Integrating Task and Data Parallelism by means of Coordination Patterns*

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, by means of some examples, the suitability and expressiveness of a pattern-based approach to integrate task and data parallelism. Coordination skeletons or patterns 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. On the one hand, the use of domains, i.e. regions together with some interaction information, 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 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) [13] 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 [10], a common computation structure in image processing, signal processing or computer vision; multi-block codes containing irregularly structured regular meshes [1]; multidisciplinary optimization problems like aircraft design [5]. 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 [12][11][19]. Integrating the two forms of parallelism cleanly and within a coherent programming model is difficult [2]. In general, compiler-based approaches are limited in terms of the

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

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 [4] and structured parallel programming [15] 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 [16][6].

In [9] we presented DIP (Domain Interaction Patterns), a new approach of integrating task and data parallelism using skeletons. 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. It 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. In this paper, we prove how it can be successfully used on the solution of domain decomposition-based problems and multi-block codes. Moreover, we show how 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 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 [16][6]. 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 an efficient implementation of the communication and synchronization among them. In DIP, unlike in other proposals [11][6], 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 confirm 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. Section 2 gives an overview of DIP. In section 3, the expressiveness and suitability of the model to integrate task and data parallelism is 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 integration of task and data parallelism. We shall state a few of them and discuss the relative contributions of our approach.

The Fx model [19] expresses task parallelism by providing declaration directives to partition processors into subgroups and execution directives to assign computations to different subgroups (task regions). These task regions can be dynamically nested. The new standard HPF 2.0 [12] of the data parallel language HPF provides approved extensions for task parallelism, which allow nested task and data parallelism, following a similar model to that of Fx. These extensions allow the spawning of tasks but do not allow interaction like synchronization and communication between tasks during their execution and therefore might 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 [11], 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 HPF 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.

Another coordination language for mixed task and data parallel programs has been proposed in [16]. The model provides a framework for the complete derivation process in which a specification program is transformed into a coordination program. The former expresses possible execution orders between modules and describes the available degree of task parallelism. The latter describes how the available degree of parallelism is actually exploited for a specific parallel implementation. The result is a complete description of a parallel program that can 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.

Possibly the closest proposal to DIP is taskHPF [6]. It is also a high level coordination language to define the interaction patterns among HPF tasks in a declarative way. Applications considered 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: `ON PROCESSORS` directive fixes the number of processors assigned to an HPF task and `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 has also a pipeline pattern with similar directives. However, the differences are substancial: a) we work with domains, without considering data types at coordination level, which can improve pattern reusability; b) our pattern do not force the utilization of ending marks, such as `END_OF_STREAM`, in the computational part, i.e. inside an HPF task; c) 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 make the language suitable for the solution of domain decomposition-based problems and multi-block codes.

The implementation of taskHPF is based on $COLT_{HPF}$ [14], 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 [8], 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.

Finally, it is worthy of remark some others skeletal coordination approaches with goals quite different from those of DIP. In [7], 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 [18], it is described the way the Network Of Tasks model [17] is used to built programs. 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 the performance of the component nodes and the visible graph structure.

2 The DIP Coordination Language

In this section we only give a brief summary of the DIP language. More detailed description of DIP appears in [9].

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. Besides defining the interaction among tasks, patterns also specify processor and data layouts. DIP is based on the use of domains, i.e. regions together with some interaction information that will allow efficient inter-task coordination.

We have initially established two patterns in DIP. The `MULTIBLOCK` 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. This pattern specifies the different blocks or domains that form the problem and also establishes the coordination scheme among tasks. For the latter role, it defines the borders among domains and establishes the way these borders will be updated.

The other skeleton provided by DIP is the `PIPE` pattern, which pipelines sequences of tasks in a primitive way. It is also based on the use of domains, which avoids that computational aspects such as data types have to appear at the coordination level, improving pattern reusability. In this case, no border among domains has to be explicitly specified, since all data associated to a domain are involved in the interaction. Nested pipeline patterns are allowed so that complex structures can be built in a declarative way.

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 (`PUT_DATA`, `GET_DATA`, `CONVERGE`). A new type (`DOMAINxD`) and a new attribute (`GRIDxD`) have also been included.

3 Two simple examples

3.1 Example 1. Laplace's equation

The following program shows the `MULTIBLOCK` pattern for an irregular problem that solves Laplace's equation in two dimensions using Jacobi's finite differences method with 5 points.

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

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:

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

```
MULTIBLOCK Jacobi   u/1,1,Nxu,Nyu/, v/1,1,Nxv,Nyv/
  solve(u:(BLOCK,BLOCK)) ON PROCS(4,4)
  solve(v:(BLOCK,BLOCK)) ON PROCS(2,2)
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 domains in which the problem is divided are shown in Figure 1 together with a possible data distribution and the border between domains. Dotted lines represent the distribution into each HPF task. A domain definition is achieved

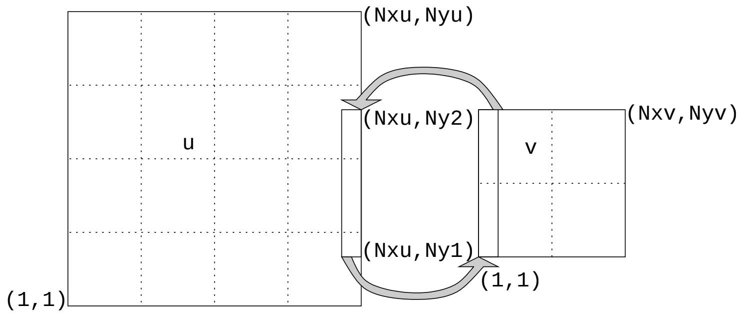

Fig. 1. Communication between two HPF tasks.

by means of an assignment of Cartesian points, i.e. the region of the domain is established. For example, the expression `u/1,1,Nxu,Nyu/` assigns to the domain `u` the region of the plane that extends from the point `(1,1)` to the point `(Nxu,Nyu)`. A border is specified by means of the `<-` operator. For example, the expression `u(Nxu,Ny1,Nxu,Ny2) <- v(2,1,2,Nyv)` indicates that 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)`.

In the task call specification, the name of the domain (say `u`) to be solved by the task (`solve`) and the data distribution (for example `(BLOCK,BLOCK)`) are specified. The processor layout is also indicated (for example `ON PROCS(4,4)`). 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. The knowledge at the coordination level of data distribution 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.

Finally, the code of subroutine `solve` used for the two tasks specified in the pattern is shown below. Line 2 declares a domain variable for the received domain. Line 3 uses the attribute `GRID` to declare two record variables `g` and `g_old`. After domain and grid declarations, line 4 is a special kind of distribution which produces the distribution of the field `DATA` (an array of real numbers) and the replication of the field `DOMAIN` (the associated domain) of both `g` and `g_old` variables. In lines 5 and 6, arrays `g%DATA` and `g_old%DATA` are dynamically created at the same time that the domain `d` is assigned to the fields `g%DOMAIN` and `g_old%DOMAIN`, respectively. The initialization of `g%DATA` is performed in the subroutine called in line 7. Statement 9 produces the assignment of the two variables with `GRID` attribute. Since `g_old` has its domain already defined, this instruction will just produce a copy of the values of the field `g%DATA` to `g_old%DATA`.

Lines 10 and 11 are the first where communication is achieved. Data from g%DATA needed by each task are communicated. Local computation is accomplished by the subroutines called in lines 12 and 13 while the convergence is tested in line 14. The instruction CONVERGE causes a communication between the two tasks. In this case, the communicated data is the value of the variable error. The maximum value (calculated by function maxim) obtained in each process is assigned to the variable error once the execution of CONVERGE is finished.

```
 1)subroutine solve (d)
 2)DOMAIN2D d
 3)double precision,GRID2D :: g,g_old
 4)!hpf$ distribute(BLOCK,BLOCK)::g,g_old
 5)g%DOMAIN = d
 6)g_old%DOMAIN = d
 7)call initGrid (g)
 8)do i=1, niters
 9)  g_old = g
10)  PUT_DATA (g)
11)  GET_DATA (g)
12)  call computeLocal (g, g_old)
13)  error = computeNorm (g, g_old)
14)  CONVERGE (g, error, maxim)
15)  Print *, "Max norm: ", error
16)enddo
17)end subroutine solve
```

3.2 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 [11][6]. 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 [11] and [6]. Figure 2 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
  cfft(d/1,1,N,N/:(*,BLOCK)) ON PROCS(4)
  rfft(d:(BLOCK,*)) ON PROCS(4)
END
```

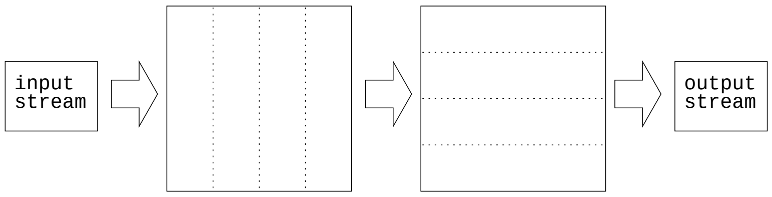

Fig. 2. Array distributions for 2-D FFT.

The code below shows the two stages. The stage `cfft` reads an input element, performs the 1-D transformations and calls `PUT_DATA(a)`. The stage `rfft` calls `GET_DATA(b)` to receive the array, performs the 1-D transformations and write 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 cfft (d)
DOMAIN2D d
complex, GRID2D :: a
!hpf$ distribute a(*,block)
a%DOMAIN= d
do i= 1, n_images
   call read_stream (a%DATA) ! read input
!hpf$ independent
  do icol = 1, N
    call fftSlice(a%DATA(:,icol))
  enddo
  PUT_DATA (a)
enddo
end
```

```
subroutine rfft (d)
DOMAIN2D d
complex, GRID2D :: b
!hpf$ distribute b(block,*)
b%DOMAIN= d
do i= 1, n_images
  GET_DATA (b)
!hpf$ independent
  do irow = 1, N
    call fftSlice(b%DATA(irow,:))
  enddo
  call write_stream (b%DATA) !write output
enddo
end
```

An alternative solution that shows how nested PIPE patterns are used is given in the following code.

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

A list of input/output domain definitions must appear after the nested pattern name. One of the predefined word IN, OUT, INOUT must precede each domain definition. Here, the domain involved in the pattern FFT2D is defined as an input/output domain. This pattern is "called" in the second stage of the main PIPE pattern. Subroutines cfft and rfft shown above must be modified since read/write operations are now carried out by Input and Output stages. So, line 7 in subroutine cfft is substituted by a call to GET_DATA(a), and line 12 in subroutine rfft is now a call to PUT_DATA(b).

Although the first solution is shorter, the second one establishes a more useful PIPE FFT2D pattern, since it can be reused in more complex applications, such as Convolution, which is a standard technique used to extract feature information from images. It involves two 2-D FFTs, an elementwise multiplication, and an inverse 2-D FFT and it is applied to two streams of input images to generate a single output stream.

4 Implementation Issues and Results

In order to evaluate the performance of DIP, 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 proposal. Here, we show the results for Jacobi's method and the 2-D FFT problem explained above.

For designing our initial prototype, we have built a compiler that translates our DIP code to BCL code. 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 [3]. No change to the HPF compiler has been needed. In a BCL program there are one coordinator process and one or several worker 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. So, in our DIP to BCL transformation phase, we have created the coordinator process from the coordination pattern and the worker processes from our computational tasks.

Table 1. HPF/DIP ratio for Jacobi's method

Domains	HPF/DIP ratio		
	4 Procs.	8 Procs.	16 Procs.
2	1.03	1.27	1.49
4	1.04	1.57	2.38
8	0.93	1.57	2.90

Table 2. HPF/DIP ratio for 2-D FFT

Size	HPF/DIP ratio		
	4 Procs.	8 Procs.	16 Procs.
32 by 32	1.59	2.08	1.47
64 by 64	1.09	1.44	1.83
128 by 128	1.03	1.08	1.25

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).

Table 1 reports the ratio between the HPF and DIP execution times for Jacobi's method for the different domains of the problem and numbers of processors exploited. We have considered 2, 4 and 8 domains with a 128×128 grid each one. The program has been executed for 20000 iterations. Table 1 highlights the better performance of the mixed task and data-parallel implementation. When the number of processors is equal to the number of domains (only task parallelism is achieved) DIP has also shown better results. Only when there are more domains than available processors, DIP has shown less performance because of the context change overhead among weight processes.

Table 2 reports the HPF/DIP ratio for different problem sizes of the 2-D FFT application. Again, the performance of DIP is generally better. 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 [11]. In this situation, HPF performance is quite good and so, the integration of task parallelism does not contribute so much.

5 Conclusions

We have used DIP, a Domain Interaction Pattern-based high level coordination language, to integrate task and data parallelism. The suitability and expressiveness of the model have been proved by means of some examples. We have also confirm the efficiency of the approach discussing some experimental results obtained with an initial prototype. 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.

References

1. 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
2. Bal, H.E., Haines, M.: Approaches for Integrating Task and Data Parallelism. IEEE Concurrency, **6(3)** (1998) 74–84
3. Brandes, T.: ADAPTOR Programmer's Guide (Version 7.0). Technical documentation, GMD-SCAI, Germany. (1999) `ftp://ftp.gmd.de/GMD/adaptor/docs/pguide.ps`
4. Carriero, N., Gelernter, D.: Coordination Languages and their Significance. Communications of the ACM, **35(2)** (1992) 97–107
5. Chapman, B., Haines, M., Mehrotra, P., Zima, H., Rosendale, J. Opus: A Coordination Language for Multidisciplinary Applications. Scientific Programming, **6(2)** (1997) 345-362
6. Ciarpaglini, S., Folchi, L., Orlando, S., Pelagatti, S., Perego, R.: Integrating Task and Data Parallelism with taskHPF. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00), Las Vegas, Nevada. (2000) 2485–2492
7. Cole, M., Zavanella, A.: Activity Graphs: A Model-Independent Intermediate Layer for Skeletal Coordination. 15th Annual ACM Symposium on Applied Computing (SAC'00). Special Track on Coordination Models, Villa Olmo, Como, Italy. (2000) 255–261
8. 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'00), Las Vegas, Nevada. (2000) 753–760
9. Díaz, M., Rubio, B., Soler, E., Troya, J.M.: DIP: A Pattern-based Approach for Task and Data Parallelism Integration. To appear in 16th Annual ACM Symposium on Applied Computing (SAC'01). Special Track on Coordination Models, Las Vegas, Nevada. (2001)
10. 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)
11. 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
12. High Performance Fortran Forum: High Performance Fortran Language Specification version 2.0 (1997)
13. Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M.: The High Performance Fortran Handbook. MIT Press (1994)
14. 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

15. Pelagatti, S.: Structured Development of Parallel Programs. Taylor&Francis (1997)
16. Rauber, T., Rünger, 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, San Antonio, Texas. (1999) 146–155
17. Skillincorn, D.: The Network of Tasks Model. International Conference on Parallel and Distributed Computing and Systems (PDCS'99). Boston, Massachusetts. (1999)
18. Skillincorn, D., Pelagatti, S.: Building Programs in the Network of Tasks Model. 15th Annual ACM Symposium on Applied Computing (SAC'00). Special Track on Coordination Models, Villa Olmo, Como, Italy. (2000) 248–254
19. Subhlok, J., Yang, B.: A New Model for Integrated Nested Task and Data Parallel Programming. 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'97), Las Vegas, Nevada. (2000) 1–12