Distributed multiscale computing

Borgdorff, J.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Applying distributed multiscale computing to in-stent restenosis

Abstract

Nature is observed at all scales; with multiscale modelling, scientists bring together several scales for a holistic analysis of a phenomenon. The models on these different scales may require significant but also heterogeneous computational resources, creating the need for distributed multiscale computing. A par-

\[1\] The contents of this chapter are based on:
particularly demanding type of multiscale models, cyclically coupled models, brings with it a number of theoretical and practical issues. In this chapter, a cyclically coupled model of in-stent restenosis is first examined for its multiscale merits using the Multiscale Modelling Language (MML); this is aided by a toolchain consisting of MAPPER Memory (MaMe), the Multiscale Application Designer (MAD), and GridSpace Experiment Workbench. It is implemented and executed with the general Multiscale Coupling Library and Environment (MUSCLE). Finally, it is scheduled on heterogeneous infrastructures using the QCG-Broker. This marks the first occasion that a cyclically coupled multiscale model uses distributed multiscale computing in such a general way.

4.1 Introduction

Models of biomedical systems are inherently complex; properties on small time and length scales, such as the molecular or genome level, can make a substantial difference to the properties observed on much larger scales, such as the organ, full-body and even the population level [125]. We therefore need to apply multiscale approaches when modelling many biomedical problems. Example biomedical multiscale challenges include predicting the impact of a surgical procedure [137], investigating the effects of pathologies (e.g. arterial malformations or fistulas [105]), or assessing the effects of a targeted drug on a given patient [111]. In all these cases, we need to examine processes that not only occur across several time and/or length scales, but that also rely on different underlying physical and/or biological mechanisms. As a result, modelling these processes may require substantially different algorithms and varying levels of computational effort.

Historically, these problems have often been modelled using single scale approaches, focussing on those aspects of the problem which are deemed most relevant. However, applying a single scale model is frequently insufficient to fully understand the problem at hand, as additional processes occurring on different scales must be incorporated to obtain sufficient accuracy. It is this need for understanding the composite problem, rather than its individual subcomponents alone, that has driven many research groups
to explore multiscale modelling, see e.g. [41, 49, 110, 127].

From a computational point of view, different parts of a multiscale model may have different, even contradictory, hardware and software requirements. For example, take a model with one submodel using a highly-parallel fluid dynamics flow solver, requiring a cluster with InfiniBand interconnects; another submodel, an agent-based simulation parallelised with OpenMP, performing best on a large SMP machine; and finally, a cellular automaton using GPU-powered calculations. Moreover, two submodels might require different specialised proprietary software, while having no sites available with licenses for both. This situation is only exacerbated if the multiscale model is cyclically coupled, requiring frequent communication between its submodels. Such a case demands distributed multiscale computing, as was recognised by five scientific communities behind the MAPPER project[2].

From the biomedical domain, a three-dimensional model of in-stent restenosis, ISR3D, is an example of a cyclically coupled multiscale model with heterogeneous submodels [32]. It models a stenosed blood vessel after stenting to determine if and how a restenosis could occur. The two-dimensional version, ISR2D, already has published results [137, 139, 140], but ISR3D is far more computationally demanding and requires distributed multiscale computing. Preliminary computational results of ISR3D have been reported by Tahir [136].

This chapter shows how a cyclically coupled multiscale model can be described, specified, and executed, exemplified by ISR3D. First, ISR3D is described with the high-level multiscale modelling language (MML) [19, 22, 47]. Once this is done, it can be specified using the MAPPER Memory (MaMe), Multiscale Application Designer (MAD), after which the application is managed by GridSpace Experiment Workbench [17, 39, 123]. Meanwhile, ISR3D is implemented using the multiscale coupling library and environment (MUSCLE 2, see Chapter 3 [26], which handles the communication between submodels. Finally, the application is then scheduled on distributed resources using the QosCosGrid stack [86], including EGI, PRACE, and a local resource. To our knowledge, this is the first time that a cyclically coupled multiscale application had a distributed execution in such a general and automated way. The case of ISR3D forms a validation point for the aforementioned MML, the tools to convert MML into an executable experiment, and for distributing a cyclically coupled multiscale model.

4.2 Multiscale modelling language (MML)

To bridge the gap between multiscale modellers and execution environments, the multiscale modelling language (MML) was conceived [19, 22, 47]. This language introduces a well-defined multiscale modelling terminology that can be used to describe, verify, analyse, and execute a multiscale model. MML is defined in detail in Chapter 2.3, but will be summarised here from the perspective of the application and tools.

As a basis, MML defines a multiscale model as a set of coupled single scale models. The way submodel instances are coupled is called the coupling topology. When a coupling topology is cyclic, it means that there is a feedback loop within the model and that certain submodels will be revisited; we call this a cyclically coupled model. In a loosely coupled model, without a cycle, a submodel can be considered finished when it has sent its information. However, in a cyclically coupled model, submodels repeatedly depend on input from other submodels. So while submodel $A$ is waiting for input, a runtime environment should either stop $A$ and restart it when input is available, or keep $A$ running during this period.

With MML it is possible to specify submodels and submodel instances but also their scale, computational requirements, and implementation details. Couplings are made explicit using the concept of conduits that bind to specific ports of submodels. For distributing or collecting messages between submodels, so-called fan-out and fan-in mappers are used.

For human interaction, MML has a graphical representation called gMML. This representation features the elements listed above, as shown in Figure 4.1, but does not contain any implementation details or information on scales. It is useful for composing or communicating the architecture of a multiscale model.

For machine interpretation, the XML format xMML captures these features, but also a wide range of metadata. This includes scale information, possible parameter settings, a datatype system, implementation details such as number of cores needed per submodel, but also descriptive and documentation facilities. In contrast to gMML, xMML can be automatically processed and it acts as an exchange format of a model.

Once a multiscale modeller has implemented a model and fully described it with MML, it is possible for software to verify, analyse, and execute it.
4.3 Software

The conceptual framework summarised above needs a software ecosystem to be put into practice. The software ecosystem needed to do distributed execution of a multiscale model, consists of roughly three parts: model specification and composition; a simulation runtime environment; and distributing and managing the execution.

4.3.1 High level composition and execution tools

To facilitate MML-based composition and execution of multiscale applications such as ISR3D, a set of supporting tools have been developed, depicted in Figure 4.2. First, MAPPER Memory (MaMe)\(^3\) is a semantics-aware persistence store to record the MML specifications of submodels and their scales. The information from MaMe is then fetched by the Multiscale Application Designer (MAD)\(^4\) – a user friendly visual composition tool that can connect single scale models to form a multiscale model. MAD can transforms a high-level MML description into an executable experiment, containing a MUSCLE configuration file, that can be executed in the GridSpace Experiment Workbench.

MaMe is based on the idea of semantic integration \(^{63}\). It supports the exchange and reuse of MML specifications by other tools via a REST (Representational State Transfer) interface, but also provides a web-based user interface.

\(^{3}\)http://gs2.mapper-project.eu/mame
\(^{4}\)http://gs2.mapper-project.eu/mad

Figure 4.1: The elements that make up gMML. In (a) the computational elements are shown, in (b) are the edge terminals corresponding to the current operation in the sub-model.
MAD supports application composition which is implemented as a sequence of drag-and-drop operations on graphical representations of MaMe components. On a conceptual and visual level, it is used to create gMML. When connections are created between the nodes MAD is able to perform various export procedures including xMML and the GridSpace executable format. Exported xMML contains MAD annotations about the positions of the elements in the MAD tool, so that when importing xMML, the visual composition persists.

The GridSpace Experiment Workbench (EW) supports execution and result management of infrastructure independent experiments. Experiments are applications composed of code fragments (called snippets) that can be expressed either in general-purpose scripting programming languages (Bash, Ruby, Perl etc.) or domain-specific languages (CxA in MUSCLE, LAMMPS, Matlab, etc). Snippets are evaluated by respective programs called interpreters. GridSpace provides also set of so called Executors that are responsible for snippets execution on various computational resources - servers, clusters, grid via direct SSH on User Interface (UI) machine or interoperability layer such as QCG (see Section 4.3.3). Each snippet can then be run on different resource.

4.3.2 MUSCLE

Implementing a multiscale model in a modular way is possible in several coupling environments; due to a close compatibility with MML we have chosen to use the multiscale coupling library and environment (MUSCLE 2) to implement ISR3D

https://gs2.mapper-project.eu
with (see Chapter 3). For a multiscale model, MUSCLE is in charge of handling communication between different submodels. As such, submodels, mappers and conduits are explicitly defined in MUSCLE, as are conduit filters. Its core is programmed in Java but it also supports C, C++, Fortran, Python, and MATLAB, using threads, OpenMP, or MPI.

Cross-cluster communication

Majority of clusters use private IP addresses for their worker nodes, thus accessing any process running within a job is not possible without additional effort. In addition, some of sites impose restrictions on outgoing traffic. In order to distribute multiscale applications that among many clusters, MUSCLE had to be adapted for firewalls and local IP-range environments. Firstly, a solution based on the port-range technique [96] was implemented, a mechanism which limits the ports numbers MUSCLE uses to some predefined range.

Secondly, communication between worker nodes of two clusters located in different administrative domains had to be enabled. This was solved by implementing a user-space daemon: MUSCLE Transport Overlay (MTO), depicted in Figure 4.3. This daemon is deployed at an interactive node, or any other node that is accessible from both external hosts and worker nodes, of all clusters involved in a multiscale simulation. Every MTO listens on a separate address for external and internal requests. The external port must be either accessible from all the other interactive nodes or the MTO must be able to connect to the external ports of all the others MTO (i.e. uni-directional connection is needed between every pair of MTOs).

Figure 4.3: Muscle Transport Overlay (MTO) Architecture

Another issue was that private IP addresses used for worker nodes are not globally
unique. Consequently, MUSCLE port ranges are enforced to be disjoint among all sites. Under this assumption the tuple \((\text{IP address}, \text{port})\) is unique for participating clusters.

4.3.3 Cross-cluster execution with QosCosGrid

Running multiscale application in cross-cluster environment requires addressing the following issues: co-allocation of heterogeneous resources; coordination of spawning application processes at multiple sites; and finally, enabling communication between systems with firewalls and Network Address Translation (NAT).

Co-allocation of heterogeneous resources

All modern HPC systems are managed by Local Resource Management Systems \cite{84}, often referred to as batch systems. In such environments a user will submit an application for execution (called a job), together with its resource requirements instead of running it directly. At a later time, a batch system will start the application when the requested resources are available and all local policies are met, thus preventing oversubscription of resources. With cross-cluster multiscale applications, submodels that are cyclicly coupled must be started at approximately the same time. However, the start time of jobs are not known prior to submission. One possible solution of this problem, known as resource co-allocation, is exploiting the advance reservation mechanism. An advance reservation consists of a list of users, a time slot and a set of resources. Once a reservation is created, the system guaranties the availability of those resources for the listed users during the given time slot, as long as no system failure occurs.

The QCG-Computing service\cite{98} uses the programmatic advance reservation mechanism available in almost every modern batch system, in order to co-allocate resources belonging to two or more resource providers. The whole process is managed by the QosCosGrid meta-scheduler: the QCG-Broker service\cite{86}. Users provide an upper limit on application runtime and a time window within which the application should start. QCG-Broker tries to find the earliest time when the requested amount of resources can be booked, and creates an advance reservation for it through the QCG-Computing service.

\begin{itemize}
\item \url{http://www.qoscosgrid.org/trac/qcg-computing}
\item \url{http://www.qoscosgrid.org/trac/qcg-broker}
\end{itemize}
Sites that do not have the QCG-Computing service installed can accept manually created advance reservations. QCG-Broker then tries to create a schedule based on the manual reservation and availability of other resources as depicted in Figure 4.4. This process has some similarities with the Two Phase Commit Protocol [149] known from transactions systems, i.e., when advance reservations are created successfully at all sites, the job is submitted (COMMIT); otherwise, all reservations are cancelled (ROLLBACK).

Coordination of application spawning

In most parallel toolkits used within single clusters there is a master process that spawns worker processes either using SSH or batch systems native interfaces. This make the task of exchanging contact information (e.g., listening host and port) between master and workers relatively easy as the master is always initialised before the workers. With a co-allocated distributed application the master and workers are started independently, and exchanging information is less trivial. In the QosCosGrid stack, the QCG-Coordinator service accepts contact information from the master, and provides it to any requesting workers. This relaxes the requirement that MUSCLE instances must be started in a particular order.

4.4 A three-dimensional model of in-stent restenosis

Coronary heart disease (CHD) causes about 7.3 million deaths per year worldwide, and it is one of the most common causes of death [112]. Typically, CHD is expressed as arteriosclerosis, which corresponds with a thickening and hardening of blood vessels caused by build-up of atheromatous plaque. Where arteriosclerosis causes a significant decrease in luminal area of the blood vessel, it is called a stenosis. A common inter-
vention for stenosis is stent-assisted balloon angioplasty, where a balloon with a stent is inserted in the blood vessel and inflated at stenosed location, consequently deploying the stent. The stent acts as a strut or scaffold for the blood vessel, compressing the plaque and holding the lumen open. Occasionally, however, this intervention is followed by in-stent restenosis (ISR), an excessive regrowth of tissue due to the injury caused by the stent deployment [80, 109]. The pathophysiological mechanisms and risk factors of in-stent restenosis have not yet been fully elucidated, although there have been multiple suggestions [79].

By modelling in-stent restenosis with a three-dimensional model (ISR3D) it is possible to explore which mechanisms and risk factors are likely to be main contributors to in-stent restenosis. After evaluating the processes involved in in-stent restenosis [46], ISR3D is modelled with the hypothesis that smooth muscle cell proliferation drives the restenosis, and that this is affected most heavily by wall shear stress of the blood flow regulating endothelium recovery and by growth inhibiting drugs diffused by a drug-eluting stent. With the model, the effect of different drug intensities, physical stent designs, vascular geometries and endothelium recovery rates can be evaluated. The predecessor of ISR3D, a two-dimensional model of in-stent restenosis (ISR2D), has a similar model architecture [32] and has published results [137, 138]. These clearly show correlations with re-endothelialisation and deployment depth are being reproduced in ISR3D. However, ISR2D is inherently limited by its two-dimensional design, which could not account for a full stent design, realistic cell growth, or exact blood flow. On the other hand, ISR3D requires far more computation; both cell proliferation and blood flow calculation are an order of magnitude more expensive in 3D.

From a multiscale modelling perspective, ISR3D spans multiple timescales with four submodels: smooth muscle cell proliferation on a timescale from hours to days, which explicitly models the cell cycle, cell growth, and physical forces between cells; initial thrombus formation due to the back-flow of blood, in the order of hours; blood flow (BF) and the resulting wall shear stress on individual smooth muscle cells, in the order of a second; and, drug diffusion of the drug eluding stent through the tissue and applied to the individual smooth muscle cells, in the order of minutes. The smooth muscle cell submodel is an agent based model on the cellular scale, which is in addition validated on the tissue level. All other submodels act on a cartesian grid representation of those cells. Figure 4.5 shows the cell proliferation in a simulation; in it, each sphere individually interacts with other spheres.
Figure 4.5: Cell growth simulated by ISR3D. The grey structure models the stent, the dark blue spheres model smooth muscle cells, the light blue spheres model endothelial cells and the red spheres form a layer that models the tunica externa.

Figure 4.6 shows the MML description of ISR3D. It shows that the model has a cyclic coupling topology, with smooth muscle cell proliferation as the coarse-scale submodel, which needs additional information in each iteration on wall-shear stress and drug diffusion. For each iteration, the smooth muscle cells locations are converted to a cartesian grid, and after wall-shear stress and drug diffusion is calculated those values are mapped back to the original cells.

The submodels act independently, apart from exchanging messages, and are heterogeneous. Smooth muscle cells proliferation has a C++ code, drug diffusion Java, thrombus formation Fortran, and blood flow uses the external C++ Lattice Boltzmann library Palabos. Only blood flow uses MPI. The MML specification of ISR3D contains a few mappers not mentioned above, which do simple data transformations but are necessary to ensure that the different single scale models are not aware of other submodels and their internal representation or scales.

Clinical directions

The ISR3D models main goal is to suggest which biological pathways dominate in the process leading to in-stent restenosis. If successful, this will have two effects on clinical
Figure 4.6: The MML description of ISR3D. SMC simulates smooth muscle cell proliferation, voxel maps its particle description to a grid and sends the mapping to in. Blob simulates initial thrombus formation and sends it to add, which combines the original geometry with the thrombus. The out mapper send the geometry to DD and BF for computing drug diffusion and wall shear stress, and they send their values back to in, which maps them to the particle description of SMC.

practice: first, it suggests which factors are important for in-stent restenosis which in turn gives clinicians more accurate estimates of what the progression of the disease will be; second, it may spur further directed clinical research of certain pathways, which will help the next iteration of the model give more accurate results.

The methods to achieve this are divided in two directions: general model validation and experiments; and, virtual patient cohort studies. For general model validation the literature and experiments are consulted. For ISR3D, this concerns porcine data. When this is done, virtual patient cohort studies assess the in-stent restenosis risk factors of virtual patients with different characteristics. In the clinical practice, this will not lead to personalised estimates, but rather to patient classifiers on how ISR3D will progress. Once the determining factors leading to in-stent restenosis are more evident, a simplified model could be made based on ISR3D, which takes less computational resources and runs within a hospital.

Both model validation and cohort studies need a large amount of computing resources, which why the MAPPER infrastructure is required for this application. Since no personalised data is used, there are no legal or ethical issues with distributing ISR3D over multiple European computing resources.
4.5 Results

In Section 4.4 an MML description of ISR3D was created. To create a software application it needs to be integrated with a software framework that can run a cyclically coupled multiscale model. To this end, each of the submodels and mappers of ISR3D use the MUSCLE library.

The information on ISR3D that is presented in Section 4.4, such as its scale separation map and MML, could be entered in straightforward manner in MaMe and MAD. First, the individual submodels and mappers are entered in MaMe, including information on scales, submodel ports and datatypes. In MaMe, it is also possible to enter preliminary or default parameter settings. Once this is done, the gMML of ISR3D was constructed using MAD, by connecting the respective ports of submodels and mappers of ISR3D together, and exported to a MUSCLE configuration file in GridSpace EW. This configuration file contained all parameters set in MaMe and all couplings defined in the MAD. In the GridSpace EW, the machines that different submodels should be scheduled on can be specified. Then, it was straightforward to run the simulation by simply pressing start.

An example scenario of a cyclically coupled model, ISR3D, running on distributed resources, will be described below and is depicted in Figure 4.7. One of the unique features of this scenario is the integration of resources provided by EGI, PRACE and local infrastructures. As can be seen from the figure, both drug diffusion and blood flow were computed on different hosts from the other submodels. For BF there is a very good reason, it is the only submodel of ISR3D that is extremely well parallelised, and which can make use of a many-core machine. DD could also have been computed on the same host as SMC, however, this scenario is also to show the viability of the approach sketched.

With MUSCLE handling the communication between submodels, problem described in Section 4.3.3 presented itself, where Huygens did not allow MUSCLE to open ports of worker nodes to the outside. This was fixed by using the MUSCLE Transport Overlay to relay communications to the Reef machine, which had a more liberal security policy.

Meanwhile, QCG-Broker made the reservations to the machines that were scheduled in GridSpace EW, aiming to create a co-allocation. It started by using the manual reservation on Huygens and then proceeded to make an advance reservation on Zeus. Using the reservation, it then submitted jobs to both systems to start MUSCLE, with
the blood flow submodel on Huygens and the other submodels on Zeus. MUSCLE communicated between both systems using the MUSCLE Transport Overlay. The IP address of the main MUSCLE instance was registered at the QCG-Coordinator and provided to the other instance. Once the submodels were finished, QCG-Broker collected the generated data and returned it to a host accessible to GridSpace, where it can be collected by the user.

During the time that the model was running, two hosts were reserved, however, not all resources were used efficiently in this process. Notably, the Huygens machine sat idle when BF was not computing, for instance when SMC was computing. During the run, only few iterations of ISR3D were performed, for scientific results more iterations will be run.

ISR3D runtimes have been measured in several settings. First of all, each of the submodels are used and tested locally. However, to do a coupling between the sub-
Table 4.1: Machines that are referred to in the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Type</th>
<th>Processor</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huygens</td>
<td>Amsterdam, NL</td>
<td>HPC</td>
<td>IBM Power6 4.70 GHz</td>
<td>64</td>
</tr>
<tr>
<td>Zeus</td>
<td>Krakow, Poland</td>
<td>cluster</td>
<td>Intel Xeon 2.40 GHz</td>
<td>8</td>
</tr>
</tbody>
</table>

models a more advanced machine is needed, since then multiple submodels will run at once. The goal is to know both the runtime and the efficiency of the setup used.

Performance test setup

For testing the performance of ISR3D, it is executed in four different scenarios, on a Polish national grid resource Zeus and on a PRACE tier-1 machine Huygens. Table 4.1 lists their statistics. Since the runtime behaviour ISR3D is cyclic, determined by the number of smooth muscle cell iterations, we measured the runtime of a single iteration for each. Queuing times are highly variable and do not add to the asymptotic runtime, so they are not measured.

It was first run on only Zeus and Huygens, and then on a combination of Zeus and Huygens. On Huygens, MPI was used to compute blood flow (BF), and all other submodels ran sequentially.

To use the reservation on Huygens efficiently, we created a setup which alternates the blood flow calculations of one simulation with the blood flow calculations of another, both running in the same reservation. This would in principle doubling the efficiency of the model on resource usage. Efficiency is defined here as CPU time used divided by CPU time reserved. This scenario treats the two blood flow calculations as mutually exclusive, as enforced with wait/notify semantics. This is implemented as two light-weight wait/notify mappers with no busy wait.

Results

In Table 4.2 the runtimes of the scenarios above are given. The efficiency is calculated as the CPU time taken, divided by the cores reserved times the time taken per iteration. The purpose of this measure is to ascertain what percentage of a reservation was actually used.

By using the PRACE resource the runtime went down drastically, but the efficiency was also severely decreased. This was somewhat ameliorated by using a scheme
Table 4.2: Runtimes with different scenarios. The first column is the scenario, the final column the efficiency and the others are runtimes in minutes.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>BF</th>
<th>Other submodels</th>
<th>Coupling</th>
<th>Total</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zeus</td>
<td>35</td>
<td>19</td>
<td>1</td>
<td>55</td>
<td>80%</td>
</tr>
<tr>
<td>Huygens</td>
<td>8</td>
<td>21</td>
<td>1</td>
<td>30</td>
<td>27%</td>
</tr>
<tr>
<td>Huygens-double</td>
<td>8</td>
<td>22</td>
<td>2</td>
<td>40</td>
<td>45%</td>
</tr>
<tr>
<td>Zeus-Huygens</td>
<td>8</td>
<td>16</td>
<td>2</td>
<td>26</td>
<td>32%</td>
</tr>
<tr>
<td>Zeus-Huygens-double</td>
<td>8</td>
<td>18</td>
<td>3</td>
<td>29</td>
<td>56%</td>
</tr>
</tbody>
</table>

where two simulations alternated on the same resource. Interestingly, the distributed simulation between Zeus and Huygens was faster and more efficient than running on Huygens alone. The reason is that the Blob submodel, which is a serial Fortran code, only compiled with the GNU gfortran compiler, which is not fully optimised for the IBM Power6 processor of Huygens. Zeus, however, has an Intel processor and since gfortran is well optimised for this Blob ran 3 times faster (reduced from 12 to 4 minutes). In all cases, the time spent in coupling code such as mappers and conduits, including communication time, took about $\frac{1}{10}$th of the total time.

These numbers will be put into context in the next chapter, where we compare the performance of a number of multiscale models, including ISR3D.

4.6 Conclusions and discussion

In this contribution we believe to have shown the first generalisable distributed multiscale execution of a cyclically coupled multiscale model, in this case, ISR3D. This was achieved by using recent foundations by way of the Multiscale Modelling Language, and tools based on that language: MAPPER Memory and the Multiscale Application designer. Since these tools were integrated with the application manager GridSpace Experiment Workbench, that in turn supported MUSCLE and QCG-Broker as execution tools, ISR3D was executed on heterogeneous infrastructure.

For ISR3D, doing distributed multiscale computing is a viable option. Compared to running on a general cluster, distributed multiscale computing cuts the runtime cost by more than a factor 2, although it is only 70% as efficient. Coupling cost played a small role in this, taking about $\frac{1}{10}$th of the total time. The resource management cost is larger though, so another viable option is to do only computation on Zeus.
where we could launch many jobs at once and have a very decent efficiency. Once the SMC code is parallelised, these options should be reevaluated.

By executing this scenario, ISR3D has the possibility to generate many more results. Other cyclically coupled multiscale models in the MAPPER project are anticipated to follow the same approach, further steadying and substantiating it. With this approach gaining more users, also its performance will have to be measured and compared with others.

One aspect in particular, resource usage and scheduling, should be explored further. In the scenario that was sketched here, the Huygens machine was partially idle while the blood flow submodel was not active. By using more advanced load balancing mechanisms, such as running multiple applications simultaneously to keep all resources active, this may be circumvented. Alternatively, by supporting the task graph for MML [22], and dividing an execution of a multiscale model, submodels could be dynamically scheduled to resources, creating no unnecessary idle reservations.