Architecture of Middleware to Provide the Multiscale Modelling Using Coupling Templates

Alexey Liniov[1], Valentina Kustikova[1], Alexander Sysoyev[1], Maxim Zhiltsov[1], Igor Polyakov[1], Denis Nasonov[2], and Nikolay Butakov[2]

[1] Lobachevsky State University of Nizhni Novgorod, Nizhny Novgorod, Russia
[2] ITMO University, St.Petersburg, Russia
alin@unn.ru✉, valentina.kustikova@gmail.com, sysoyev@vmk.unn.ru, zhiltsov.max35@gmail.com, polykovio@mail.ru, denis.nasonov@gmail.com, alipoov.nb@gmail.com

Abstract. The Multiscale Modelling and Simulation approach is a powerful methodological way to identify sub-models and classify their interaction. The execution order and interaction of computational modules are described in the form of workflow. This workflow can be executed as a single HPC cluster job if there is a middleware which schedule modules execution on allocated resources. We present an architecture of such middleware called Wrapper which provides internal module execution scheduling, interconnection functionality, module migration between allocated resources and storing intermediate state of computations. This middleware is compatible with CLAVIRE (CLoud Applications VIRtual Environment) platform and acts as its execution mechanism.

Keywords: Supercomputing technologies · Parallel computing middleware · High-performance computing · Multiscale modelling · CLAVIRE

1 Introduction

By the moment a great number of high-performance purpose-oriented software has been developed to solve problems in different application fields. Most computational modules are developed by applied specialists using various numerical models, programming languages and parallel programming technologies. In most cases only source code and binaries are available. Joint usage of such modules requires integration of data formats, used technologies and platforms. First of all it is necessary to determine principles of combined use of different models implemented in such modules. However, the employment of such modules is rather difficult even in cases when interaction with the code developer is possible.

One of the approaches progressing in the field of composite applications description is Multiscale Modelling [1,2], presenting templates to combine computational modules. Multiscale Modelling offers several standard methods applying some models of different time and spatial scales: Extreme Scale Computing

(ES), Hierarchical Multiscale Method of Computing (HMM), Replica Computing (RC) [3]. It defines the way to identify sub-models, classify the sub-model interactions as full or partial overlap of scales and specify the relation between the sub-models that could be represented as a task graph or workflow.

The development of the first multiscale modeling environments is carried out in specific applied fields. Among such environments, one can single out the Computational Materials Design Facility (CMDF) [4] that allows multi-scale multi-paradigm simulations of complex materials phenomena. This framework is based on a generic scripting environment, with the objective to enable simple setup of complex multi-scale simulation tasks. Interfaces between different modules, along with a central data structure allow straightforward communication between different simulation engines. CMDF uses the Python programming language to control the computational flow between disparate processing cores written in compiled languages (C/C++/Fortran) that carry out physicochemical calculations for multiscale/multiparadigm under a unified data model.

Morpheus [5] is another example of multiscale modeling environment. It allows the simulation and integration of cell-based models with ordinary differential equations and reaction-diffusion systems. It allows rapid development of multiscale models in biological terms and mathematical expressions rather than programming code. Morpheus separates modeling from numerical implementation by using a declarative domain-specific markup language.

Also, note Multiphysics Software Environment (MUSE) [6] for multiscale modeling in astrophysics. MUSE facilitates the coupling of existing codes written in different languages by providing inter-language tools and by specifying an interface between each module and the framework that represents a balance between generality and computational efficiency. MUSE has layered architecture. The top layer (flow control) is connected to the middle (interface layer) which controls the command structure for the individual applications. These parts and the underlying interfaces are written in Python, whereas the applications can be written in any language. The only constraint that code must meet to be wrapped as a module is that it is written in a programming language with a foreign function (C/C++, Fortran, C#, Java, Haskel etc.).

Later, ideas are formulated about the need to develop a universal environment that provides the possibility of carrying out a multiscale experiment, regardless of the specifics of the applied field. In this connection, the concept of a multiscale model is formalized and their classification is introduced [7]. Based on this classification the Multiscale Coupling Library and Environment (MUSCLE) [7] and its improved version MUSCLE 2 [8] are implemented. MUSCLE 2 is a component-based modeling tool inspired by the multiscale modeling and simulation framework, with an easy-to-use API which supports Java, C++, C, and Fortran. It assumes that a multiscale model is split into multiple coupled single scale submodels [7]. As a result, each submodel has inputs and outputs that can be coupled in a general way. Within one simulation, one submodel could for instance use hundreds of cores on a supercomputer, whereas another may have to make use of GPU-computing, and yet another needs high I/O performance.

Each submodel is managed by its own instance controller. The controller is an intermediary for any messages that a submodel sends or receives [8].

The proposed environment Wrapper is based on the same theoretical foundations of multiscale modeling as MUSCLE 2 [8]. In contrast to MUSCLE 2 we attempt to organize centralized submodels scheduling in accordance with statistics on its resource usage. To utilize computational resources efficiently one need to analyze parameters and statistical data related to utilization of hardware resources, execution of individual computational modules and composite application taken as a whole. In case of specific modules the relatively simple algorithms can be used [9,10], but provided huge computational facilities are implemented and complex applications are executed we shall use the more comprehensive approaches, such as Knowledge-Based Resource Management [11]. The middleware which we develop, is oriented to coupling with CLAVIRE (CLoud Applications VIRtual Environment) [12] which allows building composite applications using domain specific software available within distributed environment. CLAVIRE builds the workflow, reserve resources of high-performance computing system and launches Wrapper middleware in allocated resources.

2 Purpose of Wrapper Middleware

Wrapper middleware is a MPI program. Wrapper is launched in computational cluster by CLAVIRE scheduler, assumed as being executed in the node external towards the computational cluster. CLAVIRE scheduler ensures the delivery of input data to the cluster, the analysis of cluster resources and features of workflow to be executed, job formation for the cluster management system, launching Wrapper, as well as download the output data from the cluster.

Wrapper provides the following functionality.

- Collecting information about allocated resources of computational cluster.
- Dynamically assigning of computational modules to cluster nodes (with possibility of migration).
- Launching workflow execution.
- Data transmission between computational modules.
- Completing the workflow execution and release of computational cluster resources.

Wrapper architecture makes possible to schedule the execution of computational modules within the allocated resources, however the scheduler is not its subsystem. Scheduling algorithm implements by default static allocation of computational modules on cluster nodes, but we are going to provide integration with adapted version of CLAVIRE scheduler.

Fig. 1 illustrates the structural diagram of subsystems interacting with Wrapper middleware.

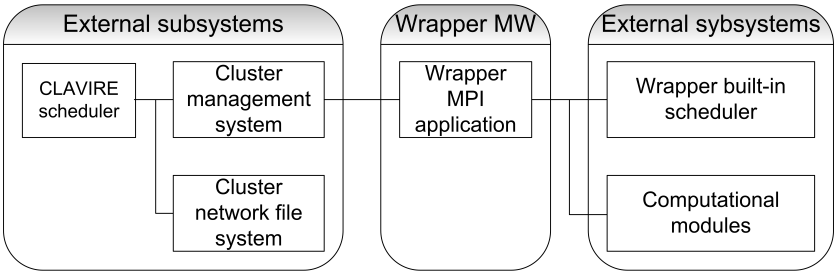

Fig. 1. Interacting subsystems: structural diagram

3 The Workflow Model

3.1 Introduction to the Workflow Model

Wrapper is oriented to use execution patterns presented as a workflow. The workflow contains information about composition of computational modules, possible sets of input as well as output data. The input data of computational module can be built based on output data of other modules or received from the scheduler (such option is required to launch the workflow execution). Fig. 2 shows the example of workflow, consisting of the scheduler (S), 3 computational modules (CM-1, CM-2, CM-3) and description of relations between them. In this example the computational module can be executed if all input data has been received.

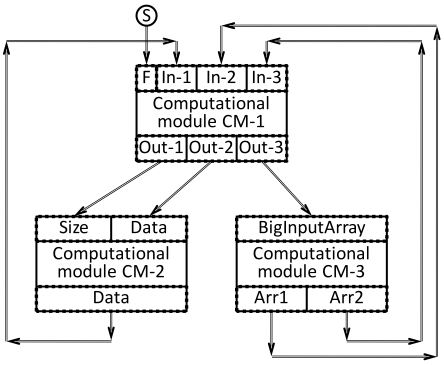

Fig. 2. Workflow: example. “S” is a scheduler. “CM-1”, “CM-2”, “CM-3” are computational modules. “F” and “In-1, In-2, In-3” are the sets of inputs of the module “CM-1”; “Out-1, Out-2, Out-3” is a set of outputs of the module “CM-1”. “Size, Data” is a set of inputs of the “CM-2”; “Data” is a set of outputs of the “CM-2”. “BigInputArray” is a set of inputs of the “CM-3”; “Arr1, Arr2” is a set of outputs of the “CM-3”

3.2 Concept "Set of Inputs/Outputs"

"The set of inputs" is a set of input data of computational module (hereafter – CM), sufficient to launch the module execution. Each CM input can be included only into one set. The description of all possible sets of inputs for each CM is represented in the workflow. On each cluster node Wrapper collects input data for computational module assigned to this node; once any full set of inputs is collected it is transmitted to computational module (or CM is launched with the prepared set).

"The set of inputs/outputs" is required to ensure the integrity of CM input data structure as well as to support the concept of "launching output" (see below). Here we shall point out that empty output and absence of output are completely different situations because an empty output can be used (and required) to form the set of inputs for another module.

The sets of inputs make it possible, using workflow, to describe the Multiscale Modelling templates including a number of integrated models (in time and space); for example templates shown in Fig. 3 and Fig. 4 can be presented as workflows on that figures.

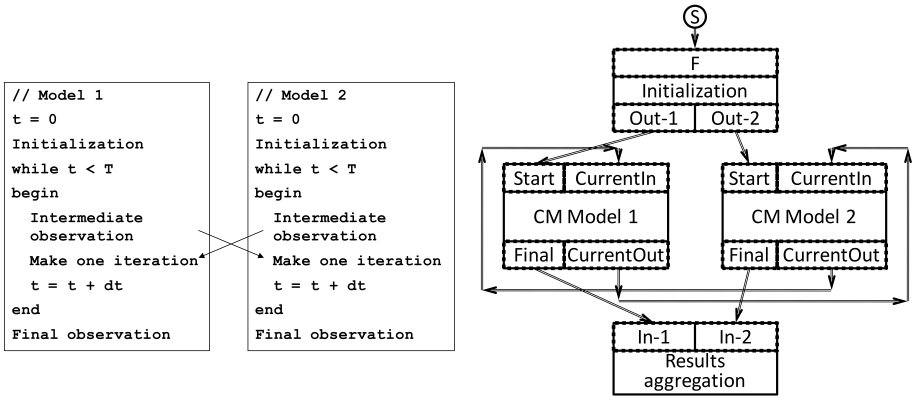

Fig. 3. The Multiscale Modelling with 2 models and integration in space: calculation scheme and workflow. "CM Model 1", "CM Model 2" are computational modules implementing "Model 1" and "Model 2" respectively. "F" and "Out-1, Out-2" are sets of inputs and outputs of the Initialization module. "Start" and "CurrentIn" are sets of inputs, "Final" and "CurrentOut" are sets of outputs of "CM Model 1" and "CM Model 2". "In-1, In-2" is a set of inputs of the "Results aggregation" module

In both cases the execution starts after the scheduler sends the set of data including one "S" output to the input of initialization module "F". For this module one set of inputs is assigned containing only "F" input; that is why once this input received the execution of the initialization module will start and the outputs will be formed sufficient to launch CM models (two in the first case

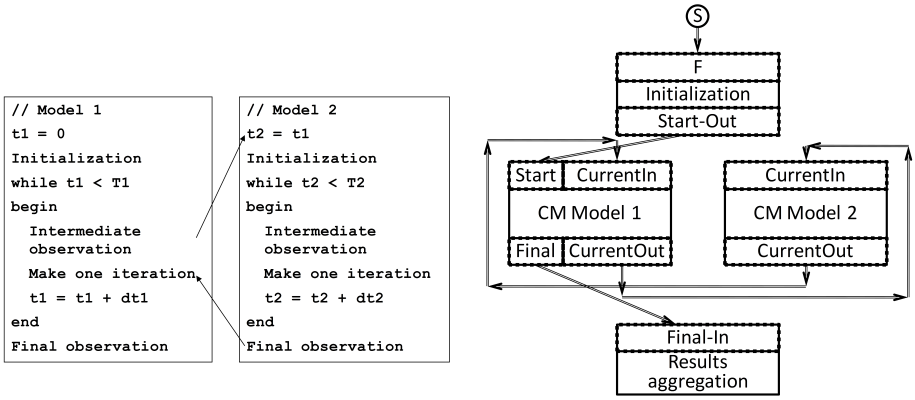

Fig. 4. The Multiscale Modelling with 2 models and integration in time: calculation scheme and workflow. “CM Model 1”, “CM Model 2” are computational modules implementing “Model 1” and “Model 2” respectively. “F” and “Start-Out” are sets of inputs and outputs of the “Initialization” module. “Start” and “CurrentIn” are sets of inputs, “Final” and “CurrentOut” are sets of outputs of “CM Model 1”. “CurrentIn” and “CurrentOut” are sets of input and outputs of “CM Model 2”. “Final-In” is a set of inputs of the “Results aggregation” module

and one in the second case). For example in the first case two outputs will be formed launching CMs for models 1 and 2. Thereafter CM for models 1 and 2 will be executed in a parallel way, in each iteration sending each other input sets sufficient for their next iteration. After the modeling is completed CMs form resulting data and transmit them to the module of result aggregation, which performs the final processing and completes the workflow execution.

Thus, workflow with sets of inputs can be implemented to organize the Multiscale computations based on Extreme Scale Computing (ES).

3.3 Workflow Modification. The “Module Instance” Concept

Usage of Multiscale computation template Hierarchical Multiscale Computing (HMM) implies the execution of many launches for CM model of less scale in one computation step of the model of larger scale. The number of launches for CM model of less scale can be unknown in advance and vary from iteration to iteration; and for efficient computations its necessary to parallel launch several copies of less-scaled CM model on different cluster nodes.

To support the HMM pattern the concepts “launching set of outputs”, “module instance”, “aggregating set of inputs” shall be introduced into workflow.

– “Module instance” is a computational module launched for processing one or several sets of inputs. Each module is identified by the pair (module identifier, instance identifier). Wrapper uses these pairs as CM addresses.

– "Launching set of outputs" (LSO) means that one or several module instances which receive the input from this set can be launched. The number of instances is determined by the Wrapper scheduler which, at the moment of transmission of regular output set, determines if the outputs included into the set will be sent to the already launched CM instances or additional instances will be launched and the data shall be sent to them. A unique identifier is assigned to each LSO and it will be inherited by the output data of following modules. LSO are sent in blocks of arbitrary size, thereby simultaneous sending is not compulsory. As soon as the numbers of the first and last LSO in the block are known they are sent to the corresponding modules with aggregating set of input data.
– "Aggregating set of inputs" is used to collect several outputs of several instances of one CM into one input of other CM. In the workflow "Aggregating set of inputs" is obviously linked with "Launching set of outputs" which generates the parallel processing followed by aggregation. CM with "Launching set of outputs" for each block of outputs sent for processing shall obviously transmit numbers of the first and the last sets (corresponding outputs and inputs are automatically created in the sets of inputs/outputs) to all modules with aggregating input. The block of sets of inputs is transmitted to the module only after receiving all sets according to the first and last numbers.

Fig. 5 shows the workflow example using HMM template.

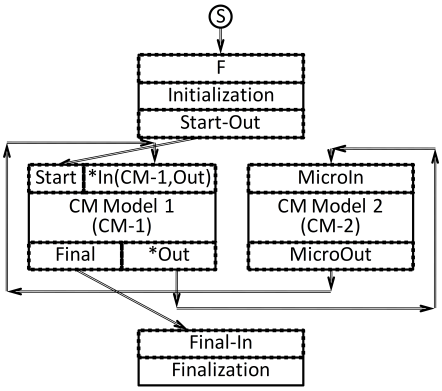

Fig. 5. Workflow for the Multiscale Modelling using Hierarchical Multiscale Computing (HMM). "CM Model 1 (CM-1)", "CM Model 2 (CM-2)" are computational modules. "F" and "Start-Out" are sets of inputs and outputs of the "Initialization" module. "Start" and "*In(CM-1, Out)" are sets of inputs of the "CM-1". "Final" and "*Out" are sets of outputs of "CM-1". "MicroIn" and "MicroOut" are a sets of inputs and outputs of the "CM-2". "Final-In" is a set of inputs of "Finalization" module

"CM Model 1" has one launching set of outputs which includes "*Out" output. Correspondingly, the arbitrary number of instances of "CM Model 2" can

be launched. “CM Model 1” has also one aggregating set of inputs “*In(CM-1,Out)”, where input data “In” are aggregated according to the blocks of outputs generated by the “Out” output of the same “CM Model 1”.

4 Wrapper Architecture

First of all lets introduce the interconnection diagram of CM (Fig. 6).

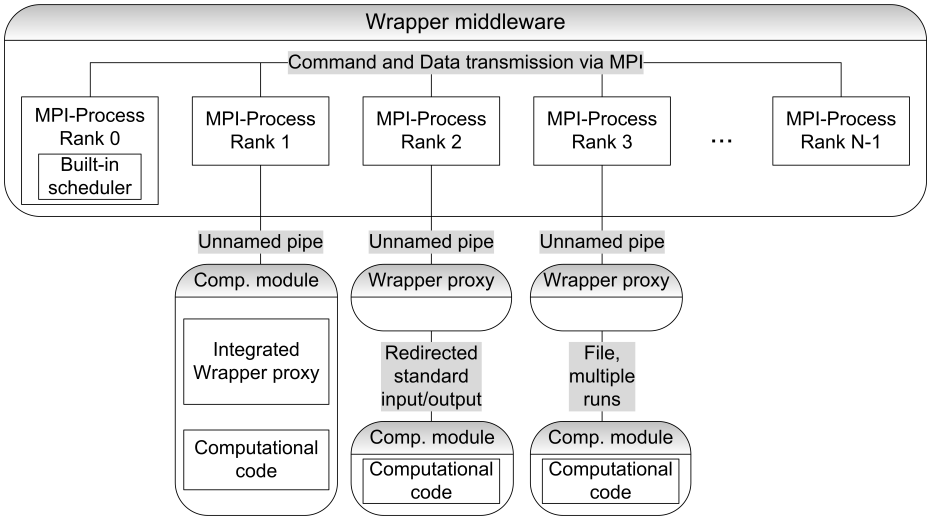

Fig. 6. Wrapper and CMs interconnection diagram. Wrapper middleware is a MPI program. Built-in scheduler is integrated into the Process with Rank 0. Another processes provide launching of computational modules and their communication

Adaptation of computational modules to execution with Wrapper can be performed by the following methods.

1. Integration between computational module and Wrapper at source code level: CM compiled and linked with a set of Wrapper functions providing the transmission of commands and data using mechanism of unnamed pipes (integrated or built-in Wrapper proxy). The computational module is launched one-time when the workflow execution starts, thereafter it is executed constantly, receiving and transmitting messages through the unnamed pipes.
2. The computational module is developed using the set of Wrapper functions which provide reading and parsing of input data and building the set of output data. The computational module is launched every time when Wrapper builds the full set of its input data (external Wrapper proxy). While launching the module 3 parameters are transmitted to it via environment variables:

input file name, output file name, file name of the module state (the last parameter is used if the module shall save some data between iterations).

3. The computational module is compiled and operates independently receiving and transmitting input/output data using the standard input/output (external Wrapper proxy). The computational module is also launched one-time when the workflow execution starts and executed constantly, receiving and transmitting messages through the redirected standard input/output.

In the first mode Wrapper proxy requires to know the command lines in order to launch and stop the computational module. The computational module uses a provided specific interface for reading the input data and saving the output data.

The second mode is available only for computational modules in C and C++ programming languages. To integrate proxy into the computational module the following shall be done:

– design the computational module in a specific way;
– include the Wrapper proxy header files into CM source code;
– create a proxy object in the computational module;
– define the computation function to callback from proxy;
– launch proxy from the computational module.

The Wrapper structural diagram is shown in Fig. 7.

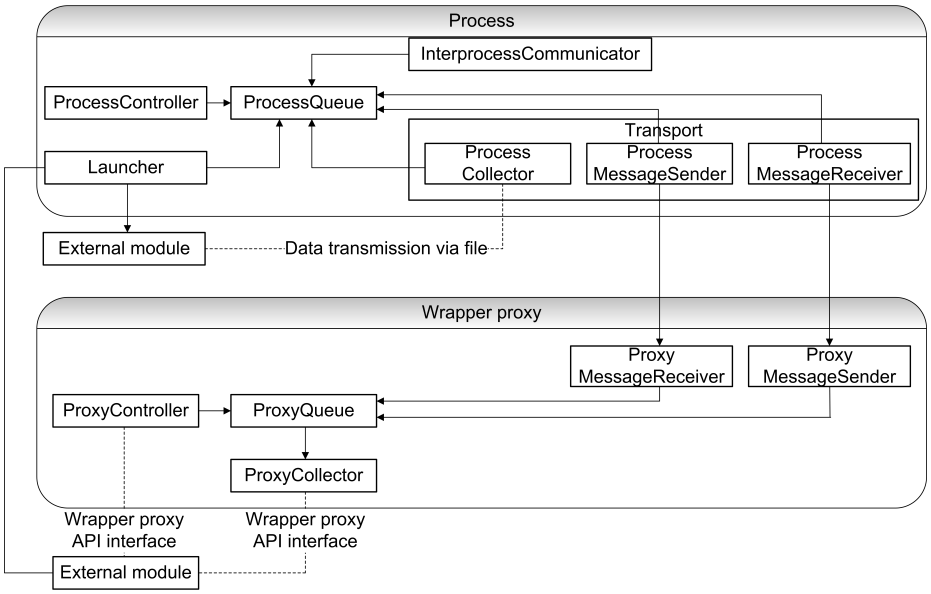

Fig. 7. The Wrapper middleware structure

Process contains components executed in MPI-process, Proxy contains components executed in the built-in Proxy. Structural components has the following purpose.

- ProcessQueue, ProxyQueue are message queues through which the interaction of other components is performed.
- InterprocessCommunicator provides the data transmissions between Wrapper processes using MPI technology.
- ProcessCollector, ProxyCollector receive input data for the computational modules and form the sets of inputs, distribute the sets of outputs and send them to the receivers. Besides collectors receive information about module migration. If collector does not prepare full set of inputs for module execution then it transfers current set of inputs back to the message queue. After that, InterprocessCommunicator sends data to the target computational module which was migrated.
- ProcessMessageSender, ProcessMessageReceiver, ProxyMessageSender, ProxyMessageReceiver provide data transmission between Wrapper and the computational modules with integrated Proxy.
- Launcher implements the launching of the computational modules.
- ProcessController, ProxyController manages allocation/release of other components.

5 The Results of Experiments

5.1 Computational Infrastructure

We used UNN Lobachevsky supercomputer. Nodes of the Linux segment we used have 2x Intel Sandy Bridge E5-2660 2.2 GHz processors (8 cores), 64 GB RAM, QDR InfiniBand network. We employed the Intel MPI and Intel C++ Compiler from the Intel Parallel Studio XE Cluster Edition 2017.

5.2 The Test Workflow

To perform the tests the following computational diagram has been used (see Fig. 8, hereafter "test workflow").

Modules 0–4 do not perform any computations, they only receive the input data and send the output data. Module 0 sends the data block of the fixed size to Module 1, Module 1 sends the copies of this block to Modules 2 and 3, Modules 2 and 3 send an empty message to Module 0.

5.3 Results

The performance of the test workflow has been estimated in terms of data transfer rate for two mechanism of data transmission to computational modules:

- using of external wrapper proxy and data transmission through the file;

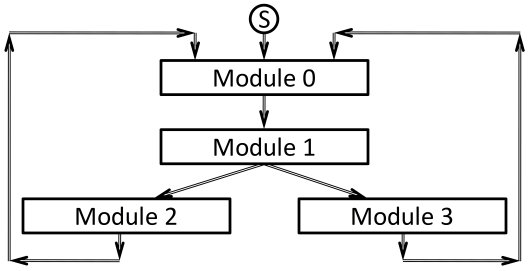

Fig. 8. The test computational diagram

– using proxy, integrated with the computational module.

During tests several iterations of workflow has been performed and average time for one iteration has been calculated. Data blocks from 10 bytes to 1 billion bytes has been used. Table 1 and Fig. 9 show the average execution time values for one iteration using data transmission through the file.

Table 1. Test workflow iteration times for external Wrapper proxy

#	Block size (B)	Average iteration time (s)
1	10	0.13
2	100	0.14
3	1 000	0.13
4	10 000	0.16
5	100 000	0.14
6	1 000 000	0.29
7	10 000 000	0.76
8	100 000 000	17.98
9	1 000 000 000	196.02

Test results show that the overhead for single iteration is approximately constant and makes up about 0.15 s. The transfer time starts to have a value only when the block size exceeds 1MB.

Table 2 and Fig. 10 show the average execution time values for one iteration using built-in Wrapper proxy and data transmission via unnamed pipes.

The overhead for one iteration is again constant and makes up about 0.05 s. Switching from using an external proxy to a built-in one reduces the transfer time of 1 GB data block from 196 s to 62 s. In general, the results of the experiment show the advantage of using the built-in proxy, and acceptable performance.

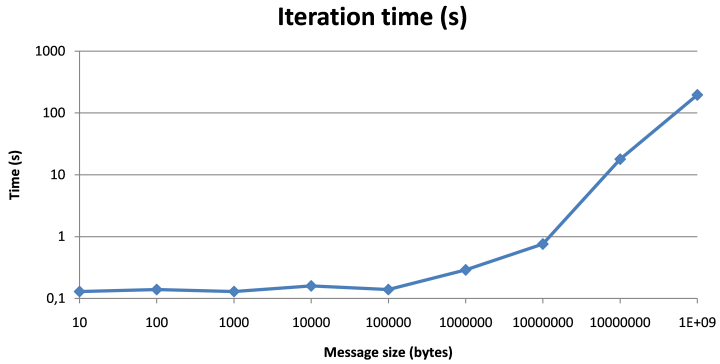

Fig. 9. The test workflow iteration times for external Wrapper proxy

Table 2. Test workflow iteration times for built-in Wrapper proxy

#	Block size (B)	Average iteration time (s)
1	10	0.054
2	100	0.053
3	1 000	0.05
4	10 000	0.06
5	100 000	0.05
6	1 000 000	0.06
7	10 000 000	0.12
8	100 000 000	6.39
9	1 000 000 000	61.92

6 Application of Wrapper Middleware for "Restenosis" Modeling

Within our study we have adopted "Restenosis" application (computation of barrier reconstruction in blood vessels) in order to use the Wrapper middleware. Original version uses 8 computational modules and MUSCLE library, providing modules launch and data exchange between them. Adaptation includes the following stages to do.

1. Analyze launching methods of the computational modules.
2. Analyze data transmission workflow between the modules.
3. Analyze principles and mechanisms for implementation of interactions between modules using the MUSCLE library interface.
4. Develop "Restenosis" source code modifications which enable collecting and saving the sets of inputs and outputs. Make test launches and save the data sets (for further testing of adapted version).
5. Develop the set of Wrapper functions which implement reading and parsing of inputs as well as building sets of Wrapper outputs in programming lan-

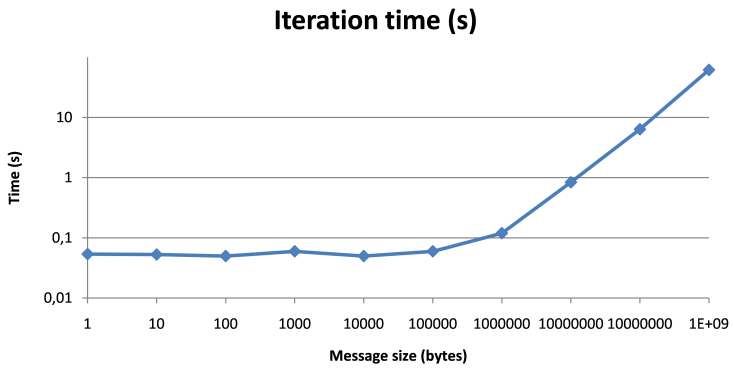

Fig. 10. The test workflow iteration times for built-in Wrapper proxy

guages used in the computational modules of "Restenosis" application (C, C++, Java).
6. Exclude the MUSCLE library from source code of the computational modules. Add Wrapper code.
7. Test the adapted computational modules.
8. Launch the adapted version of "Restenosis" using Wrapper middleware.

The adapted version of "Restenosis" on the test problem generates output that coincides with the original version, and shows comparable performance.

7 Conclusion

The paper describes extensions to the workflow model which enables the execution of composite tasks based on the Multiscale Modelling templates. The developed architecture of Wrapper middleware provides combined usage of the computational modules developed with different programming languages and technologies. Migration of the computational modules between cluster nodes becomes possible as well. Performance tests show acceptable results. The use of Wrapper in the modeling of "Restenosis" shows the possibility of using it for solving applied problems. The authors continue to develop and plan further use of Wrapper middleware.

Acknowledgments

This research financially supported by Ministry of Education and Science of the Russian Federation, Agreement #14.587.21.0024(18.11.2015). Unique Identification RFMEFI58715X0024.

References

1. Hoekstra, A.G., Lorenz, E., Falcone, J.-L., Chopard, B.: Toward a Complex Automata Formalism for MultiScale Modeling, International Journal for Multiscale Computational Engineering 5 (6), 491-502 (2007)
2. Borgdorff, J., Falcone, J.-L., Lorenz, E., Bona-Casas, C., Chopard, B., Hoekstra, A.G.: Foundations of distributed multiscale computing: Formalization, specification, and analysis, Journal of Parallel and Distributed Computing 73, 465-483 (2013)
3. Alowayyed, S., Groen, D., Coveney, P., Hoekstra, A.: Multiscale Computing in the Exascale Era. In: arXiv preprint arXiv:1612.02467 (2016)
4. Computational Materials Design Facility (CMDF). `http://web.mit.edu/mbuehler/www/research/CMDF/CMDF.htm`
5. Starruby, J., Backy, W., Brusch, L., Deutsch, A. Morpheus: a user-friendly modeling environment for multiscale and multicellular systems biology // Bioinformatics 30(9), 1331-1332 (2014)
6. Zwart, S.P., et al.: A Multiphysics and Multiscale Software Environment for Modeling Astrophysical Systems // LNCS 5102, 207216 (2008)
7. Borgdorff, J., et al.: Foundations of distributed multiscale computing: Formalization, specification, and analysis // J. Parallel Distrib. Comput, 73, 465-483 (2013)
8. Borgdorff, J., Mamonski, M., Bosak, B., Groen, D., Belgacem, M.B., Kurowski, K., Hoekstra, A.G.: Multiscale computing with the multiscale modeling library and runtime environment. Procedia Comput. Sci. 18, 10971105 (2013)
9. Kishimoto, Y., Ichikawa, S.: Optimizing the Configuration of a Heterogeneous Cluster with Multiprocessing and Execution-Time Estimation. Parallel Computing 31(7), 691710 (2005)
10. Dolan, E.D., Mor, J.J.: Benchmarking Optimization Software with Performance Profiles. Mathematical Programming 91(2), 201213 (2002)
11. Kovalchuk, S., Larchenko, A., Boukhanovsky, A.: Knowledge-Based Resource Management for Distributed Problem Solving // Knowledge Engineering and Management, Springer AISC 123, 121128 (2011)
12. Knyazkov, K.V., Kovalchuk, S.V., Tchurov, T.N., Maryin, S.V., Boukhanovsky, A.V.: CLAVIRE: e-Science infrastructure for data-driven computing // Journal of Computational Science 3(6) , 504-510 (2012)