ProActive Parallel Suite: From Active Objects-Skeletons-Components to Environment & Deployment

Denis Caromel and Mario Leyton

INRIA Sophia Antipolis, Université de Nice Sophia Antipolis, CNRS - I3S.
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.
First.Last@sophia.inria.fr

Abstract. The Proactive Parallel Suite offers multiple layers of abstraction for parallel and distributed applications which include both programming and the environment/deployment abstraction layers.
At the core of ProActive's programming abstractions are *active objects* with transparent futures and wait-by-necessity. Other abstractions offered by ProActive, such as *typed groups*, *algorithmic skeletons*, and *hierarchical distributed components* among others; are constructed on top of active objects. This pluralism of abstractions offers programmers a wide choice of expressiveness for coding parallel and distributed applications.
Additionally, an environment/deployment layer offers abstractions that simplify the interaction with the infrastructure. A deployment descriptor and a super-scheduler abstractions manage deployment of application on distributed resources, while the IC2D tool provides an abstraction to monitor debug and profile parallel and distributed applications.

1 Introduction

The relevance of parallel programming is evident. There has never been a point in time where we have had a dearer need for parallel programming abstractions to harness the power of increasingly complex parallel systems [40]. On one side large scale distributed-memory computing such as cluster and grid computing [29]; and on the other parallel shared-memory computing through new multi-core processors [7].

The difficulties of parallel programming have led to the development of many parallel programming models, each having its particular strengths. One thing which they have in common is that parallel programming models pursue a balance between abstractions (simplicity) and details (expressiveness). As applications increase in complexity, a single programming abstraction lacks expressiveness to adequately satisfy the whole application. Instead an approach where multiple abstractions are used for particular parts of the application is better suited. This papers describes a library providing such pluralism of programming models, the ProActive Parallel Suite.

The *ProActive Parallel Suite* is a 100% Java library, which aims at achieving seamless programming for concurrent, parallel, distributed, and mobile computing. It does not require any modification of the standard Java execution environment, nor does it make use of a special compiler, pre-processor, or modified virtual machine. Released under the GPL license, ProActive is a Java library for parallel, distributed, and concurrent computing; also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API which simplifies the programming of applications distributed on Local Area Networks (LAN), clusters, and Grids.

ProActive provides two levels of abstractions. First, a set of *programming model abstractions* such as: active objects, typed groups, algorithmic skeletons, distributed components, etc. The programming model abstractions are all implemented on top of the core programming abstraction, active objects, because they provide good properties such as determinism and orthogonality of future update policies [21]. Second, ProActive is also concerned with the complexity of deploying distributed applications. ProActive provides deployment descriptors which abstract low level information from the application source code. Users can deploy their applications on different infrastructures by providing the corresponding deployment descriptor, without changing the application code. Also, for more dynamic environments, ProActive supports batch like deployment of applications through an active object based scheduler. Additionally, ProActive provides a monitoring, debugging and profiling tool IC2D. Among others, IC2D provides a visual representation of an application's active objects and their communication.

This paper is organized as follows. Section 2 describes the related work. Section 3 describes the parallel programming abstractions in ProActive, starting with the active object model in Section 3.1, typed groups in Section 3.2, Calcium's algorithmic skeletons in Section 3.3, and the GCM hierarchical components in Section 3.4. Then Section 4 describes the Environment and Deployment abstractions. Finally Section 5 provides the conclusions and future work.

2 Related Work

ASSIST [2] is a programming environment which provides programmers with a structured coordination language. The coordination language can express parallel programs as an arbitrary graph of software modules. The graph describes how a set of modules interact with each other using a set of typed data streams. The modules can be sequential or parallel. Sequential modules can be written in C, C++, or Fortran; and parallel modules are programmed with a special ASSIST parallel module (*parmod*).

Condor [35] is a distributed computing system for batch processing. Condor provides job management, scheduling, resource monitoring and resource management. One of the key features of Condor is its matchmaking mechanism. Both jobs and resources describe their requirements using a ClassAd language, and the matchmaking determines if a resource is suitable for the execution of a job. Jobs can be ordered using DAGMan to define the dependencies between jobs, and a Master-Worker system is available for parameter search applications. Condor also monitors the job's progress and informs of completion to the user.

GAT stands for *Grid Application Toolkit* [6, 34], which defines a platform independent API to access resources and services. The API focuses, among others, on resource management (job submission and migration), and data management (file transfer, file access and communication pipes). A GAT Engine dispatches API calls to available services via *adaptors*. Adaptors are the interface between the GAT Engine and the third party services. They are analogous to providers in Java CoG. When a call to the API arrives, the GAT Engine executes suitable adaptors until one succeeds, or all fail, to perform the operation.

Globus Toolkit [30] is a rich set tools capable of interoperating to run applications on distributed and Grid infrastructures. These tools are concerned with deployment, data management, monitoring, and security among others. For deployment Globus provides GRAM which is the module responsible of the resource acquisition, configuration, executable staging, and program's execution. Data management is achieved by a set of tools, such as GridFTP used for data movement; but also others such as the Data Replication Service which handles replication of data.

Grid Superscalar [8] is an environment to program parallel applications for the Grid using imperative languages such as C++ and Perl. The program is specified as a set of tasks with input/output files in an interface description language. Grid Superscalar analyzes the dependencies between tasks and executes them sequentially or in parallel after having transferred the required data.

Java CoG Kit stands for *Java Commodity Grid Kit* [38], and provides services using simplified interfaces for lower level providers, in particular for the Globus Toolkit [31]. The Java CoG Kit's abstraction model follows a *provider* pattern, where abstract and generic concepts specified by programmers are translated into provider specific implementation entities. In the case of file transfer abstractions, file transfer operations are no different from other tasks, in the sense that a file transfer operation must be submitted for execution as a file-transfer-task [37, 39].

SAGA stands for Simple API for Grid Applications, and has the same objective as GAT: to construct a uniform API for the development of Grid applications [32]. Indeed, SAGA is an API standardization effort within the Open Grid Forum (OGF). The SAGA API is concerned with functional features such as job submission and management, file input/output, replica management, remote procedure calls, etc; and non-functional features such as permissions, security, monitoring, etc.

Unicore [28] is a middleware oriented towards application Grid services, where services are setup on a pre-configured Grid environment. Remote clients submit jobs to the Unicore's Grid gateway, which chooses suitable resources to run the jobs. A job is composed of one or more typed tasks. Each tasks triggers the execution of a predefined Grid service, in accordance with the type of the task. Tasks are arranged using a workflow, and can be executed in parallel or

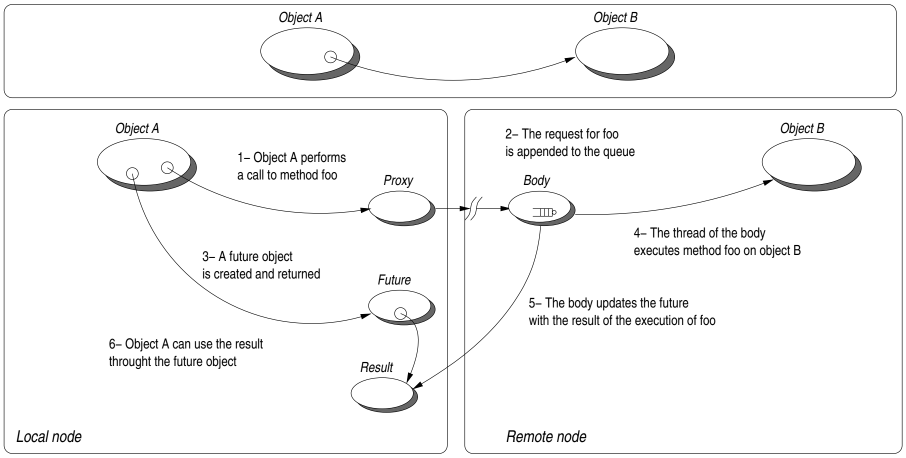


Fig. 1. Execution of a remote method call.

not. All tasks belonging to the same job share a jobspace file system. Besides the workflow, the job description also specifies which files must be imported into the jobspace before the execution of the job, and which files must be exported after the job is finished.

3 Parallel Programming Abstractions

ProActive provides several programming abstractions. This section describes only the following ones: active objects, typed groups, algorithmic skeletons, and components; which we believe provide a good overview of ProActive's pluralism of abstractions. Readers interested on some other specific programming model in ProActive, such as the Branch & Bound [19], Master-Slave, or Monte Carlo [16] APIs should refer to the ProActive documentation for further details [33].

3.1 Active Objects with Transparent Futures

At ProActive's core lies a uniform *active object* programming model abstraction. As shown in Figure 1, active objects are remotely accessible via method invocations, which are automatically stored in a queue of pending requests. Each active object has its own thread of control and is granted the ability to decide in which order the incoming method calls are served. Method calls on active objects are asynchronous with automatic synchronization. This is achieved using transparent *future objects* as a result of remote methods calls, and synchronization is handled by a mechanism known as *wait-by-necessity* [18].

Active objects are instantiated using the ProActive API, as shown in Listing 1.1, by specifying the class of the root object, the instantiation parameters, and

```
Object[] params= ...; //Constructor parameters
// instantiate active object of class B on a remote node
B b = (B) ProActive.newActive("B", params, node);

// use active object as any object of type B
R r = b.foo();
...

// possible wait-by-necessity
System.out.println(r.printResult());
```

Listing 1.1. Active Object instantiation and method invocation.

an optional location node. Invoking the method `foo()` on `b` returns a future of type `R`. Where `R` is the return type of the method foo, not a wrapper type. The computation can continue until a wait-by-necessity is reached. The thread accessing the future will be blocked only if the result is not yet available when it is actually required.

Active objects may migrate from any Java Virtual Machine (JVM) to any other using the provided *migration* mechanism. An active object with its pending requests (method calls), futures, and passive (mandatory non-shared) objects can migrate from JVM to JVM through the `migrateTo(...)` primitive. The migration can be initiated from outside the active object, but it is the responsibility of the active object to execute the migration, this is known as *weak migration*. Automatic and transparent forwarding of requests and replies provide location transparency, as remote references toward *active mobile objects* remain valid.

ProActive uses by default the RMI Java standard library as a portable communication layer, supporting the following communication protocols: `RMI`, `HTTP`, `Jini`, `RMI/SSH`, and `Ibis` [36].

3.2 Typed Groups

An extension of the active object abstraction corresponds to the *typed group communication* model [9]. Group communication is an important feature for high-performance and Grid computing, for which MPI is generally the only available coordination model [10]. Group communication allows triggering method calls on a distributed group of *active objects* with compatible type, dynamically generating a group of results. It has been shown in [9] that this group communication mechanism, plus a few synchronization operations (WaitAll, WaitOne, etc.), provides similar patterns for collective operations such as those available in MPI, but in a language centric approach [10].

The typed group communication mechanism [9] is built upon the ProActive elementary mechanism for asynchronous remote method invocation with automatic futures. The group mechanism must be thought of as a replication of more than one (say N) ProActive remote method invocations towards N active objects.

```
Object[][] paramsArray = {{...},{...},...};
Node[] nodes = {...,...,... };
A ag = (A) ProActiveGroup.newActiveGroup("A", paramsArray, nodes);
...
ag.foo(...); // A group communication

// A method call on a typed group
V vg = ag.bar();
// To wait and capture the first returned member of vg
V v = (V) ProActiveGroup.waitAndGetOne(vg);
// To wait all the members of vg are arrived
ProActiveGroup.waitAll(vg);
```

Listing 1.2. Typed Group Communications

Of course, the aim is to incorporate some optimizations into the group mechanism implementation, in such a way as to achieve better performances than a sequential achievement of N individual ProActive remote method calls. In this way, the mechanism is a generalization of the remote method call mechanism of ProActive.

The availability of such a group communication mechanism simplifies the programming of applications with similar activities running in parallel. Indeed, from the programming point of view, using a group of active objects of the same type, subsequently called a typed group, takes exactly the same form as using only one active object of this type. This is possible due to the fact that the ProActive library is built upon reification techniques.

Listing 1.2 shows an example using typed group communication. The creation of a group is analogous to the creation of an active object but using the `newActiveGroup` primitive. A group communication call is transparent and the result is stored in a future. The API allows for several utility methods like `waitAndGetOne` which waits for a single result from the group, and `waitAll` which waits for all results.

3.3 Algorithmic Skeletons

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [24]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones. All the parallelization and distribution aspects are implicitly defined by the composed skeletal structure.

As a skeleton framework we use Calcium [20, 22, 23], which is greatly inspired on Lithium [3–5, 27] and its successor Muskel [26]. Calcium is written in Java and is provided as a library. The Calcium framework is capable of evaluating the same skeleton program on different execution environments. Currently it supports parallel environments using threads, distributed environments using ProActive's active objects, and Grid like environments using the ProActive Scheduler (see Section 4.2).

In Calcium, skeletons are provided as a Java library. The library can nest task and data parallel skeleton in the following way:

$$\begin{aligned}\triangle ::= {} & \mathrm{seq}(f_e) \mid \mathrm{farm}(\triangle) \mid \mathrm{pipe}(\triangle_1, \triangle_2) \mid \mathrm{while}(f_b, \triangle) \mid \\ & \mathrm{if}(f_b, \triangle_{true}, \triangle_{false}) \mid \mathrm{for}(i, \triangle) \mid \mathrm{map}(f_d, \triangle, f_c) \mid \\ & \mathrm{fork}(f_d, \{\triangle_i\}, f_c) \mid \mathrm{d\&c}(f_d, f_b, \triangle, f_c)\end{aligned}$$

Each skeleton represents a different pattern of parallel computation. All the communication details are implicit for each pattern, hidden away from the programmer, and are classified in two types: task parallel or data parallel. The task parallel skeletons are: $farm$ for task replication; *pipe* for staged computation; *seq* for wrapping execution functions; if for conditional branching; and *while*/for for iteration. The data parallel skeletons are: *map* for single instruction multiple data; $fork$ for multiple instruction multiple data; and *d&c* for divide and conquer.

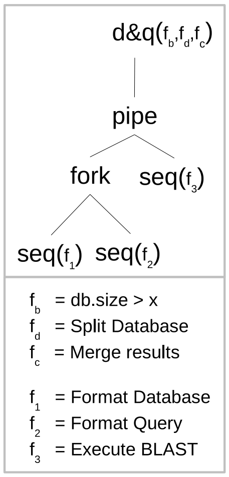
The nested skeleton pattern ($\triangle$) relies on sequential blocks of the application. These blocks provide the business logic and transform a general skeleton pattern into a specific application. We denominate these blocks *muscles*, as they provide the *real* (non-parallel) functionality of the application. In Calcium, muscles come in four flavors:

Execution	$f_e : P \rightarrow R$
Division	$f_d : P \rightarrow \{R\}$
Conquer	$f_c : \{P\} \rightarrow R$
Condition	$f_b : P \rightarrow \mathrm{boolean}$

Where P is the parameter type, R the result type, and $\{X\}$ a list of parameters or results of type X.

For the skeleton language, muscles are black boxes invoked during the computation of the skeleton program. Multiple muscles may be executed either sequentially or in parallel with respect to each other, in accordance with the defined $\triangle$. The result of a muscle is passed as a parameter to other muscle(s). When no further muscles need to be executed, the final result is delivered to the user.

Figure 2 shows an example. BLAST [15] corresponds to Basic Local Alignment Search Tool. It is a popular tool used in bioinformatics to perform sequence alignment of DNA and proteins. In short, BLAST reads a query file and performs an alignment of this query against a database file. The results of the alignment are then stored in an output file. A BLAST parallelization using skeleton programming is shown in the figure. The strategy is to divide the database until a suitable size is reached and then merge the results of the BLAST alignment.



```
//Initialization
Skeleton<BlastParams,File> blast = ...;
Environment env = new ProActiveEnv(...);
Calcium calcium = new Calcium(env);
Stream<BlastParams,File> stream =
        calcium.getStream(blast);

//Input Parameters
stream.input(new BlastParams("/home/query.1"));
stream.input(new BlastParams("/home/query.2"));

//...

//Output Results
File alignment1 = stream.getResult();
File alignment2 = stream.getResult();
```

Fig. 2. BLAST Skeleton Program

The code shown in the figure represents the usage API. An initialization phase defines the skeleton program (portrayed graphically in the example), and then instantiates Calcium with a specific environment. The skeleton program is then associated with a stream which is used to input parameters and collect the results.

3.4 Grid Component Model (GCM)

The Grid Component Model (GCM) [25] abstraction extends Fractal [17] for distributed and Grid computing [12]. As in Fractal, GCM allows for hierarchical composition, separation of functional and non-functional interfaces; but also considers deployment, collective communications [14], and autonomic behavior [1] among others.

ProActive's GCM implementation is built upon the active object model. Each component is implemented with an active object and (non-)functional requests are served from the active object queue. Invocations on component interfaces inherit asynchronism from the remote method calls of active objects. The result of an invocation is also a transparent future which can be passed to other components.

Figure 3 shows graphical example of a GCM component. External interfaces provide interaction with the environment while internal interfaces are binded with sub-components. The interface on the sides represent server (left) and client (right), while interfaces on the top correspond to non-functional services: life cycle, reconfiguration, etc.

Among others, GCM extends Fractal by providing multicast and gathercast interfaces [14]. Multicast interfaces provide abstractions for one-to-many communications, by transforming a single invocation into a list of invocations. The

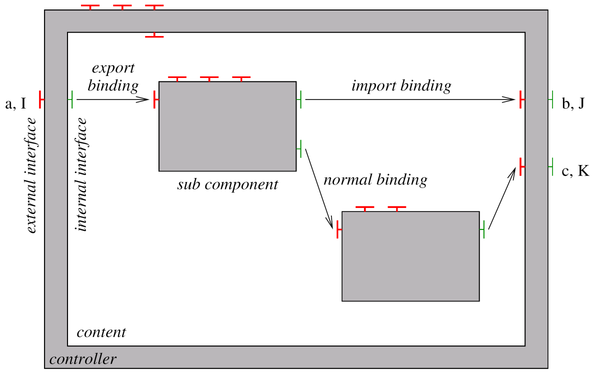


Fig. 3. Fractal based Grid Component Model

transformation is customizable (broadcast, split, etc), and the result of such invocation is a list of result or its reduction. The symmetrical interfaces are gathercast which provide abstractions for many-to-one communications by transforming a list of invocations into a single invocation. Gathercast interfaces can coordinate the invocation which is automatically redistributed to the invoking components.

4 Environment and Deployment Abstractions

4.1 Deployment and Scheduling

Descriptor Based The deployment of distributed applications is commonly done manually through the use of remote shells for launching the various virtual machines or daemons on remote computers and clusters. In heterogeneous infrastructure the deployment complexity increases thus making the deployment task central and harder to perform by the application.

To address this issue, ProActive provides a *deployment descriptor* abstraction [11], which allows the deployment of applications on heterogeneous sites without changing the application's source code. All infrastructure information related with the deployment of applications on the infrastructure is described in a *deployment descriptor* (XML). Thus, eliminating references inside the code to: machine names, resource acquisition protocols (local, rsh, ssh, lsf, globusgram, unicore, pbs, lsf, nordugrid-arc, etc...), and communication/lookup protocols (rmi, jini, http, etc...).

The deployment descriptor's architecture is shown in Figure 4. The infrastructure section contains the information necessary for acquiring remote resources. Once acquired, ProActive *nodes* are instantiated on the remote resources. The nodes are then linked with the application code via a `virtual-node` abstraction. In the application's code, a `virtual-node` name corresponds to a reference on the nodes that will be acquired during the deployment. While, on

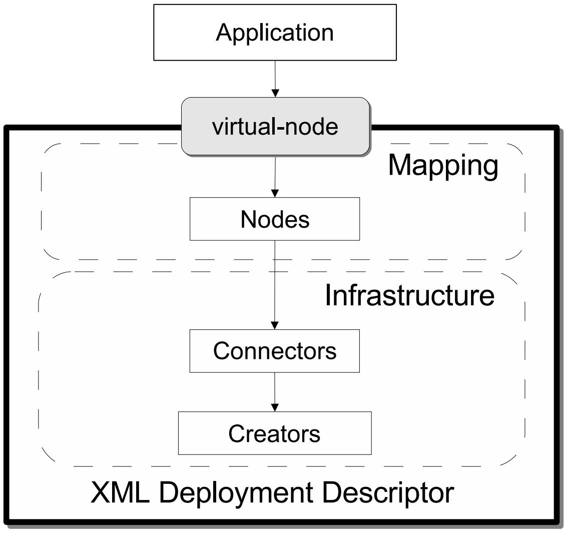


Fig. 4. Deployment Descriptor Layout

the deployment descriptor, the `virtual-node` corresponds to a set of deployment operations that will yield resources with instantiated nodes.

Consequently, the parsing of the deployment descriptor, and associated deployment operations, are triggered from the application code by calling one single method of the ProActive library. The deployment can be configured by changing the

application → `virtual-node` → nodes

mapping, to run the application on a different infrastructure, without modifying a single line of code in the application.

4.2 Scheduler

Batch schedulers provide an abstraction of resources to clients. Clients submit tasks and the scheduler is in charge of executing them on available resources. Thus, a scheduler allows several clients to share a same pool of resources. In this section, we present a super-scheduler (*scheduler* for short), capable of federating other schedulers. Clients can interact with the scheduler through different mechanism: command-line, API, GUI, and description files. In addition to the super-scheduler, we describe a *resource manager*, which is in charge of acquiring and managing resources.

The Scheduler is the central entity with which clients interact using a remote Java API, or by submitting a `Job` Description. A `Job` describes the batch process to be executed. The description specifies the code, which can be in Java by extending the `Executable` interface or any native executable; required data files; and a script for validating resources.

Currently three kinds of jobs are supported: in **Task Flow** Jobs, clients describe the flow and dependencies of tasks to execute; in **Parameter Sweeping** a single task is executed in parallel with multiple data; and in **ProActive Applications** clients submit a regular ProActive distributed application.

The scheduler also supports customized allocation policies, and provides a FIFO policy by default. Basic non-functional concerns such as security and fault-tolerance are handled both at the ProActive middleware and scheduler levels.

Finally, the management, deployment, and selection of resources is handled by a second entity, named the *resource manager*. Figure 5 shows a global overview of the whole system.

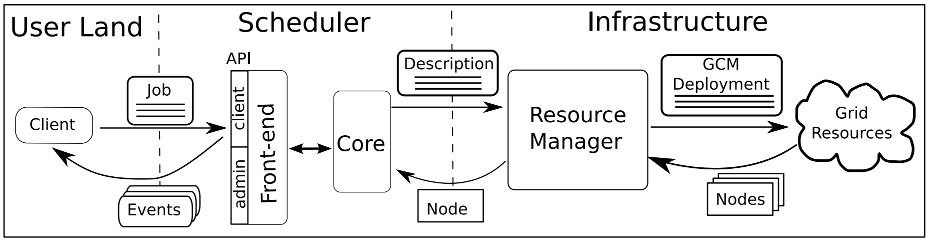


Fig. 5. Scheduler Global Overview

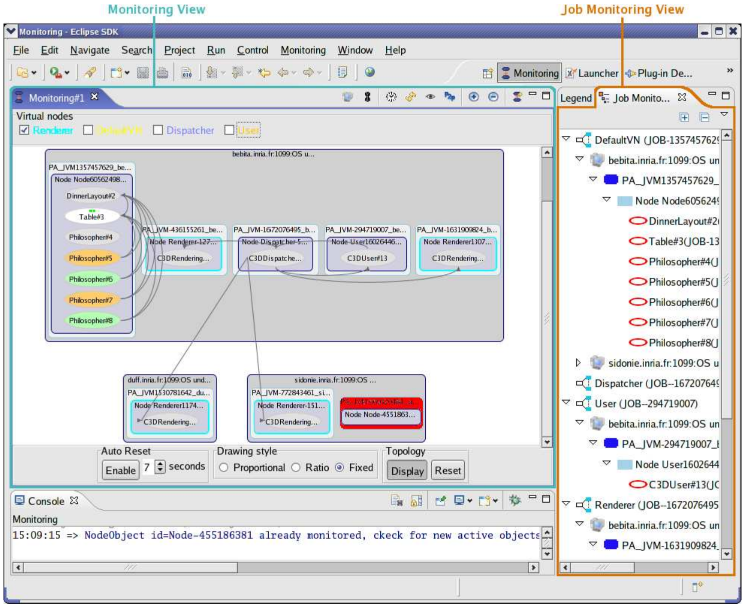
The Resource Manager (RM) is responsible for acquiring and managing resources. The RM is built on top of the deployment descriptors which provide an abstraction of how resources can be acquired.

The static acquisition of resources is handled by deployment descriptors, while the dynamic management of these resources is done by the RM. A scheduler can ask resources from a RM, and the RM will deliver a resource through a node abstraction. Once the scheduler no longer requires a node, it is returned to RM for cleaning, pooling or releasing.

The scheduler can also request specific resources, that fulfill some requirements in order to execute a particular task. Requirements can be verified with a script attached to the task, the RM uses this script to test resources. A successful execution of this script on a given resource validates the node.

4.3 IC2D Monitoring, Debugging, and Profiling

Graphical visualization and monitoring of any ongoing ProActive applications is possible with *IC2D* (Interactive Control and Debugging of Distribution) tool [13]. IC2D provides a graphical representation of hosts, java virtual machines,



Virtual nodes: ☑ Renderer □ DefaultVN □ Dispatcher □ User

Auto Reset: Enable 7 seconds

Drawing style: ○ Proportional ○ Ratio ● Fixed

Topology: Display Reset

```
Monitoring
15:09:15 => NodeObject id=Node-455186381 already monitored, ckeck for new active objects
```

Fig. 6. IC2D Example Snapshot

nodes, active objects with their queue of requests, and messages as shown in Figure 6. In the figure, outermost squares represent hosts while inner squares correspond to java virtual machines and nodes. The ellipsis correspond to active objects and the queue of pending requests is represented by dots inside active objects. Communications between active objects are shown by lines. Additionally, IC2D allows the monitoring of migrations, which can also be triggered through IC2D with a drag-and-drop.

When interfaced with Timit a profile of the application can be generated. The profile contains information such as time spent sending/waiting for requests, a timeline of activity for each active object, memory usage, and thread usage among others.

5 Conclusions

The ProActive Parallel Suite offers a variety of abstractions to ease the programming and execution of parallel and distributed applications. The programming abstraction layer is based on an active object model with transparent first class futures and wait-by-necessity. On top of this programming model, other abstractions are provided such as typed groups, Calcium's algorithmic skeletons, and GCM components among others. ProActive's programming models pluralism allows programmers to choose the most adequate abstractions for their application.

ProActive also provides an environment/deployment abstraction. The deployment process is simplified with deployment descriptors or a scheduler for more dynamic environments. Additionally, the IC2D tool provides abstract to monitor, debug, and profile parallel and distributed applications.

The current and future work of ProActive has many directions. For example the deployment mechanism is currently undergoing a complete re-write and extension. The new deployment labeled *GCM Deployment*, will harness all the experience gathered in the deployment of ProActive applications. Besides a simplified an easier description of infrastructure resources, the GCM Deployment considers an application side descriptor.

At the programming abstraction level, for active objects we are currently working on providing advanced update policies. For GCM components we are currently working on better MPI like collective communications, integration with webservices, tailored monitoring, reconfiguration, and compositional non-functional aspects.

References

1. Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Peter Kilpatrick, Patrizio Dazzi, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons in gcm: Autonomic management of grid components. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 54–63, Washington, DC, USA, 2008. IEEE Computer Society.
2. Marco Aldinucci, Massimo Coppola, Marco Danelutto, Nicola Tonellotto, Marco Vanneschi, and Corrado Zoccolo. High level grid programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32, 2006.
3. Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED, ACTA press.
4. Marco Aldinucci, Marco Danelutto, and Jan Dünnweber. Optimization techniques for implementing parallel skeletons in grid environments. In S. Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, Stirling, Scotland, UK, July 2004. Universität Münster, Germany.
5. Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.
6. Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, Andre Merzky, Rob V. van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schott, Ed Seidel, and Brygg Ullmer. The grid application toolkit: Towards generic and easy application programming interfaces for the grid. In *Proceedings of the IEEE*, volume 93, pages 534–550, March 2005.
7. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel

computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

8. Rosa M. Badia, Jesus Labarta, Raul Sirvent, Josep M. Perez, Jose M. Cela, and Rogeli Grima. Programming grid applications with grid superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.
9. Laurent Baduel, Françoise Baude, and Denis Caromel. Efficient, Flexible, and Typed Group Communications in Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press. ISBN 1-58113-559-8.
10. Laurent Baduel, Françoise Baude, and Denis Caromel. Object-Oriented SPMD. In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, May 2005.
11. F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.
12. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, pages 1226–1242, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.
13. Francoise Baude, Alexandre Bergel, Denis Caromel, Fabrice Huet, Olivier Nano, and Julien Vayssière. IC2D: Interactive Control & Debug of Distribution. Grappes 2001, `http://www.univ-ubs.fr/valoria/grappes2001`, May 2001. Invited talk.
14. Francoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective interfaces for distributed components. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 599–610, Washington, DC, USA, 2007. IEEE Computer Society.
15. BLAST. Basic local alignment search tool. http://www.ncbi.nlm.nih.gov/blast/.
16. Mireille Bossy, Françoise Baude, Viet Dung Doan, Abhijeet Gaikwad, and Ian Stokes-Rees. Parallel pricing algorithms for multi–dimensional bermudan/american options using monte carlo methods. *CoRR*, abs/0805.1827, 2008.
17. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software, Practice & Experience*, 36(11-12):1257–1284, 2006.
18. Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
19. Denis Caromel, Alexandre di Costanzo, Laurent Baduel, and Satoshi Matsuoka. Grid'BnB: A Parallel Branch & Bound Framework for Grids. In *Proceedings of the HiPC'07 - International Conference on High Performance Computing*, 2007.
20. Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 45–53, Toulouse, France, February 2008. IEEE CS Press.
21. Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.
22. Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-Par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 2007.
23. Denis Caromel and Mario Leyton. A transparent non-invasive file data model for algorithmic skeletons. In *22nd International Parallel and Distributed Process-*

ing Symposium (IPDPS), pages 1–8, Miami, USA, March 2008. IEEE Computer Society.

24. Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, Cambridge, MA, USA, 1991.
25. CoreGRID, Programming Model Institute. Basic features of the grid component model (assessed). Technical report, 2006. Deliverable D.PM.04, http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf.
26. Marco Danelutto and Patrizio Dazzi. Joint structured/unstructured parallelism exploitation in Muskel. In *Proc. of ICCS 2006 / PAPP 2006*, LNCS. Springer Verlag, May 2006. to appear.
27. Marco Danelutto and Paolo Teti. Lithium: A structured parallel programming enviroment in Java. In *Proc. of ICCS: International Conference on Computational Science*, volume 2330 of *LNCS*, pages 844–853. Springer Verlag, April 2002.
28. Dietmar W. Erwin and David F. Snelling. Unicore: A grid computing environment. In *Lecture Notes in Computer Science*, volume 2150, pages 825–834. Springer, 2001.
29. Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.
30. Ian T. Foster. Globus toolkit version 4: Software for service-oriented systems. In Hai Jin, Daniel A. Reed, and Wenbin Jiang, editors, *NPC*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.
31. Globus. http://www.globus.org.
32. Hartmut Kaiser, Andre Merzky, Stephan Hirmer, Gabrielle Allen, and Edward Seidel. The saga c++ reference implementation: a milestone toward new high-level grid applications. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 184, New York, NY, USA, 2006. ACM.
33. ProActive. http://proactive.objectweb.org.
34. E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. Gridlab: A grid application toolkit and testbed. *Future Generation Computer Systems*, 18:1143–1153, 2002.
35. Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. *Concurrency and Computation: Practice & Experience*, 17(2-4):323–356, 2005.
36. R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency - Practice and Experience*, 17(7-8):1079–1107, 2005.
37. Gregor von Laszewski, Beulah Alunkal, Jarek Gawor, Ravi Madhuri, Pawel Plaszczak, and Xian-He Sun. A File Transfer Component for Grids. In H.R. Arabnia and Youngson Mun, editors, *Proceedings of the International Conferenece on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 24–30. CSREA Press, 2003.
38. Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, /2001.
39. Gregor von Laszewski, Jarek Gawor, Pawel Plaszczak, Mike Hategan, Kaizar Amin, Ravi Madduri, and Scott Gose. An overview of grid file transfer patterns and their implementation in the java cog kit. *Neural, Parallel Sci. Comput.*, 12(3):329–352, 2004.
40. Katherine Yelick. Keynote: Programming models for petascale to exascale. In *22nd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–1, Miami, USA, March 2008. IEEE Computer Society.