Dynamic Reconfiguration of Grid-Aware Applications in ASSIST*

Marco Aldinucci[1], Alessandro Petrocelli[2], Edoardo Pistoletti[2],
Massimo Torquati[2], Marco Vanneschi[2], Luca Veraldi[2], and Corrado Zoccolo[2]

[1] Inst. of Information Science and Technologies – CNR, Via Moruzzi 1, Pisa, Italy
[2] Dept. of Computer Science – University of Pisa – Largo B. Pontecorvo 3, Pisa, Italy

Abstract. Current grid-aware applications are implemented on top of low-level libraries by developers who are experts on grid middleware architecture. This approach can hardly support the additional complexity of QoS control in real applications. We discuss a novel approach used in the ASSIST programming environment to implement/guarantee user provided QoS contracts in a transparent and effective way. Our approach is based on the implementation of automatic run-time reconfiguration of ASSIST application executions triggered by mismatch between the user provided QoS contract and the actual performance values achieved.

Keywords: Structured Parallel Programming, grid, QoS contract, Adaptive Applications

1 Introduction

A grid system is a geographically distributed collection of possibly parallel, interconnected processing elements that all run some kind of common grid middleware (e.g. Globus services). Such platforms are characterized by heterogeneity of nodes, and by dynamicity in resource management and allocation [1].

One popular approach to grid programming consists in directly exploiting middleware services within a standard programming language. This approach rapidly leads to an intolerable complexity as soon as the application is both complex and requested to exploit an user-defined QoS.

Large scale grid-aware application will require developing toolkits which support reconfigurable code and the binding of legacy code. They should enforce the minimization of developing cost by enabling the (static and dynamic) reconfiguration of the application to target different customer scenarios, while exploiting a good performance/cost ratio over a broad class of hardware platforms. They should also cope with code reuse providing the programmer with suitable bridges to interoperate with legacy code (Corba, CCM, Java Beans, DCOM, etc.). In this context, an advanced run-time support should seamlessly adapt the application structure to the current grid status, and transparently manage faulty

* This work has been supported by the Italian MIUR FIRB *Grid.it* project No. RBNE01KNFP, and Italian Project “legge 449/97” No. 02.00640.ST97.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 771–781, 2005.
© Springer-Verlag Berlin Heidelberg 2005

events and performance degradations of the underlying platform, which should be considered the standard behavior of a large-scale distributed platform.

High-level programming environments for grid aim at moving most of the grid specific efforts needed while developing high-performance grid applications from programmers to grid tools and run time systems. Among them, we mention ASSIST, GrADS, ProActive, Condor, Ibis [2–6]. In particular, ASSIST [2] has been designed along these guidelines. It supports the development of interoperable applications [7] onto heterogeneous platforms [8].

We present here a novel extension of the ASSIST environment exploiting a self-optimizing run-time targeted to fulfill QoS requirements, which are expressed at the language level by means of a *QoS contract*. QoS contract is transparently managed by the ASSIST compiler that preprocesses it and generates all the support code needed to enforce it at run-time by controlling parallelism degree, processes remapping, and algorithm selection. Programmer is just asked to express a composition of ASSIST modules, and declare a QoS contract either on each module or on the *whole* application. We experimentally show that both stateless and stateful computations may be suitably reconfigured to fulfill several kinds of contracts, and that these reconfigurations might have negligible cost enabling fine grain control on the application dynamic evolution.

ASSIST is briefly introduced in Sect. 2. QoS contracts and their managing strategy are presented in Sect. 3. ASSIST QoS-enabled run-time support is build on a set of *mechanisms* aiming to enable the run-time reconfiguration of applications. We believe that these mechanisms are an enabling technology for QoS control of grid-aware applications. Different *policies* may drive these mechanisms to target different QoS goals, such as performance and fault-tolerance. In this paper we mainly focus on these mechanisms, which are presented and evaluated in Sect. 4. Some examples of performance policies are sketched in Sect. 5, a full discussion on performance policies is outside the scope of this paper (due to lack of space). The presented policies are validated through experiments in Sect. 6.

2 The ASSIST Environment and Its Run-Time

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data. 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* on items arrival onto input streams. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran). Virtual Processes may synchronize with one other by barriers. Overall, a *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel (e.g. farm) way and may exploit a distributed shared state which survive to Virtual Processes lifespan. A module can non deterministically 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 Virtual Processes, according to

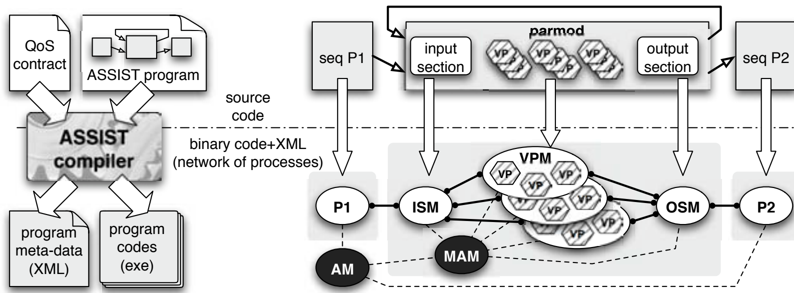


Fig. 1. An ASSIST application and a QoS contract is compiled in a set of executable codes and its meta-data [9]. These informations are used to set up a processes network at launch time: hexagons represents Virtual Processes, ovals represents processes, solid edges represent data channels, dashed edges managements channels.

the input and distribution rules specified in the parmod. Virtual Processes may send onto output streams items or parts of them which are gathered according to the output rules. More details on ASSIST environment can be found in [2, 9].

The ASSIST compiler translates a graph of modules into a network of processes. As sketched in Fig. 1, 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). The actual parallelism degree of a parmod instance is given by the number of VPMs. ASSIST run-time support also include a *Module Adaptation Manager* per parmod (MAM), which monitors the performances of the parmod, and implements reconfiguration policies; and an *Application Manager* (AM), which coordinates the QoS at the level of the whole application by coordinating MAMs. In this work we focus on MAM design. AM design is subject of current research, we refer back to Sect. 8 and [9] for a general description.

3 ASSIST Autonomic Run-Time and QoS Contracts

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 bind them to MAMs. Moreover, a MAM can asynchronously receive a different QoS contract from the 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, 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.
– *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.

The following is the QoS contract used in experiment Fig. 3 ❷:

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	$T_p = \max\{T_{ISM}, \sum_{i=1}^{n} T_w[i]/n, T_{OSM}\}$, $T_p < K$ (goal)
Deployment	arch = (i686-pc-linux-gnu ∨ powerpc-apple-darwin*)
Adapt. policy	goal_based

MAM run-time behavior may be conveniently sketched in terms of autonomic control loops [10]. In order to handle situations in which resource availability affects the performance of the applications, the run-time system of the running application has to:

1. **monitor** application actual status by collecting raw sensible performance data, and synthesize *performance features* variables;
2. evaluate *performance model*, and if it is unsatisfied, **analyze** it to discover possible causes;
3. if needed, **plan** a reconfiguration strategy according to the *adaptation policy*, with the goal of re-conveying the application in a legal status;
4. **execute** the reconfiguration, possibly allocating new resources/rebalancing the computation, possibly migrating entire modules.

4 Reconfiguration Key Concepts and Mechanisms

The modular nature of ASSIST applications and their management enable the reconfiguration of a subset of modules while neither affecting nor stopping the ones not involved in the reconfiguration, which can be distinguished in two categories: (a) involving the alteration of mapping between application activities and PEs[1]; (b) involving the variation of process graph structure, including modules

[1] ASSIST parmod supports the migration of Virtual Processes between VPMs, and VPMs between PEs, possibly migrating or remapping associated data.

parallelism degree[2]. Observe that, load balancing within a parmod can be managed by reconfigurations of kind (a): the load of a VPM (and the PE hosting it) may be decreased by moving some of its Virtual Processes to another VPM.

Independently of when the MAM decides to trigger a reconfiguration, the module is actually reconfigured on the next *reconf-safe* point. These are the time windows during a given parmod run in which its internal attributes are completely defined by the set of local attributes. Notably, the runtime does not introduce any additional synchronization w.r.t. the ones required by program semantics. It rather delays reconfiguration execution just after next natural reconf-safe point is reached. We distinguish between two kinds of reconf-safe points:

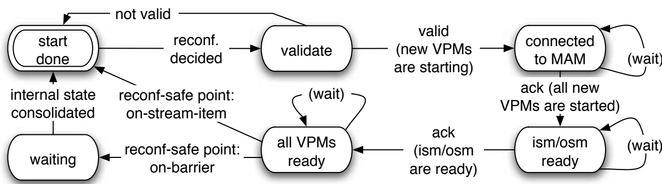
- *on-stream-item*: A new item is available in any input streams. A complete systolic synchronization is induced by the ISM process within the parmod. If needed, shared state is consolidated along the synchronization process.
- *on-barrier*: A complete synchronization has happened within a parmod due to either an explicit or implicit barrier. Barriers are issued by the programmer or the compiler to enforce the consolidation of shared state (e.g. at each step of a data-parallel iterative program).

No other points can be considered reconf-safe. Since the reconfiguration process is designed to be transparent to the programmer, we exclude the possibility of reconfiguring the parmod during the execution of an user defined function. In this way, we avoid the instrumentation of legacy code and the adoption of process dumping techniques that are hardly effective on heterogeneous platforms.

4.1 Reconfiguration Protocol

The MAM triggers a parmod reconfiguration raising a command toward all interested processes which participate, with the MAM, to a distributed reconfiguration protocol. All data exchanges (data or computation migrations) happen among VPMs following a communication schema encoded and optimized for the particular parmod semantics at compile time. These regard the static instrumentation of reconf-safe points with the minimum needed reconfiguration actions, e.g. a farm stateless parmod is not instrumented with data migration code.

The MAM participates to the protocol in order to mediate and orchestrate the interactions between AM and the Grid Abstract Machine, and to enforce all processes involved are aware and ready to start a reconfiguration at the next reconf-safe point. This should be enforced also when a complete barrier is not needed to ensure data integrity (e.g. master-slave). The latter property is guaranteed by the MAM accordingly to the following behavioral schema:



[2] ASSIST support an increment or decrement of the number of VPMs in parmods.

A parmod reconfiguration is initiated by its MAM. The reconfiguration plan is build accordingly to the proper strategy (see Sect. 5), e.g. increase the number of PEs, move to it a given number of Virtual Processes. Possibly some resources are asked to grid middleware through the Grid Abstract Machine. A reconfiguration command is synthesized, then validated (e.g. do not remove the last VPM, etc.).

The MAM waits in sequence that new started processes are connected to it; and the ISM, OSM, and all involved VPMs acknowledge the reconfiguration command. Eventually the MAM waits a reconf-safe point is reached and enforces all data and computation redistribution is completed.

4.2 Evaluation of Reconfiguration Overhead

We evaluate the cost of reconfiguration mechanisms against the following metrics (the former two are illustrated in Fig. 2):

- *Reconfiguration time* (R_t): refers to the total time spent to reconfigure the application, from the time a MAM decides the reconfiguration to the time it is completed.
- *Reconfiguration latency* (R_l): the time elapsed from the point a parmod is stopped for a reconfiguration to the time it is resumed. This is the foremost measure from users' viewpoint.
- *Reconfigurable code overhead* (R_o): the slowdown of an application when instrumented with the additional code needed to make it reconfigurable.

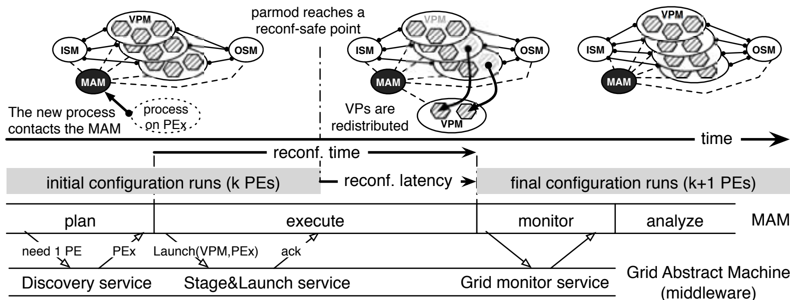


Fig. 2. Reconfiguration dynamics and metrics.

These metrics are evaluated on two different ASSIST applications 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 the reconfiguration overhead. As shown in [8], ASSIST already supports heterogeneous platforms in CPU and O.S. with less than 7% of additional communication cost, due to message marshalling. The reconfiguration mechanisms also support the deployment on to heterogeneous platforms with TCP/IP or Globus provided communication channels. The two applications are composed by one parmod and two sequential modules.

The first is a data-parallel application receiving a stream of integer arrays and computing a forall of simple function for each stream item; the matrix is stored in the parmod shared state. In this case the overall shared state has a negligible size since the experiment is designed to evaluate mechanism overhead. Preliminary experiments with realistic state size show a linear dependency between R_t and the global size of parmod shared state.

The second is a farm application computing a simple function on different stream items. Since R_t also depends on sequential function cost, in both cases we choose sequential functions with a close to zero computational cost in order to evaluate mechanism on the finest possible grain.

The reconfiguration overhead (R_o) measured during our experiments, without any reconfiguration change actually performed, is practically negligible, remaining under the limit of 0,004%, the measurement of the other two metrics are reported in Table 1.

Table 1. Evaluation of reconfiguration overheads (ms). On this cluster, 50 ms are needed to ping 200KB between two PEs, or to compute a 1M integer additions.

parmod kind	Data-parallel (with shared state)						Farm (without shared state)					
reconf. kind	add_PEs			remove_PEs			add_PEs			remove_PEs		
# of PEs involved	1→2	2→4	4→8	2→1	4→2	8→4	1→2	2→4	4→8	2→1	4→2	8→4
R_l on-barrier	1.2	1.6	2.3	0.8	1.4	3.7	–	–	–	–	–	–
R_l on-stream-item	4.7	12.0	33.9	3.9	6.5	19.1	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0
R_t	24.4	30.5	36.6	21.2	35.3	43.5	24.0	32.7	48.6	17.1	21.6	31.9

Consider the reconfiguration $x \to y$, where x and y are the number of PEs before and after the reconfiguration respectively. For the data-parallel parmod, R_l grows linearly with $(x + y)$ for both kinds of reconf-safe points, and depends on shared state size and mapping. Shared state is kept distributed during all the reconfiguration process. Farm parmod cannot be reconfigured on-barrier since it has no barriers, and achieves a negligible R_l (below 10^{-3} ms). This is due to the fact that no processes are stopped in the transition from one configuration to the next. R_t, which includes both the protocol cost and time to reach next reconf-safe point, grows linearly with $(x+y)$ for the former cost and depends on user-function for the latter.

5 Adaptation Policies

Adaptation policies are implemented as algorithms, actually methods of the MAM automatically generated by the compiler. ASSIST provides the programmer with hooks to add user-defined policies. The definition of a set of suitable policies and models to drive MAM and AM **analyze** and **plan** phases is subject of current research. For the sake of brevity, we present here a policy to automatically drive MAM of a farm-like parmod, adaptation policies for data-parallel stateful parmod have been presented elsewhere [11]. Farm parmod ex-

ploits on-demand task scheduling that guarantees load-balancing also in case of heterogeneous platforms, thus the MAM does not need to care about it. As discussed in Sect. 3, it is worth distinguishing two kinds of goals, and their adaptation policies. Ensure: (i) a *desired* service time; (ii) the *best effort* in the performance/resource trade-off.

In general, a policy should first analyze causes of module misbehavior reasoning on performance features values, then use the performance model to forecast if an adaptation may lead to contract satisfation. Eventually use mechanisms API (e.g. add_PEs) to reconfigure the module. Different policies can lead to different decisions in the same configuration: when the QoS contract is fulfilled, a policy of kind (i) would not increase the resource assigned to the module, even if it could exploit them. A best-effort policy in this case would pursue the maximum performance. As well, when incoming data rate decreases, so that some resources could be released because the module is over-dimensioned w.r.t. the input rate, a best-effort strategy will promptly release the resources, in order to optimize their usage, while a goal based policy would not, in the eventuality that the input rate will raise again.

When operating in best effort mode, the parmod acquires a new resource if the input queue is filling (its utilization is close to 1) and the output queue is emptying, i.e. the slower stage in the parmod is the processing one. It releases a certain amount of resources if exists a proper subset R of the set of VPMs that provides enough computing power:

$$B_{ISM} < \sum_{i \in R} B_w[i], \text{where} \quad B_{ISM} = \frac{1}{T_{ISM}}, B_w[i] = \frac{1}{T_w[i]}$$

in that case, the resources not in R can be released with no loss in performance.

When pursuing target (i), in the condition to release the resource, the actual bandwidth B_{ISM} is substituted by the contractually specified one: T_p. In this setting, the parmod acquires a new resource if the contract is not satisfied and the slower stage in the parmod is the processing one (as in the best-effort case); the conjunction of the two conditions prevents from adding new resources if the contract is not satisfied but due to other modules low performances.

These policies can clearly distinguish when a contract is violated due to another module misbehavior: in such case, the module manager is aware that no actions could be performed to solve the problem: we are investigating cooperation strategies among managers, to address these issues.

6 Experiments

To evaluate the effectiveness of proposed reconfiguration mechanisms and policies we tested a farm and a data-parallel parmods on several scenarios. The former parmod farms out a dummy sequential function with 2s average service time (experiments in Fig. 3 ❶, ❷, ❸). The latter computes a shortest-path like algorithm exploiting 640KB of shared state (Fig. 3 ❹). Tests are performed on the cluster described in Sect. 4.2, results are shown in Fig. 3:

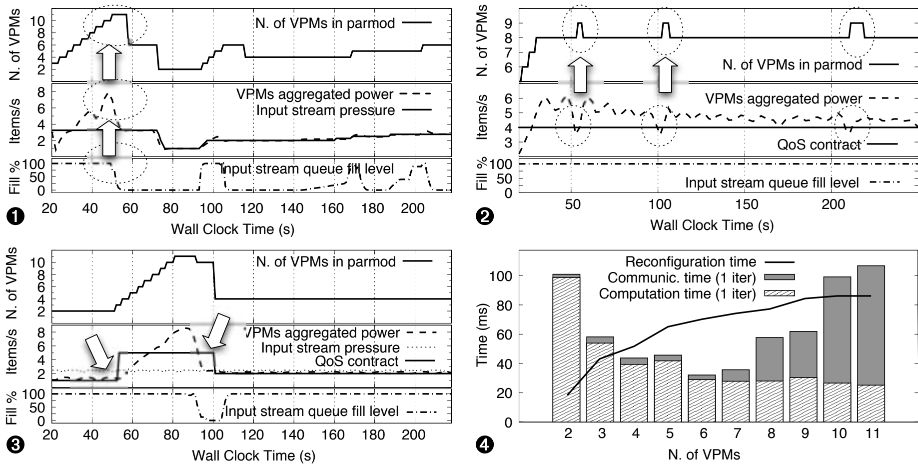


Fig. 3. Experiments on parmod reconfiguration (see Sect. 6).

❶ Farm: best effort mode: the Input stream pressure, i.e. the frequency at which parmod receive stream items, is changed along the program run. The parmod input queue tends to fill when the VPMs consume stream items slower than they are received, and vice-versa. The MAM tries to match the service time of VPMs and items arrival time by increasing or decreasing the number of VPMs. Transient VPMs may be exploited to bring back queue to a safe level.
❷ Farm: 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 (see Sect. 5) 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.
❸ Farm: a fixed service time specified in the contract and with a fixed input pressure, but the contract is changed by the AM three times along the program run. Each time, the MAM reacts by adapting the number of VPMs in order to satisfy the new contract.
❹ Data-parallel: on-barrier reconfiguration during the execution of a single forall. The MAM receives, during the program run, different contracts with fixed number of PEs (ranging from 2 to 11); it reacts by asking each time a fresh PE, mapping on it a VPM, and triggering the suitable computation and shared data redistribution. Observe that in this case the optimal number of PEs may heuristically be decided since the iteration time (computation time + communication time) exhibit a quite regular behavior. The figure also show that reconfiguration is quite efficient since its overhead is comparable to a single iteration time.

The experiments show that the approach is feasible for data-parallel computations, and that good results are obtained for the task-parallel ones.

7 Related Work

Early experiences of reconfigurable code have been presented since eighties; these include the management of process migration at O.S. kernel level [12], and libraries providing the programmer with a migration API for running processes (the libckpt [13], MPI-based DyRect [14]). With respect to them, ASSIST is able to target heterogeneous architectures, at a higher level of abstraction. The extensions of parallel programming languages (OpenMP [15], HPF [16]) proposed, are not enough flexible for a grid-like environment (e.g. they cannot acquire new PEs at run-time). The AFPAC library [17] proposes a similar approach to our in supporting reconf-safe points (AFPAC coordination protocol) for MPI applications, however the reconfigurations are not transparent since the user code should be augmented with both reconf-safe points and reconfiguration code. Java bytecode portability has been exploited to provide a user-level migration mechanism (ProActive [4]), even if it is not transparent to the application programmer.

We followed a similar approach to the GrADS project, which exploits a complete environment, including a monitoring architecture, contract negotiators and configuration 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). In particular [3], 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.

8 Conclusions and Future Work

We presented a novel extension of the ASSIST environment that seamlessly support application reconfiguration at run-time. Application reconfiguration is achieved efficiently and transparently to the application programmers through parmod reconfiguration. ASSIST parmod are self-optimizing parallel entities that can be able to respect a dynamically received QoS contract. A parmod reconfiguration does not have any direct impact on other parmods in the same application. Also, we shown a set of policies to deal with QoS contracts enabling parmods to self-adapt to a running environment that is heterogeneous and unreliable in provided performance.

On this ground we are extending ASSIST to full grid support. In particular, all MAMs can be organized in a hierarchy of managers, the root being the Application Manager (AM), that enforces a QoS contract for the whole application [9]. The AM works – in the large – similarly to MAM (see Sect. 3), and leverages on MAMs to detect parmods behaving as bottlenecks for the application: it reacts sending their MAMs a suitable new contract.

References

1. Foster, I., Kesselmann, C., eds.: The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann (2003)
2. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing **28** (2002) 1709–1732
3. Vadhiyar, S., Dongarra, J.: Self adaptability in grid computing. International Journal Computation and Currency: Practice and Experience (2005) To appear.
4. Baude, F., Caromel, D., Morel, M.: On hierarchical, parallel and distributed components for Grid programming. In: Workshop on component Models and Systems for Grid Applications. (2005)
5. Thain, D., Tannenbaum, T., Livny, M.: Condor and the grid. In: Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons Inc. (2002)
6. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: a flexible and efficient Java-based grid programming environment. Concurrency & Computation: Practice & Experience (2005)
7. Magini, S., Pesciullesi, P., Zoccolo, C.: Parallel software interoperability by means of CORBA in the ASSIST programming environment. In: Proc. of Euro-Par 2004. Volume 3149 of LNCS., Springer (2004) 679–688
8. Aldinucci, M., Campa, S., Coppola, M., Magini, S., Pesciullesi, P., Potiti, L., Ravazzolo, R., Torquati, M., Zoccolo, C.: Targeting heterogeneous architectures in ASSIST: experimental results. In: Proc. of Euro-Par 2004. Volume 3149 of LNCS., Springer (2004) 638–643
9. Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M., Zoccolo, C.: ASSIST as a research framework for high-performance Grid programming environments. In: Grid Computing: Software environments and Tools. Springer (2005)
10. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36** (2003) 41–50
11. Aldinucci, M., Campa, S., Coppola, M., Danelutto, M., Laforenza, D., Puppin, D., Scarponi, L., Vanneschi, M., Zoccolo, C.: Components for high performance Grid programming in Grid.it. In: Proc. of the Workshop on Component Models and Systems for Grid Applications. CoreGRID series. Springer (2005)
12. Zayas, E.R.: Attacking the process migration bottleneck. In: Proc. of the 11th ACM Symposium on Operating System Principles. (1987)
13. Litzkow, M.: Supporting checkpointing and process migration outside the unix kernel. In: Usenix Winter Conference. (1992)
14. E. Godard, S. Setia, E.W.: Dyrect: Software support for adaptive parallelism on nows. In: Proc. of IPDPS Workshop on Runtime Systems for Parallel Programming. (2000)
15. Scherer, A., Lui, H., Gross, T., Zwaenepoel, W.: Transparent adaptive parallelism on nows using OpenMP. In: Proc. of Principles and Practice of Parallel Programming. (1999)
16. Edjlali, G., Agrawal, G., Sussman, A., Humphries, J., Saltz, J.: Compiler and runtime support for programming in adaptive parallel environments scientific programming. Scientific Programming **6** (1997)
17. André, F., Buisson, J., Pazat, J.L.: Dynamic adaptation of parallel codes: toward self-adaptable components for the Grid. In: Workshop on component Models and Systems for Grid Applications. (2005)