Abstract

We present a global overview of the methodology developed within the MAPPER European project to design, implement and run a multiscale simulation on a distributed supercomputing infrastructure. Our goal is to highlight the main steps required when developing an application within this framework. More specifically, we illustrate the proposed approach in the case of hydrology applications. A performance model describing the execution time of the application as a function of its spatial resolution and the hardware performance is proposed. It shows that Distributed Multiscale Computation is beneficial for large scale problems.

Reference


DOI: 10.1016/j.procs.2013.05.276
International Conference on Computational Science, ICCS 2013

Distributed Multiscale Computations using the MAPPER framework

Mohamed Ben Belgacem\textsuperscript{a,\ast}, Bastien Chopard\textsuperscript{a,\ast}, Joris Borgdorff\textsuperscript{b}, Mariusz Mamoński\textsuperscript{c}, Katarzyna Rycerz\textsuperscript{d,e}, Daniel Harezlak\textsuperscript{e}

\textsuperscript{a}University of Geneva, Route de Drize 7, 1227 Carouge, Switzerland
\textsuperscript{b}Computational Science, Faculty of Science, University of Amsterdam, Amsterdam, the Netherlands
\textsuperscript{c}Poznań Supercomputing and Network Center, Poznań, Poland
\textsuperscript{d}Institute of Computer Science, AGH University of Science and Technology, Kraków, Poland
\textsuperscript{e}CYFRONET, AGH University of Science and Technology, Kraków, Poland

Abstract

We present a global overview of the methodology developed within the MAPPER European project to design, implement and run a multiscale simulation on a distributed supercomputing infrastructure. Our goal is to highlight the main steps required when developing an application within this framework. More specifically, we illustrate the proposed approach in the case of hydrology applications. A performance model describing the execution time of the application as a function of its spatial resolution and the hardware performance is proposed. It shows that Distributed Multiscale Computation is beneficial for large scale problems.

Keywords: Multiscale, application, distributed, grid, computation, MAPPER framework

1. Introduction

Multiscale applications are an important challenge for computational science and many domains of research. Most phenomena involve many spatial or temporal scales, and the interaction between various physical processes. It is therefore important to provide an effective methodology for designing, implementing and running multiscale applications. There are rather few frameworks that offer general tools and concepts to address such a challenge. Methodological papers such as [1, 2, 3] are important in this perspective as they propose examples of a conceptual approach. However, there is still a lack of a consensus vision among the many communities dealing with multiscale applications. A recent review [4] of different domains shows that existing multiscale projects tend to use radically divers approaches to build and run their applications, mostly connecting their codes with hand-written scripts and run computation on single sites.

Running simulations across multiple computing sites has been done in previous works, as illustrated for instance in [5, 6, 7]. These approaches offer user friendly API and convenient grid submission procedures but do not propose a full methodology based on theoretical concepts and a formalism as discussed below.
A few years ago, some of us contributed to the development of a multiscale framework, coined CxA for Complex Automata. The initial goal was the coupling of different Cellular Automata or Lattice Boltzmann models with different spatial and temporal scales [8, 9, 10]. This approach has been successfully applied to the so-called In-Stent Restenosis biomedical problem [11]. It also gave rise to a Multiscale Modeling Language (MML) [12], offering a powerful way to describe a multiscale problem in terms of its components and their couplings.

This approach is now further developed and generalized within the European project MAPPER 1. The goal is to show that a wide range of applications can be described, implemented and run within the formalism. An important aspect of the project is the execution phase where the concept of Distributed Multiscale Computing (DMC) is proposed [13, 14]. In short, the idea is that the different components of a multiscale-multiphysics application can be distributed on several supercomputing facilities and run in a tightly and/or loosely coupled way. DMC allows the users to access at once, and for one specific application, much more computing resources than available in a single supercomputing center.

The goal of this paper is to present the sequence of actions that are proposed in the MAPPER framework, from building to executing a multiscale-multiphysics application on the European distributed computing infrastructure, such as EGI and PRACE [15, 16]. The process will be illustrated for the case of a specific problem taken from the MAPPER application portfolio, namely the simulation of irrigation canals or rivers. The other applications studied in MAPPER are in-stent restenosis [11, 17], cerebrovascular blood flow [18], clay-polymer nanocomposites [19], nuclear fusion, and gene regulatory networks.

Hydrology is a class of problem for which multiscale tools are needed. River and waterway management is a major concern of our modern society, in particular for clean electricity production and agriculture irrigation. One of our interests is to simulate a section of the Rhone river (13 km), from the Geneva city down to Verbois water dam. In collaboration with the Geneva electricity company, we are investigating the possibility to study specific questions, namely, the way to flush sediments in some critical area by changing the water levels. In order to address this problem, a multiscale, multiphysics approach is required as one cannot afford a fully resolved 3D, free surface flow with sediment transport along the full 13 km. We are thus developing a numerical approach where some sections are described with a 1D shallow water solver whereas more critical sections are solved by a 3D free surface model with sediment. The MAPPER framework allows us to easily connect these different components and run them on a distributed grid of supercomputers [20].

Note that for hydrology applications at least, MAPPER offers a problem solving environment, that is a framework in which several multiscale, multiphysics components can be predefined and coupled at will to build any desired complex hydrology system. We described this feature as a “lego based philosophy”.

2. Formalism

We refer the reader to [12, 10, 13, 14, 21] for a detailed discussion of the theoretical concepts underlying our approach. Here we would like to adopt the point of view of the end-user aiming at developing, or adapting, his application within the proposed framework.

The MAPPER formalism is based on four steps, as illustrated in Fig. 1. These steps are called here Modeling, computation design, multiscale implementation and execution. Each of them is discussed below.

2.1. Modeling

In this step, a given multiscale-multiphysics phenomena is decomposed in several coupled single-scale submodels, by selecting only the most relevant scales of the problem, and selecting the relevant physical processes that take place in the system. From a practical point of view submodels correspond to numerical solvers, with given spatial and temporal resolution. By coupling these submodels in an appropriate way, one expects a good enough approximation of the original problem with all scales, with yet a substantial reduction of computational needs.

The coupling between the submodels is realized by components called mappers (junctions) or conduits. They implement the required scale bridging techniques and transfer of data. Note that the knowledge of the different

---

1http://www.mapper-project.eu
scales of the system is in the junctions only, making the submodels autonomous and independent solvers, blind to the coupling structure.

The decomposition in submodels and their couplings is a difficult yet central task. It is usually done according to the specific knowledge that the scientist has of the given problem. This process is however facilitated by representing the system on a Scale Separation Map (SSM), as suggested for instance in [10].

An important aspect of the formalism is that submodels are assumed to be described by a generic time loop containing only a few abstract operators, named initialization, observation, computation, and boundary condition. The coupling between submodels can then be thought of as a link between a pair of such fundamental operations. In order to be MAPPER compatible, a submodel should have input and output ports for the above generic operations. At the programming level, these ports are implemented with a coupling framework called MUSCLE [22, 23].

In our approach, the links between submodels are also computational processes. Submodels do not have to know about the rest of the system, only links know the properties and scales of the submodels they connect. Simple links simply transfer data from one side to another, but they can also be programmed to do some transformation (filtering or interpolation). An important type of link is provided by a component called mapper which may have several inputs or several outputs.

Several coupling patterns can be identified between submodels. They depend on the relation between the submodels in the SSM [10] and can be cyclic or acyclic [13].

The conceptualization of a multiscale application according to the above methodology can be expressed, independently of the implementation, with MML, our multiscale modeling language [12]. Within this language, the submodels, their scales and their couplings are described in an abstract way. Fig. 2 illustrates a graphical version of the MML formulation of a physical system where two canal sections are connected through a gate.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Real phenomena</td>
<td>MML description</td>
<td>Coding + MUSCLE2 API</td>
<td>CxA file</td>
</tr>
<tr>
<td>- Abstract description</td>
<td>- Definition of submodels</td>
<td>- Submodel implementations</td>
<td>workflow</td>
</tr>
<tr>
<td>- Functional decomposition (CxA formalism)</td>
<td>- Submodel definitions database</td>
<td>- Java binaries</td>
<td>GridSpace</td>
</tr>
<tr>
<td>- Scale based decomposition</td>
<td>- MAD tool</td>
<td>- Fortran binaries</td>
<td>AHE</td>
</tr>
<tr>
<td>Design graphically the computation</td>
<td>- CxA file</td>
<td>- C/C++ binaries</td>
<td>QCG</td>
</tr>
<tr>
<td>MML description</td>
<td>- workflow</td>
<td>European computing infrastructure</td>
<td>egi</td>
</tr>
</tbody>
</table>
and enables users to drag-and-drop in a graphical design space in order to compose a cyclic or acyclic computational workflow.

It is worth noticing here that the submodels/junctions can be implemented in different languages (C++, Java, etc.) and have no information regarding the other components, thus, illustrating the “Lego based philosophy” of our approach.

Fig. 3 illustrates the design step of a multiscale application where a computational workflow is designed on the fly. The connection between components is handled by the defined operators and conduits (arrows in the figure) and inter-scale data are transmitted throughout conduits accordingly to the data type described in the MaMe database.

The MAD tool generates a configuration file, named CxA file in reference to the theoretical concepts mentioned above. This file describes the coupling of all the components as illustrated in Listing 1 for the case of two canal sections separated by a gate. Here, the flow in the canal sections are computed by two submodels, SW1D1 and SW1D2, both implementing a Lattice Boltzmann 1D shallow water flow simulation (hence the name SW1D).

Listing 1. Example of CxA file

```python
# declare kernels
2 cxa.add_ker neoliberal('SW1D1', 'com.unige.irigcan.kernel.d1.SW1Dkernel')
3 cxa.add_ker neoliberal('SW1D2', 'com.unige.irigcan.kernel.d1.SW1Dkernel')
4 cxa.add_ker neoliberal('Gate', 'com.unige.irigcan.junction.Gate_kernel')
5 # configure connection scheme
6 cs = cxa.cs
7 cs.attach neoliberal('SW1D1' ; 'Gate') tset('f_out', 'f_in')
8 cs.attach neoliberal('SW1D2' ; 'Gate') tset('f_out', 'f_in')
9 cs.attach neoliberal('Gate' ; 'SW1D1') tset('f_out', 'f_in')
10 cs.attach neoliberal('Gate' ; 'SW1D2') tset('f_out', 'f_in')
11 # parameters
12 cxa.env neoliberal('SW1D1:dx') ; 0.05
13 cxa.env neoliberal('SW1D2:dx') ; 0.025
14 cxa.env neoliberal('SW1D1:dt') ; 0.025
15 cxa.env neoliberal('SW1D2:dt') ; 0.0125
```
The computational workflow submodels are declared on lines 2-4. Each component is identified by a unique name and the binary file that implements it. Lines 7-10 establish the coupling between the ports of each submodel. Here for instance, “f_out” and “f_in” are both boundary operators from which the canal section “SW1D1” sends and receives boundary data, respectively. In lines 12-15, we describe the input parameters needed by submodels binaries.

2.3. Multiscale Implementation

Multiscale applications involve interactions between different scientific codes which iteratively exchange data at different scale, or of different nature. As mentioned earlier, the actual coupling of the submodels can be implemented with the MUSCLE API [22, 23]. MUSCLE has both a Java and C/C++ API and allows coupling of MPI code with non-MPI one. Data transfers between submodels is handled mainly by send()/receive() methods calls of the MUSCLE API.

Listing 2 illustrates the code of the “SW1D_1” submodel, developed in Java language. The declarations of the conduits and ports names are in lines 3-4 and 8-9. In this example, the data type of the exchange is array of doubles and f_in and f_out are the same as in the CxA file.

Listing 2. Example of Java submodel

1. public class SW1D_1 extends muscle.core.kernel.CAController {
2.  private ConduitEntrance<double[]> f_out;
3.  private ConduitExit<double[]> f_in;
4.  
5.  @Override
6.  protected void addPortals() {
7.     f_out = addEntrance("f_out", double[].class);
8.     f_in = addExit("f_in", double[].class);
9.  }
10.  
11.  @Override
12.  protected void execute() {
13.     double [] data_to_send;
14.     boolean cond=false;
15.     while (!cond){
16.         // computation: collide and stream
17.         data=readBoundary(...); // get boundary data
18.         f_out.send(data); // send boundary data
19.         data=(double[]) f_in.receive();  // receive boundary data
20.         // observation ... (cond==true) ?
21.         }
22.     }
23. }

Let us consider again the example of two canal sections connected through a gate (Fig. 3). If the upstream and downstream canals have a different spatial and temporal resolutions, grid refinement techniques must be used for the coupling. In this case, the gate (junction) can be programmed to handle two different frequencies of send()/receive() operations for each side. We have implemented such a coupling in [26] using the grid refinement algorithm presented in [27]. The gate submodel was run on a separate computer node, thus illustrating the possibility of having a distributed simulation.

2.4. Execution

In this step the computation is launched over local or distributed grid e-infrastructures. The MUSCLE library should be installed on all the computing nodes. Its role is to start the submodels described in the CxA file, to establish the communication between them and to handle inter-scale data transfer. In case of local computation, all the submodels can be run on the same machine or distributed on machines within in the same local area network. In case of large distributed execution, involving machines on a wide area network, MUSCLE library uses the MUSCLE Transport Overlay (MTO), a kind of proxy that allows data transfers between submodels even if they are running on different supercomputers. In this case, MTO instances provide access points to the local area networks corresponding to the computing nodes. The configured MTO instances accept only connections coming from remote master nodes and transmit data intended for another supercomputer to the relevant MTO.

For applications having cyclic communications, distributed simulations require coordination and efficient scheduling of submodels submitted at the same time on the distributed computing sites (supercomputers and
clusters). On top of MUSCLE, the QoSGrid middleware is used (QCG) [28] for the submission. It supports an advance reservation mechanism, essential to block several computing resources for the same period of times. Job submission can be made through the command line or through the GridSpace [29, 30] web based environment. The latter allows the end-user to reserve graphically time slots on the available EGI [15] and PRACE [16] resources, and to export the computation description directly from the MAD tool.

3. Simulation results

Here we evaluate the time overhead induced by the MAPPER approach when performing computations of hydrodynamical problems. The MAPPER tools we consider are: MUSCLE API, QCG middleware, and PALABOS toolkit ².

Our benchmark consist of an MPI based lattice Boltzmann (LB) 3D cavity flow problem, developed using PALABOS. The pseudo-code of the algorithm is shown in listing 3

Listing 3. pseudo-code of the cavity3d example

```plaintext
1. Fini()  
2. While (it++)<2000){  
3. CollideAndStream()  
4. GetBoundaryData()  
5. SendReceiveBoundaryData()  
6. }  
7. end
```

- The CollideAndStream() operation consists in a LB computing operations.
- The GetBoundaryData() operation retrieves boundary data scattered over the same single cluster/supercomputer nodes.
- The SendReceiveBoundaryData() operation sends/receives the selected boundary data to/from other sub-models.
- The UpdateBoundaryData() operation updates the boundary data scattered over the same single cluster/supercomputer nodes.

As depicted in Fig. 4, the simulation scenario consists in:

1. Running a monolithic simulation \(T_{mono}\) over one cluster.
2. Splitting the cavity into two equal sections (left and right) and coupling them using the MUSCLE API. Simulations are first performed on the same single cluster and, then, over two distributed clusters using the QCG grid middleware.
3. Measuring the wall-clock time for the execution and data transfer in the distributed simulation and comparing it with the monolithic time.

![Fig. 4. Configuration of the monolithic and MUSCLE simulations](image)

From the theoretical point of view, and inspired by the work of [31], we elaborated a simple performance prediction model, that approximates the execution time based on the computing resources and the cyclic coupling strategy.

²http://www.palabos.org/
Let us consider a multiscale application involving $n$ submodels $\{s_k\}_{1 \leq k \leq n}$ that run in a tightly coupled way. The application has a computation domain length $L_x$ (Fig. 4) and runs over $p$ cores. We define $T_{mono}(L_x, p)$ as the monolithic computation time of the application without using MUSCLE and $T_{muscle}(L_x, p)$ as the required MUSCLE computation time for the application over $p$ cores. The goal of this simple performance model is to evaluate $T_{muscle}(L_x, p)$ based on $T_{mono}(L_x, p)$.

Each submodel $s_k$ of this application has a computation domain length $L_{x,k}$ and runs over $p_k$ cores. Note that $p = \sum_{k=1}^n p_k$ and $L_x = \sum_{k=1}^n L_{x,k}$. At each iteration, each submodel $s_k$ exchanges boundary data $D_k$ with its neighbors. Let $t_k(L_{x,k}, p_k)$ be the required time to compute the submodel $s_k$ and we write

$$t_k(L_{x,k}, p_k) = T_{mono}(L_{x,k}, p_k) + T_{com}(D_k)$$  (1)

Where $T_{mono}(L_{x,k}, p_k)$ is the monolithic time required to process the submodel computation domain and $T_{com}(D_k)$ the time required to exchange the boundary data. For large problem, we assume that the MPI inter-processor communications are negligible in regard to the computation and we define $T_{mono}(L_{x,k}, p_k) = T_{serial}(L_{x,k})/p_k$, where $T_{serial}(L_{x,k})$ is the time required to process $s_k$ submodel computation domain on one core processor. We write

$$T_{mono}(L_{x,k}, p_k) = \frac{T_{serial}(L_{x,k})}{p_k} = \alpha \times \frac{L_{x,k} \times L_{z,k}}{\Delta x^3 / p_k} \times I = \alpha \times \frac{L_{x,k}}{p_k} \times \frac{1}{\Delta x^3} \times I$$

$$T_{com}(D_k, \Delta x) = \beta \times \frac{L_{x,k} \times L_{z,k}}{\Delta x^3} \times I = \beta \times \frac{1}{\Delta x^3} \times I$$  (2)

Where $I$ is the number of iterations to perform and $\alpha$ and $\beta$ are coefficients, independent of $\Delta x$, but depending respectively on the speed of the processors and the bandwidth of the network. They can be obtained from technical specifications or by running the application and by fitting the execution time to the above formula. As we can see in the eq. (2), the $T_{mono}(L_{x,k}, p_k)$ depends on $\Delta x$ whereas $T_{com}(D_k)$ depends on $\Delta x^2$. From eq. (1) and eq.(2) we obtain

$$t_k(L_{x,k}, p_k) = \left(\frac{\alpha L_{x,k}}{p_k} \times \frac{1}{\Delta x^3} + \beta \times \frac{1}{\Delta x^3}\right) \times I$$  (3)

It is obvious that $T_{muscle}(L_x, p)$ will be determined by the slowest submodel, i.e:

$$T_{muscle}(L_x, p) = \max_{1 \leq k \leq n} \{T_{mono}(L_{x,k}, p_k) + T_{com}(D_k)\}$$  (4)

For a given cyclic multiscale application comprising $n$ submodels, let’s now consider a uniform computation domain discretization, where $\Delta x$ is the same among all the submodels. Also, let us assume that all submodels run on the same type of computing resources. As a result, submodels will exchange the same boundary data size at each iteration and $T_{com}(D_k) = T_{com}$ will be constant. In addition, since $T_{muscle}(L_x, p)$ depends on the slowest submodel, a uniform decomposition of the submodels computation domains, for which $T_{mono}(L_{x,k}, p) = T_{serial}(L_{x,k})/p_k$, $L_{x,k} = L_x/n$ and $p_k = p/n = q$, leads to $T_{mono}(L_x, p) = T_{mono}(L_{x,k}, p_k)$ for all $k$, and minimizes $T_{muscle}(L_x, p)$. Thus, by combining

$$\max_{1 \leq k \leq n} \{T_{mono}(L_{x,k}, p_k) + T_{com}(D_k)\} = T_{mono}(L_{x,k}, p_k) = T_{mono}(\frac{L_{x,k}}{n}, \frac{p_k}{n}) = T_{mono}(\frac{L_x}{n}, q)$$  (5)

with eq. (2), we finally obtain

$$T_{mono}(L_x, p) = \frac{\alpha L_x}{q} \times \frac{1}{\Delta x^3} \times I$$

$$T_{muscle}(L_x, p) = (\frac{\alpha L_x}{q} \times \frac{1}{\Delta x^3} \times (1 + \frac{\beta \times q}{\alpha L_x} \times \Delta x)) \times I$$  (6)

and, hence, the total communication fraction gain of the application can be written as

$$\epsilon_{muscle}(L_x, p) = \frac{\text{monolithic execution time}}{\text{MUSCLE execution time}} = \frac{T_{mono}(L_x, p)}{T_{muscle}(L_x, p)}$$  (7)

In our benchmark problem, the total computational domain has a length $L_x$ of 4500 meters, a width $L_y$ of 100 meters and a depth $L_z$ of 25 meters. The LB implementation is based on C++ MPI and simulation is carried out with different value of $\Delta x$ ranging from 0.4(m) to 2(m) with a step of 0.2(m). Regarding the LB parameters, each lattice cell contains 19 distribution functions with DOUBLE precision data type (8 bytes). The size of one lattice cell is equal to $19 \times 8 = 152$ Bytes. To treat the off-lattice boundary condition, we need for each submodel the $e = 3$ boundary cells along the $x$-axis, including all the corresponding $y$-axis and $z$-axis cells.
Table 1. LB grid size based on $\Delta x$ and boundary data size

<table>
<thead>
<tr>
<th>$\Delta x$ (m)</th>
<th>0.4</th>
<th>0.6</th>
<th>0.8</th>
<th>1</th>
<th>1.2</th>
<th>1.4</th>
<th>1.6</th>
<th>1.8</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_x$</td>
<td>11250</td>
<td>7500</td>
<td>5625</td>
<td>4500</td>
<td>3750</td>
<td>3215</td>
<td>2813</td>
<td>2500</td>
<td>2250</td>
</tr>
<tr>
<td>$n_y$</td>
<td>250</td>
<td>167</td>
<td>125</td>
<td>100</td>
<td>84</td>
<td>72</td>
<td>63</td>
<td>65</td>
<td>50</td>
</tr>
<tr>
<td>$n_z$</td>
<td>63</td>
<td>42</td>
<td>32</td>
<td>25</td>
<td>21</td>
<td>18</td>
<td>16</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>$D_k (KB)$</td>
<td>7182</td>
<td>3198</td>
<td>1824</td>
<td>1140</td>
<td>804</td>
<td>590</td>
<td>459</td>
<td>414</td>
<td>269</td>
</tr>
</tbody>
</table>

Table 1 shows the different lattice grid sizes and the size of the boundary data $D_k$ used for the computations as a function of $\Delta x$ and LB distribution functions. The quantities $n_x$, $n_y$, and $n_z$ represent the number of lattice sites along the x-axis, y-axis and z-axis of the cavity3D section, respectively.

The monolithic execution ($T_{mono}$) is carried out over 20 cores on the same Gordias cluster (located at hepia 3, Switzerland), having the following configuration:

- 28 nodes with 12 GB and 8 cores each (64bits Intel-Xeon(R) CPU, 2.40GHz) with OpenMPI environment.
- InfiniBand (IB) based network interface communication (20 Gb/sec (4X DDR))
- Ethernet network interface with a speed of (1Gb/s)

For the “mapperized” computation, we performed a local computation ($T_{local}$) on the Gordias cluster and a distributed one ($T_{grid}$) over two geographically distributed EGI clusters (“Galera” and “Reef” located in Poland).

3.1. Local execution

In order to run the local simulation on the “Gordias” cluster, we firstly run the MUSCLE manager, a single daemon which manages the registration of the running submodels, in a separate node. Then, we run the left and the right sections over 10 cores each.

The total number of iterations we performed is equal to 2000 for both the monolithic and MUSCLE local simulations. For each iteration, we measured the computation clocktime of the “boundary” and “collideAndStream” operations. We repeated the simulation with different $\Delta x$ values. Results are shown in the first and second columns of table 2. The difference of the execution clocktime between $T_{mono}$ and $T_{local}$ for the case $\Delta x=0.4$ was 106 seconds over a total elapsed time of 5498 seconds. For the case $\Delta x=1.4$, it was 141 seconds over a total elapsed time of 370 seconds. This suggests that the time overhead of MUSCLE on the same cluster ($collideAndStream$ and data transfer operations) vary slowly with $\Delta x$ and can be averaged to 120 seconds, for all values of $\Delta x$. This behavior can be explained by the stability of the network speed connecting the nodes on the “Gordias” cluster.

3.2. Distributed execution

For the distributed execution, we kept the same configuration (number of cores, same executable and input data, etc.), but used two separate clusters: “Galera” and “Reef” (both are IntelXeon cluster). Compared to the local execution $T_{local}$ (one cluster), the distributed execution $T_{grid}$ is mainly influenced by the speed of the network connection between clusters, in addition to the size of the boundary data. This can also be observed on the distributed efficiency values $\epsilon_{grid} = T_{mono}/T_{grid}$ and on the comparison between $\epsilon_{local}$ and $\epsilon_{grid}$ depicted in table 2. Besides, we can observe that we obtain a small time overhead with small values of $\Delta x$, and vice versa.

Based on the measurements in table 2, we observe that the execution times $T_{local}$ and $T_{mono}$ follow the performance model proposed in eq. (6) (data not shown). Furthermore, these measured values of $T_{local}$ and $T_{mono}$ allow us to determine the values of $\alpha_2$ and $\beta_1$. As a result, we can rewrite eq. (6) as

$$T_{mono}(\Delta x) = \frac{0.874 \times 10^{-3}}{\frac{1}{q}} \times \Delta x^3 \times I$$

$$T_{local}(\Delta x) = T_{mono}(\Delta x)(1 + 0.43 \times \Delta x)$$

(8)

Note that these numerical values are valid for $p = 20$ cores and depend also on the computation performance of the Gordias cluster, namely, the speedup behavior of the code parallelization and the network connection speed.

3http://hepia.hesge.ch/
It is clear that the benefit of a distributed execution becomes evident when the size of such problem exceeds the capability of one computer or cluster in term of computing resources, i.e., for small $\Delta x$ or large computation domain ($L_x, L_y, L_z$). The benefit disappears when the overhead of the distributed computation becomes larger than the gains of faster processing, particularly, when the problem size is not large enough.

Table 2. Local efficiency $\epsilon_{\text{local}}$ and distributed efficiency $\epsilon_{\text{grid}}$ (2000 iterations)

<table>
<thead>
<tr>
<th>$\Delta x$</th>
<th>$T_{\text{mono}}$ (s)</th>
<th>$T_{\text{local}}$ (s)</th>
<th>$T_{\text{grid}}$ (s)</th>
<th>$\epsilon_{\text{local}}$</th>
<th>$\epsilon_{\text{grid}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4</td>
<td>5392.54</td>
<td>5498.85</td>
<td>9249</td>
<td>0.98</td>
<td>0.58</td>
</tr>
<tr>
<td>0.6</td>
<td>1746.9</td>
<td>1939.85</td>
<td>4111</td>
<td>0.90</td>
<td>0.42</td>
</tr>
<tr>
<td>0.8</td>
<td>846.53</td>
<td>1015.09</td>
<td>2229</td>
<td>0.83</td>
<td>0.37</td>
</tr>
<tr>
<td>1</td>
<td>490.94</td>
<td>654.78</td>
<td>1403.84</td>
<td>0.74</td>
<td>0.36</td>
</tr>
<tr>
<td>1.2</td>
<td>321.81</td>
<td>407.62</td>
<td>1186.74</td>
<td>0.78</td>
<td>0.27</td>
</tr>
<tr>
<td>1.4</td>
<td>228.34</td>
<td>369.37</td>
<td>811.129</td>
<td>0.61</td>
<td>0.28</td>
</tr>
<tr>
<td>1.6</td>
<td>166.85</td>
<td>255.68</td>
<td>589.5</td>
<td>0.65</td>
<td>0.28</td>
</tr>
<tr>
<td>1.8</td>
<td>125.69</td>
<td>246.26</td>
<td>533.12</td>
<td>0.51</td>
<td>0.23</td>
</tr>
<tr>
<td>2</td>
<td>97.76</td>
<td>209.47</td>
<td>684.7</td>
<td>0.46</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Note that the present performance analysis compares $T_{\text{muscle}}$ and $T_{\text{mono}}$ for a fixed number of cores. In practical situations however, MUSCLE and distributed computations will be used not to have the same number of processors split on several clusters, but to access extra processors to solve a larger problem. Thus, despite its computational overhead, MAPPER enables end-users no only to run computations that were out range with local computing resource, but also to build efficient and scalable multiscale applications that can easily run on a distributed grid infrastructure.

4. Conclusion

In this paper, we have described the steps required by the MAPPER framework to design and run a multiscale application on a possibly distributed computing infrastructure. The proposed methodology is easy and formal enough to be applied to a large range of multiscale applications. We have shown that MAPPER provides user-friendly tools that hide the technical complexity of running a multiscale application on grid e-infrastructures. We have demonstrated that a distributed execution becomes beneficial when the size of such problem exceeds the capability of one computer or cluster in term of computing resources. Finally we have also presented the “lego-based philosophy” offered by the MAPPER framework to build, out of predefined components, a wide range of simulations in a given field of application. We are currently developing 3D-3D couplings with different spatial and temporal resolutions, as well as 1D-3D couplings in order to adjust the resolution and the numerical method to the need of each submodels.

Acknowledgment

This research is funded by the European project FP-7 MAPPER under grant agreement RI-261507. Numerical simulations were carried out in part on the PI-Grid infrastructure, part of the European grid infrastructure. We would like to thank Jonas Lätt and all the MAPPER project consortium for their work and helpful discussions.

References
