COMPAT

Computing Patterns
for High Performance
Multiscale Computing

D2.2 First Report on Multiscale Computing Patterns and Algorithms

Due Date	Month 18 (first submission) Month 24 (resubmission)
Delivery	Month 18 +1week, PO agreed (first submission) Month 23 (resubmission)
Lead Partner	UvA
Dissemination Level	PU
Status	final
Approved	Internal review yes
Version	2.0

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 671564.

DOCUMENT INFO

Date and version number	Author	Comments
22.03.2017 v0.1	Alfons Hoekstra	Skeleton of report, first intro text.
23.03.2017 v0.2	Saad Alowayyed	Partial sections 2.3 and 2.4
24.03.2017 v0.3	Alfons Hoekstra	Update, including input by partners
27.03.2017 v0.4	Saad Alowayyed	Summary and conclusion, improved pictures, added more input from pattern taskforce
28.03.2017 v0.5	Alfons Hoekstra	More additions to 2.3 and 2.4, links to other WPs
29.03.2017 v0.6	Saad Alowayyed	Cleaning up, submit for internal review
04.04.2017 v1.0	Alfons Hoekstra	Including feedback by internal reviewers, final document to project manager for submission
31.08.2017 v.2.0	Alfons Hoekstra	Revision of submitted D2.2 (v1.0) to incorporate M18 reviewers' comments

CONTRIBUTORS

The contributors to this deliverable are:

Contributor	Role
Alfons Hoekstra (UvA)	PI and WP leader
Saad Alowayyed (UvA)	contributor
James Suter (UCL)	contributor
Oliver Hoenen (MPG)	contributor
Derek Groen (UBRUN)	Contributor
David Coster (MPG)	Internal reviewer
Tomasz Piontek (PSNC)	Internal reviewer

TABLE OF CONTENTS

1 Executive summary

After a short review of the Multiscale Computing Patterns, this deliverable introduces in more detail the specific architecture of the Multiscale Computing Patterns software and its components, in line with the overall ComPat architecture as described in Deliverable D5.1. The Multiscale Computing Patterns and Algorithms software consists of *description*, *optimisation* and *service* parts. The *description* part is where the task graph, entered by the user using xMML, is translated to understand the type of pattern. Moreover, this part includes submodel definitions. The *optimisation* part is where, after detecting the type of the pattern in the description layer, algorithms are chosen and applied to find the most suitable mapping between submodels and HPC resources. Finally, the *service* part is the middleware layer, where the submodels are mapped to number and type of physical resources and distributed, based on the suggestions from the *optimisation* part and other execution elements such as queueing time and availability of resources. Three examples of using the Multiscale Computing Patterns and Algorithms software are illustrated, and examples of cost functions are worked out, showing that a wide range of variables for Multi-objective Optimisation algorithms can be chosen. The idea is that Multiscale Computing Patterns software will automatically detect which cost functions and algorithms to pick, based on the type of the pattern and user requirements.

2 Main body of the report

2.1 Introduction

The goal of Work Package 2 is, quoting from Part A of the DoA, to “create a general mapping from a multiscale model to a multiscale computing pattern. Reusable software components will be created for each of the three computing patterns. MML specifications of multiscale models will be modified to include these components, and the modified MML will be converted to input for the high-level tools (WP4) and middleware (WP5). The performance of the patterns will be tested and predicted.”

This deliverable D2.2 should then “report on the algorithms and components used to construct the Multiscale Computing Patterns, details of the actual implementation, and preliminary results on performance measurements.” and builds upon deliverable D2.1 on Existing Software Suitability and Adaptation. It is mainly based on work performed in task 2.3 (Development of the multiscale computing patterns) but also on task 2.2 (Implementing a method for converting MML for ComPat) and 2.4 (profiling and performance measurement).

Please note that this deliverable D2.2 was originally scheduled in M24, but because of the change of the reporting period from 12 months to 18 months during the grant preparation phase, was moved back from M24 to M18. As consequence, we cannot yet report on "preliminary results on performance measurements", as the corresponding task 2.4, although started in M13, has not yet produced tangible results. We do refer to deliverable D4.2 where the first important results on profiling and measuring performance of multiscale applications are reported, and where input the type of information that will be needed by the Multiscale Computing Patterns was provided by WP2. WP4 delivers the tools that we need in task 2.4 to allow us to carry out the intended performance measurements.

In D2.1 we wrote that "converting MML for ComPat, task 2.2, has partly been addressed" and "…we will therefore continue to put effort in this task, slightly deviating from the original DoA and extending this task into the second year of the project". We have now reached a point, given the current design of the Multiscale Computing Patterns (MCPs), that the information needed by them should not be embedded in MML, but provided as separate data files. This specifically relates to performance data of the single scale models, as described in section 2.3. We have however decided to not close task 2.2 yet, as e further development of the MCPs may warrant updates to MML.

The development of the actual patterns, task 2.3, has been the major focus of WP2 during M13 to M18 of the project. Building upon the conceptual design of MCPs as reported in D2.1, and in collaboration with WP3 and WP5, we have now designed a specific architecture for the Multiscale Computing Patterns and Algorithms software, in line with the overall ComPat architecture described in deliverable D5.1. We partly implemented its main features, and worked out specific examples to test these concepts, and thus created first implementations of the MCPs.

Milestone 8 (Month18): *First prototype of the ComPat multiscale development and execution environment. First version of MML specifications of Multiscale Computing Patterns* has been reached, in the sense that we have implemented first version of Multiscale Computing Patterns and Algorithms software, and in the sense that we have now concluded that adaptations of MML are not (yet) required and that MML descriptions of our multiscale applications are sufficient to specify MCPs. For details, we refer to section 2.3.

This deliverable will first quickly review the MCPs (as described in detail in deliverable D2.1), then introduce specific Multiscale Computing Patterns and Algorithms software and describe its components, and finally describe three examples of using the Multiscale Computing Patterns and Algorithms software.

We would like to point out that this deliverable D2.2 with no doubt reflects original results obtained in ComPat in the first 18 months of the project and should be considered 100% ComPat foreground knowledge. Section 2.2 shortly summarizes the theory behind Multiscale Computing Patterns, as reported in deliverable D2.1 (and also being 100% ComPat results). It then continues in section 2.3 with the design and implementation of the MCP algorithms and software and in section 2.4 with case studies of acutally using the software for two MCPs. All of this are original ComPat results, and there is actually no overlap whatsoever with earlier obtained results in the MAPPER project.

2.2 Multiscale Computing Patterns

As a courtesy to the reader, and to keep this deliverable self-contained, this section provides a short summary of Multiscale Computing Patterns, as described in detail in deliverable D2.1. We do refer the reader to deliverable D2.1 for all details, including worked out examples.

We define Multiscale Computing Patterns (MCP) as high-level call sequences that exploit the functional decomposition of multiscale models in terms of single scale models. We have identified three computing patterns that we believe are most relevant for high performance multiscale computing, namely

- Extreme Scaling (ES),
- Heterogeneous Multiscale Computing (HMC), and
- Replica Computing (RC).

The *Extreme Scaling* computing pattern represents a specific class of multi-scale applications where one (or perhaps a few) of the single scale models in the overall multiscale model dominates all others, in terms of computational and/or energy cost, by far. Such a dominating *primary model* is expected to scale to very large systems (i.e., multi-petascale or above) and the efficiency of the primary model largely determines the efficiency of the multiscale application. Consequently, one of our goals is to ensure minimal interference by the other single scale models, so-called *auxiliary models*. These typically have a much lower computational and/or energy cost, and could even be sequential codes. Load-balancing, decentralized communication, and computation overlapping are some of the techniques we can use here, depending on the relation between the primary and auxiliary models.

In the *Heterogeneous Multiscale Computing* pattern, we couple a macroscopic model to a large and dynamic number of microscopic models. The basic philosophy is to apply a numerical solver to the macroscale equations and to provide the missing macroscale data using an appropriate microscale model. The number of microscale models required in HMC depends on spatial properties of the macroscale model, and can in some cases easily be in the order of 10^7 or more. The large number and size of the microscale models causes them to dominate the computational and energy cost of the multiscale application, and are therefore cost-critical.

Replica Computing is a multiscale computing pattern that combines a potentially very large number of terascale and petascale simulations (also known as 'replicas') to produce scientifically important and statistically robust outcomes. The replicas are not part of a larger spatial structure (as is the case in, for example, Heterogeneous Multiscale Computing), but they are applied to explore a system under a broad range of conditions. Replica Computing is set up through an initialization stage, which determines the simulations required to explore or incorporate a given parameter space. This initialisation is then followed by one or more sequences of simulation and data processing.

An important result from deliverable D2.1 was the realisation that the MCPs will be expressed on the level of the task graph. The key idea is that we define generic task graphs for each MCP, such that application specific task graphs can be embedded in the generic task graphs. We use the generic task graph to obtain an optimized mapping of the application to an HPC resource, and try to find generic algorithms for this. What exactly is meant by an ‘optimal’ mapping needs to be defined, or can be made application specific. In any case, it should be optimized with respect to several dimensions (efficient use of resources, power consumption, wall clock time, load balancing, fault tolerance). The way we proceed is that for each generic task graph we will specify sets of optimal execution profiles, or define constrained optimization problems that should be easily solvable when fed with details of the specific applications. An MCP therefore is a tuple of a generic task graph plus data or models on the performance of single scale models, a specification of a specific multiscale application in terms of the xMML and a set of algorithms and heuristics that combine this into detailed input/configuration files for the execution environment.

We have worked out generic task graphs for the *Extreme Scaling* (ES), the *Heterogeneous Multiscale Computing* (HMC), and *Replica Computing* patterns, see deliverable D2.1.

In the generic task graph for ES, a collection of auxiliary models can either be executed in parallel with the primary model, or in series with the primary model. Depending on the execution behaviour of the primary and auxiliary models on HPC machines, a specific execution of the ES graph is considered (for an in-depth example we refer to deliverable D2.1).

For HMC a large and dynamic number of microscale simulations is coupled with one macroscale model, with a database in between. The role of the database is to prevent computing of previously computed results, to interpolate between earlier computed results, and to submit microscale simulation jobs when needed.

For RC we find two variants, which capture the behaviour of the three types of replica computing that we defined. In both cases a potentially large set of replicas are executed independently and then feed into a second master process. In RC, if a replica fails, a restart is not immediately needed, as long as the overall statistical quality of the ensemble that is computed by the RC application is maintained. This is a main distinction with HMM, where if a microscale simulation fails it must be restarted, as the database requested output from this microscale simulation. So, although in terms of load balancing the HMC and RC patterns are pretty close, in terms of fault tolerance they have quite different constraints.

2.3 The Multiscale Computing Patterns and Algorithms software

2.3.1 Design

The Multiscale Computing Patterns and Algorithms software consists of three parts, namely the submodels *Description Part* (the user input of the multiscale computing patterns software), the *Optimisation Part* (performance oriented services and tools) and the *Services Part* (underlying resource allocation service, i.e. Quality in Cloud and Grid "QCG"). Figure 1 shows the relationships between the different parts and their components. At the top, the description of the multiscale model to be executed is shown, with the task graph expressed in textual form, submodel definitions, and simulation input and configuration data. The description part also includes any restrictions in relation to resources that the single scale simulations can use. These are fed into a translation step to convert them from their original formats into a format suitable for the patterns performance services.

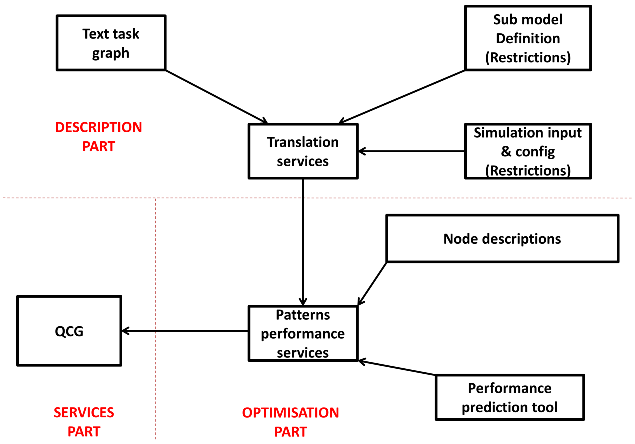

Text task graph
Sub model Definition (Restrictions)
DESCRIPTION PART
Translation services
Simulation input & config (Restrictions)
Node descriptions
QCG
Patterns performance services
Performance prediction tool
SERVICES PART
OPTIMISATION PART

Figure 1: Architecture of the Multiscale Computing Patterns and Algorithms software

The optimisation part contains the patterns performance services and performance prediction tool. The performance prediction tool collects measurements of performance and efficiency of the submodels under various execution scenarios and uses that information to compute execution on available resources. These data are the input of the patterns performance services, together with a description of the available classes of compute resources (e.g. CPU nodes, GPU nodes, high-memory nodes, etc.). The patterns performance services then, using the requested performance metrics requested by the user (such as the multiscale efficiency, throughput, fault tolerance, energy usage, or a combination), suggests a small collection of most suitable execution scenarios to QCG for execution. QCG will then select the most optimal one, also considering predicted queuing times for each execution scenario. Finally, the execution scenarios themselves are based on the MCPs and the underlying generic task graphs, as discussed in deliverable D2.1.

In the next part of this section, we will illustrate an overview of how the Multiscale Computing Patterns and Algorithms software would work, and describe in more detail their components. We will provide more information on how we designed the components, and the status of these components. In deliverable D3.2 more detail of instantiations of specific applications using the Multiscale Computing Patterns and Algorithms software can be found. D3.2 also provides the user perspective of the software, what are the tools, what to do as a user, etc. Here we provide more detail from a design perspective.

2.3.2 Description Part

This part provides the logical description and the requirements of a multiscale application. It builds on and uses formats and software from the Multiscale Modelling and Simulation Framework (that was described in deliverable D2.1). Three data files make up the description part, which are the *text task graph*, the *submodel definitions* and the *simulation input and configurations*. This information is converted by the translation services into a set of standardised inputs for the patterns performance services. Below we provide more information on each of these parts.

1. Text task graph

   This file holds a pseudo representation of the task graph in the form of a highly adapted xMML. One important issue here is to identify which information to retrieve. Currently this is:

   - Motif components, which are the submodels and their dependencies,
   - Type of the pattern (i.e. ES, RC or HMC),
   - Multiscale model time, a formula composed by the translation service to express the overall time of the multiscale applications in terms of the execution time of the single scale models and the specific multiscale computing pattern.

2. Submodels definitions

   This file gathers all the information required for the single-scale model to run, in other words, all the requirements needed to be able to execute the multiscale simulation. This includes:

   - models to be loaded,
   - submodel specific applications,
   - specific number of cores (or set of cores) per submodel, if required,

- set of restrictions on the level of the submodel such as available environment modules, specific library, allowed resources, etc.

This could rely on earlier developed MAPPER tools such as MAD/MaMe[1]. MaMe, the Mapper Memory, is a database application for storing descriptions of submodels, mappers and filters, and MML graphs in xMML format. MAD (Multiscale Application Designer) is a graphical development environment for multiscale models. It enables the user to draw an MML diagram, using submodels from MaMe or defining them directly by writing code. MaMe could thus be a source of submodel descriptions for input into the multiscale computing patterns software. Currently we do not use these tools, but in the next phase of the project we intend to investigate if and how the tools could be used to integrate the multiscale computing patterns software into the overall Multiscale Modelling and Simulation Framework

3. Simulation input and configuration file

   Here, the user specifies a path to a folder that contains configuration files and input parameters for a multiscale application. Users specify the required total number of cores for the simulation. These submodels and configuration files will be the input required for the optimisation part. Also, a set of restrictions on the whole multiscale simulation have to be defined in a generic way. These restrictions refer to e.g. available environment modules, specific library (e.g. MUSCLE2), middleware used, allowed resources, etc.

4. Translation services

   The purpose of the translation services is to gather the user-supplied input (task graph for the simulation, simulation configurations and restrictions) with submodel information (definitions and restrictions) to identify the patterns contained within the task graph and communicate the required information to the optimisation part of Multiscale Computing Patterns and Algorithms software. Currently, the translation tool is application- specific, but in the second phase of the project, we will develop patterns recognition to identify the multiscale computing patterns present in the task graph. The output of the translation service are two xml files, specific for the multiscale pattern. One file (*matrix.xml*) stores all information about single scale submodels and components: name of the submodels, codes instantiating each submodel and for each code, possible restrictions, information about resources where it is available, benchmarking details and initial performance results. The other one (*multiscale.xml*) contains information about the multiscale model expressed as a set of single scale submodels coupled together: choice of codes which implement each submodels, coupling topology (by pairing submodels), and specific information on the application (paths, inputs, environment, pre and post-processing steps). For clarity, we choose to use one xml file for submodule level of information (submodel definitions and restrictions and the performance matrix per submodel) as shown in
   Listing **1**. The second output concerns the coupled multiscale information (simulation information and QCG script data) in Listing 2.

[1] http://www.mapper-project.eu/web/guest/mad-mame-ew

```
<submodel name="turb" class="MPIKernel">
            <instance name="GEM">
                <restrictions>
                    <cpu>
                        <number> (2^x)*k:k=8 </number>
                        <min_cores> 64 </min_cores>
                        <max_cores> 2048 </max_cores>
                    </cpu>
                </restrictions>

                <benchmark_input>
                    <file type="URL">gsiftp://input_to_GEM.xml</file>
                    <iterations> 1000 </iterations>
                </benchmark_input>
                <scalability_formula> NY </scalability_formula>

                <available_resources>
                    <resource name="supermuc" nodeType="thin"/>
                    <resource name="eagle" nodeType="haswell_128"/>
                </available_resources>

            </instance>
        </submodel>
```

Listing 1: Submodel definitions (matrix.xml), this snippet shows the part of defining a singlescale model (turbulence) as a part of the multiscale fast track (fusion).

```
<multiscale>
    <info>
        <job appID="compat-test" project="compat">
            <computing> ES </computing>
            <modeltime> ETS + CHEASE + GEM </modeltime>
            <task persistent="true" taskId="ES-task-1">

                <numberofcores>
                    <min> 64 </min>
                    <max> 2048 </max>
                </numberofcores>
            </task>
        </job>
    </info>

  <topology>
    <instance id="CONTINUE" helper="init"/>
    <instance id="ETS"      helper="transp"/>
    <instance id="GEM"      submodel="turb"/>
       ...
    </topology>

    <middleware name="QCG">

        <execution type="compat">
            <executable>
                <application name="muscle2" version="compat-1.2"/>
            </executable>
```

Listing 2: multiscale coupling file (multiscale.xml), the snippet shows the input requirements to run the multiscale application, such as minimum and maximum number of cores required by the user to run a simulation, instances, coupling topologies and middleware specific requirements.

2.3.3 Optimisation part

The optimisation part operates after receiving the required input from the description layer. This part can be logically divided into three main components, namely *node descriptions*, *patterns performance services* and *performance prediction model*.

1. Node descriptions

   In these descriptions, we specify a node in two way:

   - Node type: as a spefic physical hardware configuration,
   - Node class: as a logical group of node types equivalent from the point of view of the performance of selected criterion of the sub-model.

     a. Node type

     A single node type represents a set of the same or very similar physical computation nodes, basically the same hardware configuration including interconnects. The types of nodes should be defined based on knowledge about resources gathered a priori by the multiscale computing patterns software from the infrastructure provider. Table 1 shows the current node types used in this phase of the project.

Table 1: node types, an input to the patterns performance

Type name		**# of nodes**	**Processors / node**	**Cores / node**	**Threads / node**	**RAM / node**	**Processor name**
host	**Type**					**(GB)**	
eagle	**haswell_64**	492	2	28	28	64	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
	haswell_128	460				128	
	haswell_256	52				256	
supermuc	**Thin**	9216	2	16	32	32	Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz
	Fat	205	4	40	80	256	Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz
stfc	**Default**	118	2	16	16	64	Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz

b. Node class

Node classes are sets of different node types where computations of a single kernel have a comparable performance. Node class is specific for every single scale model, as different single scale models usually achieve different performance on the same type of nodes. The node class will be created by the patterns performance services during creation of a single scale model performance matrix. During this phase, the single scale model restrictions that might limit the set of the available types of nodes is also taken into account.

2. Patterns performance services

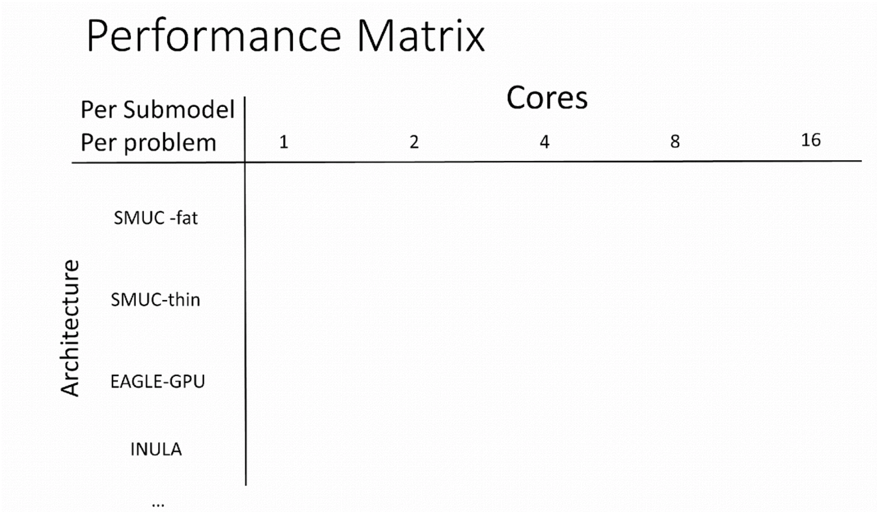
The patterns performance services combine two main components:

- Performance Matrix
- Performance Model

Each component deals with a different set of issues. The Performance Matrix holds the data required for optimisation. In the performance model, requested criteria are computed (e.g. overall efficiency, etc.) on the set of execution scenarios received from the performance matrix.

a. Performance Matrix

The performance matrix, shown in Figure 2, is the main data structure to be used in the performance model. Currently, this matrix contains only wall clock time information from the submodels. This is ideally done by measuring the overall time of the submodel using 1 to n cores for the first node, followed by measuring or estimating the overall time of the submodel utilising 2 to N nodes. Note that production codes may usually not fit in one node at all, in which case this will become a constrained on that specific single scale model and the performance matrix will hold data only for the minimal possible number of nodes N_{min} to N nodes. Likewise, other codes may only run up to a maximum number of nodes N_{max}, again leading to a constrained on that single scale model. These performance measurements should be repeated for all available types of nodes. Based on performance results, types of nodes will be grouped into classes, where a single class will contain types of nodes with comparable performance. Moreover, we propose to construct one performance matrix per submodel per problem size to fulfil all the requirements and to supply the patterns performance services with the ability of scaling with different problem sizes, using performance models that will be developed in the second phase of the project. This could e.g. be done using interpolation as a combination of problem size of the multiscale model and the resources, relying on performance models. The matrix might contain specially marked values (e.g NA), for type of nodes where a specific single scale model is not supported (see also discussion above). Note that such information is available in the “sub-model definition” component. Listing 3 shows an example.



Performance Matrix

	Per Submodel Per problem	Cores				
		1	2	4	8	16
Architecture	SMUC -fat					
	SMUC-thin					
	EAGLE-GPU					
	INULA					
	...					

Figure 2: Performance Matrix layout (table is intentionally left empty)

```
        <performance>
        ...
        <submodel name="turb">
            <instance name="GEM">
                <resources name="supermuc">
                    <nodeType> thin </nodeType>
                    <numberOfCores> 64;128;256;512;1024;2048</numberOfCores>
                    <wallClockTime> 767.9; 408.5; 197.2; 93.7; 45.2; 32.4
                    </wallClockTime>
                    <numberOfNodes> 4;8;16;32;64;128</numberOfNodes>

                </resources>

                <resources name="eagle">
                    <numberOfCores> 64;128;256;512;1024;2048</numberOfCores>
                    <wallClockTime> 863; 517; 255; 85; 44; 34</wallClockTime>
                    <numberOfNodes> 3;5;10;19;37;74</numberOfNodes>
                </resources>

            </instance>
        </submodel>

    </performance>
```

Listing 3: The performance section (in matrix.xml) for the earlier shown submodel (turb). Two architectures are shown "supermuc" and "eagle". In this benchmark a node type in supermuc is considered namely "thin" node type. In eagle, another one node type "haswell_128" was used to measure the performance of the benchmark.

b. Performance Model

The main aim of the performance model is to provide a mapping of submodels to classes of nodes, based on the available performance data, given the specific MCP, and based on user defined criteria (w.r.t. e.g. energy usage, fault tolerance, or other criteria). The mapping will be established by computing a cost function, given the user requirements. The cost function that we so far applied is the overall efficiency (see section 2.4), because larger efficiency implies better usage of resources. In the second phase of the project, we will turn to more involved scenarios, also taking into account energy awareness and fault tolerance. The patterns performance service will generate a small number of alternative execution tuples, allowing QCG to pick the one that can be deployed and executed given actual status of the resources. An Execution Tuple will hold, for every kernel K_i, implementing single scale model i of the multiscale model, the number of required cores N_i, and the required node class C_i. The importance of the alternatives here is to give the middleware the freedom to choose from a set of resources with comparable performance per kernel. This component is subject to ongoing development and will be enhanced with relevant features during the second phase of the project. We will enhance this component to extend the patterns with capabilities to also consider issues related to energy awareness and fault-tolerance. For each pattern we will formulate constrained optimization problems that as output will deliver alternative execution profiles to QCG.

3. Performance Prediction tool

   This tool will be based on Performance Matrices and will provide information about performance of kernels (submodels) on a specific number of cores/nodes. For now, a Performance Matrix is filled with exact values measured during execution of kernels. In the final implementation, it will be possible to make an optimisation of the construction of the performance matrix by interpolating some values and avoiding extra calculations. Note that exactly in this tool we can also include performance predictions for non-existing emerging exa-scale configurations and as such assess how MCP could optimally benefit from such hypothetical machines.

```
<plans>
  <plan id="plan1">
    <criteria>
      <time>PT0H10M</time>
    </criteria>
    <group>
      <kernel refid="GEM"></kernel>
      <class refid="c1">
        <cores>1024</cores>
      </class>
    </group>
  </plan>
```

Listing 4: A snippet of patterns performance services output, a plan is a group of single scales models (can be the whole multiscale model or part of it) with a number of classes of nodes and cores assigned to each submodel in a group. This is input of the services part of multiscale computing patterns software. For details of plans, we refer to D5.2.

The output of the patterns performance services will be several allocation plans, where a plan is a specific mapping of the multiscale model to resources. Details of these plans (that is, the input to QCG) is described in D5.2.

Listing **4** shows the plan part of this output. Note that in this example we show one allocation plan, whereas in reality the output will typically have in the order of three such plans.

2.3.4 Services part

The main component here is the middleware, which is extensively described in deliverables from WP5. The service part will select the best allocation plan based on the Execution Tuples presented earlier by mapping of computational kernels to the concrete number of physical resources of a specific type and then pick the efficient and reliable execution of the application on the distributed heterogeneous infrastructure (EEE). Moreover, this choice is based on other factors such as availability of the requested classes of nodes and average queueing time for a specific job. For more details on the service provided by the tool, we refer to D5.2.

2.4 Using the Multiscale Computing Patterns and Algorithms software

2.4.1 Introduction

In this section, the input files in the description part, some of the calculations of the patterns performance services and the output to QCG for the Extreme Scaling computing pattern (based on MUSCLE2) and Replica Computing pattern (Binding Affinity Calculator (BAC) based on scripts) are presented. For details of specific applications as well as a user perspective of using the software we refer to deliverable D3.2.

2.4.2 Extreme Scaling

It all starts with a formal description of the task graph using the XMML format, which describes each submodel, their time and space scales and inputs/outputs. This can be transformed into different files by different tools of the translation services:

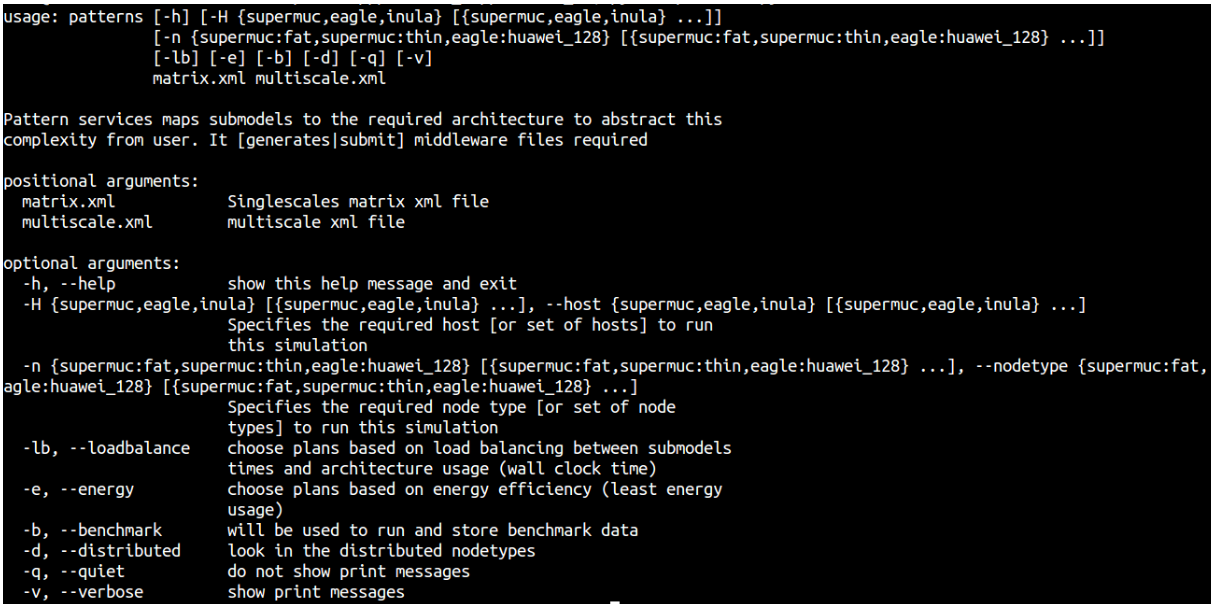
- Using the jmml tool[2], the skeleton of a MUSCLE2 configuration file can be produced (cxa file in Ruby) to assist the developer when implementing the targeted MUSCLE2 application (optional in case when the cxa file was already implemented). In addition, to give a visual representation of the coupled application, the scale separation map, the

[2] The Java Multiscale Modeling Language implementation, developed in the MAPPER project, see https://github.com/blootsvoets/jmml

coupling topology and a task graph representing a few iterations can be generated also with jmml.

- Using a generic python script, a skeleton for both matrix.xml and multiscale.xml can be inferred from the XMML description. The developer can complete them by adding all required information, as described in the previous section.

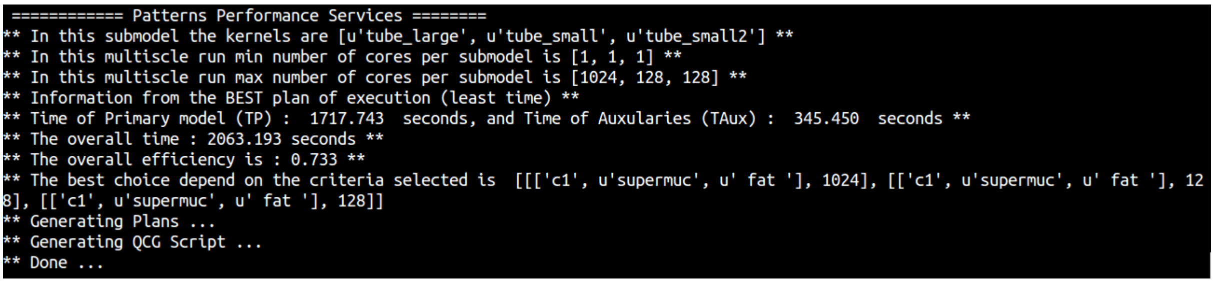
Next, by using the command line, with input as shown in Figure 3, a user can pass both matrix.xml and multiscale.xml to the patterns performance services. Along with this input, the user will have the ability to specify the benchmark flag '-b', which will run a set of ready-made benchmark files to fill the performance matrix automatically. Note that not all functionality is available yet, we will make this available in the next phase of the project.



```
usage: patterns [-h] [-H {supermuc,eagle,inula} [{supermuc,eagle,inula} ...]]
                [-n {supermuc:fat,supermuc:thin,eagle:huawei_128} [{supermuc:fat,supermuc:thin,eagle:huawei_128} ...]]
                [-lb] [-e] [-b] [-d] [-q] [-v]
                matrix.xml multiscale.xml

Pattern services maps submodels to the required architecture to abstract this
complexity from user. It [generates|submit] middleware files required

positional arguments:
  matrix.xml            Singlescales matrix xml file
  multiscale.xml        multiscale xml file

optional arguments:
  -h, --help            show this help message and exit
  -H {supermuc,eagle,inula} [{supermuc,eagle,inula} ...], --host {supermuc,eagle,inula} [{supermuc,eagle,inula} ...]
                        Specifies the required host [or set of hosts] to run
                        this simulation
  -n {supermuc:fat,supermuc:thin,eagle:huawei_128} [{supermuc:fat,supermuc:thin,eagle:huawei_128} ...], --nodetype {supermuc:fat,
agle:huawei_128} [{supermuc:fat,supermuc:thin,eagle:huawei_128} ...]
                        Specifies the required node type [or set of node
                        types] to run this simulation
  -lb, --loadbalance    choose plans based on load balancing between submodels
                        times and architecture usage (wall clock time)
  -e, --energy          choose plans based on energy efficiency (least energy
                        usage)
  -b, --benchmark       will be used to run and store benchmark data
  -d, --distributed     look in the distributed nodetypes
  -q, --quiet           do not show print messages
  -v, --verbose         show print messages
```

Figure 3: current and proposed command line inputs

Before submission to QCG, the user specifies the required number of cores for the overall multiscale simulation (in *multiscale.xml*). Next, the patterns performance service generates a list of possible resource allocation plans. Based on the criteria chosen by the user, a first set of scenarios can be immediately excluded. Then, the rest are examined against the selected cost function (see below). The best three scenarios (and a combination of node classes) are then sent to QCG as multiple plans. The patterns performance services output for the SandBox application (see deliverable D3.2) is shown in Figure 4. In this run, we used the sandbox application and specified minimum time as a cost function to generate suggestions to QCG.



```
 ============ Patterns Performance Services ========
** In this submodel the kernels are [u'tube_large', u'tube_small', u'tube_small2'] **
** In this multiscle run min number of cores per submodel is [1, 1, 1] **
** In this multiscle run max number of cores per submodel is [1024, 128, 128] **
** Information from the BEST plan of execution (least time) **
** Time of Primary model (TP) :  1717.743  seconds, and Time of Auxularies (TAux) :  345.450  seconds **
** The overall time : 2063.193 seconds **
** The overall efficiency is : 0.733 **
** The best choice depend on the criteria selected is  [[['c1', u'supermuc', u' fat '], 1024], [['c1', u'supermuc', u' fat '], 12
8], [['c1', u'supermuc', u' fat '], 128]]
** Generating Plans ...
** Generating QCG Script ...
** Done ...
```

Figure 4: patterns performance services commands standard output

2.4.2.1 Cost function(s)

Initially, we analyse the pattern of the multiscale application and currently manually decide which cost function is suitable. This will be automated at a later stage of the project. In this subsection, we will show initial trials of the cost functions used per application.

Figure 5 shows the task graph for the fusion fast track application. It falls in the class of an Extreme Scaling application with serial auxiliary models. For this serial execution, we will use the total time of execution as the cost function. However, we will also include efficiency as a proof of concept. Figure 6, illustrates the time and efficiency of the primary model, while the other single scale models (ETS and CHEASE) are serial auxiliaries.

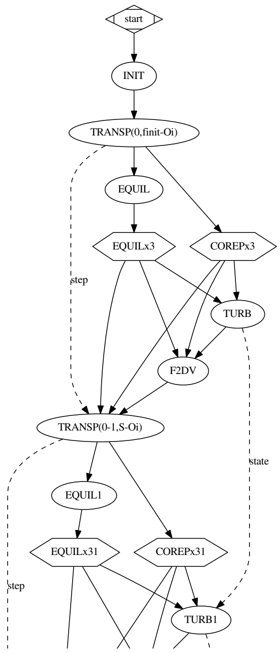


Figure 5: task graph for the fusion application

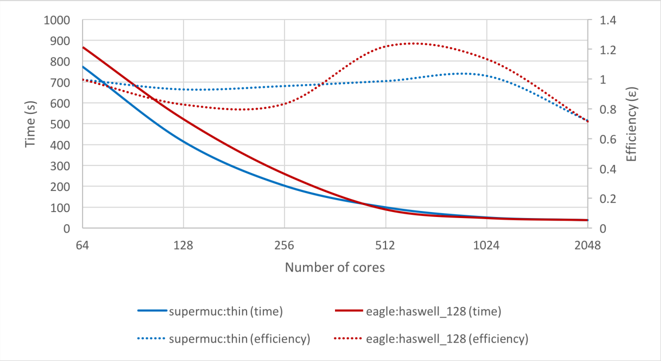


Figure 6: Runtime and efficiency on different resources, for one iteration of the primary submodel of the fusion application.

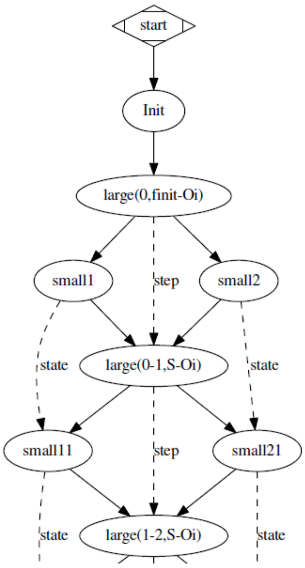


Figure 7: task graph for the SandBox application

One motif of the task graph for an instance of the sandbox application is shown in Figure 7. In this example, the execution of one of the auxiliaries is comparable to the primary ($T_{pr} \sim T_{aux}$). For this execution, we will use resource usage (core-hour consumption) as a cost function. We will also take the total time as another variable. We suppose the primary model 'large' is running in other nodes than auxiliaries (same node type but different physical node). For simplicity, we chose eagle nodes for this example because the number of cores divides evenly with the number of processors requested. Next we calculated the resource usage (R) as $R = P * T$, where P is the total number of cores in the used nodes, T is the total makespan time. Figure 8 shows the resource usage and the actual use of the set of nodes where primary the model runs. The actual resources usage for the primary model is an order of magnitude less than the total usage, which indicate that a large number of resources are idle waiting for the auxiliaries to finish. To overcome this, we need to interleave between two instances of the application to increase the throughput by factor of two while using the same resources. This scenario was introduced and discussed in Deliverable D2.1. For illustration Figure 9 represents the time of the execution scenarios.

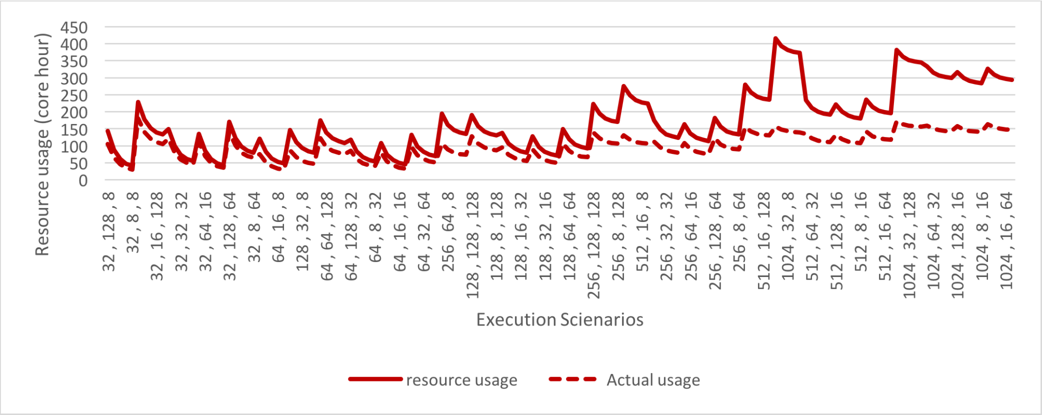


Figure 8: Resource usage of one node type (thin) executing the primary model for many different scenarios. The x-axis shows scenarios that were considered, where the first number indicated to number of cores for the first submodel and so on.

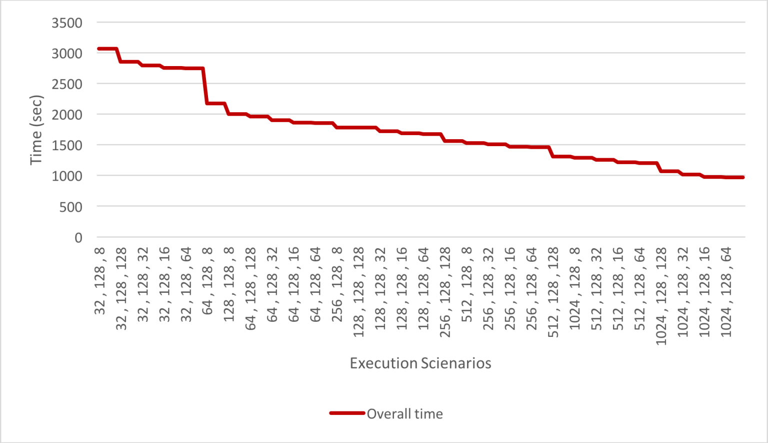


Figure 9: Execution time on one node type (thin) executing the primary model for many different scenarios. The x-axis shows scenarios that were considered, where the first number indicated to number of cores for the first submodel and so on. The scenarios are sorted on execution time

For the full description of Fusion and sandbox application, including the input output files to the multiscale computing patterns software, we refer to deliverable D3.2.

2.4.3 Replica Computing (Binding Affinity Calculator)

The procedure for the BAC is similar to that for Extreme Scaling. The starting point for the BAC application (and all RC pattern applications) is an XMML format description. As described in Deliverable 3.2, a "multiplicity" tag in the "instance" node of the XMML description indicates that multiple instances (replicas) are required for that submodel. The python script detects the presence of this tag, and identifies that the RC pattern is required, and the associated cost-function in the patterns performance software should be invoked (through setting the <computing> node as RC in multiscale.xml).

There are some differences to the ES pattern procedure detailed above due to the specific computational requirements of the BAC. Firstly, within each replica, the BAC does not use the MUSCLE coupling library, as it relies on a simple workflow (an output from one submodel (NAMD) is used as input to the next submodel (AmberTools)). Therefore, there is no need to create a MUSCLE cxa configuration file. Secondly, BAC has previously used the FabSim tool extensively to perform simulation runs and, therefore, we have added an option to the python script to allow the matrix.xml and multiscale.xml files to be completed (as much as possible) through reading of FabSim configuration files. Specifically, it uses the "machines.yml" configuration file from FabSim, which lists the configuration settings of submodels on remote resources. An example of a user-specified BAC YaML file is given in the Annex of Deliverable 3.2. Information specific to ComPat (and not required by FabSim) can also be added to this file, including restrictions on the submodel (GPU / CPU compatibility, max / min number of cores, etc.). This allows submodel information to be reused if it is required for different multiscale applications. A "-y" command line flag to the python script is used to indicate that a FabSim compatible YaML file can assist the completing of matrix.xml and multiscale.xml.

In the same manner as described for the ES pattern, the jMML tool will produce a taskgraph which can be visually inspected by the end-user. Currently this is the taskgraph within a single replica. The python script, using the xMML description and the YaML file, will generate a matrix.xml and multiscale.xml file. The fields within each field are completed using the YaML file or by the user, as appropriate. As in the ES pattern, performance information is currently entered into matrix.xml by the user, but in future this will be added automatically using the tools provided by WP 4. An example of a complete matrix.xml and multiscale.xml for the BAC is given in the Annex to Deliverable 3.2.

Following the procedure outlined above for the ES pattern, the user passes matrix.xml and multiscale.xml to the patterns performance service. Unlike the ES pattern, the user does not need to specify the required number of cores for the overall simulation. This is decided by the patterns performance service by calculating the cost function, described below.

2.4.3.1 Cost function(s)

Finding a cost function for RC that will allow to generate resource allocation plans is quite different then for ES. First, there is an obvious trade-off between the number of replicas that must be executed, the minimum number of cores that one single replica needs, and the total number of cores available for the overall job. Moreover, most of supercomputers have a limit constraint on the job queue so it is not allowed to run more than a certain number of replicas per user. The performance data for RC uses the minimum time per replica for different node types in different hosts, as shown in Figure 10 for a single BAC replica. In the simplest model, where we consider only time to solution, all replicas would be run concurrently on the node with the shortest running time per replica. However, there are several constraints that the patterns performance model must also consider, and we are currently developing more sophisticated models to reflect these constraints.

Most supercomputers have a limit on the number of jobs that can be run/queued at any moment in time. For example, on SuperMUC, the maximum number of jobs that can be run concurrently on the thin nodes in the "general" queue is 8, while there are no restrictions on the Eagle machine at PSNC.

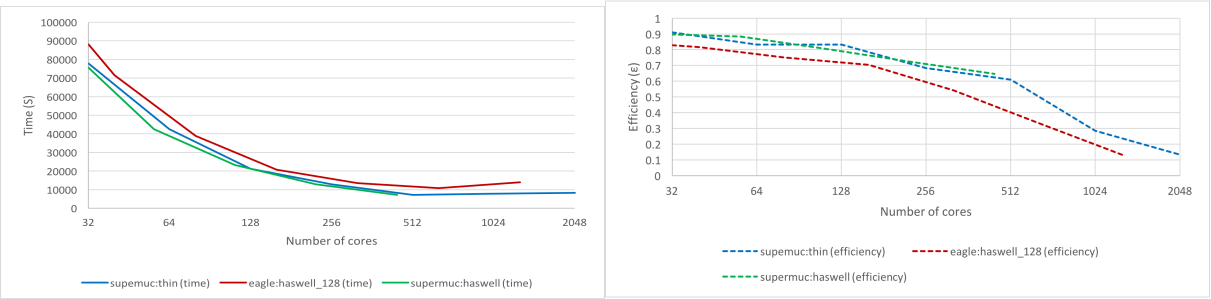


Figure 10: Time and efficiency per replica on different number of processors on different node types

As an example, if we have two RC applications which require 40 and 80 replicas respectively, the pattern performance model needs to calculate which is faster: running all replicas at one supercomputer supermuc (while taking into account the constraint of concurrently running 8 jobs per user) or distributing the jobs among different hosts, for example, across supermuc and eagle, using the functionality in QCG to run across multiple resources. To illustrate how this could be coordinated, let us take the hypothetical situation that there is also a 12 job limit on concurrent jobs running on Eagle. In Figure 11, we show the time to completion as a function of the number of "batches" running on supermuc, where a "batch" is defined as a set of 8 concurrent running jobs on supermuc. The remainder of the replicas are run on Eagle (again in "batches" of up to 12 jobs).

Figure 11 shows that for 40 replicas, the shortest time to completion is for 2 "batches" to be run on supermuc, while for 80 replicas, the minimum time to completion is for 4 "batches" to be run on supermuc. It is clear there is a limitation to this model; it will only be realistic if the time spent in the queue is very short. Otherwise, the time to completion could be very different to that predicted in Figure 11, and we could envisage the most efficient split in replicas across resources being completely changed if the queuing times are very different across the resources. In the next phase of the project, we will investigate ways to estimate queuing times and to incorporate them into the performance model. It is obviously highly non-trivial to accurately estimate queuing times and we will judge whether historical data (average queueing times, for example) is sufficient to formulate the most efficient split across resources.

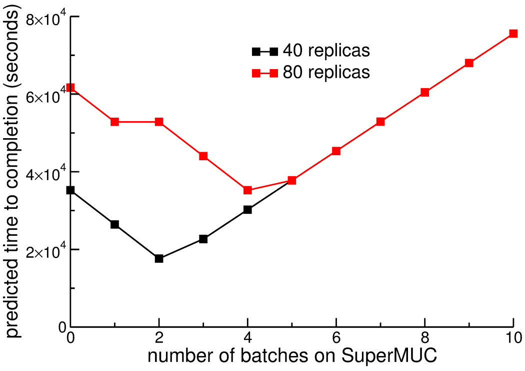


Figure 11: Time of running multi-replicas simulations across 2 resources (supermuc and eagle), as a function of number of "batches" (i.e. sets of 8 concurrent jobs) on supermuc. The remainder of the replicas are run as batches of up to 12 concurrent jobs on eagle.

Another scenario that could increase efficiently is an array style job, which involves running multiple jobs using a single job submission. In this situation, the limitation on the number of concurrent jobs on a single resource could be overcome. This functionality is currently under development in WP5.

2.4.4 Heterogeneous Multiscale Computing

The HMC pattern has not yet been implemented in the pattern service. However, note that with the realisation of the RC pattern in the pattern service, and the related QCG functionality most key functionality to implement HMC is available. During the next phase of the project we are now in the position to also realise HMC.

2.5 Plans for phase 2 (M18-M36) of the project

So far we have implemented a first version of the multiscale computing patterns and algorithms software and applied it for Extreme Scaling and Replica Computing examples. In the second phase of the project we will enhance the capabilities of the software (with more advanced cost functions, taking into account more scenarios, but also energy awareness and fault tolerance), implement the Heterogeneous Multiscale Computing patterns, implement hybrid patters, and in collaboration with WP3 instantiate all ComPat applications using the multiscale computing patterns software, and then carry out in depth performance measurements, demonstrating the capabilities of the patterns, and their benefits.

3 To Conclude

We have described in some detail the activity in WP2 in M13-M18 of the ComPat project, showing the design of the multiscale computing patterns and algorithms software, describing its current implementation and showing some of the main features and capabilities of the multiscale computing patterns software. With this deliverable we have reached Milestone 8 (Month18): *First prototype of the ComPat multiscale development and execution environment. First version of MML specifications of Multiscale Computing Patterns*. We believe WP2 is well on track we well equipped for the second phase of the project.