SBASCO: Skeleton-based Scientific Components*

Manuel Díaz, Bartolomé Rubio, Enrique Soler and José M. Troya
Dpto. Lenguajes y Ciencias de la Computación.
Málaga University
29071 Málaga, SPAIN
mdr@lcc.uma.es, tolo@lcc.uma.es, esc@lcc.uma.es, troya@lcc.uma.es

Abstract

SBASCO is a new programming environment for the development of parallel and distributed high-performance scientific applications. The approach integrates both skeleton-based and component technologies. The main goal of the proposal is to provide a high-level programmability system for the efficient development of numerical applications with performance portability on different platforms. This paper presents the system programming model, which considers two different views of a component interface: one from the point of view of the application programmer and another thought to be used by a configuration tool in order to establish efficient implementations. This can be achieved due to the knowledge at the interface level of data distribution and processor layout inside each component. The programming model borrows from software skeletons a cost model enhanced by a run-time analysis, which enables to automatically establish a suitable degree of parallelism and replication of the internal structure of a component.

1. Introduction

A parallel programming environment (PE) can be understood as a set of tools needed to design, code and debug parallel and/or distributed applications, according to a given programming model or language. Traditionally, low-level communication and synchronization libraries have been integrated into the base programming language. Well-known examples are C/MPI, C++/ACE, HPF/MPI. Although very efficient applications can be developed with this kind of systems, the provided low-level programming abstraction requires a deep understanding of the communication and synchronization mechanisms [7].

* This work was supported by the Spanish project MICYT TIC 2002-04309-C02-02

Currently, a significant effort is being carried out on the design of parallel and distributed PEs that could relieve the programmer from most of the parallelism exploitation details, while preserving his/her ability to handle its qualitative aspects. An important research area where this issue is being explored is that of software skeletons. According to Cole [6], a software skeleton is a known reusable parallelism exploitation pattern. In [18] a methodology for structured development of parallel software is proposed. This methodology is based on the establishment of a fixed set of software skeletons or parallelism constructors, which are the only way to express the parallel structure of the program. 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. The low-level aspects of parallelism exploitation are managed by either the skeleton compilers or the runtime support. Starting from this idea, some PEs were developed, such as P^3L [1] and SkIE [2]. The skeleton technology has some very interesting features for high-performance computing in terms of high-level programmability, compositionality, and performance portability owing to the existence of a cost model.

On the other hand, component software technology, initially applied to the business world, is coming to the high-performance parallel and distributed computing scene, affecting the development of PEs. They get a feature that was not formerly present, namely interoperability: on the one hand, by making parallel application to look like standard components, the parallel application code become usable outside its development environment; on the other hand, by allowing to import external components, the development of new applications can be performed faster.

Component standards and implementations, such as OMG CORBA [17], Microsoft DCOM [13], Sun Java Beans and Enterprise Java Beans [10] [16], share serious shortcomings for parallel and distributed scientific applications, due to the lack of the abstraction needed by parallel and distributed programming and poor performance.

They also have trouble with the mechanism for encapsulating an existing scientific application (which might itself be a parallel-distributed application) into a component. A large effort is currently devoted to define a standard component architecture for high-performance computing in the context of the Common Component Architecture (CCA) Forum [15]. CORBA itself has been used as the basic component mechanism in different projects aimed at providing advanced parallel PEs [14] [4].

Recently, some efforts are being carried out in order to integrate both software skeletons and component technologies to develop parallel and distributed PEs in a unified approach. In this sense, ASSIST [22] is focused on high-level programmability and software productivity for complex multidisciplinary applications, including data-intensive and interactive software. Actually, ASSIST integrates features from two other different worlds, coordination languages and design patterns. These, together with software skeletons, provide suitable ways to overcome the main problem in developing parallel and distributed applications with classical communication libraries, middleware or operating system mechanisms, which consists of writing all the low-level code needed to set up an effective application.

SBASCO (Skeleton-BAsed Scientific COmponents) is a proposal of a new PE oriented to the efficient development of parallel and distributed numerical applications also integrating both technologies: skeletons and components. This unified approach provides interoperability, high-level programmability, compositionality, and performance portability. The last term makes reference to the capability of predicting performance on different platforms for a concrete component. In a first approach, three kinds of skeletons have been considered for the development of components: *multi-block*, *pipe* and *farm* skeletons. They are an evolution from those of DIP [9], a high-level skeleton-based coordination language to integrate task and data parallelism.

In SBASCO, there are two views of a component interface. The *application view* contains information related to data types of component input/output. This view is used by the programmer in order to develop his/her applications by means of a composition language. The *configuration view* extends the application view with information about input and output data distribution, processor layout and component internal structure (in terms of skeleton composition scheme).

A configuration tool uses the latter view to obtain an efficient implementation of the application on parallel/distributed platforms. The knowledge at the component interface level of data distribution and processor layout allows the system to obtain an efficient implementation of the communication and synchronization among components.

This approach has also been taken into account in other systems, such as Pardis [14], where a CORBA-based architecture for application-level parallel distributed computation is developed, in such a way that the ORB can transfer distributed arguments directly between the corresponding threads of client and server. Unlike SBASCO, Pardis is not based on skeletons, so that it lacks the performance portability feature provided by the cost model generally present at a skeleton-based system. Moreover, the client-server model of Pardis may not be the most suitable for the establishement of communication protocols in parallel and distributed scientific applications [12].

On the other hand, in SBASCO the cost model associated to each skeleton is enhanced by a run-time analysis by means of the configuration tool, which enables to automatically establish a suitable degree of parallelism and replication of the internal structure of a component. This is a contribution with respect to other environments such as ASSIST, where its configuration tool requires the manual establishment of these aspects [3]. Actually, although it is also a skeleton-based system, it does not have a cost model due to the generality of the *parmod* construct it is based on. However, this generic construct allows ASSIST to face complex multidisciplinary applications. In addition, in the ASSIST model, there is no knowledge at the interface level of data distribution and processor layout, and so, it is required a major effort to optimize the run-time support.

In this paper we present the SBASCO programming model, showing the different aspects concerning the skeletons used in the system to construct components, including their associated cost models. The way the two views of a component interface are established and used is also exposed. By means of some examples, the expressiveness and suitability of the programming model are shown.

The rest of the paper is structured as follows. Next section describes the scientific components considered in our approach. Section 3 explains the two different views of a component interface. A general SBASCO program scheme is shown in section 4. Finally, some conclusions and future work are sketched in section 5.

2. Scientific Components

The internal structure of a scientific component can be established by means of the utilization of the different skeletons defined in the SBASCO programming model. So, the interaction among the different tasks integrating the component is expressed in a high-level and declarative way, according to static and predictable patterns. A skeleton definition is based on the concept of *domain* attribute, an evolution of the domain type introduced in [8]. Basically, a do-

main consists of the Cartesian points establishing a region and some interaction information such as region borders.

We have defined three skeletons:

- The *multi-block* skeleton 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 *farm* skeleton improves a task throughput as different data sets can be computed in parallel on different sets of processors.
- Problem solutions that have a communication pattern based on array interchange and may take advantage of integrating task and data parallelism, can be defined and solved in an easy and clear way by using the *pipeline* skeleton, which pipelines sequences of tasks in a primitive way.

Besides that, components which internal structure does not fit these skeletons are also considered. In this case, they will be dealt as sequential or data parallel black boxes where the only available information is related to input/output arguments.

The following three sections outline the main characteristics of these skeletons. The section 2.4 shows their associated cost model.

2.1. The MULTIBLOCK Skeleton

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) [20]. 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 `MULTIBLOCK` skeleton has been defined. The following code shows the general scheme of this skeleton:

```
MULTIBLOCK skeleton_name
 task1(arguments) processor layout
 task2(arguments) processor layout
 .....
 taskm(arguments) processor layout
WITH BORDERS
 border definitions
END
```

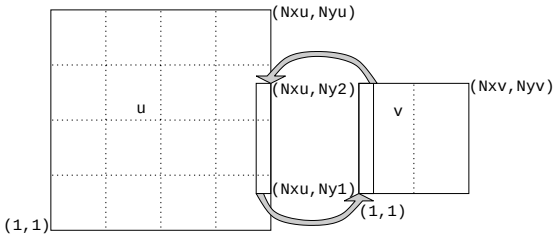

Figure 1. Communication between two tasks

Tasks solve the different domains the problem has been decomposed in. This way, a task array argument may have the `DOMAIN`x`D` attribute (where $1 \leq x \leq 4$). This means that this array contains the data belonging to a problem domain. For example,

```
complex, DOMAIN2D :: u/1,1,Nxu,Nyu/
```

declares a two-dimensional array `u` that contains data for the region of the plane that extends from the point `(1,1)` to the point `(Nxu,Nyu)`.

In general, the problem is solved iteratively, so that for each iteration a communication of the different borders is needed. These borders are defined among the specified domains. For example, if `u` and `v` are two domain-attributed arrays, 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)` (see Figure 1).

A task may be sequential or data parallel. In the latter case, the data and task parallelism integration is achieved in an elegant, simple and efficient way. Each task is solved by a disjoint set of processors. In the example of Figure 1, the task in charge of solving domain `u` is executed on a 4×4 mesh of processors while the task for domain `v`, on an array of 2 processors (denoted by dotted lines).

2.2. The FARM Skeleton

The *FARM* skeleton replicates a task in order to improve its throughput as different data sets can be computed in parallel on different sets of processors. The stream of input data for the task is accepted by a special process called emitter which schedules them to each replica. On the other hand, a collector process collects the results from each replica and merges them into the stream of output data from the task (Figure 2).

This skeleton is specified as follows:

```
FARM skeleton_name (R) task(arguments)
     processor layout
```

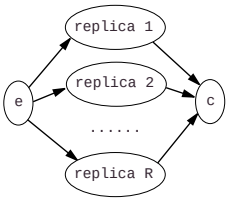

Figure 2. Structure of a farm with R replicas

where R establishes the number of replicas and processor layout indicates the number of processors where each replica is going to be executed.

2.3. The PIPE Skeleton

The `PIPE` skeleton pipelines sequences of tasks. Figure 3 depicts the structure of the general n-stage pipeline corresponding to the `PIPE` skeleton shown in the following code:

```
PIPE skeleton_name
  stage1
  stage2
  .....
  stagen
END
```

Each stage in the pipeline consumes and produces a data stream, except the first and the last stages that may only produce and consume, 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` skeleton.
- A pipeline call, i.e. the name of a nested `PIPE` skeleton together with the arguments it needs.
- A `FARM` skeleton that is used to replicate a non-scalable stage, which can be a task call or a pipeline call.

The communication between two pipe stages is specified by using the same domain-attributed array as argument in both stages, unlike in the `MULTIBLOCK` skeleton, where

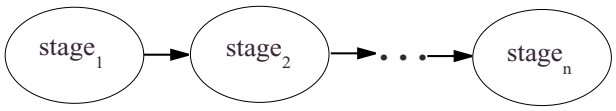

Figure 3. Structure of the n-stage pipeline

border definitions are used. The stage where this array first appears is considered the one that generates its data. If the same array appears in several stages, it means that the intermediate stages, 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. The first/last stage where an array appears may also receive/send the data if it is specified as input/output component argument.

2.4. Cost Model

In this section, we discuss a possible cost model for SBASCO skeletons described above. The cost of each skeleton is an estimation of its execution time, which depends on hardware architecture parameters, such as processor and channel speed and the system number of processors. Due to the fast evolution of hardware of both processors and inter-processor communication systems, an accurate estimation is quite difficult. So, our cost model is divided into two parts: the first one, following other approaches such as [19], establishes an analytical scheme where some parameters (as the number of processors used for each task and the number of replicas in a farm) are established by the second part, that is carried out by means of a run-time analysis.

For the MULTIBLOCK skeleton, the execution time per iteration is estimated as follows:

$$max\{T_{ti}\} + T_{cf} \quad (1)$$

where T_{ti} is the execution time for `taski`, and T_{cf} is the time required to communicate the different borders established in the skeleton. This communication time depends on the number of borders, their size and the data distribution and processor layout related to each border.

As FARM and PIPE skeletons exploit parallelism for a stream of independent data, we can define two different costs: execution time and service time. Execution time is the latency experimented to run a single stream item. Service time is the average time taken to run one item in a long stream. Formulae 2 and 3 show the execution time and the service time for the FARM skeleton, respectively.

$$T_{sch} + T_{ce} + T_r + T_{cc} \quad (2)$$

$$max\{T_{sch} + T_{ce} + T_{cc}, S_r/R\} \quad (3)$$

T_{sch} is the time required for the appropriate scheduling and possible rearranging carried out by the emitter and collector, respectively. T_{ce} and T_{cc} are the mean time needed to send the data from the emitter to a replica and from a replica to the collector, respectively. T_r is the mean time taken to

compute a stream item by a replica. If a good load balancing is ensured, R replicas with a service time S_r can compute R items in time S_r, yielding a global service time of S_r/R for a single item. Thus, the total service time of the farm is given by the slowest between $T_{sch} + T_{ce} + T_{cc}$ and S_r/R.

Finally, formulae 4 and 5 establish the execution time and the service time for the PIPE skeleton, respectively.

$$T_{s1} + T_{c1,2} + T_{s2} + T_{c2,3} + \ldots + T_{sn} \quad (4)$$

$$max\{S_{si}\} \quad (5)$$

In a PIPE, the execution time is the sum of the execution times of each stage (T_{si}) and the communication times between two stages ($T_{ci,i+1}$). The service time is given by the service time of the slowest stage (S_{si}).

3. Component Interface Views

In our approach, a scientific component interface has two different views. On the one hand, the *application view* is the typical component interface that describes the component as a black box. It contains information related to data types of component input/output. This view is used by the programmer in order to develop his/her applications by means of the composition language. On the other hand, the *configuration view* establishes information about input and output data distribution and, if possible, processor layout and component internal structure (in terms of skeleton composition scheme). A configuration tool uses this view to obtain an efficient allocation of the different application components on parallel/distributed platforms. The following sections describe both views, showing some examples in order to clarify their establishment and utilization.

It must be noted that a sequential component has neither internal structure nor data distribution and so, only its application view has to be specified.

3.1. Application View

The application view of a component interface will be used by a programmer together with the composition language described in section 4 in order to design his/her applications.

The following code shows the general scheme of the application view:

```
APPLICATION INTERFACE component_name
  [STREAM,] type, direction,
        [DOMAINxD] :: data
  .....
END
```

The interface will use as many lines as necessary in order to describe the component input/output. Each line establishes:

- *type*: data type.
- *direction*: `IN`, `OUT`, `INOUT`
- `DOMAIN`x`D`: the optional domain attribute.
- *data*: the name of the component argument. In case of an array declaration, its dimensionality and size are also specified. The size can be a constant value, the name of another argument, or it can be established at composition time. In the latter case, for domain-attributed arrays this establishment is made by means of the domain, while the `:` notation is used otherwise.
- `STREAM`: a component argument may be declared as a stream, which means that the component receives and/or generates an ordered sequence, possibly of unlimited length, of the described data.

As an example, let us consider a component that implements the Fast Fourier Transform (FFT) for a stream of two-dimensional arrays. The component receives the array stream and it generates a stream of the FFT of each input array. The array size is specified at composition time by means of the domain attribute. The component application view is the following:

```
APPLICATION INTERFACE fft_2d
  STREAM, complex, INOUT, DOMAIN2D :: a
END
```

3.2. Configuration View

Basically, the configuration view of a component interface is the application view described in the previous section together with some useful information to achieve efficient implementations of target applications. This information appears in the component argument declaration and in the skeleton describing the component internal structure. The knowledge at the interface level of input/output data distribution and processor layout belonging to a component allows the configuration tool to obtain an efficient implementation of the communication and synchronization among components. This characteristic has been proved in [9], where efficient implementations of applications that integrated task parallelism among a collection of data parallel HPF tasks were achieved.

On the other hand, the processor layout and the (farm) replication degree inside a component may be also configurable. In this case, an appropriate configuration will be decided by the tool, attending to the skeleton cost models described in section 2 enhanced by a run-time analysis.

The general scheme of the configuration view is the following:

```
CONFIGURATION INTERFACE component_name
  [STREAM,] type, direction,
        [DOMAINxD] :: data
  DISTRIBUTE [direction]
        data distribution
  .....
  [STRUCTURE
    internal structure declarations
    skeleton-based internal structure
  END]
END
```

The data distribution types are the typical ones used in data parallelism. When a data direction is declared as `INOUT`, different distribution may be specified for each direction, as we will see in the example bellow.

The `fft_2d` component of the example described above, can be structured as a pipeline of two stages in order to increase its efficiency, as proved in different works [11][5]. The first one performs independent one-dimensional FFTs on the columns of the input arrays. Then, the second stage carries out independent one-dimensional FFTs on the rows of the resulting arrays from the previous stage. Dotted lines in Figure 4 show the array distributions needed for that scheme.

This mixed task and data parallelism scheme is expressed in the component interface configuration view as follows:

```
CONFIGURATION INTERFACE fft_2d
  STREAM, complex, INOUT, DOMAIN2D :: a
  DISTRIBUTE IN a(*,BLOCK)
  DISTRIBUTE OUT a(BLOCK,*)
  STRUCTURE
    PIPE FFT2D
      cfft(a) ON PROCS(?P1)
      rfft(a) ON PROCS(?P2)
    END
  END
END
```

In this example, only one domain-attributed array is used in the pipe skeleton describing the component internal structure. The data distribution for the component input is different `(*,BLOCK)` from that for the component output `(BLOCK,*)`. This information is important in order to establish an efficient communication between a component providing data and `fft_2d` and between this and a component consuming resulting data. The notations `?P1` and `?P2` indicate that the number of processors for both stages is configurable.

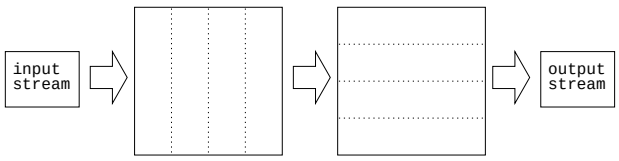

Figure 4. Array distributions for 2-D FFT (4 processors per stage).

4. SBASCO Program Scheme

The SBASCO composition language is used for both scientific component and application construction. The ”building blocks” are the scientific components previously described.

The language syntax for component composition is the same used for skeleton definition and component interface description established in previous sections. Actually, for component construction, the composition scheme is the same that the configuration view of a component. However, in case of application design, there are not input/output arguments. In addition, no data distribution is declared at the composition level, as this information will be obtained from the configuration view of the different involved components.

The general scheme of a program in SBASCO is the following:

```
PROGRAM program_name
  declarations
  STRUCTURE
    skeleton-based internal structure
  END
END
```

In order to illustrate the way the composition language is used, a simple example already presented in literature [21] and that represents the structure of a large class of image and signal processing applications is shown. The kernel application is *fft-hist*, where a stream of two-dimensional complex matrices is processed sequentially by a pipeline of three components. The first one, `inpm`, reads the matrices from I/O, and forwards them to component `fft_2d`, which performs FFT of each matrix. Finally, matrices produced by `fft_2d` are sent to component `hist`, which collects and analyzes them, and writes the results to an output file.

The problem solution using our composition language is the following. The `fft_2d` component was described in the previous section, while `inpm` and `hist` are sequential components.

```
PROGRAM fft_hist
  integer :: nrow=100, ncol=100
  STREAM, complex,
        DOMAIN2D :: a/1,1,nrow,ncol/
  STRUCTURE
```

```
    PIPE fft_hist_pipe
      inpm(a) ON PROCS(1)
      fft_2d(a) ON PROCS(?P)
      hist(a) ON PROCS (1)
    END
  END
END
```

5. Conclusions and Future Work

The programming model of SBASCO, a new programming environment for the development of parallel and distributed high-performance scientific applications, has been presented. By means of the integration of skeleton-based and component technologies, numerical applications can be designed and built using scientific components, whose internal structure is provided in order to obtain efficient implementations. The characteristics of the different skeletons used in the approach have been shown, including their associated cost model. The way the two different views of a component interface are specified and used was also presented. The expressiveness and suitability of the programming model have been shown by means of some examples.

As future work, we are currently developing the configuration tool that uses the configuration view of a component interface in order to establish efficient allocations of application components. This can be achieved due to both the knowledge at the interface level of data distribution and processor layout inside each component, and the run-time analysis of the cost model associated to the component internal structure. The latter enables to automatically establish a suitable degree of parallelism and replication inside the component.

References

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

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

[3] Baraglia, R., Danelutto, M., Laforenza, D., Orlando, S., Palmerini, P., Pesciullesi, P., Perego, R., Vanneschi, M., AssistConf: a Grid configurantion tool for the ASSIST parallel programming model, in "Proceedings of 11th Euromicro Conference on Parallel Distributed and Network based Processing ", pp. 193–200, IEEE, Genova, Italy, 2003.

[4] Beaugendre, P., Priol, T., Ren, C., "Cobra: a CORBA-compliant programming environment for high-performance computing", Technical Report PI 1141, INRIA, 1998.

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

[6] Cole, M., "Algorithmic Skeletons: Structured Management of Parallel Computation", MIT Press, Cambridge, MA, 1989.

[7] D'Ambra, P., Danelutto, M., Serafino, D., Lapegna, M., Advances Environments for Parallel and Distributed Applications: a View of Current Status, *Parallel Computing*, **28**, 12 (2002), 1637–1662.

[8] Díaz, M., Rubio, B., Soler, E., Troya, J.M., A Border-based Coordination Language for Integrating Task and Data Parallelism, *Journal of Parallel and Distributed Computing*, **62**, (2002), 715–740, doi:101006/jpdc.2001.1814.

[9] Díaz, M., Rubio, B., Soler, E., Troya, J.M., Domain Interaction Patterns to Coordinate HPF Tasks, *Parallel Computing*, *Parallel Computing*, **29**, 7 (2003), 925–951.

[10] Englander, R., "Developing Java Beans", O'Really&Associates, 1997.

[11] Foster, I., Kohr, D., Krishnaiyer, R., and Choudhary, A., A library-based approach to task parallelism in a data-parallel language, *J. Parallel and Distrib. Comput.*, **45**, 2 (1997), 148–158, doi:101006/jpdc.1997.1367.

[12] Gannon, D., Bramley, R., Stuckey, T., Villacis, J., Balasubramanian, J., Akman, E., Breg, F., diwan, S., Govindaraju, M., Developing Component Architectures for Distributed Scientific Problem Solving, *IEEE Computational Science and Engineering*, **5**, 2 (1998), 50–63.

[13] Horsmann, M., Kirtland, M., "DCOM Architecture" Microsoft White Paper, 1997. Available from http://www.microsoft.com/com/wpaper.

[14] Keahey, K., Gannon, D., PARDIS: a CORBA-based architecture for application-level parallel distributed computation, in "Proceedings of Supercomputing'97", 1997.

[15] The Common Component Architecture Forum, home page http://www.cca-forum.org.

[16] Monson-Haefel, R., "Enterprise Java Beans 3th edition", O'Really&Associates, 2001.

[17] Object Management Group, "Common Object Request Broker Architecture". CORBA page: http://www.corba.org.

[18] Pelagatti, S., "Structured Development of Parallel Programs", Taylor&Francis, London, 1997.

[19] Pelagatti, S., Task and Data Parallelism in P^3L, in "Patterns and Skeletons for Parallel and Distributed Computing, Rabhi, F.A. and Gorlatch, S. (Eds.)" Springer, London, UK, 2003.

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

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

[22] Vanneschi, M., The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications, *Parallel Computing*, **28**, 12 (2002), 1709–1732.