Performance and Scalability of a Component-Based Grid Application

Nikos Parlavantzas[1], Matthieu Morel[2], Vladimir Getov[1], Françoise Baude[2], Denis Caromel[2]

[1]*Harrow School of Computer Science, University of Westminster, HA1 3TP, U.K. {N.Parlavantzas, V.S.Getov} @westminster.ac.uk*

[2]*INRIA Sophia Antipolis, 2004, route des Lucioles, BP 93, F-06902 Sophia Antipolis Cedex, France FirstName.LastName@inria.fr*

Abstract

Component-based development has emerged as an effective approach to building flexible systems, but there is little experience in applying this approach to Grid programming. This paper presents our experience with reengineering a high performance numerical solver to become a component-based Grid application. The adopted component model is an extension of the generic Fractal model that specifically targets grid environments. The paper provides qualitative and quantitative evidence that componentisation has improved the modifiability and reusability of the application while not significantly affecting performance.

1. Introduction

As Grid technologies are becoming widely available, managing the complexity of building and evolving Grid applications is becoming increasingly important. Component-based development has emerged as an effective approach to building complex software systems; its benefits include reduced development costs through reusing off-the-self components and increased adaptability through adding, removing, or replacing components. Naturally, applying component-based development to Grid programming is currently attracting much interest. Examples of component models applicable to this field include CCA (Common Component Architecture) [9], CCM (Corba Component model) [17], and the emerging GCM (Grid Gomponent Model) [10], currently under development within the CoreGRID European project. Despite this growing interest, there is still little experience in applying components to Grid computing, and developers are not provided with adequate guidance and support.

The main aim of this work is to present our experience with applying component-based development to the domain of high performance scientific applications running on the Grid. Specifically, the work describes how a numerical solver, originally implemented as distributed object application, was reengineered into a component-based application. The adopted component model extends the generic Fractal model [7], similarly to the GCM. The model is implemented on top of the ProActive middleware [19]. We show that componentisation has increased the modifiability of the application without any significant negative effects on performance.

The rest of this paper is structured as follows. Section 2 provides background on the numerical application, called Jem3D, and the distributed object platform on which it is built. Section 3 presents our approach to reengineering this application, which comprises a general componentisation process and a Grid-enabled component model. Section 4 then describes our componentisation experience and the resulting system. Section 5 provides some performance results, and section 6 discusses related work. Finally, section 7 concludes the paper.

2. Background

This section provides background on Jem3D, the application at the focus of this paper, and the ProActive library, the distributed object platform used by Jem3D.

2.1. Jem3D overview

Jem3D is a numerical solver for the 3D Maxwell's equations modelling the time domain propagation of electromagnetic waves [4]. It relies on a finite volume approximation method operating on unstructured tetrahedral meshes. At each time step, the method evaluates flux balances as the combination of elementary fluxes computed through the four facets of a tetrahedron. The complexity of the calculation can be changed by modifying the number of tetrahedra in the domain. This is done through setting the *mesh size*; i.e., the triplet (m1×m2×m3) that specifies the number of points on the x, y, and z axes used for building the mesh. Parallelisation relies on dividing the computational domain into a number of subdomains; the domain division is controlled by another triplet (d1×d2×d3) that determines the number of subdomains on each axis. Since some facets are located on the boundary between subdomains, neighbouring subdomains must communicate to compute the values of those border facets. The original Jem3D builds on the ProActive library, outlined next.

2.2. The ProActive library

The ProActive library is a Java middleware for parallel, distributed, and concurrent programming [19]. The ProActive core supports a uniform programming model based on remotely accessible *active objects*. Each active object has its own thread of control and decides in which order to serve incoming method calls. Remote method calls on active objects are asynchronous with automatic synchronization.

Two key features of ProActive are its support for typed group communication and descriptor-based deployment. *Group communication* enables triggering method calls on a group of active objects with compatible type, dynamically generating a group of results. Invoking a group of active objects takes exactly the same form as invoking one active object, which simplifies the programming of applications with similar activities running in parallel. Moreover, group invocations incorporate optimisations that make them more efficient than sequentially invoking a set of objects. *Descriptor-based deployment* enables deploying distributed applications anywhere without having to modify the source code. References to hosts, protocols and other infrastructure details are removed from the application code, and specified in XML descriptor files.

2.3. Jem3D architecture

Figure 1 shows the runtime structure of the original Jem3D (a 2×2×1 domain division is assumed); the main elements of the architecture are outlined next.

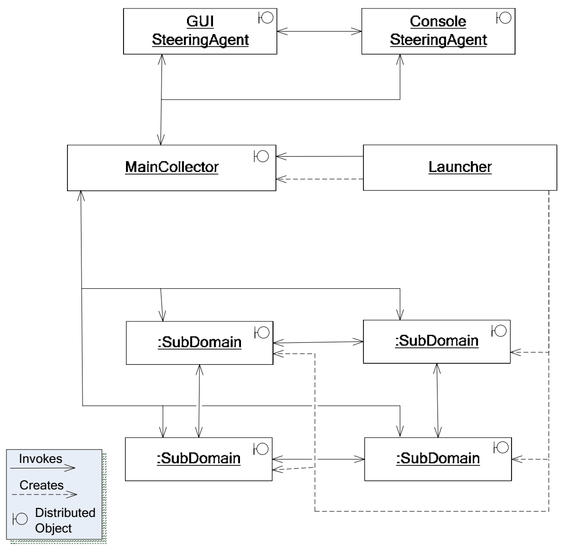

Figure 1. Jem3D Architecture

Subdomains correspond to partitions of the 3D computational domain; they perform electromagnetic computations and communicate with their closest neighbours in the 3D grid. Moreover, they send partial solutions with a predefined frequency to the main collector. The *main collector* is responsible for monitoring and steering the computation by interacting with the subdomains. The monitoring and steering functionality is used by one or more *steering agents*, which are dynamically registered with the main collector. The application includes a command-line agent and a graphical agent with visualisation capabilities. Steering agents communicate with each other to ensure that only a single agent at a time has the right to control the computation. Finally, the *launcher* is responsible for obtaining the input data, creating the main collector and the subdomains, setting up the necessary connections between them, initialising them with the necessary information, and starting the computation. Communication between the entities relies on the asynchronous remote invocation and group communication mechanisms provided by ProActive.

The original Jem3D application suffers from limited modifiability and limited reusability of its parts. This can be largely attributed to two factors. First, the application lacks reliable architectural documentation, which is essential for understanding and evolving complex software systems. Jem3D has been subjected to successive changes by multiple people without corresponding updates to the architectural information. Second, the application parts are tightly coupled together. Indeed, as in most object-oriented applications, the code includes hard-wired dependencies to classes, which limits the reusability of classes, increases the impact of changes, and inhibits run-time variability. For example, changing the subdomain implementation requires updating the source code of both the main collector and the launcher and rebuilding the whole application. As another example, although the Jem3D parallelisation follows a typical *geometric decomposition pattern* [15], no part of the application can be reused in other contexts where this pattern is applicable. To address such modifiability and reusability limitations, Jem3D was re-engineered into a component-based system.

3. Approach

This section presents our approach for addressing the modifiability and reusability limitations of Jem3D. The approach consists of a general componentisation process and the use of the Fractal/ProActive component technology, discussed in the following two sections.

3.1. Componentisation process

The purpose of the componentisation process is to transform an object-based system to a component-based system. The process assumes that the target component platform allows connecting components via provided and required interfaces, and that it minimally supports the same communication styles as the object platform (e.g., remote method invocation, streams, and events). Figure 2 shows the main activities and artefacts defined by the componentisation process. The activities are summarised next (more details can be found in [18] .

Recover Original Architecture

This activity uses as input the source code, documentation, build files, and any other software artefacts and produces an architectural description of the original system. At a minimum, the description must include a run-time view of the architecture containing executing entities, communication paths, and interactions over those paths.

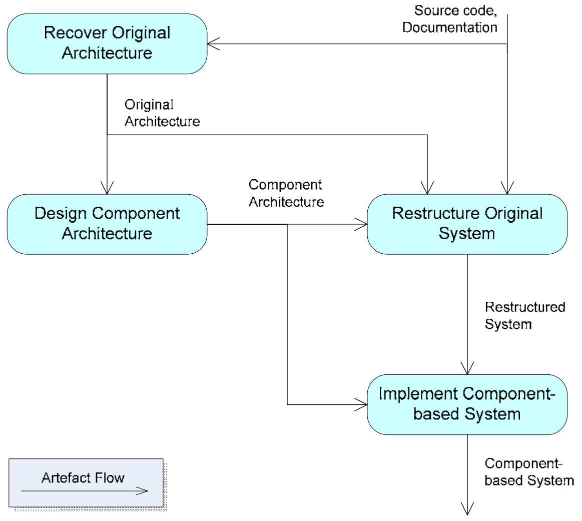
Source code, Documentation
Recover Original Architecture
Original Architecture
Design Component Architecture
Component Architecture
Restructure Original System
Restructured System
Implement Component-based System
Component-based System
Artefact Flow

Figure 2. Componentisation process

Design Component Architecture

This activity produces the target component architecture. It uses the executing entities of the original architecture as candidate components to form an initial component architecture. This initial architecture is then refined to address modifiability and performance concerns and to exploit the available features provided by the target component model (e.g., hierarchical composition in Fractal).

Restructure Original System

This activity restructures the original code to make it match closely the target component architecture, while avoiding any dependencies on the target component platform. Specifically, the activity involves implementing and testing an interface-based version of the system in which entities communicate as much as possible via explicitly identified provided/required interfaces. The motivation for the activity is to validate a large part of the target architecture at an earlier time. Moreover, the activity makes the migration to the component platform easier than it would otherwise be.

Implement Component-based system

This activity implements and tests the new component-based system. It uses as inputs the component architecture and the restructured, interface-based version. It typically involves minor changes for repackaging classes as component implementations.

3.2. Fractal/ProActive

Fractal/ProActive is a parallel and distributed component model that specifically targets Grid applications [5]. Fractal/ProActive conforms to the generic Fractal model [7] and extends it with a number of features that support Grid programming. Fractal/ProActive is implemented on top of the ProActive library [19]. Fractal and the Fractal/ProActive-defined extensions are examined in turn next.

Fractal components are runtime entities that communicate exclusively through interfaces of two types: *client interfaces* that emit operation invocations and *server interfaces* that accept them. Interfaces are connected through communication paths, called *bindings*. Fractal distinguishes *primitive* components from *composite* components formed by hierarchically assembling other components (called sub-components). This hierarchical composition is a key Fractal feature that helps managing the complexity of understanding and developing component systems. Another important Fractal feature is its support for extensible reflective facilities. Specifically, each component exposes an extensible set of *controller interfaces* for inspecting and reconfiguring internal features of the component. (e.g., for modifying the set of sub-components). Finally, Fractal includes an architecture description language (ADL) for specifying configurations comprising components, their composition relationships, and their bindings.

The Fractal/ProActive model extends Fractal in the following ways. Primitive components are specialised to obtain the properties of remotely accessible active objects. Composite components can contain multiple active objects and can be distributed over different machines. Component communication relies on asynchronous method invocations. A multicast communication style is also supported, analogous to the group communication mechanism in ProActive. Specifically, the model defines a specialisation of Fractal interfaces, called *multicast interfaces that* enable treating a set of invocations as a single invocation. As with standard interfaces, multicast interfaces can have a client or server type. Finally, the component model supports configurable component deployment based on the deployment descriptors provided by ProActive.

4. Componentising Jem3D

Jem3D was componentised using the approach presented earlier. Most of the effort was spent on the architecture recovery activity because of the undocumented and degraded structure of the system. The run-time view of the original architecture was described using UML object diagrams—such as the one in Figure 1—and UML interaction diagrams. During the component architecture design, the launcher entity (an executing Java program) was decomposed into a *subdomain factory* component and an *activator* component; the former is assigned the responsibilities for creating, initialising, and connecting the subdomains, while the latter the responsibilities for obtaining the input data, passing them to the factory, and starting the computation. The reason for the decomposition was to make the factory reusable beyond Jem3D. A later iteration of the activity grouped the factory and the subdomains into a composite *domain* component, exploiting the hierarchical composition feature of Fractal/ProActive. Implementing the interface-based version served to increase confidence in the new component architecture and drastically simplified the final component-based implementation. The component-based implementation involved wrapping classes to form Fractal components and replacing a large part of the injector logic with Fractal ADL descriptions, as seen next.

Figure 3 shows the static structure of the resulting component-based Jem3D using a UML component diagram (multicast interfaces are represented as stereotyped UML interfaces with special notation). The runtime configuration consists of multiple subdomains, logically arranged in a 3D mesh, with each subdomain connected to its neighbours via multicast interfaces. The runtime configuration also includes a dynamically varying number of steering agents. The main collector is connected to the current set of agents via a multicast interface. A multicast interface is also used to connect each agent to all other agents.

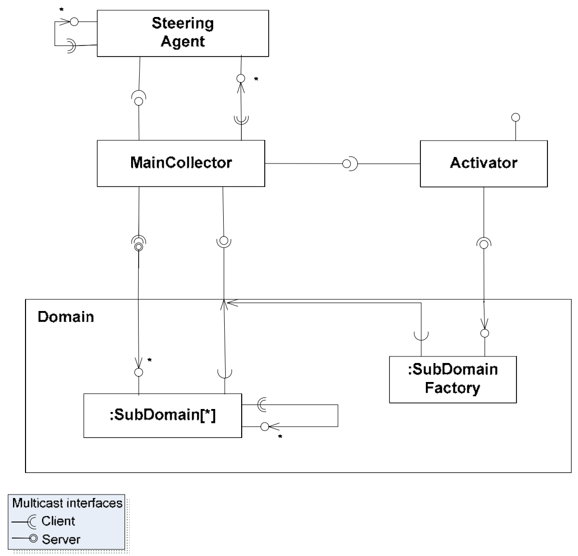

Figure 3. Component-based Jem3D structure

The initial configuration of Jem3D is described using the Fractal ADL, as seen in Figure 4 (pseudo code is used for brevity). Note that the ADL is not used to express the configuration of subdomains, which depends on the dynamically-determined domain division. Since allowable configurations follow a fixed, canonical structure in the form of a 3D mesh, a parameterised description would be useful for automatically generating subdomain configurations. However, the Fractal ADL includes currently no variability mechanisms for expressing such descriptions. The ADL does include a simple parameterisation mechanism, which is used to configure the factory with the required subdomain implementation.

```
Component ConsoleSteeringAgent
        definition = SteeringAgentImpl
Component MainCollector
        definition = MainCollectorImpl
Component Activator
        definition = ActivatorImpl
Component Domain
        Interface …   // interfaces omitted
        Component SubDomainFactory
          Definition=FactoryImpl (SubDomainImpl)

        // bindings within composite
        // interfaces names omitted
        Binding This to SubDomainFactory
        Binding SubDomainFactory to This

// bindings among top-level components
// interface names omitted
Binding ConsoleSteeringAgent to MainCollector
Binding MainCollector to ConsoleSteeringAgent
Binding Activator to MainCollector
Binding Activator to Domain
Binding MainCollector to Domain
Binding Domain to MainCollector
```

Figure 4. Initial configuration in the ADL

Evaluation

We now examine whether the new, component-based Jem3D addresses the modifiability and reusability limitations of the original system. Owning to the componentisation process, the new system has gained reliable architectural documentation, which facilitates understanding and evolving the system. Moreover, an important part of the architecture—i.e., the initial component configuration—is captured in the ADL. As a result, the component platform can automatically enforce architectural structure on implementation, which helps reduce future architectural erosion. The use of provided and required interfaces as specified by the component model minimizes inflexible, hard-wired dependencies and allows flexible configuration after development time. Considering the scenario of changing the subdomain implementation, this can now be achieved simply by replacing a name in the ADL description (i.e., the SubDomainImpl name in Figure 4). Moreover, the domain component now serves as a reusable unit of functionality that supports the geometric decomposition pattern. Specifically, the component accepts as input the subdomain implementation and the domain division and embodies the logic to create and manage the runtime subdomain configuration.

5. Performance results

Comparison between object-based and component-based versions

We first deployed the application on a single cluster so that measurements could be realized in a stable and homogeneous environment, and so that comparison would be possible. We used a fixed mesh size of 121*121*121. The mesh was sufficiently small so that the application could be deployed on a reduced number of nodes and sufficiently large so that communication time did not exceed computation time. The deployment of the Jem3D application proceeds as follows: first, the collector is instantiated on a single node. Second, a set of virtual machines is created according to the deployment descriptor, and using the standard cluster scheduling protocols. Third, active objects are instantiated on the virtual machines. For the component version, once the components are instantiated (as active objects), there are also an assembly and a binding phase to create the system dependencies. We measured the initialization time as the time between the creation of all remote virtual machines, and the beginning of the computation. We also measured the computation time for a fixed number of iterations with the Jem3d application. The benchmarks took place on one of the clusters in INRIA Sophia-Antipolis, with machines equipped with Opteron processors at 2GHz and 2GB of RAM, and connected through Gigabit Ethernet connections. The JVMs were deployed with an allocated heap size of 1500MB. The results are presented in Figure 5.

We observe that:

- The computation times are similar for the component and the object-based version, which means that there is no significant overhead induced by the component framework during the computation.

- The deployment time (referred to as initialization time in the figure) is a little longer for the component-based version. This is due to a more elaborate deployment process that not only creates component instances, but also assembles them and binds them. The overhead for deployment seems very much acceptable.
- With a mesh size of 121*121*121, and with an available heap size of 1500MB for each computing entity, the computation needs to be distributed on a minimum of 15 machines so that the mesh data can be loaded in memory (the mesh is divided among participants: the higher the number of participants, the smaller the size of the mesh for each participant).

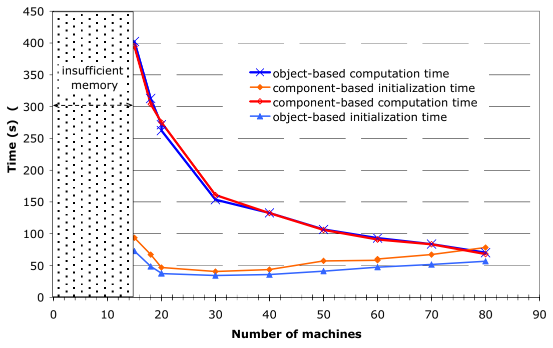

Figure 5. Comparison of execution times

Grid scalability

We used the experimental French grid infrastructure Grid'5000 [8] for performance measurements using several Grid'5000 clusters. The objective was to evaluate the scalability of the component-based version.

We ran several experiments, increasing the mesh size and the number of machines used. One sub-domain component or object is deployed on each node. We report the results in Table 1, also describing the set-up of the different experiments. The parameters that varied were the mesh size and the distribution over the different clusters (a possible distribution is shown in Figure 6). It is important to state that Jem3D does not offer control over the distribution of the computation entities (the sub-domains). All sub-domain entities are deployed on a unique virtual node, which is later mapped onto the physical infrastructure. Using several virtual nodes would allow control over the virtual distribution, hence possibly control over the physical one, however this was not possible without completely changing the design of Jem3D. As a consequence, some highly communicating neighbours may be located on separate clusters, in which case there is an induced latency overhead in their communications. In the Grid'5000 infrastructure, which uses dedicated and optimized networks, the latency between machines of a cluster is about 0.05ms, while the latency between clusters can be up to 10ms (this is the case between clusters in Sophia-Antipolis and clusters in Rennes): the latency is up to 200 times higher for inter-cluster than intra-cluster communications. Measuring computation time in this context is very difficult for a tightly-coupled application because of both lack of control over deployment and the inherent instability of Grids. This is why we preferred to present the results from a few experiments, without drawing any conclusion on the performance in this context.
The results of the experiments as reported in Table 1 demonstrate the scalability of the component framework: we managed to deploy and run a component-based version of the Jem3D application on more than 300 processors and up to 4 remote clusters. We were also able to compute with bigger meshes when increasing the number of machines.

Discussion

From our experience with the deployment and benchmarking of the Jem3D application, we can draw the following conclusions:

- Componentisation has no adverse impact on the performance of the Jem3D application. Specifically, the component framework does not induce any overhead during computation, and the initialization is only slightly longer than for the object-based version. We also demonstrated that the framework is scalable.
- Computational benchmarks for tightly coupled and highly communicating applications need to be performed on homogeneous environments, such as a single cluster. Otherwise performance measurements are unreadable, because inter-cluster communications are several orders of magnitude longer than intra-cluster ones, and because of the inherent instability of Grids: the more different administrative domains involved, the higher the chances of some local dysfunction.
- An application, to take advantage of a computational Grid, must provide a *partitioning method at design time*, which at runtime creates partitions depending on the application parameters and the runtime infrastructure. A partition identifies tightly-coupled entities which must be co-located, while the coupling between partitions is looser. Partitions can be attached to virtual nodes, which are mapped on separate deployment infrastructures at deployment time, if

needed, resulting in an efficient distribution of the application. The component-based approach provides a convenient way to design suitable partitions for both loosely-coupled and tightly-coupled applications.

- Grids, by providing large computational infrastructures, allow new categories of problems to be solved [11]. For instance, the Jem3D application can solve problems with mesh sizes over 200*200*200, which is impossible on a single cluster with machine equipped of 2GB RAM, because of memory problems.

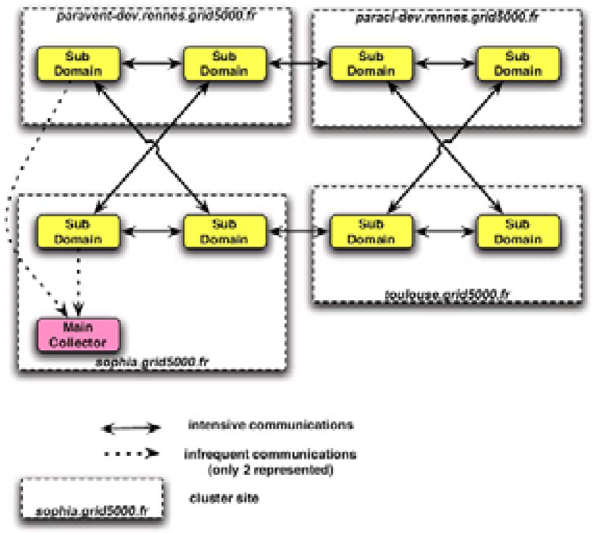
paravent-dev.rennes.grid5000.fr
paraci-dev.rennes.grid5000.fr
Sub Domain
Main Collector
sophia.grid5000.fr
toulouse.grid5000.fr
intensive communications
infrequent communications (only 2 represented)
cluster site

Figure 6. Possible distribution of Jem3D over Grid'5000 clusters

Table 1. Jem3D experiments

Mesh size	Total number of processors	Computation time (min)	Processors at Sophia Antipolis	Processors at Orsay	Processors at Rennes-1	Processors at Rennes-2	Processors at Rennes-3	Processors at Toulouse
41*41*41	20	0.46	0	0	0	0	20	0
81*81*81	70	0.94	20	10	20	20	0	0
201*201*201	130	5.15	70	0	60	0	0	0
201*201*201	138	3.85	138	0	0	0	0	0
241*241*241	258	4.29	138	0	120	0	0	0
241*241*241	308	3.72	138	0	120	0	0	50

6. Related work

As mentioned earlier, there is little experience in applying component-based development to Grid computing. Most related work to ours is that associated with CCA [9]. CCA is a component model for high-performance scientific computing that has been applied to a large range of application domains [6]. CCA components are dynamically connected through *provides* and *uses* ports. The main difference with Fractal is that CCA lacks hierarchical composition as a first-class part of the model. Ccaffeine [2] is an implementation of CCA that supports parallel computing. Ccaffeine-based components interact within a given process using CCA ports; parallel instances of Ccaffeine-based components interact across different processes using a separate programming model, typically MPI. XCAT3 [12] is another CCA implementation that supports components distributed over different address spaces and accessible as collections of Grid services compliant to OGSI (Open Grid Services Infrastructure). In [16], CCA/Ccaffeine is used to componentise simulation software for partial differential equations. Components are produced by creating thin wrappers over existing numerical libraries. A simple process for converting such libraries to components is presented in [3]; the process involves first grouping provided and used library functions to provide and uses CCA ports, and then deciding how ports are associated to components.

Beyond grid computing, several researchers have reported experiences with componentising large software systems. [13] describes the componentisation of operating system software for MPSoC (multi-processor system on chip) platforms. Componentisation relies on a lightweight Fractal implementation that targets embedded systems software. Other case studies have concentrated on componentising programmable controller software [14] and real-time telecommunication software [1]. Such work provides evidence of the positive effect of componentisation on modifiability but does not focus on the componentisation process.

7. Conclusion

This paper has presented a case study in reengineering a scientific application into a component-based, grid-enabled application built on Proactive/Fractal. The transformation from an object-based to a component-based system has followed a general componentisation process, reusable in other contexts. The paper has provided qualitative evidence that componentisation using Fractal/ProActive is beneficial to the modifiability and reusability of the application. The paper has also provided quantitative evidence that componentisation has no adverse effect on performance.

There are two main directions for future work. First, we plan to apply the componentisation process and the Fractal/ProActive component technology to other applications in diverse domains. Such work will enable a more complete assessment of their usefulness and usability, and generate further suggestions for improvement. Second, we plan to add support for dynamic reconfiguration in the component-based Jem3D application in order to accommodate variations in the availability of underlying resources. Supporting reconfiguration will involve the introduction of manager components that build on the reconfiguration primitives already provided by the component model (e.g., connect or disconnect components), without requiring any change to existing code.

Acknowledgement

This research work was carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] H. Algestam, M. Offesson, L. Lundberg, "Using Components to Increase Maintainability in a Large Telecommunication System", Ninth Asia-Pacific Software Engineering Conference (APSEC'02), p. 65, 2002.

[2] B.A. Allan, R.C. Armstrong, A.P. Wolfe, J. Ray, D.E. Bernholdt, and J.A. Kohl, "The CCA core specifications in a distributed memory SPMD framework", *Concurrency Comput. Pract. Exp.*, vol. 14(5), 323-345, 2002.

[3] B.A. Allan, S. Lefantzi, J.Ray, "ODEPACK++: Refactoring the LSODE Fortran Library for Use in the CCA High Performance Component Software Architecture", Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04), 109-119, 2004.

[4] L. Baduel, F. Baude, D. Caromel, C. Delbe, S. Kasmi, N. Gama, and S. Lanteri, "A Parallel Object-Oriented Application for 3D Electromagnetism", 18th International Parallel and Distributed Processing Symposium, IEEE Computer Society, Santa Fe, New Mexico, USA, April 2004.

[5] F. Baude, D. Caromel, and M. Morel, "From distributed objects to hierarchical grid components", In International Symposium on Distributed Objects and Applications (DOA), LNCS 2888, 1226-1242, Springer-Verlag, 2003.

[6] D.E. Bernholdt, B.A. Allan, R. Armstrong, F. Bertrand, K. Chiu, et al., "A Component Architecture for High Performance Scientific Computing", ACTS Collection special issue, *Intl. J. High-Perf. Computing Applications,* 20, 2006.

[7] E. Bruneton, T. Coupaye, and J. B. Stefani, "Recursive and dynamic software composition with sharing", In Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP2002), 2002.

[8] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, et al, "Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform", 6th IEEE/ACM International Workshop on Grid Computing, Grid'2005, Seattle, Washington, USA, Nov. 2005.

[9] CCA Forum Home Page, The Common Component Architecture Forum, 2004. http://www.cca-forum.org.

[10] Grid Component Model (GCM) Proposal, CoreGRID Deliverable, D.PM.002, Nov. 2005.

[11] A. Hoekstra and P. Sloot, "Introducing Grid Speedup Γ: A Scalability Metric for Parallel Applications on the Grid", Proc. of EGC 2005, LNCS 3470, 245–254, Springer-Verlag, 2005.

[12] S. Krishnan and D. Gannon. "XCAT3: A Framework for CCA Components as OGSA Services", 9th Intl Workshop on High-Level Parallel Programming Models and Supportive Environments, IEEE CS Press, 2004.

[13] O. Layaida, A.E. Özcan, and J.B. Stefani. "A Component-based Approach for MPSoC SW Design: Experience with OS Customization for H.264 Decoding", 3rd Workshop on Embedded Systems for Real-Time Multimedia under CODES+ISSS, New York, USA, 2005.

[14] F. Lüders, I. Crnkovic, P. Runeson, "Adopting a Component-Based Software Architecture for an Industrial Control System – A Case Study, Component-Based Software Development for Embedded Systems", LNCS 3778, 232-248, Springer-Verlag, 2005.

[15] B.L. Massingill, T.G. Mattson, and B.A. Sanders, "Patterns for parallel application programs", In Proceedings of the Sixth Pattern Languages of Programs Workshop (PLoP99), 1999.

[16] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes and B. Smith, "Parallel components for PDEs and optimization: some issues and experiences", *Parallel Computing*, vol. 28(12), 1811-1831, Dec. 2002.

[17] Object Management Group, CORBA Component Model v3.0, OMG Document formal/2002-06-65.

[18] N. Parlavantzas, V. Getov, M. Morel, F. Baude, F. Huet, D. Caromel, "Componentising a Scientific Application for the Grid", Proc. 2nd Annual CoreGrid Integration Workshop, 225-236, October 2006, Krakow, Poland.

[19] ProActive web site, ttp://www.inria.fr/oasis/ProActive/