COMPUTATIONAL METHODS IN SCIENCE AND TECHNOLOGY **12(1)**, 21-32 (2006)

High level Grid programming with ASSIST*

M. Aldinucci[1*], M. Coppola[1*], M. Danelutto[1*], N. Tonellotto[2], M. Vanneschi[2], C. Zoccolo[1]

[1]*Dept. Computer Science – Univ. Pisa*
[2]*ISTI – CNR, Pisa*
[*]*e-mail: {aldinuc|coppola|marcod}@di.unipi.it*

Abstract: The development of efficient Grid applications usually requires writing huge portions of code directly at the level of abstraction provided by the underlying Grid middleware. In this work we discuss an alternative approach, raising the level of abstraction used when programming Grid applications. Our approach requires programmers just to describe in a qualitative way the kind of parallelism they want to express. Then, compiler tools, loader tools and run time system take complete care of running the application on a Grid target architecture. This allows to move most of the cumbersome tasks related to Grid targeting and management from programmer responsibility to tools. This paper introduces the structured parallel programming environment ASSIST, whose design is aimed at raising the level of abstraction in Grid programming and discusses how it can support transparent Grid programming while implementing Grid adaptivity.

Key words: Grid, heterogeneous architectures, autonomic control, adaptivity, high performance computing

1. INTRODUCTION

Grid architectures are used and exploited in several different areas. Classically, Grids are used for high performance computing, for high availability data storage as well as for ubiquitous/global computing. In any case, Grids can be viewed as heterogeneous and dynamic collections of processing and data storage elements [25]. Both the heterogeneous and the dynamic keywords strongly characterize Grids, in particular, distinguishing them from more traditional cluster/networks of workstations. Heterogeneity comes on the scene in two ways. On the one hand, different processing elements, in terms of chipset, memory hierarchy and operating systems are commonly used in Grids to execute parts of the same application. On the other hand, the Grid resources used to execute Grid applications often belong to several different logical entities and therefore are managed using totally independent policies and rules. In the former case, programmers must set up proper architectural neutral data formats to get rid of different processors and operating systems. In addition, object code targeting all the different architectures involved should be produced and managed, in such a way that the proper object code is run on each node of the heterogeneous Grid architecture at hand. In the latter case, the single system image usually exported by Grid middleware has to be exploited or explicit mechanisms have to be inserted in the Grid application code to properly log and use remote nodes.

Dynamicity also comes from several different sources. First, Grid resources are usually shared with other users. Therefore the load of processing nodes may vary out of the control of the application programmer and, consequently, clever load balancing strategies must be programmed to achieve efficiency. Second, the exploitation of public network infrastructure with shared connection links and resources often makes the performance of the resource-to-resource links unpredictable. Third, features of the application at hand may require different resources for the different stages of the application execution. Eventually, the users can ask more powerful resources when observing the execution of an application, as they recognize an interesting, compute-intensive application stage. Again, load balancing as well as adaptive implementation strategies should be programmed in the application code to take care of all these factors and therefore be able to achieve efficiency.

Grid applications are currently developed using some kind of Grid middleware. Grid middleware provides the programmer with a set of tools and with an API to access these tools. Overall, the tools and the API give the programmer the complete control over Grid resource, process and communication management. As an example, using the Globus toolkit, programmers can look for available processing elements, stage code and data to a subset of them, start remote processing and eventually gather the results computed on the remote processing elements. Therefore

* This work has been partially supported by Italian national FIRB project no. RBNE01KNFP GRID.it and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

application code eventually looks like a mix of application specific code, computing the application results out of the application inputs, and of "system" code, managing all the interactions with the middleware subsystem. Both kinds of code are completely in charge of the application programmers. Consequently, application programmers should be both expert in the specific application field and in the Grid middleware at hand.

Middleware systems such as the Globus toolkit are huge software packages and require non-trivial knowledge to be used correctly. In several documents, the EU expert group on next generation Grids pointed out how an invisible Grid concept should be provided through suitable programming tools [35] and how the level of abstraction needed to program Grid applications should be raised [33]. This is nothing but a reformulation of the motivations used since long time to introduce parallel programming environments not necessarily requiring programmers to write sends and receives in their code, actually [12, 16, 22].

The invisible Grid is indeed a very appealing concept. Users/programmers should not even perceive that there is a Grid somewhere. They should simply use the facilities provided through the Grid. The programmers should be enabled to develop efficient Grid applications without actually making explicit actions targeting the Grid features.

On the other hand, programmers must be provided with high level programming models and tools that allow to program (Grid) applications in a concise, Grid independent way. This is to allow them to concentrate exclusively on the application specific, algorithmic aspects of the application.

In this work we discuss the innovative features of a high performance programming environment targeting workstation networks and clusters as well as Grids. The programming environment provides suitable ways to directly express most of the parallelism exploitation patterns used in Grid programming. In the meanwhile, those patterns are provided in such a way that the programmer is not concerned at all with most of the Grid management features. Resource, process and communication management are all dealt with by the programming environment in an automatic way. In particular, all the aspects related to adaptivity management are dealt with automatically by the compile and run time tools. Section 2 introduces the ASSIST programming model. Section 3 introduces ASSIST implementation and Section 4 outlines the kind of performance results that can be achieved using ASSIST. Eventually Section 5 discusses some of the most innovative and significant results achieved using ASSIST.

2. ASSIST

ASSIST provides programmers with a structured coordination language. This language can be used to express parallel programs at a very high level of abstraction. In particular, programmers can express complex parallel applications without actually writing any single line related to process decomposition, mapping and scheduling or even to communication and synchronization handling.

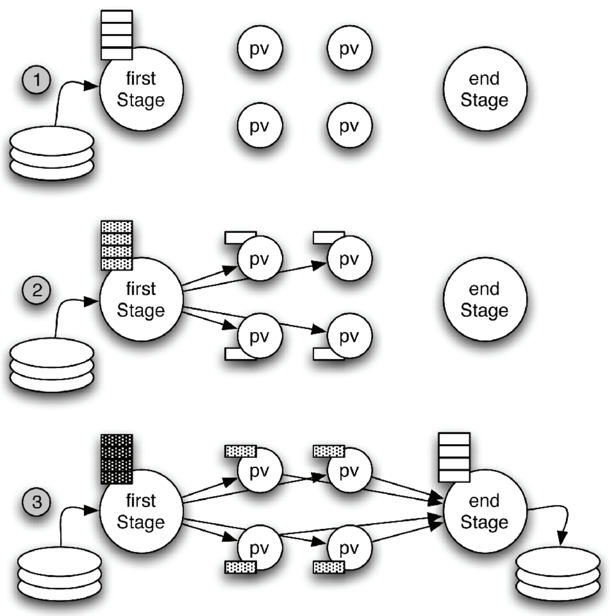

Fig. 1. Process schema for the three-stage pipeline sample application

The ASSIST coordination language programs are build of two specific parts: a module graph, describing how a set of modules, either parallel or sequential, interact with each other using a set of data flow streams, and a set of modules, implementing each one of the nodes of the graph. The modules in the graph can be programmed as sequential or as parallel modules. Sequential modules are basically procedure-like wrappings of sequential code written in C, C++ or Fortran code. Parallel modules are programmed instantiating an ASSIST parmod (parallel module).

To give a rough idea of the expressive power of the ASSIST coordination language, we assume a programmer wants to develop a three stage pipeline parallel application: the first stage generates a stream of two-dimensional matrixes reading them from a file, the second one is a data parallel stage, and the third one post processes the resulting matrixes and eventually stores them to disk. In particular, the second stage processes the input matrixes A according to an iterative data parallel pattern. A^0 being the input matrix, a new matrix A^k is computed such that $A^k_{i,j}$ is a function of $A^{k-1}_{i,j}$ and all its neighbours $A^{k-1}_{i,j-1}$, $A^{k-1}_{i,j+1}$, $A^{k-1}_{i+1,j}$, $A^{k-1}_{i-1,j}$. The process is iterated until the whole computation converges, that is, until $\forall i,j : \left| A^k_{i,j} - A^{k-1}_{i,j} \right| < \varepsilon$. We assume that the user wants to perform the computation of the new values of the matrix A in parallel on all the matrix rows. The corresponding process schema is outlined in Fig. 1. Each circle in the picture represents a logically parallel activity, assuming, for the sake of simplicity, that the matrixes only have four rows each.

```
/* definition of the application module graph */

generic main()
{
    stream double[N][N] A;
    stream double[N][N] res;

    firstStage (output_stream A);
    secondStage (input_stream A output_stream res);
    endStage (input_stream res);
}
/* definition of sequential modules */

firstStage (output_stream double A[N][N]) {
  read_from_disk (output_stream A);
}

proc read_from_disk (output_stream double A[N][N])
inc<"fstream", "iostream", "string">
$c++{
    // C++ code reading tmpA from disk here ...
    assist_out(A, tmpA); // then output tmpA on stream }
c++$

endStage(input_stream double A[N][N]){
    write_to_disk (input_stream A);
}

proc write_to_disk (input_stream double A[N][N])
inc<"fstream", "iostream", "string">
$c++{
    // C++ code writing A to disk here ...
}c++$
```

Fig. 2: Sample ASSIST coordination language code (module graph and sequential stages code)

Figure 2 and 3 show the ASSIST coordination language code of this application. The application module graph is defined in the first part of the code of Fig. 2. The streams declared in the generic main section of the program are basically data flow communication channels. In case a module process an input stream to produce an output stream, no explicit statement is needed to manage streams. The module is declared as a procedure with an input stream and output stream parameter and the corresponding code just reads the input stream variable and writes the output stream one as if they were plain variables. In case a module produces a new stream (as in the firstStage) an explicit assist_out is required to place the contents of a variable onto the output stream.

The first and third stages are sequential stages, and they are shown in the second part of Fig. 2. The second, parallel pipeline stage code is shown in Fig. 3. It is worth pointing out that this is actually the only code needed to program the three-stage pipeline application, apart from some sequential code such as the computeStencil or init that are normal sequential procedures. No explicit communication code appears there. The programmer is not required to write any kind of code to deploy and execute the program code onto the distributed executing nodes. The only thing the programmer must do in order to run this ASSIST program is a two-step process consisting in:

- compiling the program issuing an astCC program.ast command at the shell prompt, and
- run the program, once compiled, issuing an assistrun program command at the shell prompt.

Parallel modules can be expressed in ASSIST using the parmod module. The parmod relative to the second stage of our sample parallel application is shown in Fig. 3. The parmod is a generic *parallel module* construct. It is basically meant to provide programmers with a high level way of expressing sets of *logically* parallel activities.

```
parmod secondStage (input_stream double A[N][N]
                output_stream double risultato[N][N]) {
    topology array [i:N] Pv;
    attribute double S[N][N] scatter S[*i0][] onto Pv[i0];
    attribute bool diff replicated;
    stream double ris[N];

    do input_section {
        guard1: on ,, A{
            distribution A[*k0][] scatter to Pv[k0];
        }
    } while (true)

    virtual_processors {
        compute_secondStage (in guard1) {
            VP i=0 {
                init(in A[i][] out S[i][]);
                sync;
                do {
                    // do nothing
                } while (reduce (diff, ||) == true);
                assist_out (ris, S[i][]);
            }
            VP i=N-1 {
                init(in A[i][] out S[i][]);
                sync;
                do {
                    // do nothing
                } while (reduce (diff, ||) == true);
                assist_out (ris, S[i][]);
            }
            VP i=1..N-2 {
                init(in A[i][] out S[i][]);
                sync;
                do {
                    computeStencil (in S[i][], S[i-1][], S[i+1][]
                            out S[i][], diff);
                } while (reduce (diff, ||) == true);
                assist_out (ris, S[i][]);
            }
        }
    }

    output_section {
        collects ris from ALL Pv[i] {
            static double risult[N][N]; const double *el;
            AST_FOR_EACH(el) {
                for(int j=0;j<N;++j) risult[i][j]=el[j];
            } assist_out(res,risult);
        }<>;
    }
}
```

Fig. 3. Sample ASSIST coordination language code (parallel module definition)

The programmer names a set of virtual processes (*i.e.* logically parallel activities) with the topology keyword. In this case, N parallel activities are defined, named Pv[1] to PV[N]. Each parallel activity will process a single row of the input matrix, interacting with the other parallel activities to get the proper neighbor values needed to compute the row next iteration values.

The programmer may declare some data shared among the parallel activities with the attribute keyword. In this case, a shared boolean and a shared matrix are declared. The boolean is replicated at all the virtual processes and it will be used to determine termination. The matrix is distributed, one row per virtual process and it is used to host the input matrixes in such a way that the neighbor matrix values can be accessed to compute next iteration row values.

Parmod input section defines how data appearing on the input stream are handled. In this case, each matrix coming onto the input stream is scattered among the virtual processes, again one row per virtual process.

The virtual_processors section is the central one. It is used to state what the parallel activities do. In this case, all the virtual processors initialize the state variables with the input matrix values (init calls, the code is not shown there for the sake of simplicity). The first and the last rows just initialize their state variables, as they represent the border values, those not assumed to change during the computation. The central rows compute the new values of the row items based on the old ones and on the ones from the neighbor rows. The computation happens in a loop that lasts until a global OR on the diff copies returns true.

Eventually, the output section states how the final results are collected to deliver the output matrix onto the output stream, the one connecting secondStage to the endStage.

The number of processing elements actually used to run this program will be decided at launch time, depending on the resources available and on the user requests. The resources available will be discovered either using the Globus services or consulting proper configuration XML files hosting the informations concerning all the available resources. The user requests are provided either using pragmas (*i.e.* inserting in the code meta statements stating the number of processing elements to be used in the execution of the program) or supplying and XML performance contract to the ASSIST launcher program. In the former case, a line such as

```
#pragma parDegree 10 in secondStage
```

is added to the source code. In the latter, the user specifies, using a proper XML document schema, a contract stating the number of the processing elements to be used to compute the program. The contract is stored in a file whose name is then given to the compiler tools. Actually, the user supplied contracts can also state that a parmod or an entire ASSIST program should process a given number of input tasks per second. This version of the contract is the one used and discussed in the experiments of Section 5.

Although this sample application code shows most of the principal ASSIST features, there are many features that are not covered: external libraries and objects can be called from within the sequential code wrapped in the sequential modules or in the virtual processes bodies. Entire ASSIST programs can be compiled into CCM components [31] or even into standard Web Services. Furthermore, CCM components or standard Web services can be invoked from within the sequential portions of code wrapped in sequential modules or in the virtual processes bodies. These two possibilities guarantee interoperability with these other standard frameworks. Last but not least, ASSIST code can be compiled into GRID.it components and used within any other GRID.it component program. The interested reader can refer to [42, 4, 3] or to the ASSIST home page, hosting the available documentation at http:www.di.unipi.it/Assist.html.

3. LAYERED IMPLEMENTATION

The ASSIST support tools manage to compile and run ASSIST applications on two distinct kind of target architectures: Globus Grids (currently, only Globus 2.4 Grids are considered) and Grids made up of POSIX processing elements reachable *via* the ssh/scp tools. In particular, both in the Globus version and in the POSIX one, ASSIST can target heterogeneous architectures, that is architectures hosting different processing elements (with respect to CPUs and Operating Systems) at the same time [6]. The ASSIST environment is structured in layers, as shown in Fig. 4. A *compiler* layer is in charge of compiling ASSIST source code into C++ code hosting calls to the ASSISTlib library. The compiler tool (astCC) actually produces three distinct items: a set of C++ files with the abstract "object" code, a set of makefiles that can be used to generate actual object code for the target architectures considered (currently only Linux/Intel and Mac OS X/PowerPC architectures) and an XML configuration file hosting all the code/library dependencies, the parametric process network description and the configuration parameters relative to the process network that will eventually implement the ASSIST application onto the target architecture. All the static optimization techniques are exploited at the compiler level.

A run time layer is in charge of supporting the ASSIST object code execution. This layer hosts several items, such as the ASSISTlib, the AdHOC library and the application and module *managers*. ASSISTlib provides a set of "task code" classes implementing the instructions of the compiler target abstract architecture. AdHOC library provides a shared data abstraction [9]. Eventually the application and module managers provide autonomic control in the ASSIST applications and in the single parmod modules taking care of monitoring the execution of applications and adapt their execution to the dynamic target Grid features.

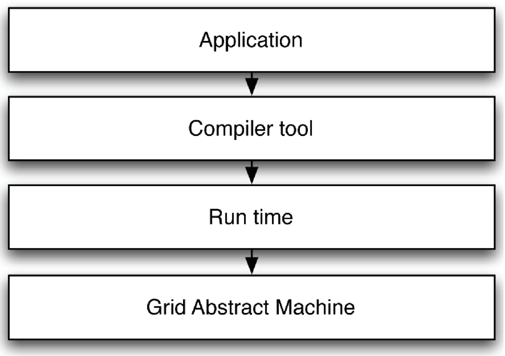

Fig. 4. ASSIST support tools

A *Grid abstract machine layer* (GAM) decouples the compiler and run time layer from the actual Grid middleware used, providing a suitable interface/API to the mechanisms used in the ASSIST framework for code and data staging, remote commanding, communications and synchronizations, *etc*.

In particular, ASSIST application code is assumed having no possibility to directly access the Grid middleware, even in the version mediated by the GAM. All the interactions with the underlying Grid are implemented by the compiler/run time layer couple. This, in conjunction with the adaptivity mechanisms described in Section 5, implements the "invisible" Grid concept advocated, for instance, in [35, 29].

Overall, this layered implementation schema is implemented as depicted in Fig. 5. The ASSIST compiler is invoked to produce the intermediate object code, then this code is run invoking the ASSIST run tool that in turn relies on the services provided by the Grid Execution Environment (GEA). GEA manages to read the configuration file, to compile the proper object code files using the makefiles produced by the compiler, to stage code and input data to the remote nodes, to start the remote processes computations, and eventually to stage back the result files and suitably terminate the remote processes.

The GEA tool is quite complex, actually. It is derived from former ASSIST program deployment tools ASSIST-conf and ASAP [21]. Its structure is depicted in Fig. 6. GEA provides two distinct kinds of interfaces to client process (the ASSIST run command): one is based on a simple protocol running on top of plain TCP/IP sockets, the other one is actually a Web Service interface. Those interfaces can be used to invoke the services of GEA, namely the ones providing code and data staging, remote process control and monitoring. The GEA engine (ASAP4G) is written in Java and is a plug-in based architecture. Proper plug-in can be provided taking care o proper code staging, launching and synchronization procedures. The ASSIST plug-in is provided by default, that takes care of stage, run and synchronize the ASSIST code and its support code (ASSISTlib, AdHOC, *etc.*).

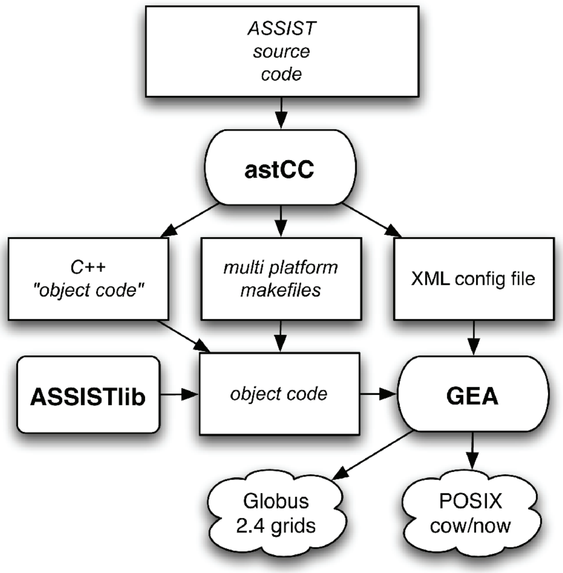

Fig. 5. ASSIST support tools

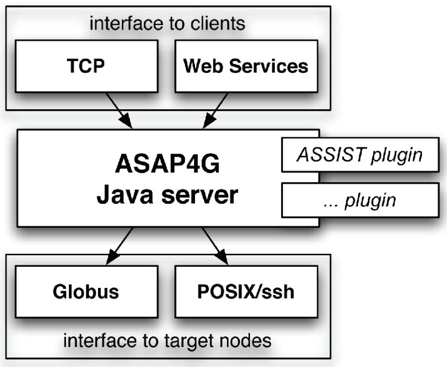

Fig. 6. Structure of GEA

Parmods, the peculiar ASSIST coordination language constructs modeling generic, customizable parallel modules are dealt with in the ASSIST support environment as shown in Fig. 7. The upper part of the Figure represents the logical view of a parmod: a set of logically parallel activities processing data coming from the input stream(s) to produce data on the output stream(s), possibly interacting with external objects and libraries. Parmod implementation is outlined in the lower part of the picture. Each parmod is implemented by a set of processes implementing a peculiar process network. An Input Stream Manager process (ISM) manages the input streams delivering the input data items to a set of Virtual Processor Managers (VPM). VPMs manage virtual processes. Each VPM can be set up to manage a set

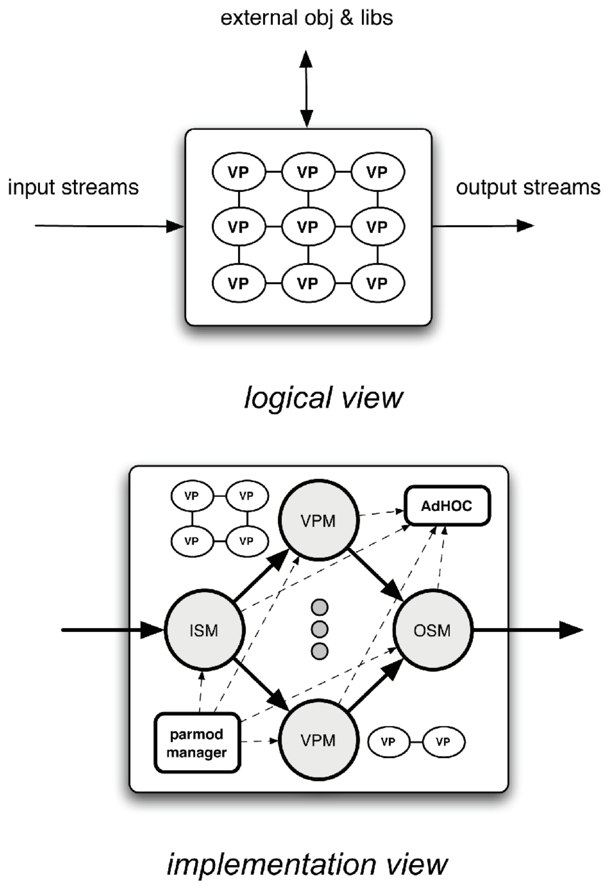

Fig. 7. Parmod implementation

of virtual processes. One VPM is run for each target architecture node available, depending on the requirements stated in the user pragmas or performance contracts. Eventually, an Output Stream Manager (OSM) is run to gather data from the VPMs and deliver them onto the proper output stream(s). The parmod manager process is run to constantly monitor the parmod execution and possibly take any corrective action in case of faults or load imbalances, as discussed in Section 5. The AdHOC process is run to support data sharing across processing elements running ISM, OSM and VPMs and to (partially) support interactions with external objects and libraries. It is actually used also to implement data flow streams connecting the parmod module to the other program modules running on distinct target architecture nodes.

4. PERFORMANCE RESULTS

ASSIST is currently being used by our group at the University of Pisa and at the ISTI/CNR, and by several groups participating to the GRID.it national research project [26]. In particular, it has been used to program computational chemistry applications [19], image processing applications [15], bioinformatics applications [32] and different applications processing SAR satellite images as well as to run several benchmarking applications including data mining code [18] and numerical kernels. The typical results achieved are those shown in Fig. 9. The plot is relative to the performance of a multimedia application. The application is structured as a pipeline with data parallel stages (see Fig. 8). The first stage requests the rendering of a sequence of scenes while the second renders each scene (exploiting the PovRay rendering engine), interpreting a script describing the 3D model of objects, their positions and motion. The third stage collects images rendered by the second one, and builds Groups Of Pictures (GOP), that are sent to the fourth stage, performing DivX compression. The last stage collects DivX compressed pieces and stores them in an AVI output file.

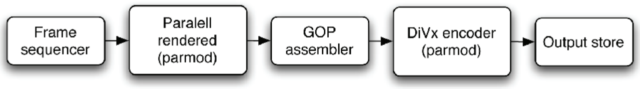

Fig. 8. Structure of the multimedia application

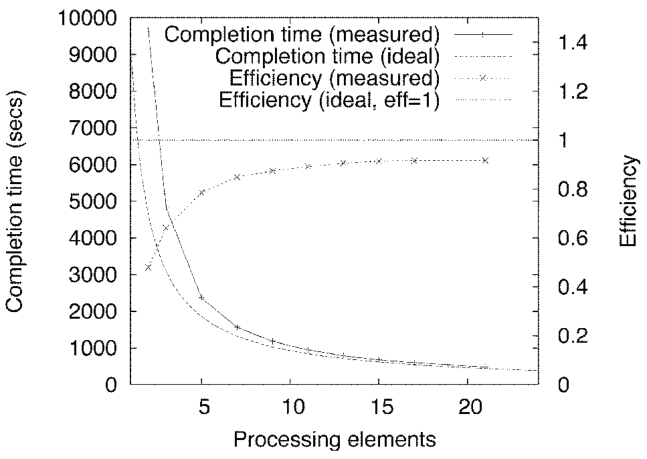

Fig. 9. ASSIST performance results: multimedia application

Overall, the ASSIST environment demonstrated to scale on both cluster and network of workstations (up to 10 to 100 nodes) and on Globus Grid architectures hosting nodes distributed in a geographical network (up to 10 (possibly cluster) nodes). These results have been achieved using medium to coarse grain parallel applications. Furthermore, as the ASSIST/GEA environment schedules interacting parallel activities on the remote nodes, the Globus Grid was used without any kind of process scheduler at the remote Grid nodes. This is necessary to be able to schedule communicating processes on different Grid nodes without incurring in communication starvation and deadlocks.

5. ADAPTIVITY

In this Section, we discuss how Grid adaptivity is implemented in the ASSIST framework. Adaptivity, in conjunction with the transparent compile and run process described in Section 3, achieves the invisible Grid goal, in that programmers may program entire, working and efficient applications without the need to write a single line of code or a single line command directly related to the target architecture middleware at hand.

Adaptivity is needed in Grid computing to take care of the varying features of the Grid nodes, as well as of the different kinds of faults that can be verified on large scale Grids. Node features can vary basically due to the fact that nodes are usually shared with other Grid applications and users. Therefore, resources with a small load can become very busy or resources with a high load can become suddenly available to accept further load. This is true not only for computing resources, but also for interconnection resources, such as communication links and routers. Faults can be related to actual hardware faults (link and processing element faults) as well as to temporary unreachability of physical resources due to link load hot spots.

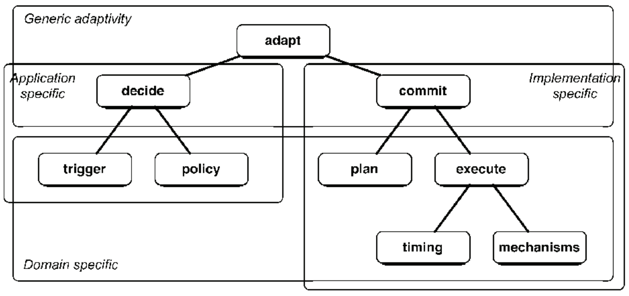

Fig. 10. General schema for adaptivity

In a joint work with collegues in IRISA/INRIA [1] we developed a general adaptation schema, that is being currently used in ASSIST such as the one depicted in Fig. 10, that can be used to control Grid programs adaptivity. We clearly recognized two distinct phases in the adaptivity process: in the first, decide phase some event triggers the adaptivity process and a policy library is consulted to figure out which kind of solutions can be devised to overcome the event triggering the process. As an example, the triggering event can be generated by a monitoring process figuring out that some parts of the parallel program do not perform as expected. In case the parallel application is structured as a task farm, the policy library may suggest that either the number of worker processes has to be increased or some "slow process" has to be moved to a faster resource. The second, commit phase is aimed at implementing the decisions taken in the first phase. Therefore a plan is figured out stating how the taken decision is to be implemented, then an execute phase takes care of figuring out which mechanisms have to be used and when they have to be exploited.

Such an abstract adaptation schema is exploited in the parmod manager process. The manager process receives a performance contract from the user code, stating the kind of behavior expected from the parmod. The contract may state, as an example, that the parmod should be able to process a new input data set each *t* (milli) seconds. When the parmod is actually executed, special monitor process are run that constantly monitor the performance of parmod. In case a poorer performance is measured with respect to the user supplied performance contract, the manager process is informed and a corrective action is planned. In this case, the policy that can be adopted requires to increment the number of resources employed in the parmod execution. Therefore, a commit phase is started that performs the following steps:

1. while the original parmod is still running, new available resources (processing elements) are searched and one of them is selected to run a new VPM taking care of executing a sub partition of the virtual processes in the parmod. In case there are no new resources to recruit to the current computation, or in case either having new resources available, the overhead required to recruit them is supposed to be higher than the expected benefit, the process is stopped and a unsatisfied performance contract event is raised to the user. The user may decide then if the computation has to continue or it has to be restarted.
2. then, the parmod is stopped as soon as it reaches a synchronization point. Synchronization points are not explicitly inserted by programmer in the application code (as it happens in AFPAC, for instance [11]). Rather, they can be figured out looking at the structure of the parmod computations. They are rather inserted in the object code by the compiler considering the structure of the particular parmod,
3. after stopping the parmod computation the virtual processors and the parmod state are redistributed across the existing VPMs and the new one added on the newly recruited processing resource,
4. eventually the parmod computation is restarted with one more VPM and the monitor process generating the triggering events is restarted as well.

In case a consistently higher performance is measured with respect to the one indicated in the user supplied performance contract, instead, a similar process is started to dismiss one of the resources used to compute a VPM and to redistributed the VPs managed by that VPM to the other VPMs.

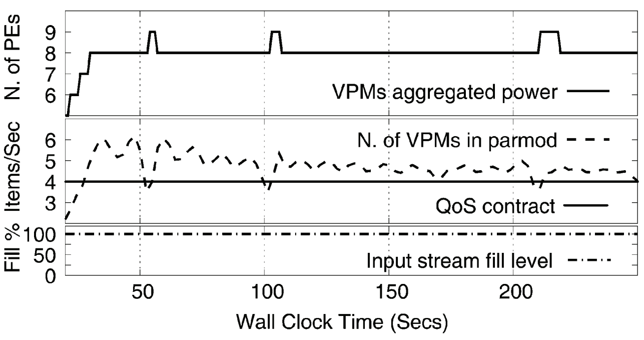

Fig. 11. Effect of adaptivity process implemented in a parmod manager

All the process can be performed in this way as the computation managed by the parmod manager process is a structured parallel computation whose features are completely known (from the compiler) to the manager.

Figure 11 shows the results achieved while running a parmod with the manager on a set of workstation whose load has been varied by putting additional load on one of the processing elements involved. At time 50 and 100, for instance, additional load was put on one of the processing elements involved in the parmod computation. The monitoring process detected the loss of performance under the user supplied performance contract, asking to process 4 items per second and the manager recruited a new resource to correct the problem. Immediately after, as the contract was "over satisfied" one resource is released, possibly not the one just recruited but somehow the "slower" in the pool of resources running VPMs.

Interested readers can found more information concerning ASSIST (component) adaptivity in [5, 8, 7].

6. HIGHER LEVEL COMPONENTS

The experience gained with the implementation of parmod managers and of the related adaptation policies was exploited in the GRID.it framework by developing two specific "supercomponents": a task farm component and a generic graph component. These components have been developed in the GRID.it component framework [5], that is the project component framework using ASSIST to implement the parallel components and either wrapped sequential C, C++ or Fortran code or even Web Services or CCM components as sequential components.

The task farm supercomponent can be customized providing a worker component to model any task farm (or master slave) parallel application. The component comes with an embedded manager that implements (best effort) policies to adapt the task farm performance to the aggregate power of the resources used to run the workers by recruiting new (releasing) workers in case of performance loss (gain) with respect to the user supplied performance contract.

The generic graph supercomponent can be customized providing the set of components that have to be interconnected in a generic graph (or in a pipeline, as a sub-case) and specifying the generic graph through proper data flow streams. This component is also provided with a manager that takes care of ensuring the user performance contract by providing to balance the input/output bandwidths of the components participating in the graph.

Both these supercomponents can be used to program high performance Grid parallel applications basically without requiring the programmers any effort to program all the structure code needed to run the components in a farm (or in a generic graph) structure, nor any code to keep care of adapting their run time behavior to the features of the Grid target architecture at hand. Preliminary experimental results have been achieved that show perfect functionality of these components as well as good efficiency in the related runs on both workstation networks and Grids.

7. RELATED WORK

The HOC project at Muenster is probably the most related project to our work on supercomponents [10, 23, 24]. HOC (High Order Components) is a programming environment allowing users to use predefined components that implements notable parallelism exploitation patterns including the ones implemented by ASSIST supercomponents, *i.e.* task farms and pipelines. These notable parallelism exploitation patterns are derived from the algorithmical skeletons originally developed for cluster architectures, such as those present in [28, 20, 17, 13]. Differently from ASSIST, the HOC implementation exploits Web Service technology to deploy components on remote Grid processing elements. However, HOC component do not support at the moment any kind of adaptivity.

The Condor Grid programming system also provides methods to program task farm computations [36]. In this case adaptivity is achieved by suitably programming the scheduling policies of the Condor tool. However, the kind of computations addressed are basic batch computations rather than full parallel programs as in the ASSIST framework, where task farms can be included in any one of the modules appearing in the program module graph.

Ibis [40] is a Grid programming system that implements the invisible Grid concept while leaving the programmer the full control over a set of communication mechanisms fully integrated in plain Java. While the mechanisms provided in base Ibis are not too high level (in particular they require programmers the full control over process decomposition and over the related communications), extensions of the environment provide users with higher-level parallel patters. In particular Satin [39] provides handy ways of implementing divide&conquer parallel applications. Adaptivity and heterogeneity are taken into account as well in Ibis: heterogeneity is solved by using plain Java byte code (with extensions to implement serialization in more efficient way than standard Sun serialization). Adaptivity is handled exploiting adaptive load balancing in the implementation of Satin divide&conquer [41].

Adaptivity in Grid computations is provided within the AFPAC framework [11, 14]. This framework was originally meant to model SPMD computation written using MPI and recently evolved to cover generic Grid components as well. AFPAC programmers have complete control over the adaptivity process. This means that a set of mechanisms are provided within AFPAC to implement an adaptivity schema fitting the one of Fig. 10 (that in fact has been developed in a joint work with the AFPAC researchers) but programmers must take care of inserting proper "adaptation points" in their code as well to completely take care of implementing adaptivity policies and commit phases.

COPS [30] is a design pattern based Java programming environment that allows users to design experiment and include in the environment new parallel patterns. To our knowledge this is the only one structured parallel programming environment providing such possibility to the users. Despite the fact that ASSIST parmod parameters are thought to allow users to express a variety of different parallelisms exploitation patterns, both ASSIST and its supercomponents are far from reaching the same degree of reconfigurability allowed in COPS.

Other approaches aimed at providing Grid programming environments are far from achieving the invisible Grid goal and do not provide any kind of support for adaptivity. The only project with a deep support to user provided performance contracts and adaptivity is the GrADs project in the U.S. [27]. The GrADs project also accommodated the development of higher level programming tools for Grids, such as the Grid-RPC model [38]. This programming model, although higher level than classical Grid programming at the Grid middleware level is far from the abstraction level provided by either ASSIST or HOC, however. Recent papers on GrADs adaptivity present results that can be considered preliminary to the ones achieved in ASSIST [37].

The Fractal component framework developed in France by INRIA and France Telecom [34], although not explicitly designed for parallel or Grid processing, defines a concept of "component membrane" which is meant to encapsulate component controllers (or managers) in a way which is very similar to the way of instrumenting ASSIST components with managers.

8. CONCLUSIONS

We shortly outlined the main features of the ASSIST Grid-programming environment, and we discussed its layered implementation. We then discussed new results concerning adaptivity. Overall, the ASSIST environment can be considered a programming environment fulfilling the "invisible Grid" goal, that is, allowing programmers to write efficient Grid programs without actually being concerned with all the details related to the (efficient) usage of the Grid middleware. ASSIST is currently being released under open source license and can be downloaded from the ASSIST web site at the address www.di.unipi.it/Assist.html.

Acknowledgments

We wish to thank all the people that participated in the development of ASSIST and of the related tools: Carlo Bertolli, Sonia Campa, PierPaolo Ciullo, Gianni Giaccherini, Silvia Magini, Andrea Paternesi, Paolo Pesciullesi, Alessandro Petrocelli, Edoardo Pistoletti, Laura Potiti, Roberto Ravazzolo, Massimo Torquati, Gianni Virdis, Luca Veraldi, Pietro Vitale.

References

[1] M. Aldinucci, F. Andr´e, J. Buisson, S. Campa, M. Coppola, M. Danelutto and C. Zoccolo, *Parallel program/component adaptivity management,* In: PARCO 2005: Parallel Computing, Malaga, Spain, 2005. To appear.

[2] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, L. Veraldi and C. Zoccolo, *Self-Configuring and Self-Optimising Grid Components in the GCM model and their ASSIST Implementation*, Proceedings of the HPC-GECO/Compframe workshop held in conjunction with HDPC-15, IEEE Press, Paris, June 2006.

[3] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi and C. Zoccolo, *ASSIST demo: a high level, high performance, portable, structured parallel programming environment at work,* In: 9th Intl Euro-Par: Parallel and Distributed Computing, volume 2790 of LNCS, 1295-1300, Springer Verlag, Klagenfurt, Austria, August 2003.

[4] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi and C. Zoccolo, *A framework for experimenting with structure parallel programming environment design*, In: Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, vol. 13 of Advances in Parallel Computing, 617-624, Dresden, Germany, Elsevier 2004.

[5] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi and C. Zoccolo, *Components for High-Performance Grid Programming in GRID.it*, In: Component modes and systems for Grid applications, CoreGRID. Springer Verlag, 2005.

[6] M. Aldinucci, S. Campa, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati and C. Zoccolo, *Targeting heterogeneous architectures in ASSIST: Experimental results*, In: 10th Intl Euro-Par: Parallel and Distributed Computing, vol. 3149 of LNCS, 638-643, Pisa, Italy, August 2004. Springer Verlag.

[7] M. Aldinucci, M. Danelutto and M. Vanneschi, *Autonomic QoS in ASSIST grid-aware components,* In: Euromicro PDP 2006: Parallel Distributed and network-based Processing, Montb`eliard, France, February 2006. IEEE. to appear.

[8] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi and C. Zoccolo, *Dynamic reconfiguration of grid-aware applications in ASSIST*, In: 11th Intl Euro-Par: Parallel and Distributed Computing, vol. 3648 of LNCS, 771-781, Portugal, Springer Verlag, August 2005.

[9] M. Aldinucci and M. Torquati, *Accelerating apache farms through ad-HOC distributed scalable object repository*, In: 10th Intl Euro-Par: Parallel and Distributed Computing, vol. 3149 of LNCS, 596-605, Pisa, Italy, Springer Verlag, August 2004.

[10] M. Alt, J. Dünnweber, J. Müller and S. Gorlatch, *HOCs: Higher-order components for grids*, In: V. Getov and Th. Kielmann (ed.) *Component Models and Systems for Grid Applications*, CoreGRID, 157-166. Springer Verlag, June 2004.

[11] F. André, J. Buisson and J.-L. Pazat, *Dynamic adaptation of parallel codes: toward self-adap table components for the Grid,* In Workshop on component Models and Systems for Grid Applications, June 2005.

[12] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti and M. Vanneschi, *P3L: A Structured High level programming language and its structured support*, Conc. Practice and Experience, **7(3)**, 225-255, 1995.

[13] A. Benoit, M. Cole, S. Gilmore and J. Hillston, *Flexible Skeletal Programming with eSkel,* In: 11th Intl Euro-Par: Parallel and Distributed Computing, vol. 3648 of LNCS, 761-770, Lisbona, Portugal, Springer-Verlag August 2005.

[14] J. Buisson, F. Andr´e and J.-L. Pazat, *Enforcing consistency during the adaptation of a parallel component*, In: Proceedings of the 4th Int. l Symposium on Parallel and Distributed Computing, July 2005.

[15] A. Clematis, D. D'Agostino and V. Gianuzzi, *Parallel Compression of 3D Meshes for Efficient Distributed Visualization*, In: Proc. of International Conference on Parallel Computing 2005 (PARCO 2005), 2005.

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

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

[18] M. Coppola and M. Vanneschi, *Parallel and Distributed Data Mining through Parallel Skeletons and Distributed Objects,* In: Data Mining: Opportunities and Challenges, pp. 106-141. IDEA Group Publishing, 2003.

[19] S. Crocchianti, A. Laganà, L. Pacifici and V. Piermarini, *Parallel skeletons and computational grain in quantum reactive scattering calculations*, In: Parallel Computing: Advances and Current Issues. Proceedings of the International Conference ParCo2001, 91-100. Imperial College Press, 2002.

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

[21] M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, R. Baraglia, T. Fagni, D. Laforenza and A. Paccosi, *HPC application execution on grids,* In: Future Generation Grids, CoreGRID series. Springer-Verlag, November 2005.

[22] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu and R. L. While, *Parallel Programming Using Skeleton Functions,* In: M. Reeve A. Bode and G. Wolf (ed.) PARLE'93 Parallel Architectures and Langauges Europe. Springer Verlag, June 1993. LNCS, No. 694.

[23] J. Dünnweber and S. Gorlatch, *HOCSA: A grid service architecture for higher-order components*, In IEEE International Conference on Services Computing, Shanghai, China, pp. 288-294. IEEE Computer Society Press, September 2004.

[24] J. Dünnweber and S. Gorlatch, *Component-based Grid Programming using the HOC-Service Architecture*. In: I. H. Fujita (ed.) *New Trends in Software Methodologies, Tools and Techniques, Frontiers in Artificial Intelligence and Applications,* IOS Press, 2005. Accepted for publication.

[25] I. Foster and C. Kesselmann, *The Grid: Blueprint for a New Computing Infrastructure,* Morgan Kaufmann, 1998.

[26] The GRID.it home page, 2005. http://www.grid.it.

[27] 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.

[28] H. Kuchen, *A Skeleton Library*, In: Euro-Par 2002, Parallel Processing, no. 2400 in LNCS, 620-629. Springer Verlag, August 2002.

[29] D. Laforenza, *Grid programming: some indications where we are headed,* Parallel Computing, **28(12)**, 1733-1752, (2002).

[30] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron and K. Tan, *From Patterns to Frameworks to Parallel Programs,* Parallel Computing, **28(12)**, 1663-1683 (2002).

[31] S. Magini, P. Pesciullesi and C. Zoccolo, *Parallel software interoperability by means of CORBA in the ASSIST programming environment.* In 10th Intl Euro-Par: Parallel and Distributed Computing, vol. 3149 of LNCS, 679-688, Springer-Verlag, Pisa, Italy, August 2004.

[32] I. Merelli, L. Milanesi, A. Clematis, D. D'Agostino, M. Vanneschi and M. Danelutto, *Using Parallel Isosurface Extraction in Superficial Molecular Modeling*, In: Proc. 1st Conference on Distributed Frameworks for Multimedia Applications (DFMA'05), 288-294. IEEE Computer Society, 2005.

[33] Future for European Grids: GRIDs and Service Oriented Knowledge Utilities Vision and Research Directions 2010 and Beyond, January 2006. available at http://www.cordis.lu/ist/grids.

[34] ObjectWeb Consortium. The Fractal Component Model, Technical Specification, 2003.

[35] D. Snelling and K.-Jeffrey *et al*., *Next Generation Grids 2 – Requirements and Options for European Grids Research 20052010 and Beyond, 2004*. available at ftp://ftp.cordis.lu/pub/ist/docs/.

[36] D. Thain, T. Tannenbaum and M. Livny, *Condor and the grid,* In: F. Berman, G. Fox and T. Hey (ed.) *Grid Computing: Making the Global Infrastructure a Reality*, John Wiley & Sons Inc., December 2002.

[37] S. Vadhiyar and J. Dongarra, *Self adaptability in grid computting,* Concurrency & Computation: Practice & Experience, **17(2-4)**, 235-257, 2005.

[38] S. Vadhiyar, J. Dongarra and A. YarKhan, *GrADSolve – RPC for high performance computing on the Grid.* In Proc. of the Euro-Par 2003, no. 2790 in LNCS, 394-403. Springer, August 2003.

[39] Rob V. van Nieuwpoort, J. Maassen, Th. Kielmann and H. E. Bal, *Satin: Simple and efficient Java-based grid programming,* Scalable Computing: Practice and Experience, **6(3)**, 19-32 (2005).

[40] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, Th. Kielmann and H. E. Bal, *Ibis: a flexible and efficient Java based grid programming environment*, Concurrency and Computation: Practice and Experience, **17(7-8)**, 1079-1107 (2005).

[41] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, Th. Kielmann and H. E. Bal, *Adaptive load balancing for divide-and-conquer grid applications*, Accepted for publication in Journal of Supercomputing, 2004.

[42] M. Vanneschi, *The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications*, Parallel Computing, 12, December 2002.

Marco Aldinucci got the PhD in Computer Science in 2003, he has been researcher at the Institute of Information Science and Technologies of the Italian National Research Council (ISTI-CNR, 2003-2006), and he is currently research associate at Computer Science Dept. of the University of Pisa, Italy. He is author of about 40 papers appearing in journals and international refereed conference proceedings, together with more than 15 different co-authors. He has been and is currently participating in more than 10 national and international research projects concerning parallel computing and Grid topics, including the Grid.it Italian national project, CoreGRID EC Network of Excellence, GridComp EC-STREP, BEinGRID EC-IP, XtreemOS EC-IP, GridCoord EC-SSA. His main research is focused on parallel/distributed computing in network of workstations and grids, and in particular on models and tools for high-level parallel programming, component-based frameworks, autonomic computing, and distributed shared memory systems. He contributed to the design and the development of a number of tools for parallel processing, including compilers, libraries and frameworks, both in industrial and academic teams.

Massimo Coppola was born in 1969 in Civitavecchia, Italy. He obtained the MS and PhD in Computer Science from the University of Pisa (October 2002), with a PhD thesis on the application of Structured Parallel Programming to Data Mining. He has been Research Assistant of the C.S. Dept. of Pisa (2002-2003), working on the same topic. He collaborated with QSW-Alenia Spazio (1997), and Consorzio Pisa Ricerche (1997, 2002), and has been Assistant Professor with the ISTI/ CNR institute in Pisa (2003-2006). He is currently Research Associate with the Department of Computer Science of Pisa, where he teaches Advanced Parallel Programming (CCP). He mainly works on High-level Parallel Programming Environments, collaborating to the development of the ASSIST environment, and on Component programming Models for Computational Grids, in the framework of several Italian (Grid.it) and European Research Projects (CoreGRID, GridComp, XtreemOS). His interests include languages, tools and performance models for structured parallel computing, run-time support, deployment and distributed management of parallel, self-adapting applications over large-scale heterogeneous Grids, and paradigms for parallel, I/O intensive applications to efficiently exploit distributed I/O over large Clusters.

Marco Danelutto received the PhD in Computer Science in 1990 and since 1998 he is an associate professor at the Department of Computer Science of the University of Pisa. His main research interests are in the field of parallel, distributed and grid architectures and in the design and implementation of structured parallel programming environments for such kind of architectures. In the '90s, he has been one of the main designers of P3L, the Pisa Parallel Programming Language, a skeleton based structured parallel programming environment targeting clusters and workstation networks. Recently, he actively participated in the development of the ASSIST programming environment and he currently maintains the muskel Java skeleton library. He is author of more than 80 scientific publications on international journals and conferences. He is member of the program committees of several conferences, including Europar, that he organized and co-chaired in 2004. In the recent years, he has been the national coordinator of the activities of work package 8 in the GRID.it Italian national project, aimed at developing the ASSIST component based parallel programming environment, and he is currently leading the Programming model Institute of the EU CoreGRID network of excellence and member of the CoreGRID executive committee.

Nicola Tonellotto received the "Laurea" degree (cum Laude) in Computer Engineering in 2002 at the faculty of Engineering, University of Pisa. Currently, he is a PhD student in Information Engineering at the Information Engineering Department of the University of Pisa jointly with the University of Dortmund. He holds a research fellowship at the High Performance Computing Laboratory of the Information Science and Technologies Institute of the Italian National Research Council from 2002. He is author of about 10 papers appearing in journals and international refereed conference proceedings. He participated in the development of several Grid-related softwares. He has been and is currently participating in more than 5 national and international research projects concerning high performance computing and Grid topics, including the Grid.it Italian national project, CoreGRID EC Network of Excellence, GridComp EC-STREP, XtreemOS EC-IP. His main research is focused on parallel/distributed computing in network of workstations and grids, and in particular on models and tools for high-level parallel programming, component-based frameworks and Grid middlewares integration.

Marco Vanneschi is a full professor at the Department of Computer Science of the University of Pisa. His research and teaching activity is mainly related to Computer Systems Architecture, and in particular High Performance Computing systems: hardware and firmware architectures and organizations, computational models for parallel computing, high performance enabling platforms, programming tools and environments for the development of parallel and distributed applications, Grid computing. In this area, he has participated to some European projects and he has coordinated several national projects. He is the coordinator of the National Basic Research Programme on Grid Computing (Grid.it: Enabling Platforms for High Performance Computational Grids oriented to Scalable Virtual Organizations), funded by MIUR. He is author of more than 170 scientific papers published in international journals and conferences, of three books on computer architecture and parallel programming, and he is scientific editor of six international books.

Corrado Zoccolo graduated "cum Laude" at "Scuola Normale Superiore" of Pisa and got the PhD in Computer Science in 2005 at University of Pisa. He is author of about 15 papers appearing in journals and international refereed conference proceedings. He has been a research collaborator of the Computer Science Dept. of the University of Pisa, Italy. He partecipated in 4 national and international research projects on High Performance and Grid computing, as well as in industrial projects where these technologies were applied. His research work is focused on high-level programming tools and abstractions for high-performance computing. His main interest is in automatic QoS enforcement on performance unstable platforms. Currently, he is Senior Software Engineering at Ask.com Italy, a division of IAC Search & Media.