Managing Multi-Concern Application Complexity in AspectSBASCO

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

Department of Languages and Computer Science, University of Málaga, 29071 SPAIN
{mdr, sromero, tolo, esc, troya}@lcc.uma.es

Abstract. AspectSBASCO is a new programming environment that integrates modern technologies (i.e. software components, parallel skeletons and aspects) to support the development of parallel and distributed scientific applications. This multi-paradigm approach provides high-level parallelism, reusability, interoperability and a clearer separation of concerns. This paper is focussed on a case study in which the programming model of AspectSBASCO is applied for the efficient solution of a relatively complex reaction-diffusion problem. In the application, the system of non-linear PDEs is solved in parallel using skeleton abstractions for domain decomposition methods. In addition, other concerns including distributed simulation persistence, mesh adaptation procedures, dynamic processor re-mapping and state variables communication are implemented in a modular way using (un)pluggable aspects. This style of application development leads to a better system decomposition, which is the key to improving software evolution, maintenance, productivity and reliability.

1 Introduction

Software engineering advances in sequential computing are often difficult to spread on the high-performance computing (HPC) scene, where parallel programming models have to deal with an additional dimension of complexity (i.e. concurrency), and where portability is restricted by the types of underlying hardware architectures. Examples of relatively recent technologies which are proven to achieve high-quality decomposition of sequential and distributed applications are those based on software components and aspects.

Component-oriented programming (COP) [8] is the paradigm that proposes the construction of applications plugging stand-alone and reusable pieces of software, so called components. However, the most extended component models and implementations (e.g. Microsoft COM+, Sun EJB, OMG CCM) lack the abstraction needed for scientific applications. For example, a component in these models cannot encapsulate (at least, in a natural way) a parallel application which must be executed on a group of processors and interact efficiently with other processors. For this reason, specific solutions for HPC have emerged in the last few years. Some examples are CCA [1] and ASSIST [14].

When designing and building complex systems (even when components are used) it is difficult to produce designs that modularize all the system requirements. Typically, there are some characteristics that do not fit well into any component structure chosen. This is particulary true for concerns such as logging, debugging, communication, synchronization, security, and so on. Design alternatives often lead to code where the same concern spreads over (i.e. cross-cuts) many system modules. Aspect-oriented programming (AOP) [9] enables developers to capture the cross-cutting structure so that concerns can be programmed in a centralized way. Although AOP is very extended for sequential application development, the paradigm is hardly ever applied to HPC. Some interesting work is that of [7] and [12], both based on using the popular aspect-oriented language AspectJ for the modularization of high-performance concerns.

In a different context, structured parallel programming [10] proposes the use of skeletons, which are reusable parallelism exploitation patterns. The idea behind skeletal programming is to provide the programmer with a collection of predefined parallelism constructors (the skeletons) which can be combined to declaratively express the parallel structure of the application. Using this paradigm the developer is free from implementing low-level operations (e.g. task creation, data communication) and thereby she/he can focus on the numerical algorithms. Muskel [3] and eSkel [2] are representative examples of skeleton-based systems.

AspectSBASCO [6] is a new programming environment for HPC applications which provides a multi-paradigm approach that combines the above-mentioned technologies (i.e. software components, parallel skeletons and aspects). Components and aspects have been previously unified successfully in work dealing with non-parallel models [13]. Components and skeletons were used together in some parallel approaches [14]. However, and to the best of our knowledge, our proposal is the first attempt to combine the three technologies in the context of HPC.

This paper presents a case study on applying the AspectSBASCO programming model for the efficient solution of a complex reaction-diffusion problem, which is solved using skeleton abstractions for parallel domain decomposition methods [11]. Additional application concerns are implemented using aspects (which can be plugged, or unplugged, if necessary). Some examples are distributed simulation persistence, mesh refinement and dynamic processor remapping, which aim for improvements in different directions (e.g. accuracy, performance). Using AspectSBASCO, these kinds of "extra-functional" concerns of vital importance in scientific applications can be separate from the numerical code and become easier to develop and evolve as they are well-modularized.

A similar problem, though considering a simpler set of application requirements, was previously studied using SBASCO [4][5], the predecessor of AspectSBASCO which only combined coarse-grain scientific components and skeletons.

2 Overview of AspectSBASCO

There are two types of components in AspectSBASCO: Scientific Components (SCs) which implement the tasks that solve the numerical problem, and Aspect

Components (ACs) for encapsulating the cross-cutting functionality in applications. In addition, a family of three parallel skeletons is defined: *multiblock* is used for the solution of domain decomposition and multi-block problems; *farm* improves the throughput of a task as different data sets can be processed in parallel; *pipe* enables a sequence of tasks which can be executed concurrently to be pipelined. A detailed explanation of these skeletons can be found in [4].

Scientific components can implement sequential or parallel tasks. A parallel SC can use skeleton declarations to establish its internal parallel structure in a high-level way. Otherwise, the developer is free to implement "ad-hoc" parallelism in the component (e.g. unstructured, data-parallelism). Skeletons are not only used for internal component structure but also for application parallelism. For instance, an application can consist of a group of SCs which interact following the multiblock paradigm. Skeletons represent an elegant and declarative style of integrating different parallelism types (e.g. task and data parallelism).

The interaction between SCs follows a data-flow style and is based on two communication primitives called `get_data()` and `put_data()`. The *configuration interface* of a SC influences this type of communication. This interface describes the input/output arguments, their data-distribution, the processor layout and, (if possible) the internal structure in terms of skeleton declarations. Exposing this kind of information at the component interface level enables the system to implement efficient data communication and task interaction.

Aspect components implement the aspect code which will be executed at different points (join points) of the application control flow. Interactions between SCs and ACs are based on method calls (instead of data-flow). An AC can provide one or several interfaces of operations describing aspect functionality. A new type of elements called Aspect Connectors (ACNs) encapsulate the SC-AC interaction information. Specifically, an ACN indicates the methods on the corresponding ACs to be invoked at specific pointcuts (namely subsets of join points) that refer to the participant SCs. Expressing the interaction information in a separate layer of ACNs improves component development and reuse.

ACN declarations exploit the AspectSBASCO join point model which defines valid points for the potential execution of aspect code. The union of two different sets of points is considered. First, some internal actions identified as generic for our skeleton-based applications (e.g. calculation of new time step, communication based on `get_data()` and `put_data()`), and second, any method call carried out on *external components*, which are components that represent functionality reused by several SCs (for instance, a wrapper to a legacy code library).

The implementation of an AC may require access to the internal state of the SCs. For this reason, the latter can define additional interfaces that expose SC internal variables and properties. These interfaces are so-called *aspect interfaces* and are described using a subset of the OMG IDL language. Aspect interfaces are designed to enable the plugging and execution of aspects by means of the traditional *provide-use* pattern followed in component models such as CCM.

More detailed descriptions of the mechanisms that enable the definition and execution of aspects in our programming model can be found in [6].

3 Case Study: Reaction-Diffusion Equations

This section describes the numerical problem, the application design in AspectS-BASCO and some issues regarding implementation.

3.1 Numerical Problem Definition

The reaction-diffusion problem considered is characterized by a system of two time-dependent PDEs which are coupled by a non-linear source term. Basic combustion and heat transfer phenomena can be modeled as follows.

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + S(U), \qquad U = (u, v)^T, \qquad S = (-uv, uv - \lambda v)^T \tag{1}$$

The variables u and v represent the concentration of a reactant and the temperature, respectively, t is time, x and y denote Cartesian coordinates, λ is a constant (in this paper, $\lambda = 0.5$), and the superscript T denotes transpose.

Eq. (1) was discretized in time and space by means of finite differences and implicit, linearized, θ-method where the non-linear term $S_{i,j}^{n+1}$ was approximated using a Taylor polynomial to obtain a system of linear algebraic equations.

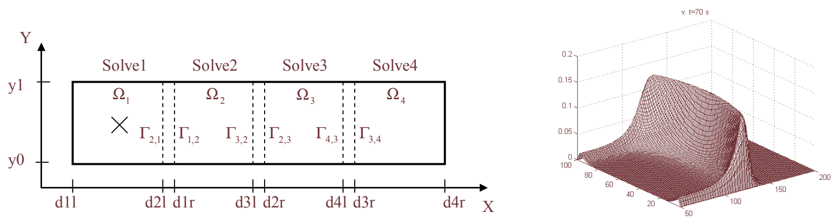

Fig. 1. Decomposition of the problem using four overlapping domains (left) and graphical representation of variable v, temperature, at $t = 70$ (right)

The problem is studied on a two-dimensional geometry similar to the one shown in Fig. 1 (left). The point marked with a cross denotes the ignition point, which is the temperature peak needed to start the reaction. The same figure depicts a snapshot of the solution, which corresponds to a traveling reaction wave front that propagates from the ignition point to the rest of the domain.

The numerical solution is calculated by means of a overlapping domain decomposition (i.e. Schwarz) method which admits parallel processing. The algorithm used proceeds as follows: the geometry is divided into four equal-sized domains which are overlapped, as illustrated in Fig. 1, where the symbol Ω_i denotes the domain number i, and $\Gamma_{i,j}$ is the part of the border of Ω_i that penetrates Ω_j. Specific initial conditions are the Dirichlet data on external boundary and, on the entire surface, unit value for reactant and inverse exponential function centered at the temperature ignition point. The solution on the different domains

Ω_i is computed concurrently. Then, the borders are interchanged, which means that every border $\Gamma_{i,j}$ is updated with the current solution on Ω_j. The process is repeated iteratively until the borders remain unchanged in two iterations (i.e. $\|\Gamma_{i,j}^{k} - \Gamma_{i,j}^{k-1}\|_\infty < 10^{-15}$). Then, the next time step can be computed.

In addition to the equation definition, other requirements are considered. Due to the steepness and high-curvature of the solution (see Fig. 1) a large number of grid points is needed for an accurate result. However, it is a fact that the solution is virtually constant in the areas which have not yet been reached by the wave front, so it can be considered a waste of computational power to use many grid points on such areas (also for the areas the wave leaves). As an alternative, grid-adaptive procedures which dynamically concentrate more grid points where they are needed can be used. In this application, when the traveling front reaches a domain Ω_i, its number of grid points is duplicated. Cubic spline interpolation on the $x-$ axis is used to calculate initial values for the new unknowns.

3.2 Application Design

The application features four scientific components, named `Solve1` to `Solve4`, each one computing the solution on a single domain Ω_i. The SCs use internal data-parallel *red-black* iterations to solve the domain, so every SC itself is a parallel task that divides its domain data to be computed on various processors.

```
PROGRAM reaction-diffusion
  integer :: y0=1, y1=100, d1l=1, d1r=200, d2l=181, d2r=380,// ...
  complex, DOMAIN2D :: omega1/d1l,y0,d1r,y1/
                       omega2/d2l,y0,d2r,y1/
                       // ...
  STRUCTURE
    MULTIBLOCK reaction-mb
      Solve1(omega1) ON PROCS(4),
      Solve2(omega2) ON PROCS(4),
      // ...
    WITH BORDERS
      omega1(d1r,y0,d1r,y1) <- omega2(_)
      omega2(d2l,y0,d2l,y1) <- omega1(_)
      // ...
END
```

```
CONFIGURATION INTERFACE Solve1
  complex, INOUT, DOMAIN2D :: d
  DISTRIBUTE d(BLOCK,*)
END
// ...
```

Fig. 2. Program structure and multiblock (left). Configuration inferface (right)

The structure of the main program in AspectSBASCO is based on a *multi-block* skeleton declaration, as shown in Fig. 2. Once the extension of the domains (i.e. two-dimensional arrays of complex numbers) is defined, the participant SCs, the domains assigned to them and, finally, the communication of borders are indicated using a specific syntax. As an example, the first line of `WITH BORDERS` means that the part of `omega1` delimited by the points `(d1r,y0)` and `(d1r,y1)` will be updated at each iteration by the part of `omega2` delimited by the same points. In this program, it is expressed that every SC executes using four processors. The right side of the picture depicts the configuration interface of `Solve1`. Interfaces for the rest of the SCs are similar. The sole component argument is a two-dimensional domain which, in this case, is distributed by rows.

It should be noted that the border interchange operation can involve complex communication and synchronization among parallel tasks, where the number of processors, data distribution and border extensions need to be considered. By using the multiblock skeleton, the developer is abstracted from these details. She/he only invokes two primitives, `get_data()` and `put_data()`, for receiving and sending border data, respectively, while the system performs efficient communication exploiting the information declared in the skeleton.

Although the procedure described in the last paragraph of Section 3.1 enables better (more accurate) numerical solution, its use may negatively affect the performance. Let us assume that one of the domains increases its size in our executing parallel application. The SC processing such a domain will take longer (in comparison with the other SCs) to finish its task. In addition, the communication of borders (operation repeated several times per iteration) represents a synchronization point for the entire application. For this reason, parallel domain decomposition programs must maintain a similar computation load for every domain, otherwise processors of the domains computed faster will be idle while waiting for new border data. In order to overcome this shortcoming, our application carries out processor re-mapping at run-time. This functionality is implemented using pluggable aspect components, as described later.

The next step entails describing the layer of aspect interfaces which enables the composition and execution of aspects. This application features a total of six aspect components to encapsulate different functionality. Fig. 3 (left) shows component declarations indicating the provided and used interfaces. Interfaces themselves are described in the same figure (right). The four SCs are declared as subclasses of a component root named `Sc` which has common declarations. The idea is that the SCs, influenced by connector declarations (ACNs), will invoke methods on the ACs in order to execute aspect code at different points of the control flow. In this application, the ACNs exploit a set of join points which are predefined in AspectSBASCO and refer to static points (in *multiblock* programs) such as domain initialization, computation of a new time step, evaluation of a new iteration (in current time step), border communication, variation in the number of processors of a SC, and estimation of convergence result.

As can be observed in Fig. 3, aspect component operations usually define their first argument as a reference to the caller SC. The AC implementation can use this reference to invoke operations on the caller component, if necessary. The next paragraphs provide brief component and interface descriptions.

Using `IAccess`, the ACs can modify the state of the SCs. The `Domain2D` object which hosts the solution on the current and the previous time steps can be accessed. The input and output borders, as well as some parameters (e.g. physical system parameters, convergence result), can be manipulated. An AC can modify the domains using `changeDomain()`. The object of type `MapResult` encapsulates the current processor mapping scheme for this application.

Component `LinearSolver` implements `ISolver` and calculates the solution of a linear system $A\mathbf{x} = \mathbf{b}$. At the beginning of each time step, the vector $\mathbf{b}$ is adjusted using `evalFixedTerms()`. Then, the solution $\mathbf{x}$ is obtained calling

```
// COMPONENT DECLARATIONS
////////////////////////

component Sc {
  provides IAccess access;
  uses ISolver solver;
  uses IConvergence convergence;
  uses IStorage storage;
  uses IAdaptor adaptor;
  uses IMapping mapping;
  uses IState state;
};

component Solve1 : Sc {};
component Solve2 : Sc {};
component Solve3 : Sc {};
component Solve4 : Sc {};

component LinearSolver {
  provides ISolver solver;
  uses IAccess access;
};

component BorderTest {
  provides IConvergence convergence;
  uses IAccess access;
};

component Store {
  provides IStorage store;
  uses IAccess access;
};

component GridAdaptor {
  provides IAdaptor adaptor;
  uses IAccess access;
};

component Scheduler {
  provides IMapping mapping;
  uses IAccess access;
};

component State {
  provides IState state;
  uses IAccess access;
};
```

```
// ASPECT INTERFACE DECLARATIONS
////////////////////////////////

interface IAccess {
  Domain2D getDomain(in unsigned currentOrPrevious);
  sequence<Domain2D> getBorders(in unsigned inputOrOutput);
  double getParam(in string param_name);
  void setParam(in string param_name, in double value);
  void changeDomain(in Domain2D currDom, in Domain2D prevDom);
  MapResult getMapResult();
  void setMapResult(in SchResult schRes);
};

interface ISolver {
  void setup(in Sc sc, in Domain2D currDom);
  void evalFixedTerms(in Sc sc, in Domain2D prevDom);
  void eval(in Sc sc, in Domain2D currDom, in Domain2D prevDom);
  void checkDomain(in Sc sc, in Domain2D currDom, in Domain2D prevDom);
  void checkMapResult(in Sc sc, in MapResult mapRes);
};

interface IConvergence {
  void setup(in Sc sc, in sequence<Domain2D> inputBorders);
  void copy(in Sc sc, in sequence<Domain2D> inputBorders)
  boolean evaluate(in Sc sc, in sequence<Domain2D> inputBorders);
  void checkMapResult(in Sc sc, in MapResult mapRes,
                      in sequence<Domain2D> inputBorders);
};

interface IStorage {
  void load(in Sc sc, in Domain2D currDom);
  void save(in Sc sc, in Domain2D currDom);
};

interface IAdaptor {
  void setup(in Sc sc, in Domain2D currDom, in Domain2D prevDom);
  void adapt(in Sc sc, in Domain2D currDom);
  void initPut(in Sc sc);
  void finishGet(in Sc sc);
  void communicateGridMode(in Sc sc, in MapResult mapRes);
  void finishApp(in Sc sc, in Domain2D currDom);
};

interface IMapping {
  void setup(in Sc sc);
  void startTime(in Sc sc);
  void stopTime(in Sc sc);
  void remap(in Sc sc);
  void communicateMapResult(in Sc sc, in MapResult mapRes);
};

interface IState {
  void initRemap(in Sc sc, in Domain2D currDom, in MapResult mapRes)
  void finishRemap(in Sc sc, in Domain2D currDom, in MapResult mapRes)
};
```

Fig. 3. Component and aspect interfaces used in the reaction-diffusion problem

`eval()` iteratively. The functions `checkDomain()` and `checkMapResult()` are executed, respectively, when the domain is replaced and when the mapping of processors changes. These functions adapt the component to the new settings.

Component `BorderTest` calculates the convergence of the method. Each time a SC invokes `get_data()` to receive new borders, the code of `copy()` stores the values of previously used borders. The function `evaluate()` compares successive border data to determine if the calculation of a time step must stop.

Component `Store` implements a simple persistence characteristic. The execution time of this application can range (from several minutes to dozens of hours in modest parallel computers) simply changing the domain size and convergence criteria. `Store` enables the final distributed numerical result to be the initial condition of a subsequent execution. This way a simulation can be completed in several sessions. The functions `load()` and `save()` manage the serialization.

The grid adaptation procedure is encapsulated in `GridAdaptor`, which implements `IAdaptor`. The method named `adapt()` changes both the current domain and the $x-$ space step length. This means either increasing or decreasing the number of grid points in function of the variable v. The component uses cubic

```
ACN adaptor_acn on component Sc {

  advice before on put_data_call {
    getAdaptor()->initPut(this);
  };
  advice after on resize_call {
    getAdaptor()->communicateGridMode(this, getMapResult());
  };
  advice after on init_call {
    getAdaptor()->setup(this, getDomain(0), getDomain(1));
  };
  advice before on step_call {
    getAdaptor()->adapt(this, getDomain(0));
  };
  // ...

};
```

Fig. 4. Aspect connector declaration for grid-adaptive procedure management

splines to interpolate the new points in the $x-$ axis. In addition, this component manages the borders in accordance with the type of domain being used. For instance, each time borders are received, the function `finishGet()` adapts received data considering the real extension of the new domain.

Component `Scheduler` determines how processors should be re-distributed for a better computation balance. The methods `startTime()` and `stopTime()` are called just before and after the code blocks which will be monitored. These functions allow `Scheduler` to calculate the execution time of the SCs. When the function `remap()` is invoked, `Scheduler` consults time information to establish a better processor mapping. The algorithm used in this application is quite straightforward and it consists of comparing two SCs which are the components having the highest and slowest execution time. If the difference is greater than a threshold the “fast” SC releases one processor which is taken by the “slow” SC.

Finally, `State` is in charge of communicating the computation state in processor re-mapping operations. For instance, if a SC acquires one more processor, the current domain data has to be re-distributed among a higher number of processors. Other internal variables have also to be communicated. The methods `initRemap()` and `finishRemap()` implement the state communication and are called, respectively, just before and after any change in the scheme of processors.

The execution of aspect code is governed by the so-called aspect connectors (ACNs). An ACN is defined on (i.e. affects) a group of SCs and contains one or several *advice declarations*. Every advice consists of an *advice header* indicating a combination of join points at which the *advice body* will be executed. The syntax chosen for advice body is C++. The only code statements allowed here are invocations to the AC operations (i.e. interactions). Advice header usually indicates the instant at which advice body is executed. Allowed values are *before* and *after* the corresponding join point. In Fig. 4, part of an ACN declaration for the execution of `GridAdaptor` is shown. For instance, the first advice means that just before border communication routine `put_data()` is executed on any SC, the operation `initPut()` on `GridAdaptor` is invoked. The second advice expresses that `communicateGridMode()` is invoked on the same AC after the number of processors of any participant SC changes. A reference to the component `GridAdaptor` is retrieved using `getAdaptor()`. Similar ACN structures are needed for the management of the other ACs. Although in this example the use of predefined join points was enough, advice headers can refer to methods on the aspect interface as alternative join points. AspectSBASCO provides mechanisms to express the precedence of a set of advices affecting the same join point.

3.3 Implementation Issues

The current implementation of AspectSBASCO consists of a runtime system based on MPI, a programming framework in C++ and a source-to-source compiler to produce target language code structures. Scientific and aspect components are implemented using C++ and MPI. SCs inherit from the class `ScRoot` which provides access to distributed arguments. This class declares `get_data()` and `put_data()` as member functions. ACs are C++ classes that implement all operations of the corresponding aspect interfaces. An instance of the class `Framework` providing a set of common services can be retrieved at any point in the application. For example, `Framework` has methods that return up-to-date MPI intra- and inter-communicators to implement component parallelism.

An application consists of a collection of MPI programs. Specifically, one MPI program (so-called *worker*) is associated with each SC. Each worker manages one single SC instance together with a set of AC instances. The program executes using a group of processes whose size was expressed in the high-level composition language. When an aspect affects different SCs, the corresponding AC appears in several workers, and so aspect instances are created in several groups of processes.

Aspect weaving is the mechanism by which aspects and base code are combined to produce final applications. In our approach, this process is carried out at compile-time by modifying the source code of the SCs. More specifically, the aspect weaver examines ACN structures in order to include advice body interaction code at the corresponding points indicated in connector declarations. Every time a SC executes AC functionality (as a result of using the ACNs) the method invocation is carried out (in parallel) in all processes that contain both component instances. Then, it is the responsibility of the aspect developer to implement functionality that may range from simple local processing to complex parallel computation involving several groups of processes. The important thing is that our approach avoids any type of complex runtime structure to support aspect execution. The overhead of invoking ACs in final applications is equivalent to the cost of a standard C++ method call (the one declared in advice body).

Communication based on `get_data()` and `put_data()` (namely border interchange in this case study) is based on point-to-point message passing managed by the runtime system. This communication is efficiently carried out due to the information expressed in component configuration interfaces and skeletons.

4 Conclusions

This paper describes the use of AspectSBASCO for the development of a reaction-diffusion parallel problem. The complexity of functional and non-functional application concerns is managed effectively by means of a combination of paradigms.

The resulting approach leads to improved interoperability and application evolvability. In this case study, heterogeneous concerns such as grid adaptation and processor re-mapping procedures are implemented as well-modularized aspects. Pluggability is also provided. For instance, simply plugging the component that manages processor re-mapping, the application achieves the same numerical result more efficiently due to a better balance of the computation load. In addition, using high-level parallel skeletons raises the abstraction level of the development.

References

1. Armstrong, R., *et. al.*, The CCA Component Model for High Performance Scientific Computing, *Concurrency and Computation: Practice and Experience*, **18** (2), pp. 215 - 229, 2006.
2. Benoit, A., Cole, M., Gilmore, S., Hillston, J., Flexible Skeletal Programming with eSkel, in Proc. of the 11*th* International Euro-Par Conference, Lisboa, Portugal, pp. 761 - 770, 2005.
3. Danelutto, M., QoS in Parallel Programming Through Application Managers, in Proc. of the 13*th* Euromicro Conference on Parallel, Distributed and Network-Based Processing, Lugano, Switzerland, pp. 282 - 289, 2005.
4. Díaz, M., Rubio, B., Soler, E., Troya, J.M., SBASCO: Skeleton-based Scientific Components, in Proc. of the 12*th* Euromicro Conference on Parallel, Distributed and Network-Based Processing, A Coruña, Spain, pp. 318 - 324, 2004.
5. Díaz, M., *et. al.*, Using SBASCO to Solve Reaction-Diffusion Equations in Two-Dimensional Irregular Domains, Proc. of the 3*rd* International Workshop on Practical Aspects of Parallel Programming, Reading, UK, pp. 912 - 919, 2006.
6. Díaz, M., *et. al.*, Adding Aspect-Oriented Concepts to the High-Performance Component Model of SBASCO, (to appear) in Proc. of the 17*th* Euromicro Conference on Parallel, Distributed and Network-Based Processing, Weimar, Germany, 2009. (This paper can be accessed from http://www.lcc.uma.es/~tolo/publications.html).
7. Harbulot, B., Gurd, J., A Join Point for Loops in AspectJ, in Proc. of the 5*th* International Conference on Aspect-Oriented Sofware Development, Lancaster, UK, pp. 122 - 131, 2006.
8. Heineman, G.T., Council, W.T., *Component-Based Software Engineering: Putting the Pieces Together.* Addision Wesley, 2001.
9. Kiczales, G., *et. al.*, Aspect-Oriented Programming, in Proc. of the European Conference on Object-Oriented Programming, Jyvskyl, Finland, pp. 220 - 242, 1997.
10. Pelagatti, S., *Structured Development of Parallel Programs.* Taylor & Francis, 1998.
11. Quarteroni, A., Valli, A., *Domain Decomposition for Partial Differential Equations.* Oxford Science Publications, 1999.
12. Sobral, J.L., Incrementally Developing Parallel Applications with AspectJ, in Proc. of the 20*th* International Parallel and Distributed Processing Symposium, Rohdes, Greece, pp. 10, 2006.
13. Suvée, D., Fraine, B., Vanderperren, W., A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development, in Proc. of the 9*th* International SIGSOFT Symposium on Component-Based Software Engineering, Stockholm, Sweeden, pp. 114 - 122, 2006.
14. Vanneschi, M., The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, **28** (12), pp. 1709 - 1732, 2002.