# STRUCTURED IMPLEMENTATION OF COMPONENT BASED GRID PROGRAMMING ENVIRONMENTS[*]

M. Aldinucci, M. Coppola
*ISTI/C.N.R. – Via Moruzzi 1, 56100 PISA, Italy*
{aldinuc,coppola}@di.unipi.it

S. Campa, M. Danelutto, M. Vanneschi, C. Zoccolo
*Dept. Computer Science – Largo Pontecorvo 3, 56122 PISA, Italy*
{campa,marcod,vannesch,zoccolo}@di.unipi.it

**Abstract** The design, implementation and deployment of efficient high performance applications on Grids is usually a quite hard task, even in the case that modern and efficient grid middleware systems are used. We claim that most of the difficulties involved in such process can be moved away from programmer responsibility by following a structured programming model approach. The proposed approach relies on the development of a layered, component based execution environment. Each layer deals with distinct features and problems related to the implementation of GRID applications, exploiting the more appropriate techniques. Static optimizations are introduced in the compile layer, dynamic optimization are introduced in the run time layer, whereas modern grid middleware features are simply exploited using standard middleware systems as the final target architecture. We first discuss the general idea, then we discuss the peculiarities of the approach and eventually we discuss the preliminary results achieved in the GRID.it project, where a prototype high performance, component based, GRID programming environment is being developed using this approach.

**Keywords:** Components, structured programming, parallelism, application manager, heterogeneous architectures, fault tolerance

[*]This work has been partially supported by Italian national FIRB project no. RBNE01KNFP *GRID.it*, by Italian national strategic projects *legge 449/97* No. 02.00470.ST97 and 02.00640.ST97, and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265)

## 1. Introduction

The development of efficient high performance grid applications requires a consistent programming effort and a huge amount of knowledge on both the Grid technology and the Grid middleware. Grid architectures are basically distributed, wide area, heterogeneous and dynamic networks of computing resources sharing a common middleware. As a wide area distributed architecture, the grid inherits all the problems typical of distributed computing/programming, made even worse because of the high latencies involved in communications. As a heterogeneous network, important actions have to be taken to allow computations to be spread across a range of different machines (i.e. different CPUs, different OS, etc.). Last but not least, as a dynamic set of computing resources, further actions have to be programmed to take into account that grid nodes can suddenly become unreachable or even that they can become more and more busy, to the point that their support to the computation at hand becomes negligible. In this work, we want to discuss a methodology that enforces the development of very efficient, high performance, grid programming environments. The focus is on *high performance*. We basically want to be able to use grid architectures to perform those computations that *need* grids as a substitute of powerful and very expensive massively parallel machines. In case the focus is on large data handling or on ubiquitous computing, rather than on high performance, different problems are to be faced and different solution can be envisaged. In particular, looking for high performance out of grids we are interested in providing several different properties, namely:

- *scalability*, that is the ability to run high performance applications on differently sized grid architectures, without incurring in any additional overhead introduced by the run time support used

- *fault tolerance*, that is the possibility of completing a high performance application execution even in presence of typical grid architecture faults, such as the temporary inaccessibility of a node due to network link failures or the shutdown of a non dedicated processing node, as an example

- *adaptivity*, that is the ability to adapt high performance computation behavior to the instantaneous features of the grid target architecture. Adaptivity, by the way, requires that both static policies are used, that is compile time policies leading to the implementation of adaptable code, and dynamic policies, that is run time policies that allow to properly react to target architecture feature changes.

Currently, grid application development is mainly performed directly exploiting in the source code the features provided by the grid middleware at hand. The classical picture from the first grid works shown in Figure 1 (left) is still

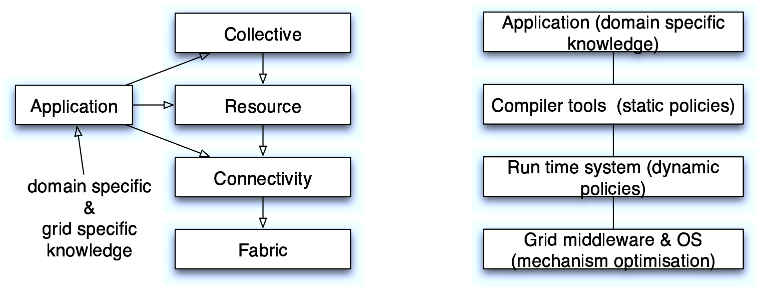
Collective
Application
Resource
Connectivity
Fabric
domain specific
&
grid specific
knowledge
Application (domain specific knowledge)
Compiler tools (static policies)
Run time system (dynamic policies)
Grid middleware & OS (mechanism optimisation)

*Figure 1.* Classic hierarchy in grid application development (left) vs. the layered alternative approach (right)

depicting the actual current grid application schema. Both in case the middleware available is very general purpose (e.g. plain Globus [21] or Unicore toolkits [19, 28]) or in case it already provides some kind of higher level programming abstraction (e.g. GridRPC [25]), the programmer is requested to go down to the middleware logic in order to implement a parallel grid application. In case the goal includes the ”high performance” keywords, the kind of knowledge required to the programmer is very high. When using plain, low level middleware systems, the programmer must explicitly program all the activities related to program decomposition, scheduling and deployment, to communication scheduling and management, to synchronization handling, to fault tolerance and possibly to adaptivity. In case higher-level tools are used, the programmer still needs to cope with program decomposition, fault tolerance and adaptivity, while other items are automatically dealt with by the compiler and/or run time support. Furthermore, when the *high performance* goal is to be achieved, the development of grid applications requires sensible *performance tuning* phase/activity. The programmer, after exploiting his knowledge on the middleware to write the grid application has to exploit the same knowledge to refine all those aspects affecting the application performance. The problem here is that most of these aspects are deeply interrelated with the features of the grid nodes used. And these features can suddenly change, thus impairing the programmer tuning actions. Even in case the features of the grid nodes does not change, a new run on a slightly different grid architecture or on the same architecture but with different nodes involved requires another tuning step to achieve high performance. We claim that there is an alternative to this way of programming high performance grid applications. The alternative we envisage is based on the adoption of a higher level programming model for grid applications *and* on the highly layered implementation of a programming environment supporting the higher level programming model (see Figure 1 right).

The higher level programming model layer should provide the programmer with tools that allow the invisible grid goal stated by NGG (Next Generation Grid) expert group to be achieved [20]. The compiler tools layer should implement all the known policies, strategies and heuristics that can be applied statically to target grid features. The run time system layer should implement all the known policies, strategies and heuristics that can be applied dynamically to target grid features. The middleware/operating system level must provide all the mechanisms (and only the mechanisms) needed to implement the upper layer policies. Therefore the classical middleware toolkits (comprehending all the three upper layers of Figure 1 left) can be placed internally to the lower layer of the right figure, provided that only mechanisms of these toolkits are used. To support our claim, we structured this paper as follows: Section 2 outlines a component based programming model aimed at representing a viable, higher level alternative to the direct usage of plain grid middleware in high performance grid application programming. Section 3 explains how such programming model can be implemented on top of existing or new grid middleware, by adopting a layered approach such as the one depicted in Figure 1 (right). Both Sections built on the studies performed and on the results achieved in the context of the GRID.it [22] project. GRID.it is a three-year project, ending in 2005 that involves major Italian universities and research institutions. Within the project, our group is responsible of a work package whose goal is to produce a prototype high performance grid-programming environment (ASSIST [32, 5, 12, 7]). In Section 4, we will discuss the results achieved in the design of ASSIST, according to the alternative approach to high performance grid application development proposed in this paper. Eventually, Section 5 discusses references to related work and Section 6 hosts the conclusions.

## 2. Component based grid programming

Some interesting, component based programming models have been proposed to be used in the grid context. In particular, the CORBA Component Model (CCM [26]) and the Common Component Architecture (CCA) component model [11] have been widely discussed in the grid context. Other models, coming from different experiences, such as JavaBeans [27], Web Services (WS [34]) and Microsoft .NET [18] are currently being considered in the field of grid programming although neither the web services model nor .NET can be properly called component models. In the context of the already mentioned GRID.it project, our group introduced a fairly new component based programming model. Components can be either parallel or sequential. Legacy CCM components and WWW web services are assumed to be usable as sequential GRID.it components via proper wrapping.

**Component interaction** GRID.it components interact using three basic mechanisms:

**use/provide ports** inherited from the classical component model. Use/provide ports are basically used to implement RPC-like component interaction.

**events** inherited from CCM. Events are basically used to implement component synchronization.

**data flow streams** These are new. A data flow stream is basically a kind of use/provide mechanism that it is used to implement efficient, one-way data flow communication between components. A component exports a data flow source port that can be used by another component via a data flow sink port. Overall this provides a way to transfer typed data items from the first component to the second one.

Even though data flow streams can be easily implemented in terms of either use/provide ports or events, they have been explicitly included in the set of primitive mechanism to enforce the concept that they provide optimized, high performance inter-component communication mechanisms. All these mechanisms are used to implement two distinct component interfaces:

**the functional interface** exposing the component functional behavior to the other components. Using the mechanisms implemented in this interface a component can use the services provided by another component to actually compute a result

**the non-functional interface** providing mechanisms that can be used to *control* the component behavior, that is, its execution features as well as its interaction with the underlying grid target architecture.

**Component representation and interoperability** GRID.it components can be described via XML descriptor files, much in the sense of Web Service Definition Language (WSDL) [33]. Each descriptor contains one item for each one of the use/provide, event and data flow stream interfaces of the component. The descriptor can be used to pick up components to be assembled in a parallel application. At the moment, exact descriptor syntax is still going to be defined. There will be some kind of `public` items in the descriptor and some kind of `protected` items, however. The former describe the functional interface of the component, that is all those ports needed by programmer to assemble components in such a way they perform the actual computation at hand. The later describe the non-functional interfaces, that will be used to *manage* the components within the component assembly. Interoperability with other component frameworks is achieved by automatically generating wrapping of

GRID.it components in the Web Service framework and in the CCM framework. The other way round, call to Web services or to CCM components is allowed and supported from within the sequential portions of code embedded in a GRID.it component.

**Parallel components** Parallel components are those components that are internally programmed as a coordinated set of parallel activities (virtual processors, according to the ASSIST jargon) using the ASSIST coordination language [32]. Basically, a parallel component is defined by qualitatively expressing the parallelism we want to exploit. This is performed defining a set of virtual processors that is logically parallel activities. The reader will refer to the available literature [7] in case he wants to understand better how parallel components (ASSIST modules) can be defined. For the purpose of this work, and in particular to describe the GRID.it component model, how parallel programs are implemented does not matter. What's worth pointing out is that a parallel GRID.it component includes a *component manager* and provides suitable ways to access the manager facilities through its non-functional interface. The component manager completely controls the parallel component behavior. In particular, it completely manages the interaction of the component with the grid and takes care of managing its internal parallelism degree in such a way that both the application needs and the target grid features are taken into account. The non functional interface hosts mechanisms that can be used to set up the component parallelism degree, to add new (eliminate) resources to (from) the set of grid resources taking care of component execution, to monitor component execution parameters, to implement fault tolerance management strategies, etc.

**Component assembly** GRID.it components can be used to build applications according to two very different strategies. On the one hand, some kind of assembly language or possibly a nice GUI can be used to compose components in the structure/pattern required to implement the grid application at hand. In this case, users just connect use and provide, data flow source and sink ports or event channels of the components they pick up to build the grid application. On the other hand, coordination components can be used to compose components according to some well-known component composition pattern (aka parallelism exploitation skeleton or parallel design pattern [13, 23, 24]). As an example, a pipeline component $P$ can be used to compose two components $S_1$ and $S_2$ in such a way that they happen to implement two stages of a pipeline. In this case, the GRID.it component framework establishes proper connections between the non-functional interfaces of the three components (the pipeline one and the two stage components) in such a way that a hierarchy of component managers is created. In the general case, managers are always composed in trees as shown in Figure 3. The top-level component manager becomes the application manager.

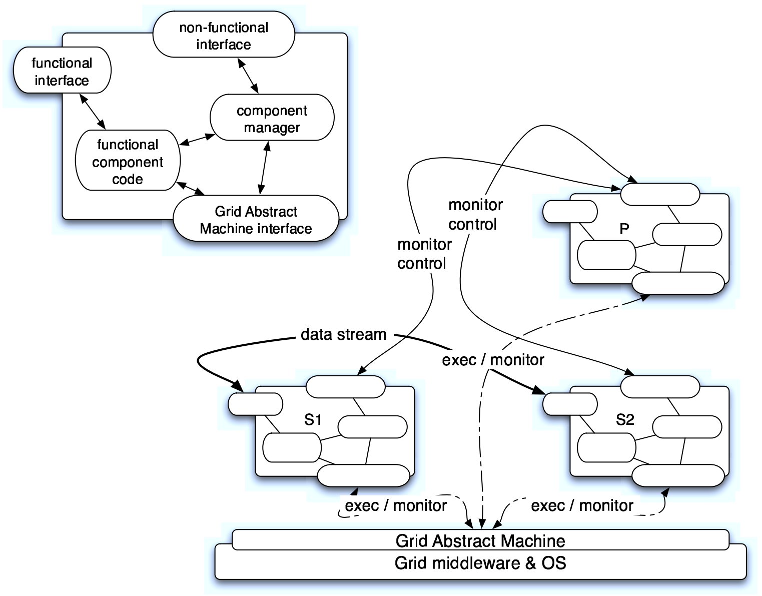

*Figure 2.* GRID.it component structure (upper left) and Sample pipeline application schema (lower right)

It is responsible of coordinating the actions of the other component managers. The leaf component managers are those in charge of more directly interacting with the underlying grid middleware or node operating system to arrange proper component execution. The pipeline component manager becomes the *application manager* and actually takes care of coordinating the activities of the two stage components in such a way the resulting pipeline turns out to be an high performance pipeline. This allows the pipeline component designers to encapsulate in the pipeline component manager all those autonomic policies that take care of efficient pipeline parallel programs executions. As an example, the pipeline component manager may *monitor* the performances achieved by the two pipeline stages using the non-functional interface mechanisms, it can *analyze* the performance values to understand if the pipeline is balanced and as a consequence it can *plan* some kind of corrective action (first stage is much slower than second one: it should be made faster, if possible) and eventually it can *execute* the corrective action (inform the first stage application manager to recruit new or better resources to support its execution).

**Target architecture management** GRID.it components interact with the target grid execution environment via calls to the *Grid Abstract Machine* (GAM) Interface. This is not a component interface actually. Grid Abstract Machine

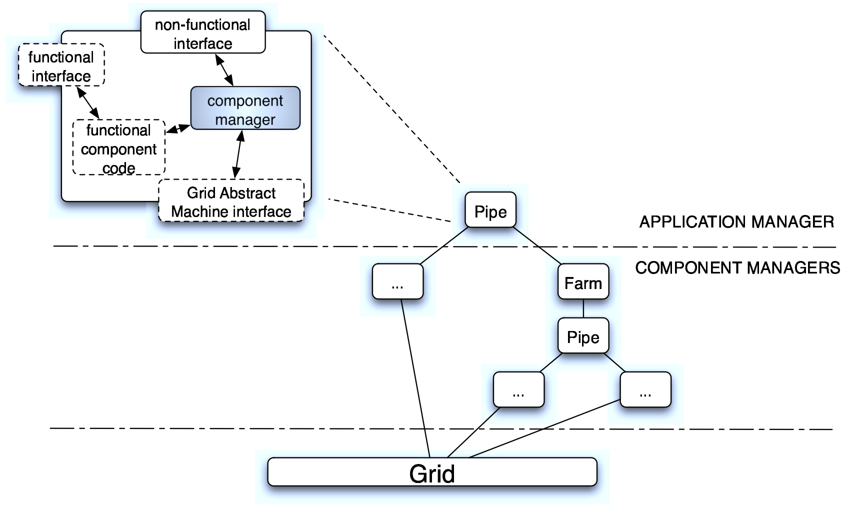

*Figure 3.* Tree structure of the component managers

is a layer built on top of existing operating systems and grid middleware that virtualizes the relatively small number of mechanisms needed to deploy and run GRID.it components on a grid: staging, point to point and collective communications, remote commanding, resource discovery. The component model of

GRID.it is currently being formalized and implemented. A better view of description of it can be found in [7, 6]. The component model we shortly discussed provides all the classical advantages of component based programming models, namely interoperability (with CCM and WS frameworks), code reuse (native component code can be written using plain C/C++ code, whole components can be reused in several different applications), modularity (applications are composition/assembly of independently developed/implemented components), software engineering (component design and implementation inherit most of the more significant software engineering techniques). In the meanwhile, the addition of autonomic control *within* each component adds a level of freedom to the programmer: the component manager, as an example, deals with all the details concerning grid execution of the component. Furthermore, the adoption of a clear, concise and effective API abstracting the grid middleware and operating system features, guarantees that portability of the whole component framework to different grid platforms just requires the re-implementation of the Grid Abstract Machine layer. Entire library of components, explicitly designed for parallel grid computing can be developed. Common grid application patterns, such as task farms, pipelines, simple DAGs, can be programmed in a GRID.it component by carefully providing the component manager code. The GRID.it framework provides standard, customizable versions of component managers to be used in the standard, simple cases. This further improves the programmability of grid applications as even in case of these *meta-components* all the pleasant properties of component systems are inherited. In addition, the meta-components will be provided by expert grid programmers and the normal users/programmers can just benefit of their existence in some reference library to program efficient grid applications. Being the access to the grid completely mediated by the component managers, and being standard component managers supplied by the component framework and sub classable to program your own GRID.it component, this contributes to implement a completely invisible grid usage.

## 3. Layered implementation

After outlining the layered approach to grid application implementation in Section 1 and then briefly discussing the component model to be provided to the user/programmer in Section 2, in this Section we discuss how the two can coexist. In particular, we discuss how the component model can be implemented exploiting a layered implementation and how such implementation can efficiently support high performance execution of grid applications. We assume that three layers of Figure 1 right implement the component based programming model Section 2. In particular, we outline in the next subsections the qualitative behavior of the different layers, concentrating on the three ones in the lower part of the Figure.

**Compiler tools** The compiler tool level is responsible of producing the actual object code of the component assembly representing the application. This means that basically each component has to be compiled in some kind of object code, and that the framework code has to be generated as well (e.g. the code needed to support component assembly). The object code produced should use the facilities provided by the Run time system layer to access grid node facilities. It cannot go directly to the grid middleware APIs, for instance. This guarantees portability of the whole component model across different architectures provided that the GAM is available on the target architectures. Heterogeneity is dealt with at the compiler tool layer by producing different object code for each one of the different node architectures present in the target grid. If the grid hosts Pentium/Windows nodes as well as PowerPC/Linux nodes, executables for both architectures are to be produced. This is completely transparent to the programmer/user. In case the actual grid used to execute the application sports both nodes, it will be a task of the run time system layer to stage and execute proper code on the grid nodes. Single components can also be separately com-

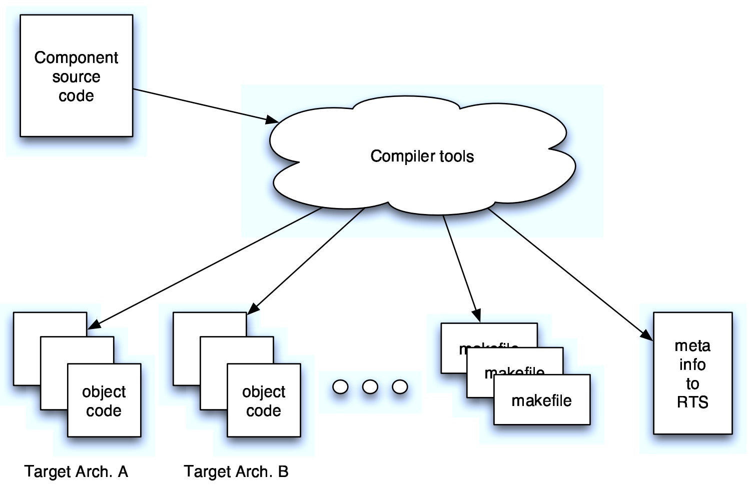

*Figure 4.* Compiler tool layer

piled. However a moment when the component assembly is processed must eventually exist. At this point the component assembly can be analyzed and specific static optimizations enforcing high performance application execution can be performed. As an example, take into account the pipeline example of Figure 2. In this case, the compiler can devise the type of the data exchanged between first and second stage, and, as the pipeline component is used to manage the two stages, it can insert any communication optimization improving the performance of the stream communication of that particular type of data. For instance, if the data to be transmitted is small size, communication aggregation can be automatically inserted in the code, to provide better latency hiding. As the pipeline component is only used to compose components that interact via (possibly infinite) data flow stream, this does not impair program semantics nor it changes the application programmer perception of the application execution, but for showing a possibly better performance. Fault tolerance is also taken into account at this level. Known techniques to implement computation checkpoints or to program handling of faulty nodes can be used to implement additional code in the component manager as well as in the component functional code. Again, this can be implemented once and for all in the component (assembly) compiler by experts of both grid technology and fault tolerance techniques, and the application programmers can be left completely unaware of the fact that fault tolerance is currently implemented in the object code. In general all the known static optimization techniques can be exploited at the compiler tools level, also exploiting the knowledge directly coming from the knowledge

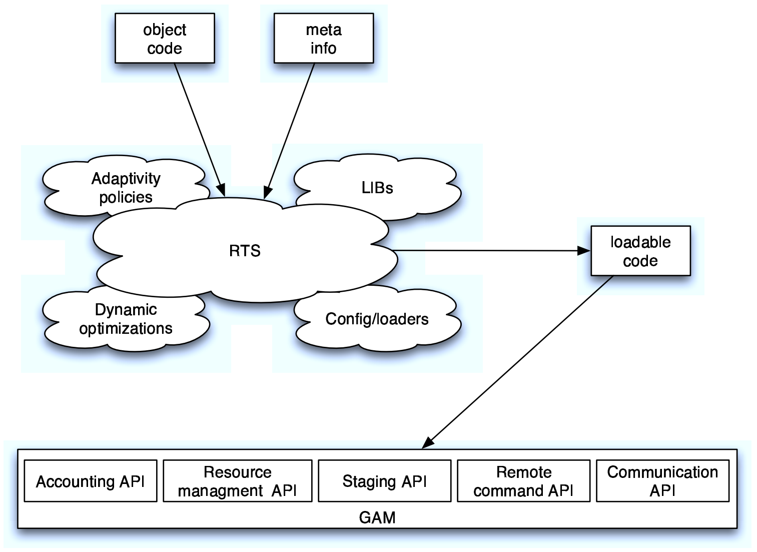

*Figure 5.* Run time system layer

of the application high-level structure, i.e. of the component assembly making the application.

**Run time system layer** The run time system layer is responsible of supporting the execution of the object code produced at the compiler tools layer. Therefore this layer includes all the libraries and component tools needed to run the component code. It also takes care of the following tasks:

- looking for the grid resources needed to execute the code, querying the grid middleware through the interfaces provided by the Grid Abstract Machine. The initial requirements of resources are produced at compile time. The application and component managers also issue resource requirements that have to be satisfied by the run time system layer

- loading the proper object code at the grid nodes selected for program execution. This possibly requires on the fly compilation of the object code, to obtain architecture specific object code out of high-level object code

- executing the proper processes/threads at the remote grid nodes used to execute the application

- supporting the execution of the component managers, that is to provide all the monitor activities needed to support manager analysis and decision planning and to allow the manger execute its plans

The component managers of the GRID.it components actually happen to be placed at this layer. They implement all those dynamic optimization strategies that are typical of high performance application support. As an example, application and component managers can plan component deployment changes, that is migration of components from one grid node to a different one, to minimize the measured communication overhead. Or they can plan to group computations (that is, components) that were originally placed on distinct node onto a single one to exploit sharing of data through efficient, in memory mechanisms. In the run time system layer performance contracts are also managed. Performance contracts are assigned to components using their non-functional interface. Basically, a performance contract is some kind of high-level description of the kind of performance behavior the programmer/user pretends to get out of the component. We are currently considering performance contracts asking for a given *service time* of an application, that is, asking the system to be able to deliver results corresponding to different, consecutive input stream data items at intervals no longer than a given time. We are also assuming a performance contract is represented by an XML file written according to a given XML document type definition (DTD). The user must provide a performance contract to the application manager, that is the manger of the top-level application component. Such performance contract may require, as an example, a given service time, that is a given inter-delivery time of the tasks computed by the component assembly out of the input stream data. The original contract is to be suitably propagated to the components managed by the top-level component. In Figure 2 right, the performance contract provided by the user/programmer to the pipeline component is propagated to the pipeline stage components $S_1$ and $S_2$. In case the performance contract originally provided to the pipeline process was a service time contract, it is simply propagated to the pipeline stage component managers, as the service time of the whole pipeline is given by the maximum of the service time of its stages. In case the original contract was a parallelism degree one, that is a contract asking the pipeline application manager to execute the application with a given parallelism degree, the pipeline manager first devises a(possibly equal) subdivision of the parallelism degree among the two stages, then propagates the requirement to the stage component managers, and eventually monitors the performance of the stage component to understand whether a different subdivision of the parallelism degree has to be implemented to keep the two stages balanced.

**Interaction with grid middleware** Most of the activities performed in the run time system layer require an interaction with the underlying grid middleware through the GAM API. As an example, the discovery of the available resources to be used to run the component application is performed querying the resource management subsystem of the grid middleware at hand. The initial necessity, in terms of grid node resources, to execute the application is derived statically by the compiler tool layer. The application manager interprets this initial need and queries the grid middleware to find out the needed resources. Eventually, when the needed resources have been discovered and recruited to the computation of the component application, code is deployed to them (using another part of the GAM API), this code execution is started, etc. Overall, the run time system uses a restricted set of the underlying grid middleware through the GAM API to execute component code. In a sense, this restricted API constitutes a sort of grid operating system API, in that it provides the basic mechanisms needed to run applications onto the grid. This minimal API must include: authentication and accounting facilities, resource management systems, supporting resource discovery and resource reservation, at least, code staging facilities, remote commanding facilities, and point-to-point and collective communication and synchronization facilities. It is worth pointing out that these features are basic *mechanisms*. All the policies are encapsulated in the run time system layer or in the code produced by the compiler tools layer and run in the run time system framework. This implies that the GAM API will be optimized to provide only those mechanisms that actually support high performance computing, leaving outside those mechanisms that cannot be used to support high performance applications because of their poor performance figures.

## 4. ASSIST: a first instantiation of our methodology

ASSIST (A Software development System based upon Integrated Skeleton Technology) [32, 5, 12, 3] is the high performance parallel programming environment being developed in the framework of the GRID.it Italian national three year project [22]. It was initially conceived as a programming environment implemented according to a layered approach such as the one discussed in the Sections above and targeting plain TCP/IP networks of POSIX workstations (namely, Linux/Intel node architectures). Subsequently, we extended the implementation to cover the grid architectures, and we are currently completing a user language review that provides GRID.it components at the top level of the programming model hierarchy.

**Basic features** The basic concept of ASSIST is that users cannot specify arbitrary parallel code. Rather, they can provide graphs of modules, interconnected by means of data flow streams and possibly sharing external (that is implemented in external libraries) objects. Each module, in turn, can be sequential module or a *parmod*. Sequential modules are plain sequential portions of code (procedures) wrapped to make clear their functional behavior, that is, the data they consume and the data they produce. Data input to a sequential module can come either from a data flow stream or it can be an external object reference. In the latter case, the reference to the object is usually passed to the sequential module through a data flow stream. Parmods, instead, are *generic parallel modules*. In a parmod, a programmer can basically define a set of logically parallel computations, the parmod virtual processors, and the way they process data. In particular, he defines how data coming from the set of module input data flow streams are non-deterministically distributed (in unicast, multicast or broadcast) to the virtual processors, and how each virtual processor contributes to generate the data eventually placed on the module output streams (that is how those data are generated either taking pieces from the data produced by each one of the virtual processors or simply delivering piece by piece the data produced by each single virtual processor as a single data item of the output stream). The single virtual processor code can be defined using sequential code such as the one used to define a sequential module. Virtual processors in a parmod are named according to a *topology*. Anonymous topology (each virtual processor performs the same computation) is used to model task farm like computations, whereas vector or array topology are used to differentiate the virtual processors either in base to the code (vector topologies having the first and the last virtual processor computing a different code with respect to the other ones, as an example) or in base to data (again, vector topologies with data coming on an input stream scattered across the virtual processors can be defined). As an example, the user may include in the parmod code the line

```
topology array[i:1024] Vp;
```

In this way, a vector of 1K virtual processors (i.e. logically parallel activities) is defined. In the following code, the programmer may assign code to be executed to different virtual processors simply specifying their index (e.g. `Vp[i]`). In any case, the virtual processor does not necessarily correspond to actual processing elements used to compute the ASSIST program. The number of processing elements used and the mapping of virtual processors (ranges) to the processing elements is actually jointly performed by the compiling tools and by the run time system of ASSIST.

**Implementation** The ASSIST programming environment is implemented using a structure such as the one depicted in Figure 1 right. The actual structure of the ASSIST environment is depicted in Figure 6. The compiler tools level take care of compiling ASSIST source code, without any kind of grid awareness in it, into a C++ object code, that is a set of process code along with all the makefiles needed to compile it on different (possibly heterogeneous) nodes of the grid target architecture. The compiler also produces some meta-information about the code in an XML configuration file. The XML describes the structure

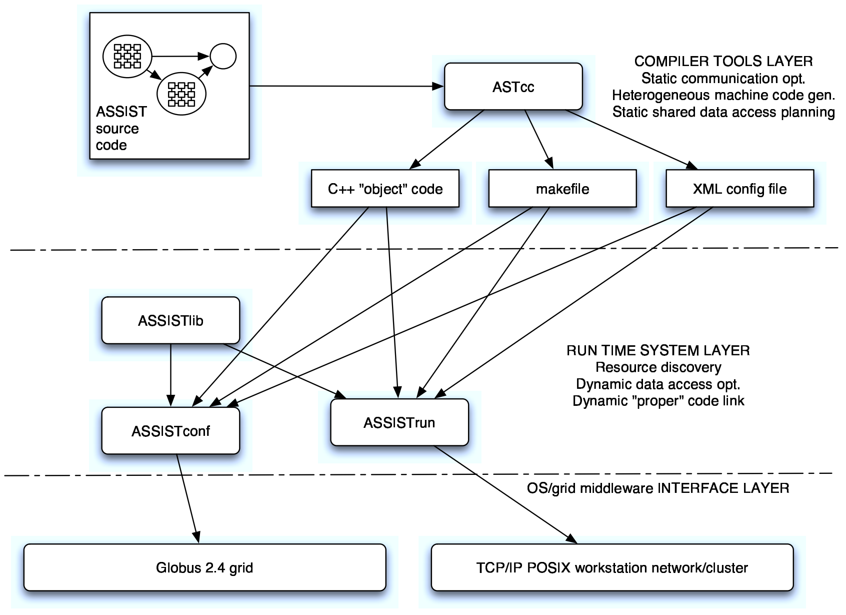
ASSIST source code
ASTcc
COMPILER TOOLS LAYER
Static communication opt.
Heterogeneous machine code gen.
Static shared data access planning
C++ "object" code
makefile
XML config file
ASSISTlib
RUN TIME SYSTEM LAYER
Resource discovery
Dynamic data access opt.
Dynamic "proper" code link
ASSISTconf
ASSISTrun
OS/grid middleware INTERFACE LAYER
Globus 2.4 grid
TCP/IP POSIX workstation network/cluster

*Figure 6.* Structure of the ASSIST environment

of the parallel program, the code needed, the libraries needed along with the parallelism degree required to execute the different modules in the source code module graph. The process code produced at this level uses calls to the ASSISTlib run time system to implement communications, data sharing, etc. The Run time system layer comprehends two items: the ASSISTlib actual run time and a loader/manager processing the XML config file and interacting with the grid middleware to achieve completely automatic ASSIST program execution. Currently, two versions of this tool have been developed: ASSISTconf is used when Globus grid target architectures are considered, whereas ASSISTrun is used when simpler, plain POSIX–TCP/IP workstation networks are targeted. In particular, the ASSISTconf tool:

- looks at the computing resources needed in the XML configuration file,

- queries Globus toolkit 2.4 Monitoring and Discovery System (MDS [9]) to retrieve such resources,

- schedules (compiles and deploys) logical nodes (modules or subsets of parmod module virtual processors) of the ASSIST program to the physical nodes recruited to the computation and

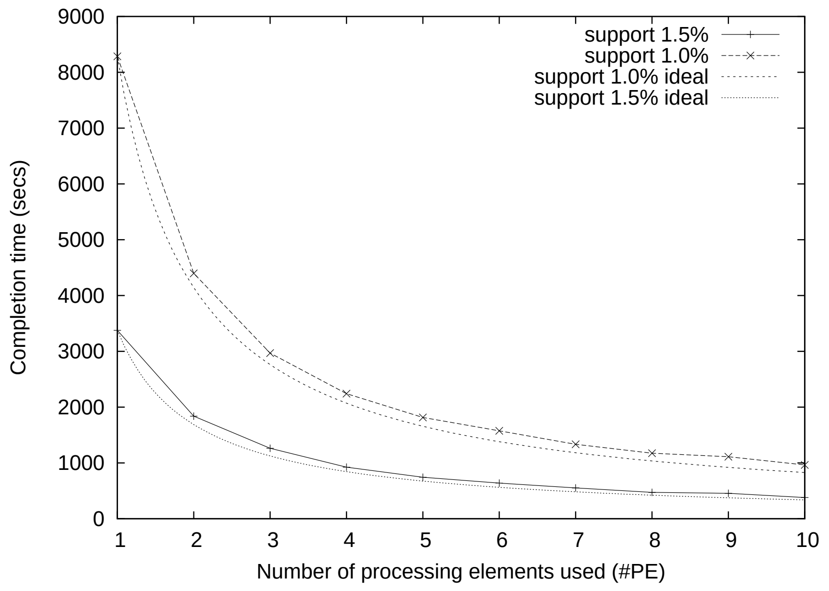

*Figure 7.* Completion times of an ASSIST data mining application on a workstation network

- eventually starts the logical nodes, waits for program completion and gathers the results produced.

In future ASSIST versions, the globus and POSIX–TCP/IP tools will be merged to have a unique configurator/launcher ASSIST tool.

As is, ASSIST perfectly matches what stated in the first part of this paper. Compiler layer performs static optimizations and take care of generating the code necessary to take heterogeneity into account. As an example, the exchange of data shared among the virtual processors are optimized at this level, producing code that uses optimized, aggregated message passing in case of virtual processors allocated on different processing elements, while it simply exploits pointers in case the virtual processors are mapped onto the same physical processor. The decision concerning which code has to be used is postponed to the run time system layer, when the physical allocation of virtual processors is known. The run time layer accounts for dynamic optimizations and grid targeting. As an example, the manger activity in charge of ensuring load balancing among virtual processors activities is performed at this layer.

**Results** This structure of the ASSIST implementation leads to very nice results. Figure 7 shows the typical performance figures achieved on a NOW (Network Of Workstations) architecture. In this case, the application was a data mining application. The different curves refer to different values of the support set parameter used. In this case, the data mining applications is based on the APRIORI data-mining algorithm. The support set represents the percentage of the data base data supporting (i.e. validating) each association rule of APRIORI. Smaller support set values usually lead to (possibly exponentially) higher computational weight, as more association rules are taken into account. The completion times measured are definitely close to the ideal ones, independently of the support set used or, in other words of the computational effort required [15]. We achieved similar results both executing other applications on a NOW target architecture and executing the same applications on Globus grid architecture, i.e. on a network of workstations running the Globus toolkit.

Figure 8 plots efficiency for an application processing MPEG-4 data using different numbers of grid nodes. The different runs use different mappings of the ASSIST logical nodes/components to the physical grid resources available. The mappings have been set up by hand intervening on the XML configuration files produced by the compiler, just to show the effect of choosing alternative mappings when executing an ASSIST program. The efficiency curves are shown for a typical ”good” mapping (run 1) and for a bad one (run 2): the former using more efficiently the processing elements at hand, the latter using them less efficiently, as an example mapping/deploying bottleneck nodes on slower machines. The superscalar efficiencies are due to the heterogeneous nodes used: some machines were more powerful than the one used to run the complete application on a single node. In this case, all the machines used where Linux/Pentium based workstations, but some of them were equipped with rather old Pentium III and others were equipped with brand new Pentium IV. Moreover, different machines were equipped with different amount of main store. The nodes were spread across a grid involving two different institutions in the Pisa area. The Figure shows how good efficiency figures can be achieved, without actually requiring the programmer any single line of code concerning process and communication set up and scheduling, or even managing interactions with the grid middleware/system. Furthermore, as the two runs only differ in some parameters of the XML configuration file produced by the compiler (modified by hand, in this case, but that is usually processed by the run time system tools ASSISTconf and ASSISTrun), this result shows how policies implemented at the run time system level (the mapping policies) can sensibly affect the overall performance of ASSIST applications, and therefore it further justifies the concept of the layered implementation.

Figure 9, plots the speedups achieved executing an irregular application using two different implementation strategies (templates) for a single ASSIST parallel module/component. The line marked as ”dynamic” (the one closer to the ideal line) is relative to a template fully exploiting macro data flow [16–17] implementation technology, while the line marked as ”static” uses compile time virtual processor partitioning. The ASSIST compiler will be able to generate

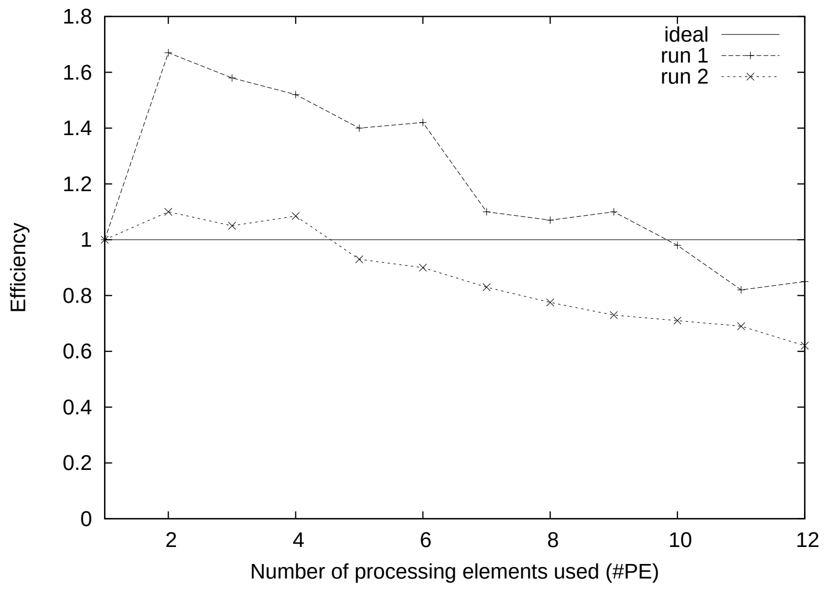

*Figure 8.* Completion times and efficiencies of an ASSIST MPEG application on a variable number of grid nodes

code for both implementation templates. Then, the run time system can use a default template (the static one, as an example). In case it observes that the computation is unbalanced the run time may dynamically decide to move to the alternative template, the dynamic one. This is possible just because the parallel component is structured (that is the parallelism exploitation pattern is exposed to the compiler/run time layers), the template properties (approximate analytical performance models) are known and the run time support is free to decide which code generated by the compiler is to be used depending on the ”observed” features of the computation at hand.

The layered implementation of the ASSIST programming environment is also exploited to tackle heterogeneous target architectures. ASSIST compiler generates code for a range of admissible host target architectures. In the current version, Pentium/Linux and PowerPC/MacOSX architectures are actually taken into account and Pentium/Windows is going to be taken into account too. Versions of the run time library ASSISTlib are provided for all the admissible target host architectures. Then, the run time tools (either ASSISTconf or ASSISTrun) decide which version of the library and of the compiled code has to be used according to the target architecture nodes chosen to run the different parts of the ASSIST application. As an example, in case a three-stage pipeline is run on two Linux and one MacOSX box, the code of the former stages is picked up

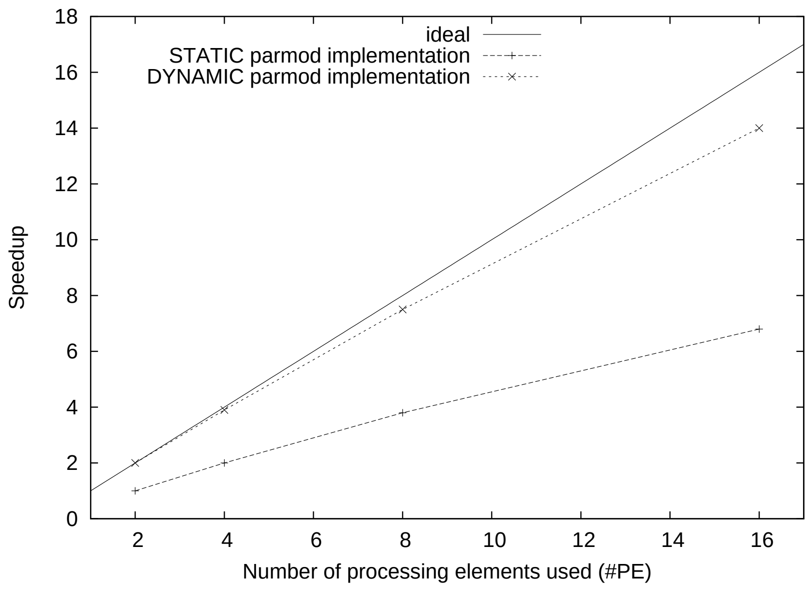

*Figure 9.* Speedup of different templates implementing the same ASSIST parallel component

from the compiler output directory relative to Linux target hosts and the code from the later is picked up from the MacOSX directory. Furthermore, proper code is inserted when communications are performed across processes running on different host target nodes, in such a way that convenient, architecture neutral external data representation is used to avoid data loss. ASSIST applications runs on heterogeneous target architectures with both Pentium/Linux and PowerPC/MacOSX nodes demonstrated almost perfect speedup, provided that the parallel program exploits a suitable grain of parallelism.

Recently, we got also results concerning the application managers activity showing that run time dynamic adaptation of parmod execution is feasible and convenient to adapt parmod execution to changed target architecture load or node availability. These results are discussed in [4]. Figure 10 shows the results of an experiment involving component managers. An application built around a single ASSIST parmod is run, after providing the parmod component manager a performance contract stating that 4 tasks per second must be processed. The manager initially looks for resources increasing the parmod parallelism degree to the point the performance contract is satisfied. After some 25 seconds, the contract is satisfied and the manager stops looking for new resources to the used to increase the parmod parallelism degree. For three times, during ASSIST program execution, the performance contract is violated due to increased load on the machines used to run the application. The manager reacts adding new

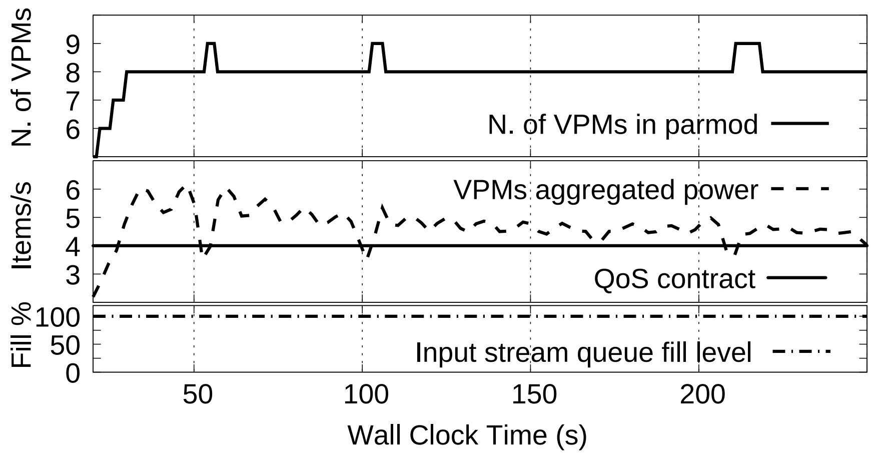

*Figure 10.* Effects of component manager activity when a performance contract is provided.

resources to the set of processing elements used to implement the parmod (8 PEs → 9). When the manager foresees that by releasing the less powerful processing element used the performance contract will be anyway satisifed, that processing element is actually released (9 PEs → 8). The component manager of the parmod performs all this work automatically.

Several other already published papers present experimental results achieved using ASSIST: [6] and [7] discuss topics more related to the component model of ASSIST. [8] discusses heterogeneity specific topics and results. [5] discusses the overall implementation of the COW/NOW version of ASSIST. A complete list of the ASSIST papers can be found on our group web site at [3].

## 5. Related work

Many projects address the problem of high performance grid application implementation. Actually, several projects are focused on the usage of RPC based programming models [25, 30]. In this cases, the implementation of applications simply relies on a further layer, the RPC one, built on top of the layers of the Figure 1 left, rather than spreading responsibilities across a compiler and a run time layer as we do. An interesting project, aimed at providing a high level-programming environment for grids, is the GrADS project [2]. GrADS uses performance contracts to manage grid application execution. It also adopts an application manager that is very close to our one. The implementation of the whole system is not clearly structured in layers, however [29]. Just taking into account the concept of manager, in [10], an approach to parallel program adaptivity is also shown, based on a notion of adapter which is very close to our application manager concept. Some programming environments designed in the frameworks of algorithmic skeletons or parallel design patterns have a layered implementation close to the one we present in this paper, although they target a different kind of architectures. In particular, $CO_2P_3S$ [24] has a layered implementation that indeed is mainly used to enhance expandability of the design pattern set. Among the other programming environments that use higher level parallel programming patterns and still provide some kind of layered implementation, IBIS [1] is a Java based programming environment whose implementation deeply optimizes several key aspects and also provides some adaptive policies for its main parallelism exploitation pattern, namely the divide&conquer pattern [31].

## 6. Conclusions

We discussed an alternative way to implement high performance parallel programming environments targeting grid platforms. This proposal is alternative to the classic grid programming figure assuming that applications are built on top of grid middleware directly using/invoking the middleware functionalities at the user code level. We propose to clearly separate static concerns, solved in the compiler tool layer, from dynamic concerns, solved in the run time system layer, much as it already happens in the classical, sequential, non-grid programming universe. We pointed out how this structuring can be exploited to perform different optimizations in the proper place, avoiding that the effects of an optimization impairs the effects of other optimizations, just taking the right decisions/applying the right policies in the right places. While discussing these items, we introduced the GRID.it component model, along with its component and application manager concept, to enforce the general figure of the structured implementation of grid applications/programming environments. Eventually, we showed how ASSIST, the prototype, component based, high performance, parallel programming environment we are currently developing in the context of the GRID.it project fits the methodology described in the first part of this work.

## References

[1] The IBIS home page. http://www.cs.vu.nl/ibis/, 2004.

[2] The grads home page. http://www.hipersoft.rice.edu/grads/, 2005.

[3] The Pisa parallel processing group home page. http://www.di.unipi.it/groups/architetture/, 2005.

[4] M. Aldinucci, A.Petrocelli, A. Pistoletti, M.Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of Grid-aware applications in ASSIST. Technical report, Dept. Computer Science, University of Pisa, Italy, 2005. Submitted to Euro-Par 2005.

[5] 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. Klagenfurt, Austria.

[6] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a Research Framework for High-performance Grid Programming Environments. In Jose C. Cunha and Omer F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer Verlag, 2005.

[7] Marco Aldinucci, Sonia Campa, Massimo Coppola, Marco Danelutto, Domenico Laforenza, Diego Puppin, Luca Scarponi, Marco Vanneschi, and Corrado Zoccolo. Components for High Performance Grid Programming in Grid.IT. In Vladimir Getov and Thilo Kielmann, editors, *Component Models and Systems for Grid Applications, Proc. of the WCMSGA Workshop of ACM ICS'04*. Springer, 2005.

[8] Marco Aldinucci, Sonia Campa, Massimo Coppola, Silvia Magini, , Paolo Pesciullesi, Laura Potiti, Massimo Torquati, and Corrado Zoccolo. Targeting Heterogeneous Architectures in ASSIST: Experimental Results. In Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, editors, *Euro-Par 2004: Parallel Processing*, number 3149 in LNCS, pages 638–643, 2004.

[9] Globus Alliance. Globus Monitoring and Discovery System homepage. http://www-unix.globus.org/toolkit/mds/.

[10] F. André, J. Buisson, and J.L. Pazat. Dynamic adaptation of parallel codes: towards self-adaptable components. In *Component Models and Systems for Grid Applications*, pages 145–156, 2005. First volume of the CoreGRID series.

[11] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 13. IEEE Computer Society, 1999.

[12] R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, R. Perego, P. Pesciullesi, and M. Vanneschi. AssistConf: A Grid Configuration Tool for the ASSIST Parallel Programming Environment. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 193–200. IEEE, 2003.

[13] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[14] M. Cole and A. Benoit. The edinburgh skeleton library home page, 2005. http://homepages.inf.ed.ac.uk/abenoit1/eSkel/.

[15] M. Coppola and M. Vanneschi. High-Performance Data Mining with Skeleton-based Structured Parallel Programming. *Parallel Computing*, 28(5):793–813, 2002.

[16] M. Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[17] M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based processing*. IEEE, 2005. Lugano (CH).

[18] Microsoft .NET Developer Center: Technology Information. http://msdn.microsoft.com/netframework/technologyinfo/default.aspx, 2005.

[19] Dietmar W. Erwin and David F. Snelling. UNICORE: A Grid computing environment. In Rizos Sakellariou, John Keane, John Gurd, and Len Freeman, editors, *Euro-Par 2001 Parallel Processing*, volume 2150 of *LNCS*, pages 825–834, 2001.

[20] D. Snelling et. al. Next generation grids 2 requirements and options for european grids research 2005-2010 and beyond. ftp://ftp.cordis.lu/pub/ist/docs/ngg2_eg_final.pdf, 2004.

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

[22] Grid.it community . The GRID.it home page, 2005. http://www.grid.it.

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

[24] 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.

[25] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, and C. Lee. GridRPC: A Remote Procedure Call API for Grid Computing. http://www.eece.unm.edu/ ?apm/docs/APM GridRPC 0702.pdf, July 2002.

[26] Object Management Group. CORBA Component Model version 3.0 Specification. `http://www.omg.org/`, September 2002.

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

[28] Unicore forum. http://www.unicore.org/, 2004.

[29] S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. http://www.hipersoft.rice.edu/grads/publications_reports.htm, 2004.

[30] Sathish Vadhiyar and Jack Dongarra. GrADSolve - A Grid-based RPC system for Remote Invocation of Parallel Software. *Journal of Parallel and Distributed Computing*, 63(11):1082 – 1104, November 2003.

[31] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal. Adaptive Load Balancing for Divide-and-Conquer Grid Applications. www.cs.vu.nl/∼kielmann/papers/satin-crs.pdf, 2004.

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

[33] W3C. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[34] W3C. Web services home page. http://www.w3.org/2002/ws/, 2003.