Autonomic QoS in ASSIST Grid-aware components

Marco Aldinucci
Inst. of Information Science
and Technologies, CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
aldinuc@di.unipi.it

Marco Danelutto
Dept. of Computer Science
University of Pisa
Largo B. Pontecorvo, 3
56127 Pisa, Italy
marcod@di.unipi.it

Marco Vanneschi
Dept. of Computer Science
University of Pisa
Largo B. Pontecorvo, 3
56127 Pisa, Italy
vannesch@di.unipi.it

Abstract

Current Grid-aware applications are developed on existing software infrastructures, such as Globus, by developers who are experts on Grid software implementation. Although many useful applications have been produced this way, this approach may hardly support the additional complexity to Quality of Service (QoS) control in real application. We describe the ASSIST programming environment, the prototype of parallel programming environment currently under development at our group, as a suitable basis to capture all the desired features for QoS control for the Grid. Grid applications, built as compositions of ASSIST components, are supported by an innovative Grid Abstract Machine, which includes essential abstractions of standard middleware services and a hierarchical Application Manager, which may be considered as an early prototype of Autonomic Manager.

Keywords: Components, Structured Parallel Programming, Autonomic Computing, Grid, Adaptive Applications.

1. Introduction

Grid computing is supposed to enable the development of large scientific applications on an unprecedented scale. The key idea behind Grid-aware applications consists in making use of the aggregate power of distributed resources, thus benefiting from a computing power that falls far beyond the current availability threshold in a single site. Despite the huge computing power potentially available on the Grid, developing algorithms able to exploit it is currently likely to be a hard task. To realize the potential, programmers must design highly concurrent algorithms that can execute on large-scale platforms, which can be assumed neither homogeneous, secure, reliable nor centrally managed. They must then implement these algorithms correctly and efficiently [13]. As results in order to build efficient Grid-aware applications, programmers have to face up classical problems of parallel computing as well as Grid-specific ones:

- code all the algorithm details, take care about concurrency exploitation, among the others concurrent activities set up, mapping/scheduling, communication/synchronization handling and data allocation;
- manage resources heterogeneity and unreliability, networks latency and bandwidth unsteadiness, resources topology and availability changes.

Hence, the number and quality of problems to be resolved in order to draw a given QoS (in term of performance, robustness, etc.) from Grid-aware applications is quite large. The lesson learned from parallel computing suggests that any low-level approach to Grid programming is likely to raise the programmer's burden to an unacceptable level for any real world application.

Therefore, we envision a layered, high-level programming model for the Grid. In such software architecture the bottom tiers should cope with key Grid requirements for protocols and services (connectivity protocols concerned with communication and authentication, resource protocols concerned with negotiating access to individual resources) and collective protocols and services (concerned with the coordinated use of multiple resources) [13]. As top tier regards, we envision a Grid-aware application as the composition of a number of coarse grained, cooperating components within a high-level programming model, which is characterized by a high-level view of compositionality, interoperability, reuse, performance and application adaptivity. Applications are expressed entirely on top of this level. This vision is currently pursued by several research initiatives and programming environments, among the others, within the ASSIST [23] and GrADS [16] projects.

The underlying idea of these programming environments is moving most of the Grid specific efforts needed while developing high-performance Grid applications from pro-

Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)
1066-6192/06 $20.00 © 2006 IEEE

grammers to Grid tools and run-time systems. This leaves programmers the responsibility of organizing the application specific code and the programming tools (i.e. the compiling tools and/or the run-time system) the responsibility of properly interacting with the Grid.

In such a scenario, QoS constraints of the whole application and their components are naturally a – *static* and/or *dynamic* – attribute of components and their composition. In both cases, the run-time system should actively operate in order to fulfill QoS requirements of the applications, since any static resource assignment may violate QoS constraints due to the very uneven performance of Grid resources over time. Therefore, the run-time support is required to exploit a certain degree of self-management (configuration, optimization), or in other words to be partially *autonomic*.

In this paper we take in account the ASSIST programming environment and we sketch how its run-time support may be enriched with some self-management features targeted to fulfill QoS requirements expressed at the language level.

In the next section we sketch the Autonomic Computing idea. In Section 3 we briefly present the ASSIST programming environment and its run-time support for the Grid, while in Section 4 we discuss ASSIST autonomic features. In Section 5 we present some experiments aiming to show the effectiveness of proposed architecture. Related works and final remarks conclude the paper.

2. Autonomic Computing

The term *Autonomic Computing* is emblematic of a vast *hierarchy* of natural self-governing systems, many of which consist of many interacting, self-governing components that in turn comprise a number of interacting, self-governing components at the next level down[1] [17]. Autonomic computing aims to attack the complexity, which entangle complex system management and optimization by equipping their parts with self-managements facilities.

IBM tries to tackle the problem with the oftenquoted five "selves": self-configuration, self-healing, self-optimization, self-protection, and, as a combination of all, self-management. As shown in Fig. 1, an autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls them. The managed element could be a hardware resource (storage, CPU, etc.), or a software resource, such as a Web service, or a software *component*. Control loops, which are known in optimization theory since (at least) the mid of the last century, can be used to apply the five

[1]The term, introduced by IBM in 2001, derives from the body's autonomic nervous system, which controls key functions without conscious awareness.

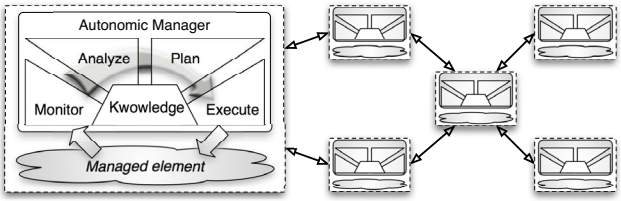


Figure 1. Structure of an autonomic element. Elements interact with other elements and with human programmers via their autonomic managers [17].

"selves". They split the optimization process into 1) a *monitoring* phase, where the symptoms are collected; 2) an *analysis* phase, where the current status is checked against the goal status; 3) a *planning* phase, where a plan is created to enact the desired alterations according to some policies; and 4) the *execution* phase, which provides the mechanisms to schedule and perform the necessary changes to the system [7, 17].

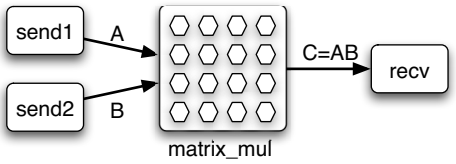
Truly autonomic systems are years away, although in the nearer term, autonomic functionality will appear in software, especially in very complex systems as Grid-aware applications. In particular, early autonomic system may threat the five "selves" as distinct aspects, with different solutions that address each one separately.

3. The ASSIST Programming Environment

Current applications are developed atop existing software infrastructures, such as Globus, by developers who are experts on Grid software implementation. They must design highly concurrent algorithms that can execute on large-scale platforms. They must then implement these algorithms correctly and efficiently. Although many useful applications have been produced this way, this approach can hardly rule increasing complexity due to application QoS control. Indeed, several research projects are targeted toward Grid high-level programming, among the other, ASSIST [23], GrADS [16], ProActive/Fractal [9, 10], Condor [20], Ibis [22], XCAT [14], eSkel [11]. In the rest of the section we sketch the ASSIST programming environment and its architecture, a complete description can be found in [1, 3].

3.1. The ASSIST coordination language

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data. Each




```
1   generic main() {
2     stream long[N][N] s1;
3     stream long[N][N] s2;
4     stream long[N][N] s3;
5     send1  (output_stream  s1);
6     send2  (output_stream  s2);
7     matrix_mul (input_stream  s1, s2)
8                 output_stream  s3);
9     recv   (input_stream s3);
10  }
11
12  ...
```

```
20  parmod matrix_mul (input_stream long M1[N][N], long M2[N][N],
21                     output_stream long M3[N][N]){
22    topology array [i:N][j:N] Pv;
23    attribute long A[N][N] scatter A[*ia][*ja] onto Pv[ia][ja];
24    attribute long B[N][N] scatter B[*ib][*jb] onto Pv[ib][jb];
25    stream long ris;
26    do input_section {
27      guard1: on , , M1 && M2 {
28        distribution M1[*i0][*j0] scatter to A[i0][j0];
29        distribution M2[*i1][*j1] scatter to B[i1][j1];
30      }   } while (true)
31    virtual_processes {
32      elab1 (in guard1 out ris) {
33        VP i, j { f_mul (in A[i][], B[][j] output_stream ris);}}}
34    output_section {
35      collects ris from ALL Pv[i][j] {
36        int elem; int Matrix_ris_[N][N];
37        AST_FOR_EACH(elem) {
38          Matrix_ris_[i][j]=elem;
39        }
40        assist_out(M3, Matrix_ris_);
41      }<>; } }
42  proc f_mul(in long A[N], long B[N] output_stream long Res)
43  $c++{ register long r=0;
44    for (register int k=0; k<N; ++k)
45      r += A[k]*B[k];
46    assist_out(Res,r); }c++$
```

Figure 2. Sample of matrix multiplication code in ASSIST

stream realizes a one-way asynchronous channel between two sets of endpoint modules: sources and sinks. Data items injected from sources are broadcast to all sinks. All data items injected into a stream should match stream type.

Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential functions that are activated and run as *Virtual Processes* (VPs) on items arriving from input streams. The VPs may synchronize with the others through barriers. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran).

A *parmod* may behave in a data-parallel (e.g. SPMD/forall/apply-to-all) or task-parallel (e.g. farm) way and it may exploit a distributed shared state, which survives to VPs lifespan. A module can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to instantiate VPs according to the input and distribution rules specified in the parmod. The VPs may send items or parts of items onto the output streams, and these are gathered according to the output rules. The simple application in Fig. 2 includes three sequential modules (`send1`, `send2`, and `recv`) and one parmod (`matrix_mul`), which take two matrixes and give their product along three different streams (lines 2–9).

A parmod is characterized by four regions of code describing its behavior: `topology`, `input_section`, `virtual_processes`, and `output_section`. The `topology` declaration specializes the behavior of the Virtual Processes as farm (topology `none`), or SMPD (topology `array`). The `input_section` enables programmers to declare how VPs receive data items, or parts of items, from streams. A single data item may be distributed (scattered, broadcast or unicast) to many VPs. The `input_section` realizes a CSP repetitive command [15]. The `virtual_processes` declarations enable the programmer to realize a parametric Virtual Process starting from a sequential function (`proc`). VPs may be identified by an index and may synchronize and exchange data one with another through the ASSIST language API. The `output_section` enables programmers to declare how data should be gathered from VPs to be sent onto output streams. More details on the ASSIST coordination language can be found in [23, 3].

The example parmod in Fig. 2 exhibits a `topology array` (line 22, NxN VPs behaving in a SPMD fashion). Once the two input matrixes are received (line 27), they are both scattered to the VPs which store them in the distributed shared matrixes A and B (lines 28–29) that has been previously declared (lines 23–24). Then, all elements of the result matrix C are computed in parallel (lines 31–33). Once all VPs completed the operation, a result matrix is collected from the distributed matrix C and sent into the output stream (lines 34–41). The code of sequential modules is not shown for the sake of brevity.

The ASSIST compiler translates a graph of modules into a network of processes. As sketched in Fig. 3 a), sequential modules are translated into sequential processes, while parallel modules are translated into a parametric (w.r.t. the parallelism degree) network of processes: one *Input Section Manager* (ISM), one *Output Section Manager* (OSM), and a set of *Virtual Processes Managers* (VPMs, each of them running a set of Virtual Processes). Also, a number of other processes devoted to application QoS control are added to the network (not shown in Fig. 3). We shall introduce them in the next sections.

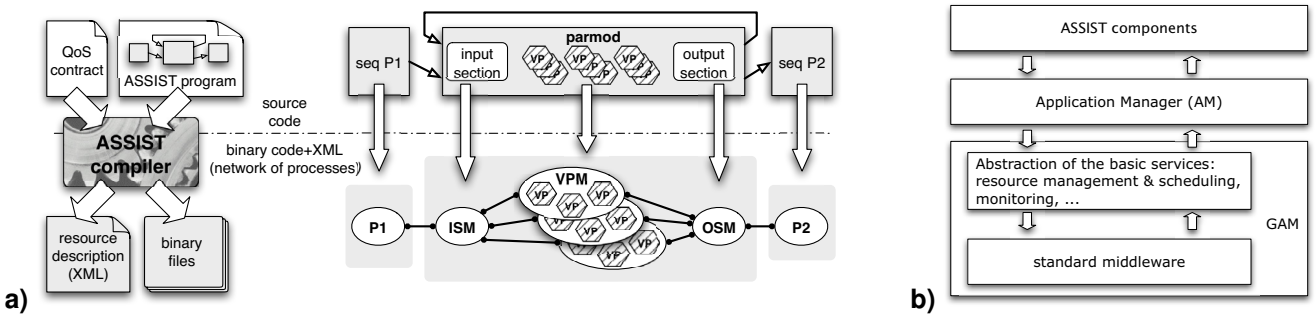


Figure 3. a) An ASSIST application and a QoS contract are compiled in a set of executable codes and its meta-data [3], this information is used to set up a processes network at launch time: hexagons represent Virtual Processes, ovals represents processes, solid edges represent data channels, dashed edges management channels. b) ASSIST software architecture.

3.2. Components and Managers

A single parmod or a graph of them may be declared as component (ASSIST native component). In addition a graph of components and the whole application may be recursively declared as a component. A component is characterized by provided and used ports, as well as by *Non-Functional ports*, which are related to component QoS control. Each port of an ASSIST native component may be configured to behave as endpoint of one-way stream connection, RPC method, or event channel. ASSIST native component may also interoperate with Corba/CCM (via IIOP based RPC) components [18] or Web services (via HTTP/SOAP) [4].

The software architecture of the ASSIST component-based parallel programming environment is organized as shown in Fig. 3 b). The run-time environment of ASSIST 1.3 is implemented on top of a *Grid Abstract Machine* (GAM). The GAM implements *abstract services*, i.e. the functionalities needed by the programming environment to support high-performance, component-based Grid-aware applications. These regard resources discovery, management and monitoring; components deployment, run, and wiring; routing of communications through networks with private addresses, and other services (accounting and so on). Whether possible, these services rely on the underlying Grid middleware, which are just abstracted out at the GAM level. In other cases, GAM services *extend* Grid middleware services (e.g. monitoring) [2, 12]. More specifically, the GAM and the related GAM deployment tool are currently implemented on top of two distinct frameworks: plain POSIX TCP/IP OS and Globus:

- POSIX `ssh/scp` tools are exploited to stage data and code and to remotely execute commands. Resource lookup is performed consulting configuration XML files. This GAM implementation is primarily thought to target clusters/networks of workstations.
- Globus CoG toolkit is used to perform code and data staging, to remotely execute commands and to access MDS, i.e. to gather information about the available Grid resources. The possibility to use XIO to implement communications and NWS to support the monitoring process is also taken into account.

In both cases, the GAM deployment tool is able to deal with multi-site deployment, even in case job schedulers manage the sites since different components of an ASSIST application do not require a strict co-allocation of resources.

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

The Application Manager (AM) has a *hierarchical structure* that can be extended at any level according to the compositionality and abstraction strategy adopted for the application. Each ASSIST module has associated a *Module Application Manager* (MAM): each MAM_i is the responsible of the configuration and QoS control of the associated module. A global strategy for the configuration control of the whole component is implemented by the *Component Application Manager* (CAM).

For example, Fig. 4 shows an application in which we recognize four components, component a consisting of modules M_1, component b consisting of modules M_2, and component c consisting of modules M_3 and M_4. The whole application exposes a provided port. Each module and component has an associated management entity: CAM_a, CAM_b, and CAM_c for components; MAM_1,

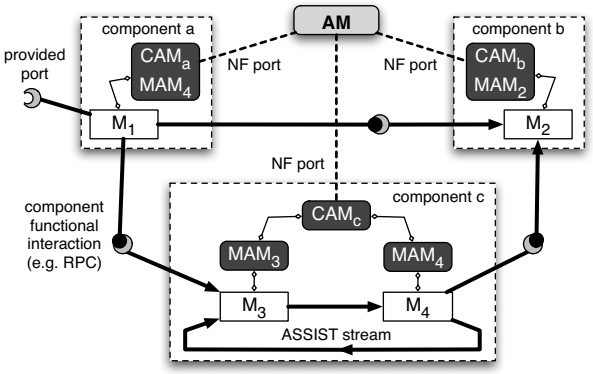


Figure 4. Four interacting ASSIST components.

MAM_2, MAM_3, and MAM_4 for ASSIST modules. Managers are arranged in a hierarchy whose root is the Application Manager coordinating the QoS of the whole application through CAMs and MAMs.

The ASSIST compiler automatically generates MAMs, CAMs, and AM, programmers are only requested to configure the management policy providing the code with a QoS contract. Also, ASSIST provides programmers an AM able to bind legacy (non ASSIST, e.g. CCM) components, provided they implement a suitable set of non-functional ports (e.g. monitor and adaptivity methods).

4. Self-Optimizing ASSIST components

In order to fully exploit Grid potentialities ASSIST rely on enhanced execution environment, which is designed (to a certain extent) to exploit an autonomic behavior (self-optimizing, at least). It should continually adapt the application to changes in Grid resources with the goal of sustaining the QoS requested for the application.

Each ASSIST component may be (statically or dynamically) set with a QoS specification, a.k.a. QoS contract. It can be specified for the *whole* application and/or for every *single* component. Currently, QoS contract is provided by a specific XML file and includes the specification of the processing bandwidth (service time) in stream-based computations, and/or the completion time, which is significant also for non-stream computations. Such service time may be constrained by the use of a maximum amount and of a given kind of processing nodes. The introduction of several other attributes is under way:

- the aggregate memory space available on a component, which is a key issue for components exploiting distributed storage;
- component protocols and their performances, which is needed to glue legacy components and to control their bandwidth;
- component fault-tolerance capability, which crucial for Grid executions.

As mentioned, each ASSIST Manager behaves as an *Autonomic Manager*. Module-AM (MAM) controlled elements are processes that implements the parmod. Component-AM (CAM) controlled elements are nested components. The application Manager (AM) is built out of all the components constituting the application. Clearly, the three kinds of managers may have different goals. All of them, anyway, contribute to satisfy the contracts, provided as XML files, according to a "best effort policy". As explained below, they initially allocate resources, then provide to cycle monitoring execution and possibly performing some performance model driven, corrective action.

4.1. Module Application Manager

The Module Application Manager (MAM) implements the configuration control of the single ASSIST parmods. Its main goal consists in to keep valid the module QoS contract, possibly by using assigned resources. Currently the performance contact can be set up dynamically by the father CAM, and may describe a mix of module service time and number/kind of resources.

The initial configuration of an ASSIST program is specified by the set of processes that are co-allocated at launch time. The configuration of a parmod is managed by its MAM, which dynamically decides the number of VPMs, and their mapping onto grid Processing Elements (PEs) acquired through Grid middleware. The ASSIST compiler prepares a *QoS contract* for each parmod and binds them to MAMs. Moreover, a MAM can asynchronously receive a different QoS contract from the CAM/AM in any moment along the application run.

Among all possible QoS goals, in this work we mainly focus on performance related ones that are achievable through adaptation within each parallel module. All aspects regarding modules coordination, as well as other QoS measures such as reliability, availability, and security are currently under investigation. We introduce the concept of QoS contract. It carries a module QoS goal and the description on how it should be achieved. In particular:

- *Performance features:* a set of variables, which can be evaluated from module static information, run-time data, collected through monitoring, and performance model evaluation.
- *Performance model:* a set of relations among *performance features* variables, some of them representing the performance goal.
- *Performance goal:* a set of inequalities involving *performance features*.

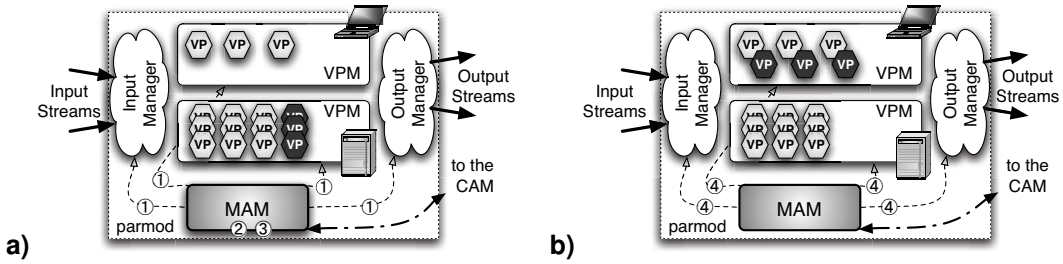


Figure 5. An ASSIST parmod before (a) and after (b) a reconfiguration. Dark VPs are migrated from one VPM to the other VPM under the control of the MAM. Along with VPs migration, some computational load is moved from one VPM to the other.

- *Deployment annotations* describing processes resource needs, such as required hardware (platform kind, memory and disk size, network configuration, etc.), required software (O.S., libraries, local services, etc.), and other all strictly required constraints to enforce code correctness.
- *Adaptation policy:* a reference to the desired adaptation policy chosen among the ones available for the module. Standard adaptation policies are represented as algorithms and embedded within MAM code at compile time.

In Section 5 an example of QoS is given. The performance models used in the ASSIST framework range from very simple and approximate analytical models, such as the one used to manage task farm parmods, to more complex models derived using advanced mathematical techniques, such as those derived in [24].

MAM main autonomic behavior consists in keeping the load balancing among module resources, despite the possible change of state/performance of underlying hardware/software resources. At this end, the MAM is equipped with a performance model that forecasts a sub-optimal mapping of VPs onto VPMs [5, 2]. The model uses VPs and VPMs historical performance data, and exploits the structured design of the parmod. MAM control loop is the following:

① *Monitor.* It collects VPMs execution times between two consecutive synchronization points that characterize module workload. These may be induced by explicit barriers (e.g. between loop cycles), or any event due to shared state synchronization or data distribution. The selection of suitable synchronization points is performed at compile time and guided by structured nature of parallelism exploitation in the parmod. It collects communication and synchronization performances.

② *Analyze.* Collected data is used to verify the MAM *performance goal*, update manager knowledge by building statistic and historical performance data. If the performance goal is broken, the possible causes are detected (e.g. load unbalance, not enough computing power or network bandwidth, insufficient input data rate).

③ *Plan.* If the performance goal is broken, a plan to reconvey the contract to a valid status is formed, i.e. a sequence of reconfiguration actions, each of them addressing a particular cause of performance degradation. Reconfiguration actions are chosen among legal ones for the particular instance of the parmod (e.g. add workers to a farm, migrate load between two VPs), and configured by using performance model instanced with data collected in previous stage (e.g. how many workers should be added to met a given service time, how much work should be moved from a VP to another). In the case no reconfiguration actions appear effective for the problem, an event is raised to the father CAM.

④ *Execute.* Depending of the previous outcome, it triggers VPs redistribution among VPMs by starting the suitable protocol, and it negotiates a resource upgrade with father CAM. The MAM can also receive an event by the father CAM indicating that it has to apply a restructuring strategy because a global variation of performance has been detected.

Note that many of the described features are really feasible due the high-level, structured nature of parallelism exploited in the ASSIST language. In particular, the pattern, frequency, and cost of communications among VPs can be derived from parmod declaration. They depend from parmod topology, data types and distribution. This information enables the definition of parametric performance models that can be instanced with monitored data (e.g. VPs completion time, communication bandwidth) to forecast expected performance. As an example, the performance gain of adding a worker in farm can be forecasted by extrapolating current performance to a scenario with more PEs.

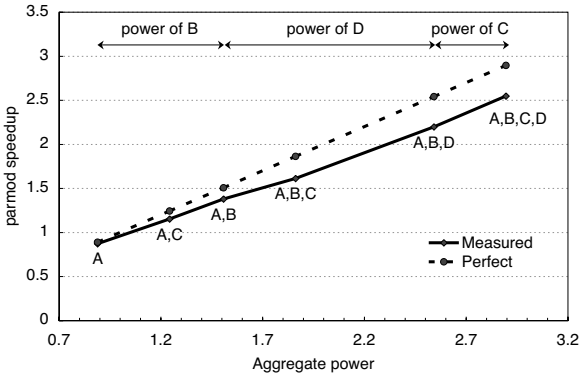


Figure 6. Speedup curve of a data-parallel application versus the aggregate power of tested environment.

4.2. Component Application Manager

Each Component Application Manager (CAM) applies control strategies *at a global level* for the associated component. Understanding the mapping from local behavior to global behavior is a necessary but insufficient condition for controlling autonomic systems. We must also exploit the inverse relationship. As indicated above, a CAM can receive proposals of restructuring by the child MAMs *(monitor)*. In this case, the CAM has to apply a global performance model in order to individuate a good solution to the restructuring of one or more of the children modules *(analyze & plan)*. Recursively, a CAM can receive reconfiguration requests from father CAMs, and can send them reconfiguration proposals *(execute)*. The root manager (AM) is the eventual responsible for the final decisions in the global reconfiguration control which, as seen, is a sort of parallel and asynchronous Divide&Conquer strategy applied along the hierarchical Application Manager structure.

The definition of a sound set of behavioral and interaction rules, that embedded in CAMs, will induce the desired global behavior is under investigation. As an example, a general strategy enabling to make sound decisions at the minimum possible level of the CAM hierarchy may significantly improve management overhead. At this end, the analysis of the application graph in terms of a queuing network seems a promising approach. This enables the detection of bottlenecks in application DAGs or sub-DAGs exploiting a data-flow behavior, i.e. ASSIST components mainly interacting via streams. Application of general graphs require a more sophisticated analysis because of the need to partition the graph in suitable sub graphs having both stable states and matching a manager that is as lower as possible in the manager hierarchy.

5. Experiments

The proposed AM organization and behavior, described in Section 3, has been evaluated and validated before actually starting its complete implementation. We integrated some ASSIST object code samples, produced by the ASSIST compiler, with code emulating the dynamic features of the run-time support and of the MAM/CAM hierarchical organization. Then we analyzed the efficacy and the peculiarities of the different implementation choices experimented. Later on we started migrating this whole experience in the actual ASSIST compiler/run time framework. The engineering of the MAM/CAM stuff in the production ASSIST compiler is still undergoing. This Section is about the results of the preliminary validation experiments performed to assess the MAM/CAM technology.

First of all, we performed a set of experiments aimed at evaluating the impact of the usage of heterogeneous processing node, such as the ones that can be recruited by the mangers in the process of adapting the execution of an application to the varying features sported by the target architecture processing elements. Good scalability can be achieved, as shown in Fig. 6. In the figure, the speedup of an application executed on a set of up to four different machines is shown. The speedup is measured in function of the aggregate power of the machines used to execute the program. Machines A, B, C and D differ in their relative power (CPU, clock speed, main memory). To have a rough idea of the relative power we used the BogoMIPS measure performed by the Linux kernel during the boot process. Normalized taking machine D as the unit power machine, machine A is rated at 0.8, machine B at 0.6 and machine C at less than 0.4 times the power of machine D. The measured speedup is definitely not far from the ideal one, also taking into account that BogoMIPS do not reflect latency and bandwidth in the network connections between the machines that also varied during the experiment.

Figure 7 shows the results achieved in a set of reconfiguration experiments. The experiments have been performed using an application whose structure was a pipeline of three stages: the first and the third stages are data servers and stream managers, and the second stage is a data parallel version of the finite difference method for solving differential equations (Jacobi method). Each stage is a parmod, wrapped in a component.

Figure 7 a) shows the effects of a perturbing overload caused by the creation of a new application onto the same processing nodes. In this case, we assume that no more processing nodes are available, thus only the load balancing solution is attempted, with a suitable redistribution of data partitions implemented directly by the run-time support. Fig. 7 b) shows a situation similar to Fig. 7 a): the difference is that more processing nodes are now available,

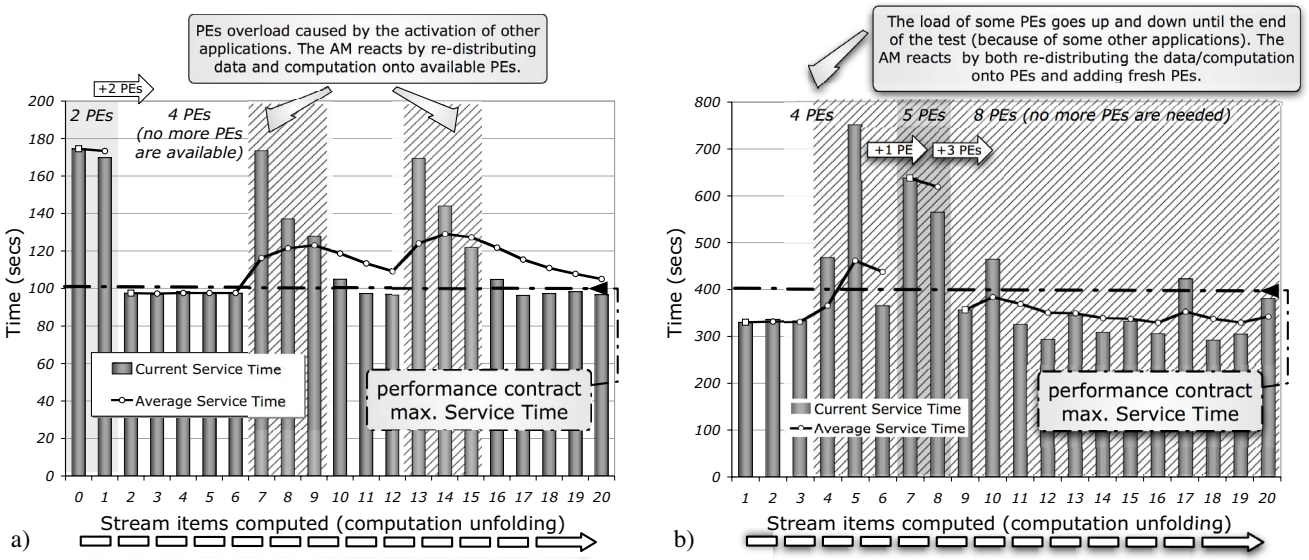


Figure 7. Experiments in dynamic restructuring of parallel ASSIST components.

and, after a first attempt of applying data redistribution, the degree of parallelism of the data parallel module is successfully increased.

Further experiments have been performed to better evaluate overhead related to manager activity. These experiments have been performed using an application using a parmod to model a plain task farm parallel pattern, and they have been performed on a dedicated Linux cluster. The cluster hosts 24 P3@800MHz PEs, connected through a 100MBit switched Ethernet. The architectural homogeneity and stability enable to precisely discriminate reconfiguration overhead, while do not affect general validity since the farm paradigm sports a self-balancing behavior through its on-demand scheduling [5]:

Fig. 8-❶ is relative to manager activity in *best effort mode*, that is, the manager was requested to arrange the better possible execution. In this case the number of tasks (i.e. stream items) per second per parmod is required to receive and compute (Input stream pressure)

Fig. 8-❷ is relative to manager activity in *goal based mode*, that is, the manager was asked to guarantee a fixed service time specified in the contract and with a fixed input pressure. Three times, along the program run, a PE is externally overloaded causing a contract violation. The MAM reacts by adding as many VPMs (one in the figure) mapped onto fresh PEs until the contract is satisfied. The MAM also knows that the contract continues to be satisfied if the overloaded PE is removed, and after a while removes it. On the whole a VPM migrates from one PE to another without stopping the parmod.

Experiment in Fig. 8-❷ is run with the following QoS contract:

Perf. features	QL_i (input queue level), QL_o (input queue level), T_{ISM} (ISM service time), T_{OSM} (OSM service time), N_w (number of VPMs), $T_w[i]$ (VPM$_i$ avg. service time), T_p (parmod avg. service time)
Perf. model	$T_p = \max\{T_{ISM}, \sum_{i=1}^{n} T_w[i]/n, T_{OSM}\}$
Goal	$T_p < K$
Deployment	arch = (i686-pc-linux-gnu $\vee$ powerpc-apple-darwin*)
Adapt. policy	goal_based

6. Related Works

The AFPAC library [6] proposes a similar approach to our in supporting application adaptability for MPI applications, however the reconfigurations are not transparent since the user code should be augmented with both synchronization points finalized to adaptability and reconfiguration code (ASSIST reconfiguration mechanisms are described in [5]). AFPAC does not support application development through assembly of components. Java bytecode portability has been exploited to provide a user-level migration mechanism (ProActive [9]), even if it is not transparent to the application programmer. ProActive include an implementation of the Fractal component model [19]. Both AFPAC and ProActive does not support automatic management of QoS.

We followed a similar approach to the GrADS project, which exploits a complete environment, including a monitoring architecture, contract negotiators and configuration

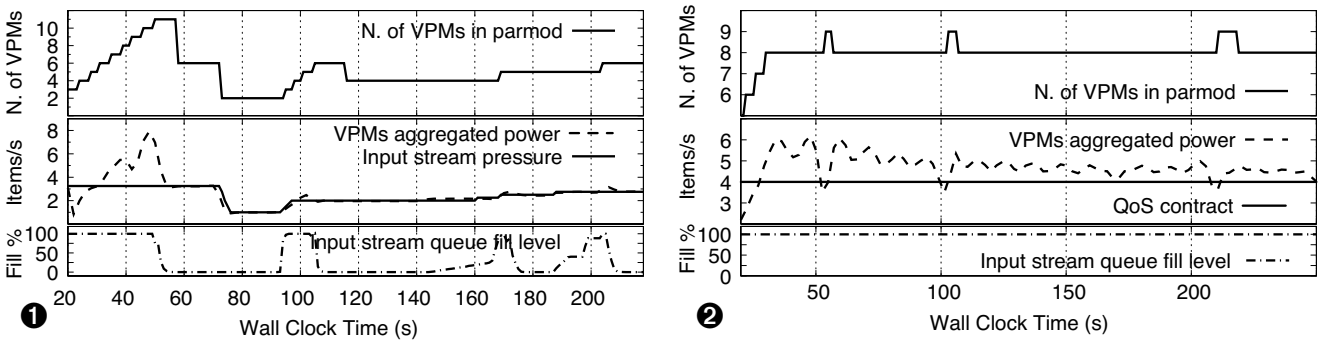


Figure 8. Experiments on farm reconfiguration.

optimizer. Differently from GrADS we can reconfigure applications in transparent manner, and with a sensibly better performance (we can join additional resources without completely stopping the application [5]). In particular [21], reports cost of minutes for reconfiguring a data-parallel application while ASSIST overheads ranges in milliseconds–seconds span. The lower reconfiguration cost diminishes the criticality of deciding a reconfiguration, and enables the use of heuristic "try-and-see" approach whether analytic modeling fails.

7. Concluding Remarks

We presented current status of ASSIST, a Grid-oriented structured parallel programming environment under development at the University of Pisa. The environment allows achieving quality of service in the execution of parallel programs by ensuring that a user defined performance contract is dynamically satisfied. This activity is completely in charge of a hierarchy of autonomic managers, which are automatically generated and configured by the ASSIST compiler. The programmer only needs to express the parallel program as a composition of components and to provide a performance contract. Components are declared by organizing sequential function according to a data or control parallel paradigm, and automatically generated by the compiler. The compiler also provides tools to integrate both legacy components and Web Services into an ASSIST application.

We discussed preliminary results on the ability of the ASSIST run-time to sustain a given QoS (service time), even whether available resources are artificially overloaded. The self-optimizing architecture of the ASSIST run-time, as well as the formalization of analysis and plan autonomic phases are topics of current research. The ASSIST programming environment is available under GPL open source license [8].

Acknowledgments. 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)

References

[1] 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 V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. Springer Verlag, Jan. 2005.

[2] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured implementation of component based grid programming environments. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer Verlag, Nov. 2005.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer Verlag, Jan. 2006.

[4] M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, and M. Vanneschi. Building interoperable grid-aware ASSIST applications via WebServices. In *Proc. of PARCO 2005: Parallel Computing*, Malaga, Spain, Sept. 2005.

[5] 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 2005: Parallel and Distributed Computing*, volume 3648 of *LNCS*, pages 771–781, Lisboa, Portugal, Aug. 2005. Springer Verlag.

[6] 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.

[7] A. Andrzejak, A. Reinefeld, F. Schintke, and T. Schütt. On adaptability in grid systems. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer-Verlag, Nov. 2005.

[8] ASSIST web site. http://www.di.unipi.it/Assist.html.

[9] F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for Grid programming. In V. Getov and T. Kielmann, editors, *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2005.

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

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

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

[13] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The Intl. Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.

[14] M. Govindaraju, S. Krishnan, A. S. K. Chiu, D. Gannon, and R. Bramley. XCAT 2.0: A component-based programming model for Grid web services. Technical Report TR562, Dept. of Computer Science, Indiana University, 2002.

[15] C. A. R. Hoare. Communicating Sequential Processes. *Communications of ACM*, 21(8):666–677, Aug. 1978.

[16] 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.

[17] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[18] S. Magini, P. Pesciullesi, and C. Zoccolo. Parallel software interoperability by means of CORBA in the ASSIST programming environment. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Proc. of the Euro-Par 2003*, volume 3149 of *Lecture Notes in Computer Science*, pages 679–688, Pisa, Italy, Aug. 2004. Springer.

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

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

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

[22] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency & Computation: Practice & Experience*, 17:1079–1107, 2005.

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

[24] C. Zoccolo. *High-performance component-based programming for heterogeneous computing*. PhD thesis, Dept. Computer Science, Univ. of Pisa, 2005.