D4.3 Final report on high level tools, including performance profiling and modelling

Due Date	Month 36
Delivery	Month 36
Lead Partner	ARM
Dissemination Level	Public
Status	Final
Approved	Internal review
Version	1.1

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
13.08.18 v0.1	Oliver Perks	Outline of the report with headings of main sections
V0.2	Keeran Brabazon	Arm internal review
V1.1	Oliver Perks	Internal ComPat review

CONTRIBUTORS

Contributor	Role
Oliver Perks	Editor, WP4 Leader
Keeran Brabazon	WP4 Contributor
Alfons Hoekstra	WP4 Contributor
Tomasz Piontek	WP4 Contributor
Tom Kirkham	WP4 Contributor
Olivier Hoenen	WP2 Contributor
Saad Alowayyed	WP2 Contributor

TABLE OF CONTENTS

1. Executive summary

This document summarises the work done in the design and deployment of the parallel debugging and performance analysis tools to be used in ComPat. In previous deliverables we have discussed the modification to the tools to support the ComPat software and hardware ecosystem. This deliverable focuses on the integration of the tools into the ComPat automated workflow and their exploitation within the project.

The document is structured as follows Section 2 details the contributions made within this document; Section 3 covers the specifics of the new features developed within the tools; Section 4 introduces the performance database where the results are stored; Section 5 documents how this historical information can be used; Section 6 introduces support for a new multiscale pattern and finally in Section 8 we conclude this report.

2. Summary of Contributions

This document provides a continuation for D4.2 [1], outlining the final state of the tools used in the ComPat project. In D4.2 we presented on the tools being used, and how they had been modified to better report on the ComPat environment.
This document will focus more on the integration of these tools into the ComPat workflow and the required modifications to support this.

The work documented in previous deliverables, specifically around the development of the MUSCLE2 integration and associated custom metric, is still of key importance to the tool use within the project. A metric refers to a user specified measurement that can be utilised by Arm MAP (performance analysis tool) to collect and present performance data requested by the user. A custom metric is one developed outside of the core metric pack within the Arm tools, for specialisation, such as the custom metric for MUSCLE2 support. The work in this deliverable builds on this existing foundation of support.

3. Tools Update for Multiscale Profiling

Since the last deliverable we have added some additional support in the tools for supporting the ComPat stack. In this section we document this effort.

3.1. SuperMUC Energy Metric

A key component for the ComPat workflow is the use of energy consumption as a factor for smart scheduling. For this we need to collect comparable energy consumption figures for jobs in MAP.

MAP has two existing energy collection methods – Intel RAPL and IPMI. Both systems have their associated merits and demerits [2], however to ensure coverage of all on-node components we decided to make use of IPMI.
Internally MAP uses its own IPMI Energy Agent [3] based upon libIPMItool [4] to query through IPMI for energy and power usage. This methodology works well for the Eagle supercomputer, however SuperMUC does not provide a suitable IPMI interface. This means we had to develop another custom metric to collect the correct energy information.

SuperMUC is an IBM system, and whilst it supports IPMI the access method used for MAP is not enabled. However, specifically for energy reporting there is another, easier, option – through the IBM Active Energy Manager [5]. We developed a custom metric to read from this file and present energy data back to the user with no additional steps. This metric is installed by default on SuperMUC.
The data collected in this custom metric is directly comparable with that collected on Eagle through the IPMI Energy Agent, and so we can use these numbers for energy to solution comparisons within the ComPat stack.

3.2. Context - QCG Integration

When performing a profiling analysis on a system some knowledge about the system is assumed. However, within the ComPat distributed Experimental Execution Environment (EEE) there are multiple machines, each with multiple partitions. Thus, when we generate a performance profile of an execution on the EEE we must maintain some form of provenance of the data collected, as we aim to re-use performance data collected in order to inform future scheduling choices.

Specifically, the information we want to preserve is the logical application name, machine name, the queue partition (which defines the node type) and selected other machine specific attributes. An example is shown in Listing 1. To enable this, Arm modified the MAP tool to enable a free form ‘Notes’ field, collected by the environment variable “ALLINEA_NOTES”.
To collect this information, we build this environment variable as a set of key value pairs, which are extensible. For example, the logical application name (what we call the multiscale application not the specific binary name) is set when the application module is set, an example is shown for the Fusion application.

```
plgoperks@client:~$ module load compat/apps/fusion
plgoperks@client:~$ echo $ALLINEA_NOTES
COMPAT_NTASKS=1:COMPAT_HPC_SYSTEM=EAGLE:COMPAT_APP=FUSION
```

Listing 1: Allinea notes environment variables

On a machine by machine level we could code to query some of the more system specific information, but this would be a replication of effort. For simplicity we query the QCG environment variables – shown in *Table 1*. These pass information about the job from the QCG level – which is often how the application user perceives their job.

Table 1: QCG environment variables used for MAP Notes field

Environment Variable	Usage
QCG_HOST	Name of the system e.g. Eagle
QCG_PROCESSES	Number of processes assigned by QCG
QCG_JOBID	QCG job ID
QCG_NODE_TYPES	Node type list e.g. eagle_haswell_128

3.3. Context – Application Specific

To understand and attribute performance to different applications we must first understand how to identify the application. During the integration of the tools it was established a lot of the interpretation stages are application specific, thus all the scripts must be engineered to identify the different applications – and interpret the data differently based on this.

As discussed in the previous subsection, we have implemented the logical application name as an environment variable in the application's module file. This makes the first step easier. However, when running the specific kernels, we must anticipate the names of the binaries – this must be hard coded into the scripts.

The main application specific context is provided via the generation of a problem size definition, this is generated by parameter extraction (from input files) and a config hash generation. The config hash is a MD5SUM hash of the input files, used to distinguish unique kernel problems.

Details of the config hash generation and parameter extraction is provided in Appendix 4.

3.4. Arm Performance Reports and Expert Advice

Tasks 4.3 and 4.4 were designed to extend the existing Arm Performance Reports tool, to focus it on multiscale models. To this end we had intended to fully develop an internal feature called expert advice – which advises the user on best practices based on values of key metrics. Such as advising users that they were bound by collective MPI communication and so the application may not scale well.

However, the progression of the project has led to a more 'automated' approach to performance data collection and storage. As we integrate the tools within the ComPat workflow, we focused less on the manual performance inspection side of things. This diminished use-case, and the required significant investment in software development means these items were de-prioritised.

The use of Arm MAP within the profiling phase is complimented with the use of Performance Reports for some of the data aggregation phase – to generate the resulting JSON stub. For this Performance Reports was updated to support the new custom metrics developed – and the workflow used – but no further ComPat specific developments were made.

4. Performance Database

Fundamentally the point of collecting all of the performance information is to allow the ComPat stack to make better decisions with regards to optimising resource usage.
As such we need to establish a tight workflow of data – and crucially this involves integration with the existing services. In Section 4.1 we detail the integration with the Pattern Service, but to enable the integration we must first store all the data in a single accessible location.

For the performance data collected from MAP it was decided that a dedicated performance database would be the best option. This database has been installed on LRZ resources and is accessible from within the PSNC QCG head-node used within ComPat. The workflow for this profiling is shown in *Figure **1***.

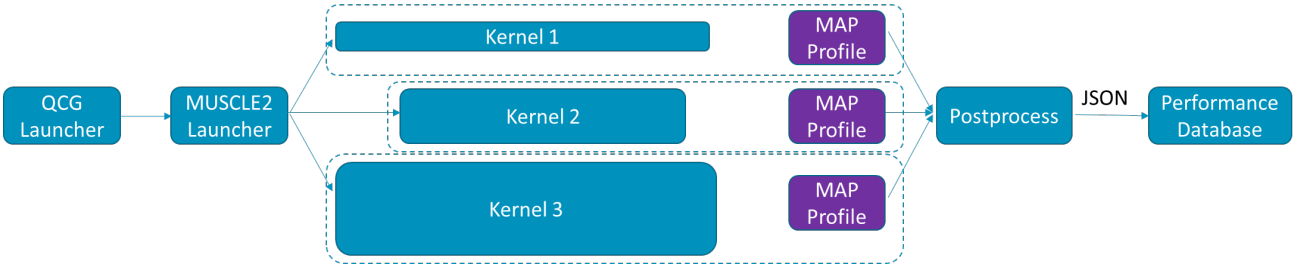

Figure 1: ComPat Performance Analysis Workflow

Whilst MAP can collect vast quantities of performance data we only want to store a limited set of this data, which will allow us to perform performance comparisons and estimations later.
During the postprocess phase we filter out the relevant information from the performance profile, and augment it with the QCG contextual information discussed in Section 3.2. This data is then stored in a json snippet which can then be uploaded to the database.

An example of the scaling data which can be extracted from the performance database is shown in Appendix 2 for running the Fusion application across the EEE resources. For technical details about the performance database please refer to deliverable D6.3 [6].

4.1. Postprocessing and Database Upload

The location and action of each of the post processing phases is critical. QCG jobs are staged onto remote systems and executed. The job's XML definition will outline which files need to be returned to the QCG head-node. Thus, we have a choice of locations for each stage in the workflow, either on the compute node as part of the job, or on the QCG head-node after the job finishes.
As the postprocessing script analyses the input files to each kernel, it makes sense that this information is captured as part of the execution of the kernel. To do this we use a wrapper script around the binary which: launches Arm MAP, runs the postprocessor and moves the resulting JSON file to a predefined location.
At the end of the run all of the generated JSON files (one per kernel) can then be staged back to the QCG head-node. From here they are copied into a shared directory (shared across the ComPat project). A Linux cron-job then monitors this shared folder for new files. These are processed and uploaded to the performance database – and the original JSON file is removed.

This whole process, whilst complex, allows the different tools in the ComPat stack access to the right data at the right time.

5. Performance Prediction and Integration with Pattern Service

The collection and storage of the performance data, discussed in Section 4, has been implemented for one single purpose – to help predict application resource usage in the future.
To this end we perform a historical analysis on the data stored in the database, for a given application kernel, and predict metrics such as runtime and energy consumption.

Accurate estimate for runtime and energy consumption can then be used by QCG for resource brokering – as discussed in [7]. The runtime estimates will also be used by the QTPS (queue time prediction service) for estimating queue time, as discussed in [6]. To this end we developed two components. Firstly, a performance predictor, based on historical information, and then an integration component with the existing pattern service.

It should be noted that the performance prediction phase is only interested in 'strong scaling' – that is scaling the number of cores assigned to a fixed problem size – with the goal of solving the problem faster. This is because at this point in the ComPat stack the user has provided the problem they wish to compute – and the service is only there to optimise the associated execution. We discuss the process of integration, with the pattern service in Section 5.2 and Section 5.3.
Further descript of the integration, from a pattern service perspective, is provided in [8].

5.1. Single Scale Performance Prediction

The database analysis is enabled for the prediction of single scale performance, based on the historical data. From this we can derive the expected execution time for a kernel, we do not compose this into a multiscale performance prediction.
The parameter space performance prediction is done over consists of: Node type, core count and problem size. That is for a given problem size, on a known node type we estimate performance for different core counts.

The concept of node type and core counts are well defined. However, that of problem size is less well defined. As this framework must support multiple applications we have implemented a generic concept which can simply be extended to new applications.

5.1.1. Linear Scaling Factor

For the definition of a generic problem size metric we have defined a linear scaling metric. This is an application specific value representing a single unit of computing, based on the parameters extracted from the problem set. It allows us to normalise the performance data to a time to solve a unit of computation.

```
mysql> mysql> SELECT * FROM Kernel WHERE name = "gem_kernelB";
+-----+-------------+----------------------------------+-----------------------------------------------------------------------+
| id  | name        | config_hash                      | config_parameters                                                     |
+-----+-------------+----------------------------------+-----------------------------------------------------------------------+
|  26 | gem_kernelB | a92f73381cdc8bd68420e7321f0c8bb7 | {"ns00": "32", "nx00": "128", "its_start": "0",                       |
|     |             |                                  |  "ny00": "128", "nftubes": "8", "its_stop": "3", "nstep": "100"}      |
| 147 | gem_kernelB | a92f73381cdc8bd68420e7321f0c8bb7 | {"ns00": "32", "nx00": "128", "its_start": "0",                       |
|     |             |                                  |  "ny00": "128", "nftubes": "8", "its_stop": "20", "nstep": "100"}     |
| 151 | gem_kernelB | 2f42ad7645ae36f188fdf1ea28deef2b | {"ns00": "32", "nx00": "128", "its_start": "0",                       |
|     |             |                                  |  "ny00": "128", "nftubes": "8", "its_stop": "3", "nstep": "100"}      |
+-----+-------------+----------------------------------+-----------------------------------------------------------------------+
3 rows in set (0.06 sec)
```

Figure 2: Database results for Fusion's Gem kernel

In *Figure 2* we show the database entries for the 'gem' kernel of Fusion. We can see that there are three entries, each representing the kernel under different 'problem sizes'. All three have the same spatial dimensions ('nx00', 'ny00', 'ns00'), gem internal timesteps ('nsteps') and number if fluxtubes ('nftubes'). However, two of these runs (26 and 151) represent the benchmark run (4 outer iterations from 'its_start' to 'its_stop' inclusive). Whereas, kernel 147 represents a larger scale production run (21 outer iterations).
Kernels 26 and 151 differ in their hash but not parameters – which means something in the cxa or xml file has changed, but the change was not in one of the parameters identified by the application developer as a proxy for performance.

We note that the parameters must correlate to a linear relationship between value and performance, to facilitate scaling properly. For example, if we had a cubic data set, of size 10, then we would need to represent this as a 10*10*10 factor not a 10 factor, as the move to a size 20 data set would represent an 8x increase in problem size, not a 2x.
Thus, for each kernel problem we can define a scaling factor, shown in *Table 3*:

Table 2: Gem scaling parameters

Kernel ID	Scaling Equation	Scaling Factor
26	1/(32*128*128*8*100*(3-0+1))	5.96046E-10
147	1/(32*128*128*8*100*(20-0+1))	1.13533E-10
151	1/(32*128*128*8*100*(3-0+1))	5.96046E-10

We can then apply this scaling factor to normalise the CPU time, defined by runtime – (MPI time + MUSCLE2 time + IO time), for each run in the database. *Figure 3* shows this analysis for the Gem kernel, on approximately 100 runs, categorised by node type.
We can see here an obvious trend – representing the increased division of work by strong scaling. Projecting the CPU time to core-hours would help account for this – but instead we choose to fit this data with a curve fitting algorithm. This allows us to better anticipate the wider trends.

We note that to make a prediction based on this ‘scaling factor’ projection we simply need to derive a new scaling factor for our ‘input’ problem and apply this to the fitted result.
All of the logic and curve fitting implementation is then application agnostic – and an application would only need to provide the logic to generate a scaling factor from the input deck, and from the parameters in the database, for each record.

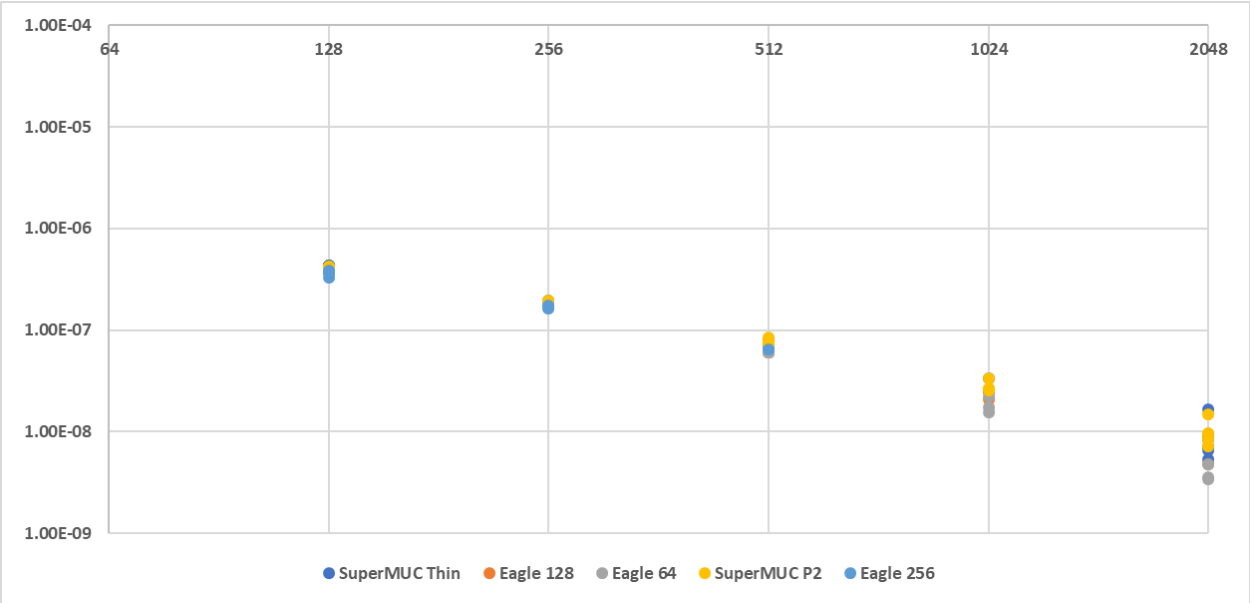

Figure 3: CPU Time Scaling Factor for Gem

We note that the benefit of this scaling factor approach is that as new data is added to the database, regardless of problem size, it can be used in new predictions. This the predictions become more accurate the more the database is used for both benchmarking and production runs.

5.1.2. Curve Fitting

Once we have normalised the performance data is to be used in the prediction of the run time or energy consumption, we extract the performance data – grouped by node type.

The more historical data is available, the larger this data set is likely to be, and in turn the more accurate the prediction based upon it.

We break up the data into two fundamental components which will exhibit different scaling characteristics: CPU time and MPI time.

We treat energy consumption at a proportional value to time – first normalising to a per-core power consumption then extrapolating from there, based on the new estimated runtime and core count.

For each characteristic we form a table of core count and value – and pass it to the curve fitting routine with the corresponding fitting function, for each node type in turn.

Using the Python SciPy 'curve_fit' [9] library we fit each characteristic to an expected scaling curve, and use known properties of the scaling to bound the estimations. These are detailed in *Table 3*.

Table 3: Curve Fitting Equations

Characteristic	Curve Equation	Bounds
CPU Time	$T(x) = a * (1.0 / x) + b$	$a,b >= 0$
MPI Time	$T(x) = a * x + b * x^2 + c * \log_2(x) + d$	

The CPU time follows a strict reciprocal relationship, as parallel performance is bounded by Amdahl's law [10]. We note that this function is monotonically decreasing – and tends to *b*.
MPI time is more complicated to model – for this we use a second order polynomial with an additional base 2 logarithmic term.

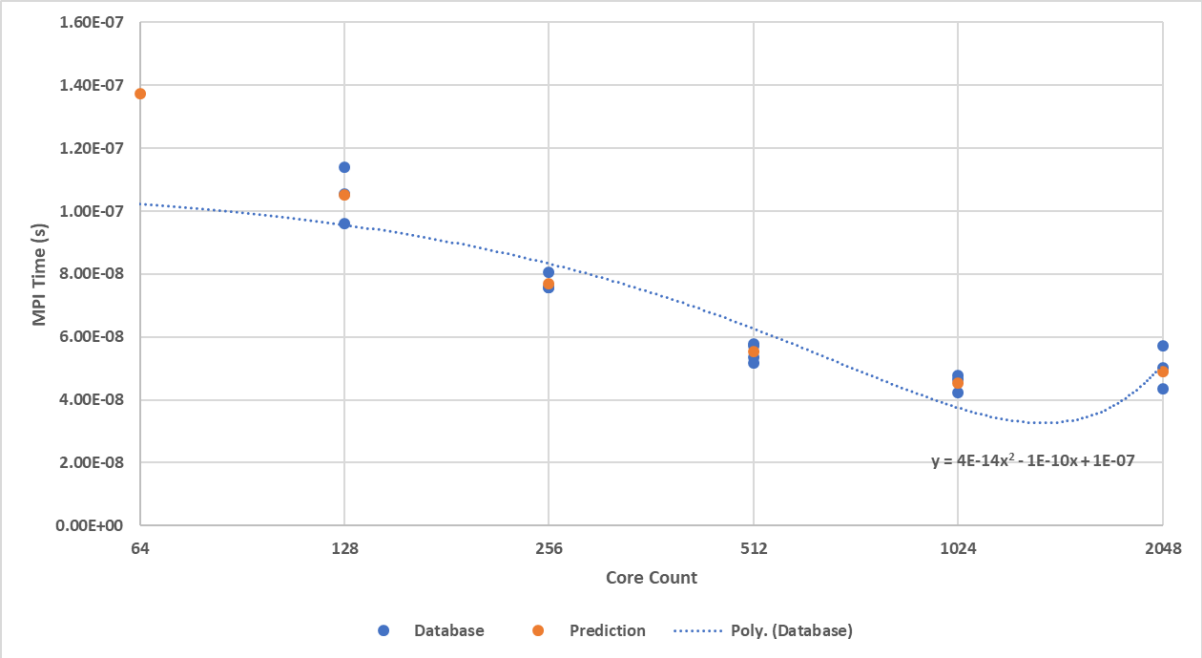

Figure 4: MPI Time Estimation

From *Figure 4* we illustrate our estimation for the MPI time (for Gem on Eagle Haswell_64) – based on our problem size normalisation, using our derived fitting function, shown in orange. We also compare this to a standard 2nd order polynomial (blue dash) fitted to the existing database values (blue dots). We see how whilst the polynomial is accurate within known data it trends poorly for unknown data.

Obviously, when there is insufficient input data we run the risk of overfitting, in such a case we operate on a simplified version of the curve equation.

5.1.3. Single Scale Validation

In this section we have presented our methodology for predicting single scale application performance, using a parameter-based normalisation. To validate this approach, we want to test the ability to predict performance for a previously unseen configuration.

For this we will stick with the Fusion example demonstrated earlier. However, we will vary two components – to ensure the test is unique. Firstly, we will query for a previously untested core count, and secondly an untested problem size. We will use the Eagle Haswell_64 node type, and execute on 64 cores with 8 outer iterations.

We query the pattern driven planner, which includes the optimisation part, for a plan which matches our requirements, and record the estimates. We then execute that plan and evaluate the resulting performance.

Table 4: Performance Prediction Validation

	Prediction (s)	**Actual Recording (s)**	**Error (%)**
Gem - CPU Time	2391.12	2483	3.84
Gem - MPI Time (s)	460.72	582	26.32
Chease – CPU Time (s)	20.96	17	18.89
ETS – CPU Time (s)	8.24	8	2.91
Imp4dv – CPU Time (s)	2.48	2	19.35
Total	2883.52	3092	7.23

Table 4 presents our kernel by kernel performance prediction and validation. Here we can see a high accuracy across the kernels – with the exception of the MPI prediction for Gem. We also note that the runtime (s) is recorded as an integer, which can distort error margins at small values.

Whilst the 26% error seen in our validation of MPI time is inaccurate we consider this an improvement over a standard polynomial fitting, as illustrated in *Figure 4*. However, it does suggest that there is scope for improvement – from a more comprehensive application specific performance model – a topic we touch upon in Section 5.4.

Energy consumption validation is more complicated. Our estimation generates a size-normalised power consumption. That is how much power (energy /s) is used per-core on a specific node type. In this case 14.35 w.

This must then be multiplied by the core count for estimation, 64, and the total runtime of a kernel (including MPI time and MUSCLE2 time). However, the MUSCLE time comes from the composition

of the multiscale model. For validation we assume a simplistic Gem runtime composition of the sum of all kernels – which is close to accurate.

Table 5: Gem Energy Consumption Validation

	Time (s)	**Estimated Energy (j)**	**Actual Energy (j)**	**Error (%)**
Estimated Time	2883.52	2,650,041	3,084,620	16.39
Actual Time	3092	2,841,641	3,084,620	8.55

The energy consumption prediction of Gem is also shown in *Table 5*. Based on our estimated Gem runtime, and our estimated power / core metric we observe a 16.4% error on our estimation. However, as shown in *Table 4* our runtime estimation (based on Gem's MPI time) is an underestimate, and runtime is a key factor in energy estimation. If we use our power estimation with the real runtime (from *Table 4*) we see this error drop to 8.5%. We see this as a very acceptable estimation.

5.2. Integration

The ability to predict performance based on historical information is only useful if we make use of the resulting data. For this we need to integrate the performance prediction into the Pattern Service workflow.

This is feasible, as all the input required by the performance prediction is the name of the application to predict for – and a definition of the input problem.

As a result, it will generate a set of permutations of execution plans with associated runtime and energy predictions.

Appendix 3 shows how the predicted runtimes can be augmented with data from other services – such as the queue time prediction service.

As the database derived performance prediction is only for single scale performance we must recompose these predictions into a multiscale simulation prediction. This work happens within the optimisation part of the pattern service, and is documented in [8].

5.3. Benchmarking

One key addition to our historical data analysis approach is the use of benchmarking. The goal is to seed the database with performance data for applications. However, inherent to their nature, benchmarks are not full executions of the simulation, and we operate on reduced size representations.

Therefore, the choice of application parameters is crucial, as for a benchmark we will want to reduce the overall runtime. This is traditionally done by reducing either the size of the problem, or the duration of the run. Thus, we must capture these parameters, and understand how to project them within the performance estimation phase.
By running multiple benchmarks across different systems at multiple core counts we can gain an approximation of application performance to allow the performance prediction service to function.
For the database the only difference between a benchmark and a production run is the problem 'size' as expressed by the parameters within the kernel definition. Thus no extra considerations need to be made.

When run through QCG, the difference between a normal execution and a benchmark execution is that there is a single execution plan, which is provided to QCG. This means that there are no options for QCG to optimise between, and the simulation is run as instructed.

5.4. Performance Models

In Section 5.1.2 we demonstrate how we apply curve fitting to historical data. This procedure is based on a generic understanding of application scaling, and the available parameters for each application.
Obviously, this is suboptimal – and where better data is available a more accurate performance prediction can be made.
Specifically, when a performance model already exists for an application we can make use of that within the performance prediction stage.

Research has been done to show that intricate parameterised LogGP performance models can operate with a high degree of accuracy, such as the model generated for wave-front codes in [11]. With appropriate data collection, and integration, such models could be exploited within the performance prediction phase.

Currently, no such detailed parameterised models exist for the applications within ComPat, but with further development an enhanced performance prediction scheme could be deployed.

6. Support for ComPat Applications

For much of the ComPat project, application support for distributed computing patterns has been focussed on representatives of both the Extreme Scale (ES) and the Replica Computing (RC) pattern [1] [12]. The Heterogenous Multiscale Computing (HMC) pattern took was phased for later in the project delivery, and so tool support also came later, we detail this through the Materials application.

Whilst most of the profiling and data storage infrastructure is generic and application agnostic, there are a few components where application specific logic is required (specifically the hash generation, and parameter extraction).
To date full support is provided for:

- Fusion
- ISR2D
- Materials

6.1. HMC Pattern - Materials

The representative Materials modelling HMC application makes use of a QCG pilot job – a job container within an allocation. Within this framework the application consists of both a micro model and a macro model. As the micro model progresses it task-farms micro model simulations within the pilot job. Both the number of these micro-models and their size (MPI ranks) are determined dynamically at runtime.

These micro-models are LAMMPS simulations, and do not make use of any external coupling library such as MUSCLE2. The start-up of these simulations is controlled via a JSON file of MPI execution lines, which are then launched by the pilot job manager.
Using the profiling tools, within this setup, is very simple as the MPI run command is simply replaced by the MAP profile framework. Each of these simulations will then generate their own profile – which will be aggregated against the QCG job ID in the performance database.

7. Adding External Applications

Throughout this report, and those prior to it, we have focussed on the specific set of applications used within the ComPat project. An ambition of the project has been to attract external applications to the ComPat software stack – to demonstrate the value in improving multiscale application development and deployment.
Whilst individually all the major components are generic (QCG, MAP, the Performance Database), and thus application agnostic, much of the integration logic is dependent on internalised knowledge about the application and kernel being run.
The additional steps required are documented in Appendix 4.

8. Conclusions

In this deliverable, we have discussed the contribution to the ComPat project provided through the use of the Arm HPC tools.
Building upon deliverable D4.2 [1] we show how the tools have progressed, and the benefit they bring to the ComPat project.

A large focus of this report has been on integration – specifically, how the performance profiling tools became an integral component of the pattern service.

The real value of the tools lays in the information they can collect, and how that information can be used. For that reason, we detail how the Arm MAP performance profiler has been integrated into the different components within the system, such as QCG and MUSCLE, to collect better data. The culmination of this integration with job execution, is the performance database. This is an ever-growing record of historical performance data from across the ComPat Experimental Execution Environment.

Whilst the performance database itself is of significant value, its true worth lays in its integration and the automation of performance prediction. Whilst this is an incredibly complex field, which has not been utilised in its most advanced forms in this project, we have demonstrated how the use of simple performance models in a distributed execution environment are very valuable. Further research and development using more sophisticated methods will be able to improve on the utility of the results outlined here, through the use of the same framework.

Annexes

1. Allinea Commands

Table 6: List of command line prefixes for launching Allinea tools, along with a brief description of files that are generated.

Command Line Prefix	Purpose	Output
ddt	Start an interactive debugging session with a GUI on the local system	No output file generated
ddt --offline	Start an offline debugging session	HTML debug report generated
ddt --connect	Start a debugger which will connect to an available parent process. This is used in order to connect to a GUI running on a remote system in a 'Reverse Connect' procedure. See **Section 3.3 of the Allinea DDT User Guide** [13] for more details.	No output file generated
map	Start a profiling session in a GUI on the current system	.map profile generated
map --profile	Start a profiling session without the need for a GUI	.map profile generated
perf-report	Start a profiling session without the need for a GUI	HTML summary profile generated

2. Performance Scaling for Fusion application

Interrogation of the performance database allows us to plot performance data from ComPat applications across the EEE resources.

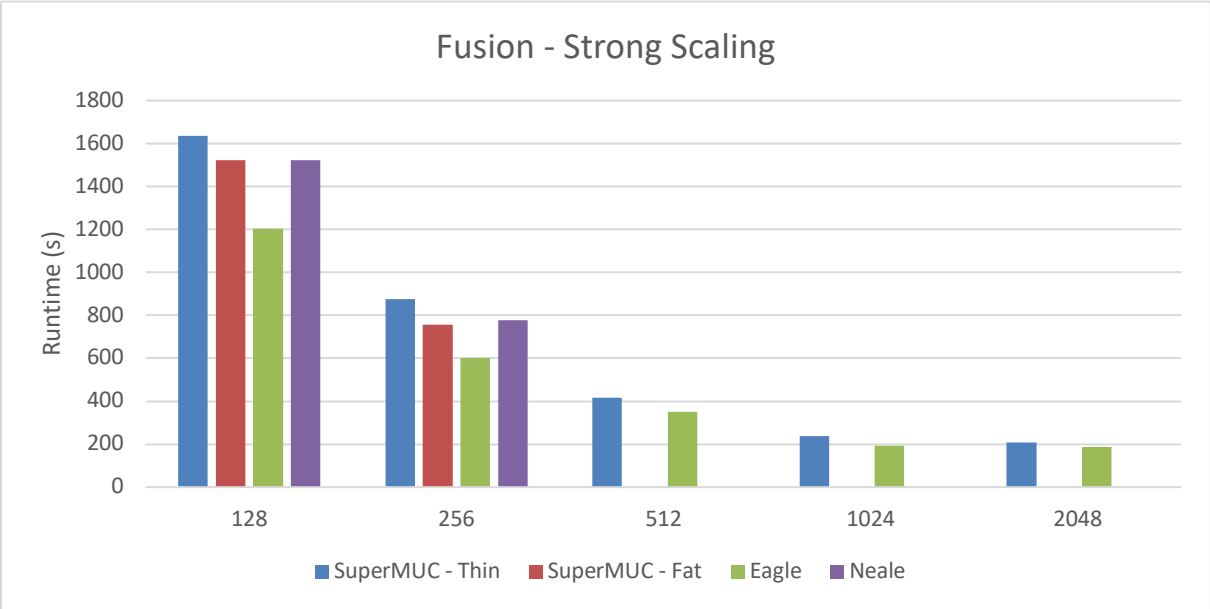

Figure 5: Fusion application strong scaling across the EEE machines

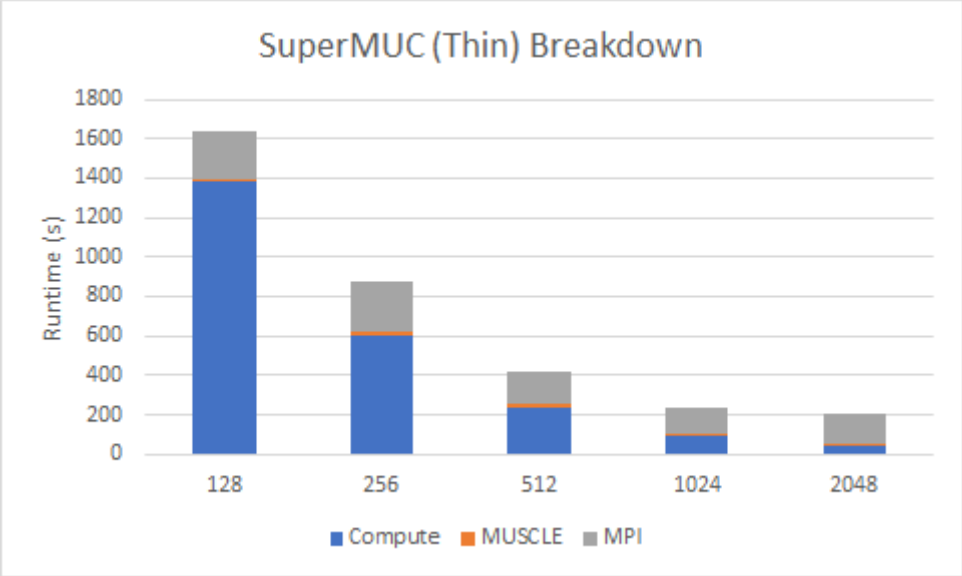

Figure 6: Fusion breakdown of scaling on SuperMUC thin nodes

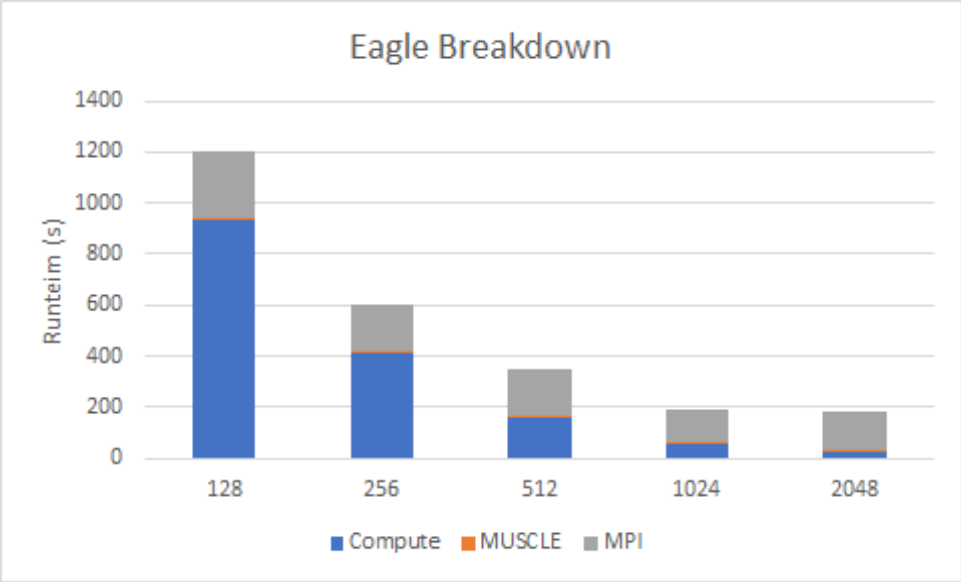

Figure 7: Fusion breakdown of scaling on Eagle nodes

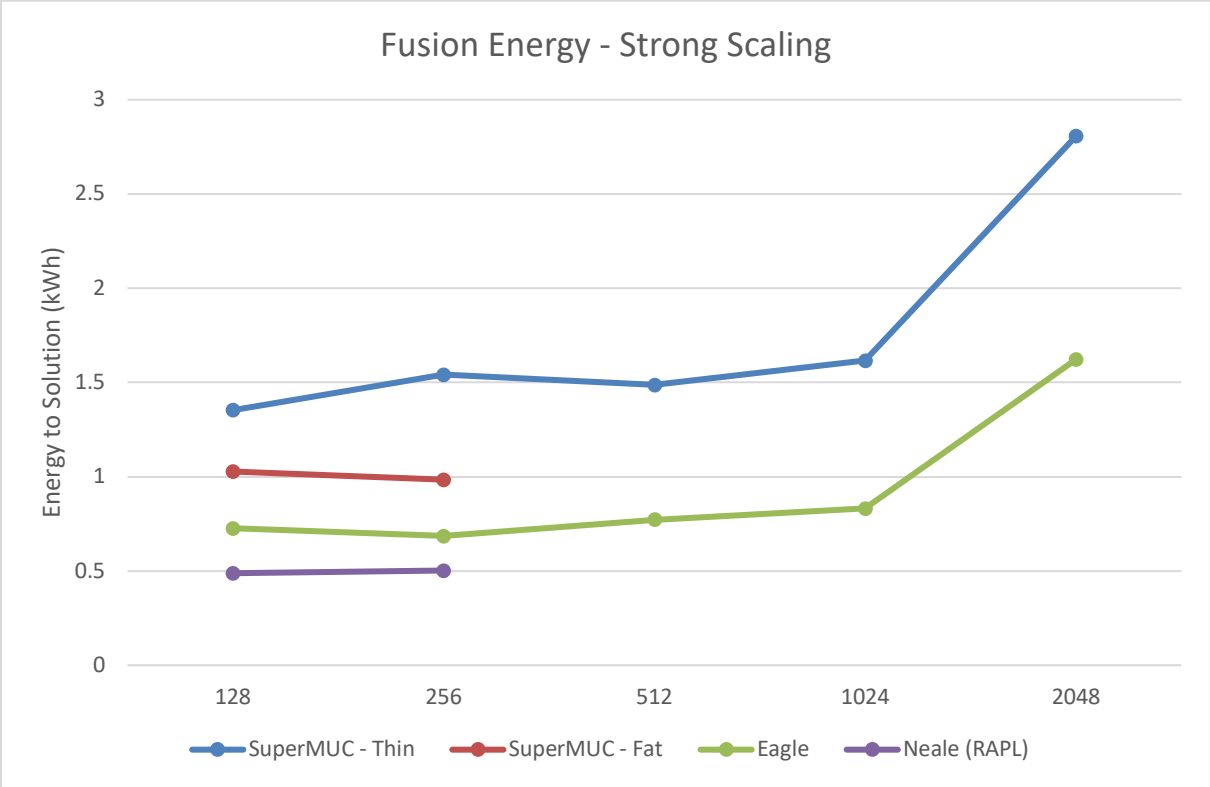

Figure 8: Fusion energy to solution comparison for EEE machines

3. Estimation and Queue Time Prediction

Once we have established data, such as the benchmark data shown in *Figure* ***6*** – we can estimate for a problem which is larger – in this case moving from 3 timesteps to 200 timesteps.

Once we have a runtime prediction from the performance prediction component we must obtain a queue time prediction. This takes the jobs runtime, an overhead assumption (in this case 10%), the size of the job and the desired node type. It then returns a queue time based on a probability of starting (in this case 80%).

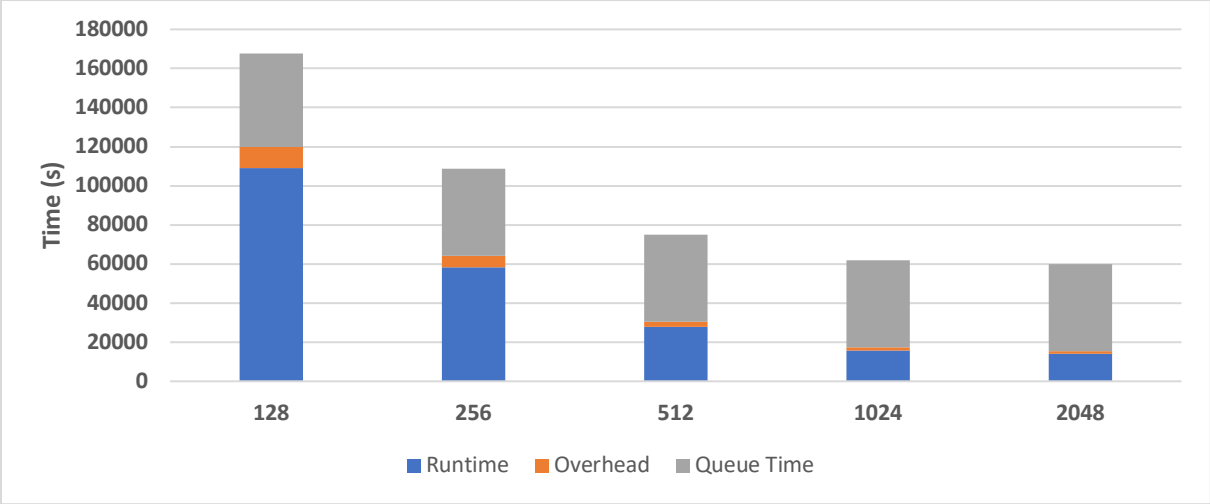

Figure 9: Fusion runtime with associated queue time prediction

4. Application Specific Configurations

To add support for a new application – within the scope of the software tools – there are a number of steps you need to follow.

Module File

On each system there should be a module file which sets up the correct environment for the application. This includes setting the appropriate environment variables for identifying the name of the application:

prepend-path ALLINEA_NOTES COMPAT_APP=<New Application>

CXA Identification

During the *kernel_postprocess.py* analysis we read the *cxa* file – for MUSCLE2 applications. We obtain the location of this file from either an application specific default – or a system environment variable. Each new application should add an appropriate entry.

```
def get_cxa(app):

    if app.upper() == 'FUSION':
       return os.environ.get('COMPAT_CXA','unified.cxa.rb')
    elif app.upper() == 'ISR2D':
       return os.environ.get('COMPAT_CXA','cxa/isr_unified_prof.cxa.rb')
```

Figure 10: Setting the CXA location

Hash Function

To establish the uniqueness of a problem we generate a hash value from the input files, or parameters, used. This exists in the *kernel_postprocess.py* file

```
def generate_kernel_hash(app, kernel):
    files=[]

    cxa = get_cxa(app)

    if app.upper() == 'FUSION':
       files.append(cxa)

       choices = {'gem_kernelB' : 'gem-ref.xml', 'chease_kernelB' : 'chease.xml', 'ets_kernelB' : 'ets.xml', 'imp4dv_kernelB' : ''}
       files.append(choices.get(kernel,''))

    elif app.upper() == 'ISR2D':
       files.append(cxa)
```

Figure 11: Hash function file setting

Parameter Extraction

```
def get_app_params(sample, app, jsonDict):

    cxa = get_cxa(app)

    if app.upper() == 'FUSION':
      if sample['kernel_name'] == "gem_kernelB" and os.path.isfile("gem.xml"):
        with open("gem.xml", 'r') as fd:
          root = etree.parse(fd)

        sample['parameters']['nx00'] = find_one_xml(root, "nx00")
        sample['parameters']['ny00'] = find_one_xml(root, "ny00")
        sample['parameters']['ns00'] = find_one_xml(root, "ns00")
        sample['parameters']['npesx'] = find_one_xml(root, "npesx")
        sample['parameters']['npess'] = find_one_xml(root, "npess")
        sample['parameters']['nftubes'] = find_one_xml(root, "nftubes")
        sample['parameters']['nstep'] = find_one_xml(root, "nstep")
```

Figure 12: Extracting runtime parameters

The next stage of insight into the application is kernel specific – and is for extracting information on the parameters to the kernel configuration. This is extracted from either the input files of runtime parameters.

Database Upload

The database upload phase makes use of the application specific data extracted from the performance data. However, it also has its own application specific behaviour. This time to manage how to aggregate

```
r.cpuhours += (mapfile['kernel_runtime'] * mapfile['kernel_cores'] / 3600)
if data['run_app'].upper() == 'FUSION':
    r.energy = max(r.energy, mapfile['kernel_energy'])
    r.runtime = max(r.runtime, mapfile['kernel_runtime'])
elif data['run_app'].upper() == 'BAC':
    r.energy += mapfile['kernel_energy']
    r.runtime += mapfile['kernel_runtime']
```

Figure 13: Data field aggregation

performance data. Specifically, this is with regards to the potential for shared resources on nodes, when using MUSCLE2, as multiple kernels can run both simultaneously and on the same node. So, to calculate the total energy used (which is reported per node) we must either sum the values – if they were executed on distinct resources – or take a maximum if on shared resources.

References

[1] ComPat, “D4.2 - Report on status of performance profiling of multiscale simulations,” 2017.

[2] T. Ilsche, D. Hackenberg, S. Graul, G. Schöne and J. Schuchart, “Power measurements for compute nodes: Improving sampling rates, granularity and accuracy,” *Sixth International Green and Sustainable Computing Conference (IGSC),* pp. 1-8, 2015.

[3] Arm, “IPMI Energy Agent,” [Online]. Available: https://developer.arm.com/products/software-development-tools/hpc/documentation/ipmi-energy-agent. [Accessed 2018].

[4] D. Laurie, “IPMItool,” [Online]. Available: https://github.com/ipmitool/ipmitool. [Accessed 2018].

[5] IBM, Implementing IBM Systems Director Active Energy Manager 4.1.1, 2009.

[6] ComPat, “D6.3 - Final report on the Experimental Execution Environment,” 2018.

[7] ComPat, “D5.3 - Report on integration of ComPat services with multiscale coupling libraries and patterns,” 2018.

[8] ComPat, “D2.3 - Final Report on Multiscale Computing Patterns Including thier Performance,” 2018.

[9] SciPy, “Curve Fit,” [Online]. Available: https://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.optimize.curve_fit.html.

[10] Wolfram, “Demonstrations of Amdahl's Law,” [Online]. Available: http://demonstrations.wolfram.com/AmdahlsLaw/. [Accessed 2018].

[11] G. Mudalige, M. Vernon and S. Jarvis, “A plug-and-play model for evaluating wavefront computations on parallel architectures,” *IEEE International Symposium on Parallel and Distributed Processing,* 2008.

[12] Allinea Software Ltd., “Deliverable 4.1: Report and software on design of tools and required actions to support performance tools for multiscale,” H2020 ComPat, 2016.

[13] Allinea Software Ltd., “Allinea DDT User Guide,” August 2016. [Online]. Available: http://www.allinea.com/user-guide/forge/DDT.html#x8-27000II.