Applied Computing (SAC'00), Special Track on Coordination Models* (ACM Press, Villa Olmo, Como, Italy, 2000) 248–254.

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

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