COMPONENTS FOR HIGH-PERFORMANCE GRID PROGRAMMING IN GRID.IT *

Marco Aldinucci[1], Sonia Campa[2], Massimo Coppola[1],
Marco Danelutto[2], Domenico Laforenza[1], Diego Puppin[1,2],
Luca Scarponi[2], Marco Vanneschi[2], and Corrado Zoccolo[2]

(1) – *Istituto di Scienza e Tecnologie dell'Informazione, ISTI-CNR – Pisa, Italy*

(2) – *Dept. of Computer Science, University of Pisa – Italy*

{Marco.Aldinucci,Massimo.Coppola,Domenico.Laforenza,Diego.Puppin}@isti.cnr.it

{campa,marcod,vannesch,zoccolo}@di.unipi.it

Abstract   This paper presents the main ideas of the high-performance component-based Grid programming environment of the *Grid.it* project. High-performance components are characterized by a programming model that integrates the concepts of structured parallelism, component interaction, compositionality, and adaptivity. We show that ASSIST, the prototype of parallel programming environment currently under development at our group, is a suitable basis to capture all the desired features of the component model in a flexible and efficient manner. For the sake of interoperability, ASSIST modules or programs are automatically encapsulated in standard frameworks; currently, we are experimenting Web Services and the CORBA Component Model. Grid applications, built as compositions of ASSIST components and possibly other existing (legacy) components, are supported by an innovative Grid Abstract Machine, that includes essential abstractions of standard middleware services and a hierarchical Application Manager (AM). AM supports static allocation and dynamic reallocation of *adaptive* applications according to a performance contract, a reconfiguration strategy, and a performance model.

Keywords: Components, Structured Parallel Programming, Programming Models, Grid, Adaptive Applications, High Performance Computing, Reconfiguration.

1.   Introduction

In the context of Grid platforms at various levels of integration [1], a Grid-aware application must be able to deal with heterogeneity and dynamicity in

the most effective way (*adaptive applications*), in order to guarantee the specified level of performance in spite of the variety of run-time events causing modifications in resource availability (load unbalancing, node/network faults, administration issues, emergencies, and so on). With respect to traditional platforms, when the Grid is taken into account it is much more important to rely on application development environments and tools that both guarantee high-level programmability, application compositionality, software interoperability and reuse, and, they are able to achieve high-performance and the ability to adapt to the evolution of underlying technologies (networks, nodes, clusters, operating systems, middleware) [2–8]. Achieving this *high-level view* of Grid application development is the basic goal of our research, in the *Grid.it* national project [9] and in associated initiatives at the national and European level.

In order to be able to design, develop and deploy such kind of high-performance Grid-aware applications efficiently, we are interested in innovative programming environments that

i) support the programmers in all the activities related to parallelism exploitation, by providing some kind of *structured* primitives for parallelism exploitation;

ii) allow to achieve full *interoperability* with existing software, both parallel and sequential, either available in source or in object form;

iii) support and enforce reuse of already developed code in other applications.

In particular, we want to exploit the experience of our group in the design and implementation of structured parallel programming environments [10–12] to target Grids composed of clusters or networks of heterogeneous workstations

Table 1. Layered software architecture for Grid-aware applications

Applications	Complex, multidisciplinary applications. Programmer only expresses the kind of parallelism, without being concerned with any detail involved in its exploitation or related to the fact that the target architecture is a Grid. The user supplies, for each parallel component, a performance contract to be satisfied.
High-performance programming environment	Exploitation of structured parallelism, basic mechanisms supporting compositionality, interoperability and reuse of existing software, support for the dynamic control of performance contracts and adaptivity.
Grid abstract machine	Functionalities and mechanism supporting the programming environment, including dynamic application management and all the needed features from the *resource*, *collective* and *connectivity* levels of Grid middleware
Basic hardware-software platform	

[13–17]. We think that a *component-based* programming environment is a suitable starting point to achieve the goals just stated.

In this work, we discuss the essential features of a programming environment that is based both on the component model and on structured parallelism. The programming environment is a layer of a larger picture, such as the one in Table 1. We will first discuss the features of a component-based parallel programming model. Then, we'll take into account how these features are currently or will be soon supported in ASSIST. ASSIST is a structured parallel programming environment that was originally designed to address cluster and networks of TCP/IP workstations only, in the framework of the Italian national project ASI-PQE2000 [18]. We show that ASSIST is a suitable basis to capture all the desired features in a flexible and efficient manner, and, in particular, we discuss how the original ASSIST environment is currently being transformed into a component-based, Grid-aware parallel programming environment. This evolution of the ASSIST environment is being performed in the framework of the Grid.it Italian national project. Grid.it is a 3 year project involving major research institutions in Italy aiming at providing innovative programming methodologies and tools for Grids. The project has a specific work-package, leaded by our group, aimed at designing and implementing a prototype high-performance parallel programming environment for Grids. Component-based ASSIST is the expected, assessed outcome of this work-package.

1.1 Related Work

Several studies recognized that component technology could be leveraged to ease the development of Grid applications. We assume as reference component standards the CORBA Component Model (CCM), because of its clean and rich component model [19–20], and the Web/Grid Services [21], because they are emerging as the standard infrastructure to integrate heterogeneous systems [21]. The Common Component Architecture (CCA) is a prominent standardization effort, aiming at the definition of a high-performance oriented component architecture [22]. We depart from CCA-based approaches like CCaffeine [23] as we explicitly deal with component composition issues (see Sect. 1.2 and Sect. 1.3.2).

Our approach differs from that of GridCCM [24], as the latter focuses on communication optimization, while our work targets application adaptivity and Grid-awareness in general.

We are closer to the GrADs project [25] with respect to the concept of adaptivity and in some architectural aspects, but we differentiate in the programming model. Our model, being based on structured parallel programming, has the ability *i)* to synthesize from the parallel structure of applications the performance models used to adapt their computation, and *ii)* to control the applica-

tion configuration at run-time, using a parametric implementation of the parallel programming constructs.

A closely related project to our one is ProActive [26], which extends the Fractal component framework for Java [27] to support parallel, reconfigurable component architectures on the Grid. We share with the Proactive project the departure from flat component models to move toward explicit component composition, the emphasis on run-time adaptivity of component structures, and the exploitation of these hierarchical structures to manage application reconfiguration. We differentiate from that research as we are not limited to Java, we have instead a well defined, language-based separation between sequential program behaviour and parallel coordination, at the intra- and inter- component levels. Our primary goal in improving component interaction is also different, as we want to exploit a broader set of interaction mechanisms than RMI. On the contrary, Proactive primarily exploits parallel and collective RMI abstractions (and their optimizations e.g. by means of futures) to extend the sequential component framework to a parallel, distributed one.

2. High-performance components for Grid-aware applications: computational model

The basic features needed to implement a component-based, high performance programming environment targeting the Grid include most of those already implemented in currently available component models, such as JavaBeans [28] or the CORBA Component Model [19]. These features are those needed to implement a distributed or a parallel program, that is they are mainly framework features and communication/interaction mechanisms. Concerning the framework mechanisms, we obviously need handy ways of both creating/instantiating and calling components across the different processing elements of the target architecture. We also need mechanisms and features that offer the programmer the possibility of controlling the parallel behavior of the Grid-aware application.

In the *Grid.it* programming environment, we want to provide such mechanisms in the most abstract way possible. In particular, we want to leave the programmer the ability of concentrate on the functional behavior of the application, as well as on the qualitative aspects of parallelism exploitation. That is, we want to relieve the programmer of the responsibility of directly handling all the details related to the quantitative aspects of parallelism exploitation, and all those related to the usage of specific Grid middleware mechanisms.

To understand the features of a high-performance components environment, it can be useful to distinguish three conceptual levels: a) computational model, b) functional and non-functional interfaces and c) support architecture for Grid-aware applications. In this section, we start to deal with the first issue. The other ones are discussed in successive sections, where the Grid.it approach

based on ASSIST is presented. From the point of view of the computational model, we propose that high performance components are studied and characterized in terms of the following features: *parallelism, component interaction, composition,* and *adaptivity*.

Parallelism.   In general, components have an internal parallel structure (*intra-component parallelism*). It must be possible to configure several, distinct *versions* of the same component, all versions having the same functional interfaces. Moving from one version to another one could be done by recompiling and/or reloading, in the most simple situations (*static versions*); however, we are interested also in parallel components that are able to change their internal structure/behaviour at run time (*dynamic versions*), depending on functional conditions (e.g. predicates on the computation state) and/or on non-functional conditions (e.g. variations in the achieved performance). In addition to intra-component parallelism, the *inter-component parallelism* is fundamental for high-performance component applications as well.

Component Interaction.   Most component-based frameworks supply a way to declare the public services provided by a component and to invoke a service provided by a (remote) component. The mechanism is based on the `uses`/`provides` port abstraction. Being essentially a new edition of the RPC/RMI paradigm, this is sufficient to guarantee proper interactions between components, when they follow this simple client/server model. As an example, task farm computations (that is, embarrassingly parallel ones) can be implemented very efficiently using these mechanisms.

However, different mechanisms are needed to implement other parallel patterns. For instance, pipelines cannot be easily expressed by means of the uses/provides port mechanism. Therefore, we assume that at least two distinct mechanisms are implemented:

- **events**, that is a way to register event handlers and to propagate events through the component network. This mechanism is already present in CCM. It can be exploited to implement all the typical asynchronous activities of parallel computations, such as monitoring.

- **streams**, that is a way to have uses/provides ports that implement data-flow-like channels for sequences of unidirectional typed communications, without incurring in the performance penalties related to the return messages and synchronizations typical of the plain uses/provides port mechanism.

In general, a component has several input streams and several output streams, that can be used in a *data-flow* and/or in a *nondeterministic* fashion. A rich set of interaction mechanisms, which are typical of the parallel computation models,

is fundamental also in order to implement higher-level abstractions, such as complex workflow-based PSEs.

Composition. Currently available component models allow components to interact in several different ways. However, only a few of them (e.g. CCM with the assembly construct) consider component composition as a primitive operation. In our opinion, composition is fundamental to allow more and more complex parallelism exploitation patterns to be developed and provided to the user as components. As derived from our experiences with algorithmic skeletons, as soon as an efficient mechanism to exploit basic parallelism patterns is available (see [29–30]) then the need for nesting/composition mechanisms arises (see [14, 10]). By exploiting pattern composition, new parallel patterns can be programmed, best suited to user's needs. Furthermore, by properly restricting the visibility of user-defined, composed parallel patterns, different degrees of programmability of parallel applications can be presented to different classes of users.

In our context, we assume to design a *structured*, component-based programming environment, and we actually want to be able to exploit composition of components to provide *new*, non-primitive components supporting the development of Grid-aware parallel applications.

In conclusion, we need to define complex computation structures by means of the parallel composition of parallel components. A composition of components can be defined and reused as a new component in more complex structures. We assume that a general, explicitly parallel structure is encapsulated into a component in order to create a basic parallel component.

Adaptivity. A *Grid-aware* application must be able to deal with heterogeneity and dynamicity in order to guarantee the specified level of performance, in spite of the variety of run-time events that can change resource availability. A component must be characterized by *non-functional interfaces*, related to the performance control, and by features that allow the programmer to specify how the computation adapts at run time. Moreover, these features have to be implemented efficiently at the run-time support level.

A strong relationship exists between the four features stated above. They must be integrated consistently in a global approach, framework, or better, in a programming model for high performance Grid programming. In Grid.it, we use ASSIST as the programming model able to satisfy this requirement.

3. ASSIST as the Basic Programming Model for High-Performance Components

We introduce ASSIST as derived from the NOW/COW programming environment as *the* way to denote parallel, high-performance, component-based, Grid applications. We also discuss how already implemented ASSIST features match the requirements emerging from the previous Section discussion or can be exploited/improved to match such requirements.

3.1 Basic Features of the ASSIST Programming Model

ASSIST programs are structured as *generic graphs* (identified by the `generic` keyword), where nodes are parallel or sequential modules and arcs represent typed *streams* of data/objects. No constraint is imposed to the form of graphs, though "structured" graphs, such as those typical of a classical skeleton model, are a notable class of cases that have efficient implementations.

All the interactions that are of interest in the composition of high-performance components are implemented easily and efficiently with the ASSIST streams. Streams are inherently *asynchronous*, however RPC/RMI interactions can be emulated effectively.

Parallel modules are expressed by a *generic skeleton*, called `parmod`. In this context, "generic" means that a `parmod` is a general-purpose construct that can be tailored, for each application, to specific instances of classical stream-parallel and/or data-parallel skeletons, and also to new forms of regular and irregular parallelism.

A `parmod` operates on multiple input streams and multiple output streams. Several distribution and collection strategies are provided for the input and output streams respectively. Moreover, input streams can be controlled in a *data-flow* or in a *nondeterministic* manner. Nondeterminism is important to model several instances of workflow structures, as well as interaction by *events*.

The parallel computation expressed in a `parmod` is decomposed in sequential units assigned to abstract executors called virtual processors (VP). The `parmod` can have an explicitly defined *internal state* for the duration of the computation. This feature is important in many cases, for example in nondeterministic/reactive computations, as well as in many irregular and dynamic computations.

As in any model for structured parallel programming, the `parmod` construct is characterized by the important property that *the implementation model is parametric*. This means that the realization of the run-time support is largely independent of the actual mapping of the virtual processors of `parmod` onto the real processors: this is true as far as it concerns the distribution of functions, the distribution of data, and the communication.

In the same way, an instance of a `parmod` is characterized by a *performance model*, which is parametric with respect to the actual realization. In the structured parallel programming community, a large amount of performance models have been provided for many stream-parallel and data-parallel skeletons. In ASSIST, we exploit this experience in order to characterize the behaviour of a `parmod` in terms of the performance it can offer according to the actual mapping. In many cases, i.e. where the parametric behaviour is predictable, the performance model is recognizable at compile-time, while in other, more dynamic cases, the association of a performance model to the computation, expressed by a `parmod`, requires some annotation by the programmer and/or the knowledge of the past history of the system or application. All the performance models can be made available, in a sort of *Performance Model Repository*, to the strategies implemented by the compiler and to the run-time support.

The "parametricity" feature (parametric implementation model and parametric performance model) is the basis for the implementation of adaptive strategies in high-performance Grid-aware applications. How this issue is dealt with in Grid.it will be discussed in detail in Sect. 1.4.1.

The "genericity" of the `parmod` construct offers an interesting opportunity to express *adaptive parallel computations* by program. That is, the same `parmod` (the same collection of virtual processors, input and output streams, and state variables) can express different parallelism forms according to the value of the internal state or of the input values. For example, in the Divide & Conquer implementation of the C4.5 algorithm [18], in different phases the Divide module has a data-parallel or a task-farm-like behaviour, in order to optimize the available parallelism in each phase of the computation. Because of the huge amount of data associated to the internal state, this flexible implementation in ASSIST is much more efficient than an equivalent version in which the data-parallel phase and the farm-like phase are expressed by different specific skeletons.

In other words, ASSIST offers a powerful feature for expressing and implementing adaptive computations: the same computation can be expressed according to several alternative strategies. As we describe in Sect. 1.4.1, alternative strategies can be associated to values generated by the program at run-time, or they can be selected according to the actual performance values with respect to the performance contract.

ASSIST modules can refer to *external objects* during any phase of the computation, i.e. to objects not defined by the ASSIST coordination language and, consequently, that are referred according to their specific interfaces/APIs. In addition to libraries and system facilities (I/O, file, data bases), current external objects in ASSIST are shared variables, CORBA remote objects, storage objects and data repositories. The existence of external objects is a further facility to express alternative strategies for adaptive computations.

ASSIST provides *full interoperability with CORBA*: not only an ASSIST program can act as a client of a CORBA server, but, most significantly, an ASSIST program can be automatically compiled as a CORBA object with RMI-like synchronous invocations or with stream-like asynchronous data passing. It has been shown [31] that the overhead introduced by the program transformation is definitely acceptable for many parallel applications.

This experience proves that interoperability features can be efficiently merged into the ASSIST model, in order to design applications as composition of components, some of which will eventually be parallel.

3.2 ASSIST and Components

We figured the road-map transforming ASSIST programs/modules to components as follows: first, we allow ASSIST modules to be encapsulated as components in existing, well known component frameworks; then we include in the component framework all the mechanisms needed to implement high-performance applications; eventually, we integrate the framework in such a way that parallel ASSIST components and existing legacy components can coexist in an high-performance parallel Grid application. Following this road-map, our current research [31] will produce the next version of ASSIST (2.0), where an ASSIST program, expressed either as a single ASSIST module or as a graph of parallel or sequential modules, is considered as *a high-performance component* which:

a) *can be composed using standard component frameworks*, in addition to the native ASSIST mechanisms,

b) *exports non-functional interfaces and features automatic support to adaptive applications*, which will be discussed in the next section.

W.r.t. standard frameworks, we are experimenting several solutions based on *Web Services* (WS) and the *CORBA Component Model* (CCM). The implementation approach is similar to the one already adopted for CORBA 2 interoperability, i.e. the compiler generates bridge ASSIST modules, which support the various kind of component ports related to the functional interfaces, as well as ports related to the non functional interfaces.

From the point of view of compositionality, the ASSIST based approach offers the following features:

i) a component (either WS or CCM) encapsulates an ASSIST (sub)graph,

ii) components can be *composed according to ASSIST mechanisms* or according to the mechanisms of the *standard component framework* adopted.

In the first solution of point *ii)*, two or more components, being ASSIST graphs, are composed by the `generic` construct, which describes the structure of the resulting ASSIST graph in terms of component modules and streams (and possibly external objects). The composed ASSIST program is automatically

compiled into a standard component. This solution can be adopted when the components are all (new or existing) ASSIST programs whose ASSIST source code is available.

In the second solution of point *ii)*, the programmer uses the interaction mechanisms of the component framework (ports) to compose two or more components. This approach is typically adopted when one or more of the components of the application are existing (legacy) components that have not been designed in ASSIST.

4. Support Architecture for Applications Based on High-Performance Adaptive Components

According to what stated in previous sections, several critical points have to be addressed when we tackle adaptivity control in Grid-aware applications. In particular, non-functional interfaces and reconfiguration strategy and application management have to be taken into account. In this section, we will discuss how these features have been taken into account in the design of our prototype of the component-based parallel programming environment ASSIST 2.0.

4.1 Non-functional Interfaces and Reconfiguration Strategy
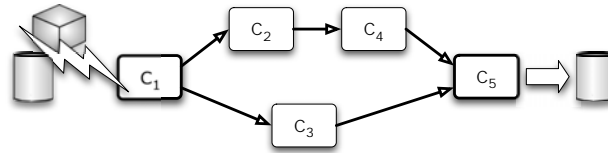
We assume that a Grid-aware application is a composition of high-performance components. That is, we restrict to the case where no legacy, non-parallel components are used in the application.

Let us consider the case in which such components are ASSIST components (graphs of sequential modules and/or `parmods`), composed by means of the `generic` graph construct. In addition to functional interfaces, that are automatically generated at compile time (out of the ASSIST code), each component is characterized by *non-functional interfaces*. They are expressed as annotations in a proper formalism, which is translated by the compiler into a run-time representation based on XML. Such annotations convey information about *performance contract, reconfiguration strategy,* and *allocation constraints*. The template of an ASSIST component thus assumes the form shown in Fig. 1.

Figure 1. Template of an ASSIST component

ASSIST `parmod` modules and their `generic` graph composition
Performance contract (annotations)
Specification of the reconfiguration strategy and performance models for Assist modules (ASSIST + annotations)
Allocation constraints (annotations)

Figure 2. Example of an adaptive application expressed by parallel components

Performance Contract. Many parameters can be used to specify the performance level that is required for the application. In this paper, we refer to the processing bandwidth (service time) in stream-based computations, and/or to the completion time, which is significant also for non-stream computations. However, the following discussion is largely independent of the specific performance parameters adopted.

The performance contract can be specified for the *whole* application and/or for every *single* component. If the required performance of every component is specified, the required performance of the whole application can be derived at compile time using the knowledge of the graph. For example, the methodology of queueing networks can be used, both in the case of asynchronous stream and in the case of RMI-like interaction. If only the whole application performance is specified, it is still possible to derive rough information at compile and run time on the performance of the single components, using profiling estimates and, most important, taking into account that an ASSIST component is a composition of ASSIST modules for each of which a performance model may be known on the basis of a Performance Model Repository.

Additional information related to communication bandwidth and latency must be estimated; of course, the reliability of this information may not be very accurate. However it is exactly because of these and other inaccuracies, which are inherent of Grid platforms, that we need a support for adaptive Grid-aware applications.

Reconfiguration Strategy. For each component, the application designer specifies which way the component has to be restructured at run-time if and when the performance contract happens to be no longer satisfied.

The reconfiguration strategy is basically expressed in ASSIST with the addition of some annotation. In Sect. 1.3.1 we saw that ASSIST allows the programmer to express alternative strategies (e.g. different parametric forms of parallelism) directly in the same program, when their activation depends on the values of some program variables (e.g. the internal state of a `parmod`). Moreover, the programmer can also specify that alternative strategies must be activated when the performance contract is violated. Let us consider the following example (Fig. 2).

a) Component C1 is an interface towards a Grid memory hierarchy, that virtualizes and transforms data sets available on the Grid into two streams of objects, the first one (whose elements have an elementary type) is sent to C2, and the other (whose elements have array type) is sent to C3. C1 may be an existing component available on the Grid, mediated by an ASSIST program.

b) C2 is a parallel component encapsulating an ASSIST program. The reconfiguration strategy of C2 specifies that

- "`by default`" C2 is a sequential module executing a certain function F;

- when the current performance level must be adjusted C2 is transformed into a farm computation whose workers execute the same function F. The actual number of workers will be determined *parametrically* at run-time, according to the performance model and to the current availability of resources.

c) C3 is a parallel component encapsulating an ASSIST data-parallel program operating on each stream element of array type. As in the case of C2, the strategy of C3 specifies that its parallelism degree (i.e. the amount of real processors onto which the virtual processors are mapped) can be modified, if needed.

d) C4 is a parallel component encapsulating an ASSIST program which, by `default`, is a sequential module, but it can be restructured into a `parmod` operating on the input stream according to a data-parallel or to a task farm style, *depending* on the values of the `parmod` state and on the input values themselves. Therefore, in this case the adaptation principle is applied at two levels: at the program level and at the run-time support level.

e) C5 is a parallel component encapsulating an ASSIST program operating nondeterministically on the input values received from C3 or C4, and transforming the two streams into a data set.

Let us assume that, at a certain time, some monitoring activity signals that C2 is becoming a bottleneck and that this causes a substantial degradation of performance of the whole application. C2 can be transformed into a version with suitable parallelism degree. In this case other components may have to be restructured (e.g. C4,C5) in order to guarantee the level of performance. As previously stated, this is possible according to a global strategy based on the knowledge of the application structure.

When restructuring data-parallel components (C3), the strategy must be applied also to the re-distribution of the data constituting the internal state of a `parmod`.
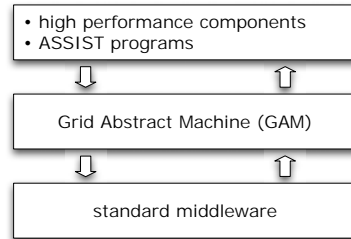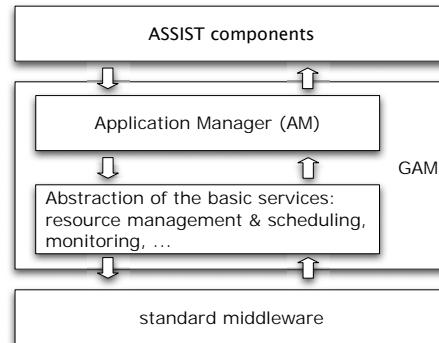
Figure 3. Grid.it software architecture

```
┌──────────────────────────────────┐
│ • high performance components    │
│ • ASSIST programs                │
└──────────────────────────────────┘
        ⇩            ⇧
┌──────────────────────────────────┐
│   Grid Abstract Machine (GAM)    │
└──────────────────────────────────┘
        ⇩            ⇧
┌──────────────────────────────────┐
│       standard middleware        │
└──────────────────────────────────┘
```

Figure 4. Grid Abstract Machine

```
┌──────────────────────────────────┐
│        ASSIST components         │
└──────────────────────────────────┘
        ⇩            ⇧
┌──────────────────────────────────────┐
│ ┌──────────────────────────────────┐ │
│ │     Application Manager (AM)      │ │
│ └──────────────────────────────────┘ │
│      ⇩            ⇧          GAM      │
│ ┌──────────────────────────────────┐ │
│ │ Abstraction of the basic services:│ │
│ │ resource management & scheduling, │ │
│ │ monitoring, …                     │ │
│ └──────────────────────────────────┘ │
└──────────────────────────────────────┘
        ⇩            ⇧
┌──────────────────────────────────┐
│       standard middleware        │
└──────────────────────────────────┘
```

More sophisticated strategies can be expressed in ASSIST than those shown in the example: the strategy of C4 could depend just on performance requirements instead of predicates on the internal state, and other alternative strategies could exploit external objects, as opposed to strategies based on the stream composition only.

Allocation Constraints.   In general, restructuring high-performance components involves resources belonging to different Grid nodes. In the example above, the new workers of C2 can be allocated onto processors of a cluster from a different Grid node. There are instead several cases in which we must put constraints on resource allocation. For instance, several components (C1 and C5, say) can be executed only on certain Grid nodes and no reconfiguration is permitted, either due to security or privacy reasons, or to requirements related to the kind of resources needed to operate on the data sets. This kind of information has to be associated with the reconfiguration strategy of every component.

4.2   Application Management for Reconfiguration

We eventually come to the point where the implementation of the high-performance component framework has to be taken into account. The software architecture of Grid.it component-based parallel programming environment is organized as shown in Fig. 3. The run-time environment of ASSIST 2.0 is implemented on top of a *Grid Abstract Machine* (GAM), which in turn is implemented on top of existing middleware (currently a version of the Globus Toolkit) and realizes the functionality needed by the programming environment to support high-performance, component-based Grid-aware applications.

As shown in Fig. 4, the GAM consists of the Application Manager and of the *abstraction of the services* for Resource Management and Scheduling, Monitoring, and other services (accounting and so on).

Application Manager Structure. The Application Manager (AM) has a *hierarchical structure.* Fig. 5 illustrates a first case of an application consisting of just one component structured as a graph of ASSIST modules

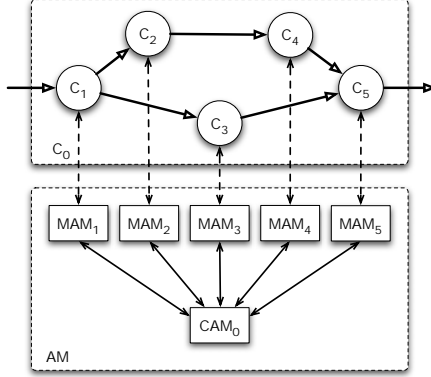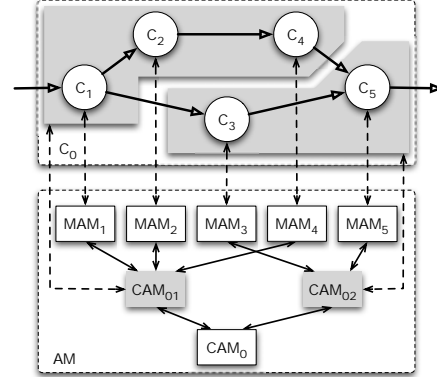*Figure 5.* Example of an ASSIST component

Figure 6. Two interacting ASSIST components

Each ASSIST module has associated a *Module Application Manager* (MAM): each MAM_i is responsible of the configuration control of the associated module. A global strategy for the configuration control of the whole component is implemented by the *Component Application Manager* (CAM).

In the example of Fig. 2, let us assume that each of the components $C_1, \ldots C_5$ consists of one module and the whole application is wrapped as a component C_0. Then, the AM consists of a two-level tree in which $MAM_1, \ldots MAM_5$ are leafs and CAM_0 is the root.

Application management is, in principle, a centralized process. It can be realized in a decentralized way, according to several strategies. We suppose that the decentralization is realized in a *hierarchical* manner. Moreover, for availability reasons, we assume that the root is designed according to principles of fault-tolerance, e.g. using redundancy and, possibly, mechanisms for checkpointing.

The hierarchical structure can be extended at any level according to the compositionality and abstraction strategy adopted for the application. For example, Fig. 6 shows the same application of Fig. 5 in which we recognize two components, C_{01} consisting of modules C_1, C_2 and C_4, and C_{02} consisting of modules C_3 and C_5. The whole application, which can be seen as the composition of C_{01} and C_{02}, is considered as a component C_0. Thus, in addition to the same leaf managers ($MAM_1, \ldots MAM_5$), we have CAM_{01} and CAM_{02} at the second level, and CAM_0 at the root.

Module Application Managers (MAMs). The MAM level is an ASSIST abstraction, independently of the fact that the application is structured as a (hierarchical) composition of higher-level components. At the MAM level we implement the configuration control of the single ASSIST modules (`parmods`) exploiting the associated performance model. As introduced in Sect. 1.3, a *Performance Model Repository* is provided in the programming environment and it is updated according to the history of the applications running in the system. The specific performance model of each module, to be found in the performance model repository, can be recognized

- by the compiler, according to the knowledge of some parallelism forms. Examples of parallelism forms which are statically recognizable in AS-SIST are farms (with and without state), data parallel computations with fixed or variable static stencils, and some mixed combinations of stream- and data-parallelism;

- by the programmer, in all the cases in which his knowledge is more accurate, and/or a new parallelism form and the associated performance model are expressed by properly and specifically instantiating a `parmod` construct.

As discussed in Sect. 1.3 and 1.4, the reconfiguration strategies of an AS-SIST module can exploit different forms of parallelism, different data distribution/collection strategies, and the usage of external objects. Moreover, in case of data-parallel behaviour, data can be redistributed at run-time according to *load balancing* strategies that cannot be (have not been) recognized at compile-time. Notice that, for stream-parallel farm-like structures, load balancing is always implemented by the run-time support .

MAM behaviour is basically *event-driven*, where the events indicate the opportunity/necessity for restructuring the associated ASSIST module. One kind of event is generated according to the outcome of Monitoring service invocations. In this case, the MAM can provide the following sequence of actions:

- a restructuring strategy is taken into account, either based on the ASSIST alternative descriptions or on load balancing for data parallel modules;

- in case of alternative parallel strategies, the performance model from the performance model repository is applied, a proposed solution to reconfiguration is derived,

- the non transient nature of the event is assessed and therefore

- the father CAM is informed about this proposal.

The MAM can also receive an event by the father CAM indicating that it has to apply a restructuring strategy because a global variation of performance has been detected. For example, in the computation of Fig. 6, CAM_{01} can ask MAM_4 to apply a reconfiguration strategy in order to increase the C_4 bandwidth in consequence of an increase of bandwidth of C_2.

Component Application Managers (CAMs). Each CAM applies control strategies *at a global level* for the associated component.

As indicated above, a CAM can receive proposals of restructuring by the child MAMs. In this case, the CAM has to apply a global performance model (e.g. queueing network based model) in order to individuate a good solution to the restructuring of one or more of the children modules. The Allocation Constraints, indicated in the non-functional interfaces of the component, are also applied during this process.

Recursively, a CAM can receive reconfiguration requests from father CAMs, and can send them reconfiguration proposals. The root CAM (CAM_0) is responsible for the final decisions in the global reconfiguration control which, as seen, is a sort of parallel and asynchronous Divide & Conquer strategy applied along the hierarchical Application Manager structure.
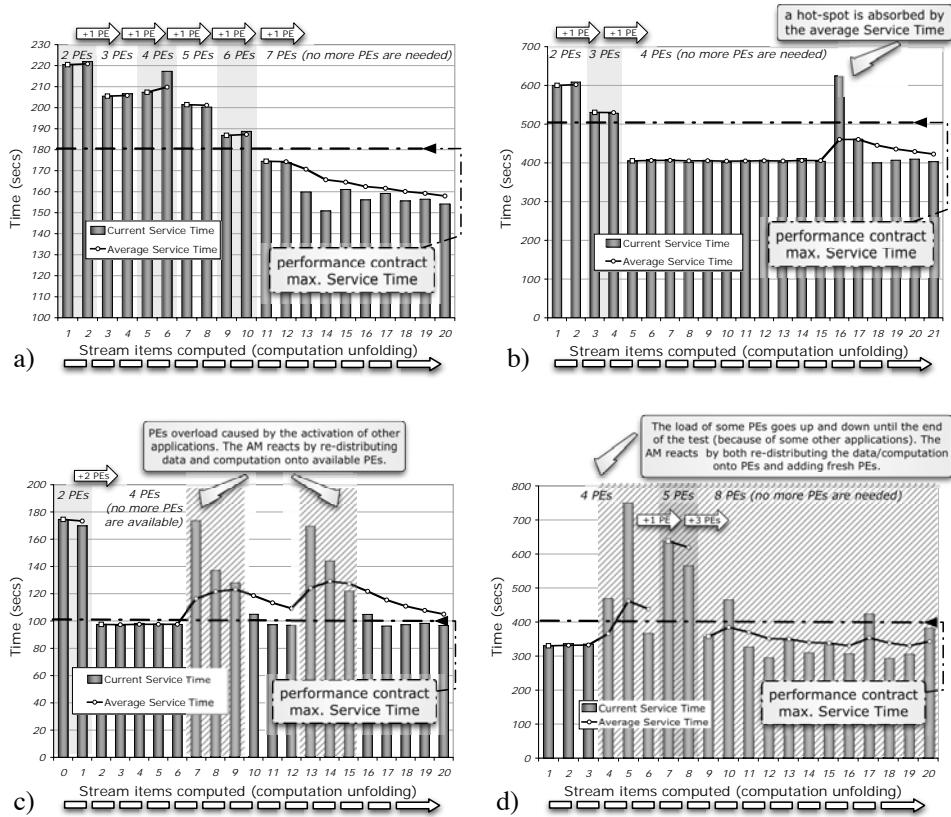
At each CAM level, the Resource Management and Scheduling services provided by the Grid Abstract machine are utilized. Notice that such services do not necessarily coincide with the services in the standard middleware, instead they represent the abstraction that are strictly needed by the Application Manager. That is, a "RISC-like" GAM is defined, though starting from a monolithic Middleware level; in the next future, this GAM service structure could be the basis for the proposal of a new Risc-like Middleware level.

5.   Experiments

The features to be included in the Grid Abstract Machine have been experimented using Lithium, a full Java, RMI based, structured parallel programming environment [16]. Lithium has been often used to experiment solutions that have been then moved to ASSIST, as the former is much more compact and easy to modify than the latter. These experiments showed that

- almost perfect scalability can be achieved, even in the case when heterogeneous resources are used for the execution. The measured execution times are usually no more than 5% away from the ideal ones.

- good tolerance to typical "dynamic" situations, such as the presence of faulty nodes, can be achieved. In presence of a number of faulty nodes not exceeding 20% of the nodes used to compute the parallel application, an increase of less than 10% of the total execution time has been measured.

Figure 7. Experiments in dynamic restructuring of parallel components

The proposed AM organization and behaviour, described in Sect. 1.4, have then been evaluated on some ASSIST examples, emulating the dynamic features of the run-time support and of the MAM/CAM hierarchical organization. The implementation of a first version of this support is on-going. Figure 7 shows the results achieved in a set of reconfiguration experiments [32]. The experiments have been performed using an application whose structure was a pipeline of three stages: the first and the third stages are data servers and stream managers, and the second stage is a data parallel version of the finite difference method for solving differential equations (Jacobi method).

Figure 7-a shows how the Application Manager can satisfy the performance contract (service time) by increasing the amount of real processors onto which virtual processors of a data-parallel stencil computation are mapped. Figure 7-b shows the effect of transient variations of system load, that have no effect in the performance in the long period (we employed an exponential mean, reset at

18

every reconfiguration). Figure 7-c shows the more serious effects of a perturbing overload caused by the creation of a new application onto the same processing nodes. In this case, we assume that no more processing nodes are available, thus only the load balancing solution is attempted, with a suitable redistribution of data partitions implemented directly by the run-time support. Fig. 7-d shows a situation similar to Fig. 7-c: the difference is that more processing nodes are now available, and, after a first attempt of applying data redistribution, the degree of parallelism of the data parallel module is successfully increased.

These experiments have been performed on a heterogeneous cluster, confirming that the Application Manager overhead is quite acceptable, and the performance obtained is the same as in the case of optimal static mapping. More intensive experiments to evaluate the Grid overhead are on-going.

6.   Conclusion and On-going Work

In this paper we have outlined the guidelines of our research in high-performance component-based programming environments which are Grid-aware, in the context of the Grid.it national project. We have shown that ASSIST is a suitable programming model on which to build all the complex features of the programming environment. In addition to showing the feasibility of component-based ASSIST, we have proposed a Grid Abstract Machine, including a hierarchical Application Manager to control resources for dynamically adaptive applications, structured by ASSIST components.

In the Grid.it project, a large amount of application case-studies provide intensive experiments and benchmarks of the proposed ideas and tools. In the short term, the on going research activity will produce a new version of ASSIST with a full implementation of all the features discussed in this paper and providing full interoperability with both CCM components and plain web services as well.

In the medium term, the research will produce ASSIST version 2.0, in which the ideas and first prototypes for the Grid Abstract Machine will be studied, implemented and evaluated and the whole, component-based, high-performance, structured parallel programming environment will be deployed.

References

[1] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*, chapter 11, The Globus toolkit. Morgan Kaufmann Pub., S.Francisco, CA, 1998.

[2] Rob van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri Bal. Ibis: an Efficient Java-based Grid Programming Environment. In *ACM JavaGrande IS-COPE 2002 Conference*, pages 18–27, Seattle, WA, November 2002.

[3] F. Berman, R. Wolski, H. Casanova, et al. Adaptive Computing on the Grid using AppLeS. *IEEE Trans. On Parallel and Distributed Systems*, 14(5), 2003.

[4] B. Ensink, J. Stanley, and V. Adve. Program Control Language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*. in press.

[5] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive Grid programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002)*, 2002.

[6] C. Prez, T. Priol, and A. Ribes. PaCO++: a parallel object model for high performance distributed systems. In *Distributed Object and Component-based Software Systems, Hawaii Int. Conf. System Sciences*. IEEE Comp. Soc., 2004.

[7] T. Priol. Programming the Grid with Distributed Objects. In *Proc. of Workshop on Performance Analysis and Distributed Computing (PACD 2002)*, 2002.

[8] D. Thain, T. Tannenbaum, and M. Livny. *Grid Computing: Making the Global Infrastructure a Reality*, chapter Condor and the Grid. Wiley, 2003.

[9] Grid.it project: Enabling platforms for high-performance computational grid oriented to scalable virtual organizations. MIUR, FIRB National Research Programme, November 2002.

[10] M. Cole. Bringing skeletons out of the closet, 2004. to appear in Parallel Computing.

[11] H. Kuchen. A skeleton library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. Springer Verlag, August 2002.

[12] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Taa. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1684, december 2002.

[13] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220, July 1992.

[14] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[15] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, December 1999.

[16] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003.

[17] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of AS-SIST, an Environment for Parallel and Distributed Programming. In H. Kosch, L. Boszormenyi, and H. Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, number 2790 in LNCS, pages 712–721. Springer Verlag, august 2003. Kalgenfurt, Austria.

[18] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, december 2002.

[19] OMG. Corba component model, v3.0, November 2001. Document ptc/2001-11-03, available at `http://www.omg.org/`.

[20] Object Management Group. The Common Object Request Broker: Architecture and Specification. `http://www.omg.org`, 2000. Minor revision 2.4.1.

[21] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, June 2002.

[22] CCA working group. The common component architecture technical specification - version 0.5 (as amended through 3/5/2001). `http://www.cca-forum.org`.

[23] Ccaffeine home page, 2003. `http://www.cca-forum.org/ccafe/`.

[24] Alexandre Denis, Christian Pérez, Thierry Priol, and André Ribes. Bringing High Performance to the CORBA Component Model. In SIAM Conference on Parallel Processing for Scientific Computing, February 2004.

[25] H. Dail, O. Sievert, F.Berman, H. Casanova, A. YarKhan, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. Scheduling in the Grid Application Development Software Project. In *Resource Management in the Grid*. Kluwer, 2003. To appear. Available on GrADS WWW home, `http://hipersoft.cs.rice.edu/grads/`.

[26] Françoise Baude, Denis Caromel, and Matthieu Morel. On hierarchical, parallel and distributed components for Grid programming. In V. Getov and T. Kielmann, editors, *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2004.

[27] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *7th International Workshop on Component-Oriented Programming (WCOP02)*, ECOOP 2002, Malaga, Spain, June 2002.

[28] Sun. Javabeans home page. `http://java.sun.com/products/javabeans`, 2003.

[29] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[30] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Europar '96*, pages 601–614. Springer-Verlag, 1996.

[31] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a Research Framework for High-performance Grid Programming Environments. Technical Report TR-04-09, Dept. Computer Science - University of Pisa, February 2004. TR-04-09 available at http://www.di.unipi.it/Ricerca/TR.

[32] Luca Scarponi. Supporti alla programmazione Grid-aware – Esperienze di allocazione dinamica di programmi ASSIST. Master's thesis, University of Pisa, 2004.