H2020-FETHPC-01-2016

DEEP-EST

DEEP Extreme Scale Technologies

Grant Agreement Number: 754304

D2.3

Benchmarking, evaluation and prediction report

Final

Version:	1.0
Author(s):	J. Corbalan (BSC)
Contributor(s):	A. Smolenko (Juelich), M.D'Amico (BSC), G. Llort (BSC), E. Mercadal (BSC), J. Giménez (BSC), C. Navarrete (BAdW-LRZ)
Date:	31.03.2021

Project and Deliverable Information Sheet

DEEP-EST Project	**Project ref. No.:**	754304
	Project Title:	DEEP Extreme Scale Technologies
	Project Web Site:	www.deep-projects.eu
	Deliverable ID:	D2.3
	Deliverable Nature:	Report
	Deliverable Level: CO*	**Contractual Date of Delivery:** 31/March/2021
		Actual Date of Delivery: 31/03/2021
	EC Project Officer:	Juan Pelegrin

*– The dissemination level are indicated as follows: **PU** - Public, **PP** - Restricted to other participants (including the Comissions Services), **RE** - Restricted to a group specified by the consortium (including the Comission Services), **CO** - Confidential, only for members of the consortium (including the Comission Services).

Document Control Sheet

Document	**Title:** Benchmarking, evaluation and prediction report	
	ID: D2.3	
	Version: 1.0	**Status:** Final
	Available at: www.deep-projects.eu	
	Software Tool: LaTex	
	File(s): DEEP-EST_D2.3_Benchmarking_evaluation_and_prediction_report_v1.0.pdf	
Authorship	**Written by:**	J. Corbalan (BSC)
	Contributors:	A. Smolenko (Juelich), M.D'Amico (BSC), G. Llort (BSC), E. Mercadal (BSC), J. Giménez (BSC), C. Navarrete (BAdW-LRZ)
	Reviewed by:	N. Burkhardt (EXTOLL) N. Eicker (Juelich)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
0.1	10/03/2021	Draft	Internal review
1.0	31/03/2021	Final	

Document Keywords

Keywords:	DEEP-EST, HPC, Exascale, Job scheduling evaluation, modular systems

Contents

Executive Summary 12

1. Introduction 14

2. Benchmarking 16
2.1. Benchmarking Guide . . . 16
2.1.1. Benchmark Infrastructure . . . 17
2.1.2. Benchmark Integration Workflow . . . 19
2.1.3. Benchmark Execution Workflow . . . 21
2.2. Benchmarking Suite . . . 23
2.2.1. Benchmarking Strategy . . . 23
2.2.2. Synthetic Benchmarks . . . 24
2.2.3. Application Benchmarks . . . 27
2.2.4. Benchmark Integration Status . . . 27
2.3. Benchmarking Evaluation . . . 29
2.3.1. mpiLinkTest . . . 29
2.3.2. mpiLinkTestGPU . . . 32
2.3.3. NEST . . . 34
2.3.4. GROMACS . . . 35
2.4. Conclusion . . . 36

3. Modelling and validation of scheduling policies 38
3.1. The Modular Workload Trace file format . . . 39
3.2. Modular trace file generation . . . 39
3.3. Evaluating conventional vs Modular systems . . . 41
3.3.1. Hardware configuration . . . 41
3.3.2. Workload . . . 42
3.3.3. Evaluation . . . 42
3.4. Workflows with dynamic dependencies . . . 43
3.4.1. Performance metrics for workflow evaluation . . . 45
3.4.2. Experiments . . . 46
3.4.3. Dynamic workflows management evaluation: Workloads with 24% of workflows . . . 47
3.4.4. Dynamic workflows management evaluation: Workloads with 33% of workflows . . . 48
3.5. Conclusions . . . 48

4. Performance modelling and extrapolation 51
4.1. BSC modelling tools . . . 51
4.2. Analysis methodology . . . 53
4.3. GROMACS . . . 55
4.4. xPIC . . . 61
4.5. NEST+Arbor . . . 69
4.6. NextDBSCAN . . . 77

4.7. EC-EARTH 83
4.8. Conclusions 88

5. Energy modeling **91**
5.1. Data input 92
5.2. PCA methodology 93
5.3. Model validation 96
5.3.1. Microbenchmark results 97
5.3.2. Real applications 102
5.4. Results 106

6. Summary **108**

Appendices **110**

A. Benchmarking **112**
A.1. Benchmark Suite 112
6.2. Synthetic Benchmarks 113
6.3. Application Benchmarks 121
6.4. Support Scripts 125
6.5. Visualisation 126

7. Modular Workload Format **128**
7.1. Modular Workload Format fields 128

List of Acronyms and Abbreviations **131**

Bibliography **133**

List of Figures

1. Benchmark Integration Workflow . . . 19
2. Benchmark Integration Execution . . . 22
3. Parameter Sets . . . 26
4. mpiLinkTest benchmarking results . . . 30
5. Cluster module node architecture . . . 31
6. CM node hardware . . . 31
7. mpiLinkTestGPU benchmarking results . . . 33
8. NEST benchmarking results . . . 34
9. GROMACS scaling benchmarking results . . . 36
10. GROMACS variation benchmarking results . . . 37
11. Execution environment for scheduling evaluation: The BSC-Slurm simulator . . . 38
12. Modular trace file generation methodology . . . 39
13. Modular list example . . . 40
14. Workflow generation example . . . 41
15. Modular system performance vs Homogeneous clusters . . . 44
16. Workflow performance metrics when having a 24% of the jobs being part of workflows . . . 47
17. Workflow performance metrics when having a 33% of the jobs being part of workflows . . . 49
18. Histogram of computation duration of LULESH benchmark on CM, DAM and ESB, showing increasingly larger computations from left to right (x-axis). Light green to dark blue gradient indicates the most frequent behavior. More spread data (red boxes) indicates higher variability . . . 54
19. Extrapolation model for GROMACS (325 thousand atoms, fixed PP-PME ratio) . . . 56
20. Extrapolation model for GROMACS (20 million atoms, variable PP-PME ratio) . . . 57
21. GROMACS timelines comparing 32- and 64-nodes runs, showing perturbations in MPI calls . . . 58
22. Zoomed view of GROMACS showing the distinct wait patterns in the communication phase . . . 59
23. Zoomed view in 64-nodes run showing process 331 of GROMACS systematically delayed, causing a cascade of increased MPI wait times in partners . . . 60
24. xPIC application structure for 1 iteration of the pure MPI version . . . 61
25. xPIC measured speed-up for runs from 1 to 256 nodes . . . 61
26. Extrapolation model for xPIC (MPI version) . . . 63
27. xPIC collapsed timeline for the beginning and the four communication phases of the main iterative step (from left to right) . . . 64
28. xPIC computation duration histogram correlated with IPC . . . 65
29. xPIC detailed views comparing real and simulated runs . . . 65
30. xPIC hybrid MPI+OpenMP structure . . . 66
31. Hybrid xPIC: MPI communications (red and yellow lines) inside OpenMP parallel regions not overlapping with computations (green phases) of other threads . . . 67
32. Extrapolation model for xPIC (MPI+OpenMP version) . . . 68

33. NEST+Arbor measured speed-up for runs from 1 to 45 processes per module . . 69
34. Global & hybrid efficiencies for NEST+ARBOR (small scale optimization enabled) 70
35. *Parallel efficiency* decomposition calculated independently for NEST and Arbor (small scale optimization enabled) 71
36. NEST+ARBOR computation (top) and communication (bottom) structure for 4-nodes (left) and 16-nodes (right) runs 72
37. Efficiencies for NEST+Arbor coupled and per-component (small scale optimization disabled) 73
38. 2 iterations of the 64 node run showing Arbor's serialization at large scales . . . 74
39. Comparison of original and simulated run at 64 nodes showing the effect on NEST of removing Arbor's serialization 74
40. Efficiencies for NEST+Arbor simulating non-existent Arbor serialization in 2 last experiments (64 and 90 nodes) 75
41. Efficiencies for NEST+Arbor coupled and per-component simulating non-existent serialization in 2 last experiments (64 and 90 nodes) 75
42. Extrapolation model for NEST+Arbor with 2 points (simulated runs at 64 & 90 nodes) predicting good scalability up to 10k processes 76
43. NextDBSCAN structure for a 44-node run 78
44. Extrapolation model for NextDBSCAN 79
45. Clustering analysis for NextDBSCAN showing the structure of computations . . . 80
46. NextDBSCAN Cluster 1 computation variability increasing with the scale 82
47. Tracking analysis for NextDBSCAN showing the evolution of all computing regions from 4-nodes (top) to 44-nodes (bottom) with increasing variability 82
48. Comparison of measured *Parallel efficiency* factors 84
49. One iteration of EC-EARTH run with 144 NEMO processes and 144 IFS processes 85
50. One iteration of EC-EARTH simulating IFS running on DAM 86
51. One iteration of EC-EARTH simulating a key computing kernel accelerated . . . 86
52. Extrapolation model for EC-EARTH comparing *Parallel efficiency* and *Load balance* between 2 different ratios of NEMO:IFS 87

53. Representation of a certain information in the original base and its transformation into the new space. 94
54. Example of PCA methodology applied to face images. 95
55. Contribution of each principal component to the global information explanation. . 95
56. Energy model for the SuperMUC-NG compute nodes using a reference frequency of 2.3GHz. 98
57. Energy model for the CM nodes using a reference frequency of 2.3GHz. 98
58. Energy model for the DAM nodes. The reference frequency is 2.3GHz. 99
59. Energy model for the ESB nodes. The reference frequency is 1.9GHz. 99
60. Energy model for the GPU accelerators. The reference frequency is 1.38GHz . . 100
61. On the CPUs of the CM nodes the best frequency to proyect energies is around 2.2GHz, 2.3GHz 101
62. On the CPUs of the DAM nodes the best frequency to proyect energies is located around 1.8GHz and 1.9GHz 102
63. On the CPUs of the ESB nodes the best frequency to proyect energies is located around 1.9GHz and 2.0GHz 103

64. Comparison of the power used by a node in the CM module while runing the NPB benchmarks. 104
65. Comparison of the power used by a node in the DAM module while runing the NPB benchmarks. 104
66. Comparison of the power used by a node in the ESB module while runing the NPB benchmarks. 105
67. Comparison of the power used by a node in the DAM and ESB cluster partitions while runing the NPB-LU benchmark. 105

68. Files created by benchmarking steps 114
69. Sample Webbrowser Visualisation 127

List of Tables

1. mpiLinkTest Parameter Permutations . . . 26
2. GROMACS Parameter Permutations . . . 28
3. Synthetic Benchmark Integrations . . . 28
4. Application Benchmark Integrations . . . 29
5. mpiLinkTest bandwidths verification . . . 32
6. DEEP-EST prototype costs distribution . . . 42
7. Application types used for modular vs homogeneous evaluation. Columns CM, DAM, and ESB show runtime ratios, e.g. CMS runs 2.25 times faster on DAM with respect to CM. . . . 43
8. Dimemas simulation parameters for the DEEP-EST prototype . . . 54
9. Efficiency model for GROMACS (20 million atoms, variable PP-PME ratio) . . . 55
10. Efficiency model for xPIC (MPI version) . . . 62
11. xPIC efficiency metrics per phase . . . 65
12. xPIC MPI+OMP efficiency metrics. . . . 67
13. Efficiency model for NEST+ARBOR (small scale optimization enabled) . . . 70
14. Efficiency model for NEST (small scale optimization enabled) . . . 71
15. Efficiency model for Arbor (small scale optimization enabled) . . . 71
16. Global efficiency model for NextDBSCAN . . . 77
17. Hybrid efficiency model for NextDBSCAN . . . 77
18. Efficiency model for EC-EARTH (ratio 1:1) . . . 84
19. Efficiency model for EC-EARTH (ratio 2:3) . . . 84
20. h5perf Parameter Permutations . . . 115
21. hpcc Parameter Permutations . . . 115
22. hpcg Parameter Permutations . . . 116
23. hpl Parameter Permutations . . . 117
24. hpl4cuda Parameter Permutations . . . 118
25. ior Parameter Permutations . . . 119
26. mdtest Parameter Permutations . . . 119
27. mpiLinkTest Parameter Permutations . . . 120
28. mpiLinkTestCrossPartition Parameter Permutations . . . 120
29. mpiLinkTestGPU Parameter Permutations . . . 120
30. stream Parameter Permutations . . . 121
31. Correlator Parameter Permutations . . . 122
32. Imager Parameter Permutations . . . 122
33. CMSSW Parameter Permutations . . . 122
34. xPic Parameter Permutations . . . 123
35. GROMACS Parameter Permutations . . . 123
36. NEST Parameter Permutations . . . 124
37. Arbor Parameter Permutations . . . 124
38. HPDBSCAN Parameter Permutations . . . 124

39. piSVM Parameter Permutations 125

Executive Summary

This deliverable presents the main objectives and results for WP2. This includes the DEEP-EST benchmark suite created in Tk2.1, the evaluation of the scheduling policies proposed in WP5 done in Tk2.2, the analysis of the main application characteristic influencing performance scalability based on the performance models created in Tk2.3 and the energy models for the different CPUs and GPU in the DEEP-EST prototype as well as their validation done in Tk2.4.

The work done by Tk2.1 concerning the benchmark suite includes the definition of the benchmarking strategy, the integration of synthetic kernels and WP1 applications in JUBE, the creation of a set of tools to manipulate the DB information and the automatic generation of graphs using experiments results. The synthetic kernel selection has been done to evaluate the main architectural components in the DEEP-EST system. We will show we are not only providing a benchmark suite but a complete workflow for automatic systems evaluation: integration, compilation, execution and analysis. The DEEP-EST benchmark suite could be adapted to be used in other projects.

The evaluation of a modular system is a new topic in the context of scheduling evaluation because it significantly differs from the traditional HPC environments. Three main contributions have been provided in the this project concerning modular systems evaluation: the specification of a new trace file format for modular systems, the specification of a methodology to create modular traces using already existing workload models, and the specification of new metrics to evaluate systems when running workflows. We will present the extensions in the Modular Workload Trace File presented in D2.1 with the additional fields introduced during the project. The whole format is included in one appendix of this document. In D2.2 we already presented the first version of our trace file generation methodology when we evaluated the module_list contribution. In this document, we introduce the complete strategy including the module_list and workflows. Finally, we have defined the set of metrics to be used when evaluating the dynamic management of dependencies in workflows. Traditional HPC metrics apply to individual jobs and do not reflect the impact of workflow optimizations.

In this document we will compare performance metrics in two types of cluster: when running in conventional (homogeneous) systems compared with a modular system. This evaluation will be done from an architectural point of view and without considering the contributions proposed in WP5. Results demonstrate the significant benefits of a modular system approach in terms of wait time and response time. The benefit is not only because of the modular architecture itself but also because of the design of the modules themselves. The fact the ESB is less expensive compared with traditional CPU+GPU nodes combined with traditional CPU nodes reduces by more than a factor of 10 the wait time and response time of jobs executed in the modular system compared to same workload executed in homogeneous systems.

The second set of results evaluates the benefits of the dynamic management of workflow dependencies. Results will show performance benefits in the execution of jobs belonging to workflows with no penalty in the rest of the workload. The evaluation of the dynamic management of dependencies in workflows shows promising results by improving the workflow response time because of the overlap between components. In the case of traditional jobs we are able to provide some overlapping in workflows without penalizing the rest of the jobs whereas when using heterogeneous jobs we also improve global metrics because we reduce the reservation

of resources.

In the context of performance models, the contribution is threefold. First, we present an analysis methodology to evaluate whether parallel applications make an efficient use of the resources, as well as to study the factors that hinder scalability. This methodology is based on the BSC efficiency model that characterizes an application with just a few efficiency factors that measure the main bottlenecks of parallel applications: data transfer, load balance and serialization issues, among others. This is applicable to homogeneous, hybrid and modular applications and we used it to conduct in-depth analyses on five codes with different use cases. Second, with the objective of gaining insight of the application's performance on the large scale, we extend the measurements taken from real runs, ranging from few hundreds to few thousand processes, with extrapolation analyses to project the aforementioned efficiency factors to larger scales, so as to foresee the potential scalability problems the applications may encounter. Since our predictions go very far from the real measurements, we use execution traces to corroborate the observations and drive a deeper analysis to understand the sources of efficiency loss. Lastly, we provide advice on how the applications could improve scalability. We support these ideas with simulated scenarios that estimate the benefits of the proposed changes, mostly targeted at removing imbalances, reducing dependency chains and accelerating key kernels. The analyses demonstrate that having access to a modular architecture helps to overcome software limitations. One of the examples shows how redistributing the application's components across the DEEP-EST modules would report significant benefits compared to an homogeneous run. Moreover, we also observe that some applications are very sensitive to small variabilities or system noise that propagate through the communications and perturb many processes. So rethinking the communication patterns to minimize dependencies and increase the application's asynchronism, as well as applying techniques such as computation-communication overlap to mask MPI costs, will be important to mitigate these effects that will increase in the exascale. Another hint for application developers, already exploited in DEEP-EST, is to promote the use of accelerators not only to speed up the computations, but also to achieve better balance and increased efficiencies in key parts of the code.

Finally, this deliverable includes the final version of the energy models for the different architectures and the validation of these models with some parallel kernels. The Energy benchmark and corresponding scripts have been adapted to the number of cores, sockets etc as well as specific performance metrics in the case of CPUs and fully migrated to NVIDIA GPUs. The average error when projecting from the different frequencies has been initially computed using a subset of the experiments reserved for this specific purpose. A deeper validation has been done with single-node applications. For validation, we have used the NAS Parallel Benchmarks BT-MZ, SP-MZ and LU-MZ in the three modules. These kernels have been executed with all the frequencies and performance counters and energy readings have been collected using DCDB. Projected energy shows an average error bellow 10% in all the cases.

1. Introduction

This deliverable includes the description of the main contributions of WP2 regarding the DEEP-EST benchmark suite, the evaluation of scheduling policy, the scalability analysis of some applications done with the performance models and the energy models for the different architectures as well as their validation.

Section 2 presents the final status of the DEEP-EST benchmark suite. The work done concerning the benchmark suite includes the selection of a set of synthetic kernels to evaluate, as much as posible, the relevant hardware characteristics existing in the project. These kernels have been integrated in the benchmarking tool JUBE and all the scripts to automatically execute them and collect the performance metrics have been created. The frequency of execution have been adapted to the project requirements, being more frequent al the beginning to identify performance variations due to changes in the software stack or the hardware. An important effort has been made in the integration of WP1 applications. In that case, the benchmark suite only includes the latest version of applications but given most of them have been significantly modified to be ported to the new architecture in some cases the integration effort had to be invested twice. The automatization of the execution of experiments saves time, but it is even better when the data analysis is also simplified. to support this, we have also included the utilization of DBs in the global strategy of benchmarking evaluation. The goal of this task is not to present application results because an exhaustive per-application analysis will be included in D1.5. However, we have include some graphs to demonstrate the usefulness of the benchmark suite to both identify performance problems because of changes in the environment (software stack or hardware configuration) and the performance evaluation when varying the application use case (number of processes, input data, etc).

Section 3 includes all the contributions to do a system evaluation and the evaluation itself. All experiments have been performed using simulations and the BSC-SLURM simulator has been used as in the previous deliverables. Contributions related to the setup of the experiments include extensions to support heterogeneous jobs, the integration of the dynamic management of dependencies in workflows, and the extension of the workload trace file generation methodology to support workflows with the API proposed. We also present the extensions we did in the modular trace file format to include energy metrics. The evaluation has been divided in two parts: first we have evaluated the benefits of a modular system compared to a conventional system. We have done this evaluation comparing three homogeneous scenarios (with 2CPU nodes, 2CPU+GPU nodes and 1CPU+GPU nodes) vs a modular system (2CPU and 1CPU+GPU nodes). The system sizes for each scenario have been determined assuming a maximum cost, where a 2CPU node costs 1unit, one 2CPU+GPU costs 2.5 units and one 1-CPU+GPU costs 1 unit. We have used the same percentage of accelerated nodes (70%) vs non-accelerated (30%) nodes as in the prototype. Using performance ratios provided by WP1 applications, we have adapted the execution times included in the trace files to take the different architectures into account.

The second set of experiments investigates the benefits of using the dynamic management of dependencies in workflows. This contributions makes the execution of workflows in this environment more dynamic reducing the wasted time of resources when running heterogeneous jobs and minimizing the wait time between components of the same workflow when using traditional

jobs. We call this new technology Dyn-WF and it reports performance benefits for workflows without penalyzing the rest of the jobs.

Section 4 includes a deep analysis of several applications from WP1: GROMACS, xPIC, NEST+Arbor and NextDBSCAN. Given there was only one modular application, we also deploy and evaluate EC-EARTH. With this set of applications we cover different use cases (strong and weak scaling), and programming models (MPI, OpenMP, CUDA). NEST+Arbor and EC-EARTH are complex use cases since they combine different programming models and binaries, posing a challenge both from the technical point of view of the tools supporting all these diverse components together, and from the analysis point of view of interpreting how each individual component impacts the global efficiency of the application.

Based on execution traces, we conduct efficiency and extrapolation analyses which provide on the one hand, an easy characterization of the application's good use of the resources summarized in a few efficiency factors, and on the other hand, preliminary hints about the application's scalability and which element hinders it the most. With this information, we have analyzed in detail the sources of efficiency loss, and have simulated scenarios where the identified problems are minimized or even removed to evaluate the benefits of potential improvements.

Our conclusions are both hints to help application developers to improve the efficiency of the next versions of their applications, as well as to help application users to select proper configurations and resources to maximize the efficiency of their experiments.

It is also worth mentioning that given the limited size of the prototype some conclusions can be affected by the reduced size and number of samples.

Finally, section 5 includes the methodology to create energy models, the energy models for the CPU and GPU in the DEEP-EST architecture and the validation of these models. The validation has been done using a subset of the experiments generated by the energy benchmark and the NAS Parallel Benchmarks. We have also included a deep analysis of the average error when projecting energy from different frequencies to the rest of frequencies. The validation with benchmarks have reported average errors slightly bigger than the preliminar evaluation with the mini kernels of the energy benchmark but always below the standard 8% and in average 4.91%.

2. Benchmarking

Benchmarking by itself is a very broad topic. One needs to define the purpose of benchmarking upfront. Otherwise, one can easily end up with spending most time with technicalities for producing potentially statistically identical results.

In the course of Tk2.1, there were a lot of benchmarks integrated. Initially, it was not completely clear, which benchmarks would lead to changes in the performances. One could just perform educated guesses at that time. The hope was to measure a possibly wide scope of the hardware and software characteristics to see as many significant system changes as possible. In addition, the parameter space of the benchmarks needed to be set up as large as possible to increase the probability of measuring interesting effects. On the other hand, benchmarking inherently is taking resources from users who could have potentially performed other tests in the meantime. The more benchmarking is performed the smaller is the amount of resources available to the users of the system. This is a complex high dimensional optimization task.

The constant and frequent benchmarking of the system lead to several situations where problems of the hardware or the software could be found in an early stage. The reason for that is that the benchmarks were testing inherently large parts of the software stack by using it. In this sense benchmarking was contributing significantly to the stability and functionality of the system by communicating these problems as early as possible. The benchmark suite is acting as a unit testing framework for the prototype hardware and software from the perspective of the user.

With increasing amount of benchmarks and partitions the amount of maintenance was increasing. The main reason for the enduring maintenance efforts was the prototype status of the system in question. This was leading to a race between finding and solving problems occurring while running a benchmark and the need for integrating new benchmarks for these new partitions.

2.1. Benchmarking Guide

Due to its versatility and complexity, one can get easily lost within the topic of benchmarking. Therefore, a clear guide of how to work on a benchmark suite is a result by itself of this task potentially simplifying future efforts for developing benchmark suites. The guide presented here was mainly developed by the efforts within the DEEP-EST project. The guide needs to be refined depending on the specific situation for which it is going to be used in the future. Nevertheless, it strives for as much generality as possible.

The topic of benchmarking can be separated into the following parts. There is a part of how to define the workflow of integrating a benchmark into the benchmark suite. Another part is the workflow of one single benchmark execution which also defines the structure of a single benchmark. Finally, the benchmarking infrastructure, which is given by the benchmark suite minus the single benchmarks, is a topic which needs special care and is going to be considered within this deliverable.

2.1.1. Benchmark Infrastructure

When integrating a benchmark suite it is natural to start right away with single benchmarks. While this is a valid approach and leads to first quick results it also leads to the problem of single benchmarks lacking for uniformity. This is especially true if there are a lot of benchmark prototypes developed by application developers and submitted to the benchmark suite maintainers. This will make the process of creating a benchmark infrastructure, which is going to be needed to execute the benchmarks on a regular basis, a very cumbersome and error prone process.

On the other hand, even if uniformity is achieved at the beginning of the process of integrating the benchmarks there will be another benchmark at some time with a workflow, which does not fit into the uniform structure. Furthermore, without first having integrated first benchmarks it is difficult to extract the relevant parts which a benchmark infrastructure will need. An enumeration of the elements needed for a benchmark infrastructure has the advantage of making a benchmark suite developer able to develop the benchmark infrastructure before the benchmarks are developed. This enables a clearer structure of the single benchmarks and a better integration into the benchmark suite with a smaller amount of errors.

With a functional benchmark infrastructure the application developers, which are naturally not motivated to submit incomplete or non-optimized application code significantly earlier than the end of the project, will have the motivation of being able to measure their progress improving the application. This performance change over time is automatically monitored and takes this part of the work away from the application developer. On the other, hand the application developers have the opportunity to analyse on a regular basis whether some changes in the prototype improved or reduced the performance of their application. This information can be fed back to the system administrator or the system software developers to ensure a specific performance of the applications.

From the experience of the DEEP-EST project the following parts are considered to be needed for a benchmark infrastructure.

- **How to execute a benchmark?** There are several ways to execute a single benchmark by itself. The tool of choice within the DEEP-EST project is JUBE, see [31]. But one can also script a benchmark execution by use of a bash script. Also Python and ReFrame, see [26], would be possible alternatives to do this.
- **How to store the system environment at execution time?** Once the benchmarks are running regularly the results will likely show step functions. On the other hand, the existence of a maintained changelog is not guaranteed. In conclusion, storing the system environment as verbose as possible is a way of how to track changes on the system and relating them to steps in the performance plots.
- **How to integrate new benchmarks?** Since the different styles of the application developers to integrate benchmarks are leading to a large complexity of the benchmarking infrastructure and even more efforts to the application developers this is an important question to answer. Naturally, one wants to simplify the work of the benchmark suite and application developers. A tool for creating prototypical benchmarking scripts would be a solution to that. This reduces the efforts for the application developers and ensures a specific form of the benchmarking scripts reducing the complexity needed for the benchmark

infrastructure. A potential solution is to offer sample JUBE scripts and an introduction into JUBE. Nevertheless, application developers submitted scripts which were significantly diverging. This effect is likely due to the different workflows of the applications and need to be kept in mind. Since the development of the applications is ongoing, the collaboriation with the application developers also continues. The applications are changing and the benchmarks are potentially breaking. They need to be maintained by the benchmark suite developers and the application developers over the whole project duration.

- **How to ensure functionality and quality?** A software project of this type includes a lot of people with a large potential for fast growing. Such type of projects should be equipped with a unit testing framework. Since the programming tools used are very diverse, the definition of the unit testing methodology is not trivial. Error code throwing and catching should be performed strictly on all levels of programming and scripting languages. Even parts which never broke in the past should be considered. If this is not possible, the unit tests should be extended aggressively every time when something breaks. The unit testing framework by itself needs to be versatile and running on a machine or infrastructure one level above the bash terminal of the used system (here the DEEP-EST prototype).
- **How to schedule the benchmarks?** On the DEEP-EST prototype we are using Slurm as a job scheduler. Nevertheless, just submitting the benchmarking jobs is not goal oriented for a benchmark suite. One has to develop some infrastructure one layer above Slurm to define dependencies of the benchmark executions. Furthermore, special care needs to be taken to define runtimes and conditions for cancelling benchmark executions if there are arising any problems or any unforeseen behaviour.
- **How to handle errors?** This is a very important point! In a prototype environment parts of the benchmark suite are breaking regularly since the benchmark suite opts for measuring as many parts of the system as possible given the constraints. If there is an error occuring this can stall the execution of all other benchmarks leading to a lack of results. Error handling taking into account as many error sources as possible, will increase the stability of the benchmark suite execution and therefore the stability of result delivery. Furthermore, one should think about a notification strategy in cases of errors. An automatic mail to the corresponding application developer could trigger and even speed up recovery of the benchmark suite result delivery and avoid an error which is not recognized. This is especially important if the benchmark suite is growing so large such that one cannot have a look at all results anymore manually and therefore an error could possibly not be found for a long time of regular benchmark execution.
- **How to make the benchmark infrastructure portable?** Different platforms are allowing for different ways of how to solve things. Or they could even use different schedulers. To enable comparison to other platforms special care needs to be taken here into account to allow for easy porting. One way how to achieve this is to separate all platform specific attributes into a configuration structure which is the only thing to be changed when porting a benchmark to another platform.
- **How to archive the results?** A stable, defensive and safe archiving of the results is mandatory. One could easily loose this topic after having a working benchmark. So this has to be taken into account for ensuring the availability of the benchmarking results for a long time frame.

- **How to analyse the results?** With the sheer amount of data produced by a benchmarking suite the need for automatized result analysis is mandatory to extract as much information as possible. A strong recommendation is a visualisation tool and an automatic notification when errors are being produced. The automatic notification is needed since it is almost impossible to have a manual look at all benchmarks with all parameter permutations to filter out which parts were not running in the last benchmark iteration. But more elaborated strategies of what to do with the data could lead to interesting results and even more outcome of a benchmarking suite.

2.1.2. Benchmark Integration Workflow

Having a clear picture in mind on how to integrate benchmarks can simplify the task of measuring interesting changes significantly. At least, having a clear picture in mind builds a starting point for adapting a benchmark integration workflow for another project. Therefore, one should picturise such a benchmark integration workflow which is going to be explained in depth in the following.

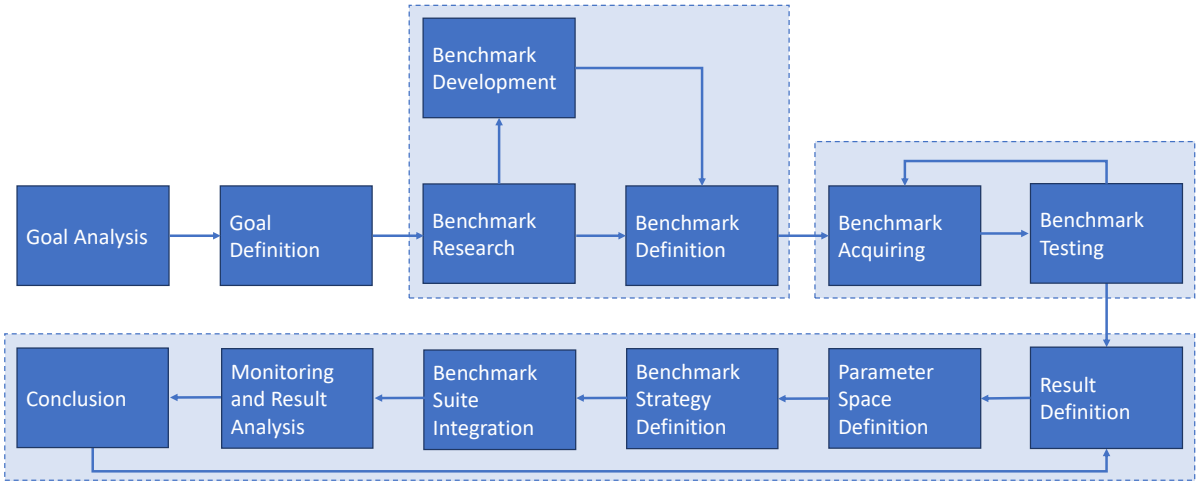


Figure 1.: Benchmark Integration Workflow

The benchmark integration workflow starts with the goal analysis. A short list of potential goals is given by the following and can be extended depending on the situation and the background.

- Monitoring of functionality of the system and/or the benchmarking software
- Monitoring of performance changes
- Comparison of different systems
- Stressing the system/simulation by creating artificial production conditions

- Single performance measurements
- Creating an environment to reproduce complex errors

Then a specific goal should be chosen and defined. Without this step the benchmark suite can easily grow exponentially in functionality not being needed. From this goal analysis follows directly the goal definition by choosing the goals of interest.

After having defined the goals of the benchmark suite a benchmark research needs to be performed. Which functionality will the benchmarks need and are there already benchmarks available to fulfil these goals? If the needed benchmarks are not available, they need to be developed at this step. In the context of the application benchmarks the application developers need to perform these developments.

The next step is the benchmark definition or selection. This is based on the benchmark research and the benchmark development performed before. This is followed by the benchmark acquiring step. In the case of publicly available synthetic benchmarks this step is just given by a download. For the case of the application benchmarks a repository needs to be defined which can be accessed frequently by the benchmark suite and will be updated frequently by the application developer. For the case of non-publicly available benchmarks it is questionable whether they are, at all, a suitable choice to be used. Benchmarking is inherently a comparison. This comparison can be done in time. But it also can be done with respect to different research groups or different systems. If the second case is foreseen, non-publicly available benchmarks should not be chosen. If they are chosen to be integrated into the benchmark suite, one needs to make sure that the goals can be achieved under the terms of this non-public benchmark.

As the next step is about benchmark testing, we try to compile, find proper software combinations to load in before compilation, test some parameters of the benchmarks and the benchmark suite and have a first look at execution results. For the case of the application benchmarks it frequently happens that the application developers have set up a private software environment or hard coded some paths into the scripts and software they use for developments. This step can take a significant amount of work and can lead to often fallbacks to the benchmark acquiring step since changes need to be made by the application developers. At this step also some knowledge about the parameters and their influences should be studied. Another important point is here to get an overview of runtimes for specific parameter values and to be able to estimate them.

Having performed first tests and knowing how to execute a benchmark in general the next step is the result definition. Results can be performance in some application specific metric, runtime, energy consumption or anything else. This needs to be defined for every benchmark independently. A complete list of possible results cannot be stated here since it depends on the synthetic benchmarks and applications at hand at the time when the benchmark suite is developed. Which results are useful to be measured has to be studied case by case. The result definition should be en par with the goal definition. Furthermore, the parameter space needs to be defined. This is a delicate topic since a benchmark suite opts to measure as much as possible but the dimensions of the parameter space can easily blow up leading to a monopolising of the prototype which is definitely not wanted. Also the amount of possible parameters to measure depends on the runtime of the parameter permutations and the overall amount of benchmarks to be performed and on the time which is allocated for the regular benchmarks.

As a next step the benchmark execution strategy should be defined. When and how often should the benchmarks be performed? Are they performed in parallel or in a serial mode? For example, for the case of an IO benchmark accessing the same file system, it depends on the goal of the benchmarks whether they should be performed in parallel or in a serial mode. How often is the software going to be compiled? For a large amount of application benchmarks the compilation can take significant amounts of time. On the other hand, one wants to perform compilations as often as possible to take a picture of the current software state.

From all these considerations a benchmark suite with its infrastructure can be developed such that the specific benchmark is integrated into this benchmark suite. Depending on the functionality supported this can lead to significant efforts. Especially in the case of multiple partitions specific parameter choices could lead to a need to handle them completely different.

Once this is done the benchmarks can go into production. The results need to be monitored and analysed frequently. This leads to conclusions. These conclusions can just be about the system status and configuration or they can lead to a need for changing parameter definitions or even result definitions since one could realise that the results are not describing specific effects happening on the system. So here is also a loop where after the conclusion one could end up at the result definition step again working the way to the conclusion step. Another result from the conclusion could be that a communication needs to be initiated with application developer, system administrator or other people maintaining the software or the hardware of the prototype to resolve a problem seen through the use of the benchmark suite.

2.1.3. Benchmark Execution Workflow

Right at the beginning of a project where many developers are involved to deliver single benchmarks for a whole benchmark suite it is advantageous to define the typical benchmark execution workflow. Having this defined, in before the developments, can lead to typical steps of the benchmark execution. This will conclude into equal or comparable structures for the single benchmarking scripts.

In addition, these steps of a typical benchmark execution can also be used to create a tool for making prototypical benchmark execution scripts. This again decreases the divergence of the benchmark execution scripts. The goal here is to make the benchmarking execution scripts as comparable as possible in view of their structure leading to a significantly easier development of the benchmarking infrastructure. If the benchmark application developers start directly from benchmark execution scripts which have comparable structure it is much easier to automatize the usage of the most current application benchmark within the benchmark infrastructure workflow.

From the DEEP-EST project there were typical steps extractable from the available benchmark execution scripts. Of course in some cases additional steps could be needed since benchmarking is a broad topic. These steps described here are more a first starting point to define a benchmark execution workflow for a given benchmark suite. After first analyses, this definition of a benchmark execution should then be updated for every benchmark suite being integrated.

The execution workflow, figure 2, starts with loading required software modules. For the DEEP-EST prototype and the production machines at JSC this is typically the loading of a lua file

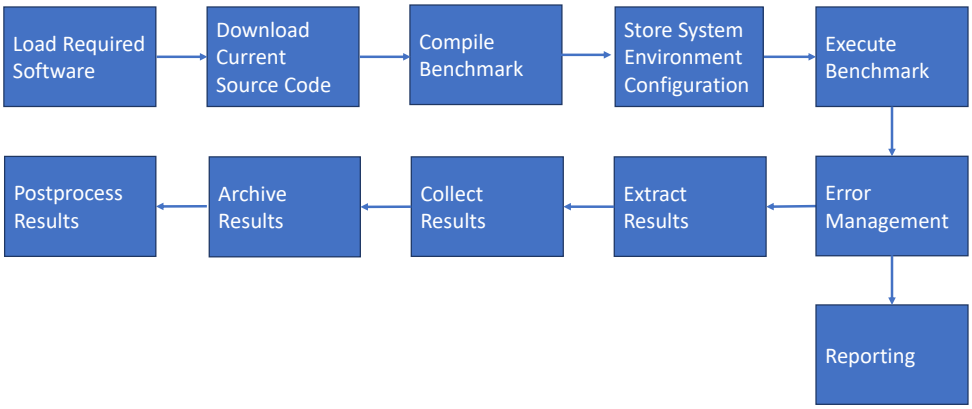


Figure 2.: Benchmark Execution

describing the updates of the environment to use preinstalled software. Already at this point, it is mandatory to define whether to load the default modules or whether to manually update the concrete version number of loaded software versions within the benchmark execution scripts.

As a next step one could download the current source code of the application or the synthetic benchmark. Depending on the system configuration here first issues can come into play. The repository could not be reachable without an ssh key or on the compute nodes. Furthermore, one has to assure that the original repository is regularly maintained by the original developer.

The next step is the compilation step. The compilation can be very fast or can take a very long time. This has to be considered when planning the workflow. For some application, like in the DEEP-EST project, the compilation was already performed depending on a parameter. In such cases these compilation steps would not be considered as a compilation. It would be implicitly performed within the step of the execution of the benchmark since there the parameter expansion is coming into play.

After having loaded the modules and performed the compilation and directly before the exeuction of the benchmarks the system environment configuration should be stored. The existence of a maintained changelog is not guaranteed. The regular archiving of this changelog, if it exists, is neither guaranteed. Therefore, one likely needs to take care of this step within the benchmark suite. This is an important step to be able to correlate system environment changes to changes in the performances of the benchmarking results afterwards.

As a next step the execution of the benchmark is performed. Here the compiled binaries are called with different parameter permutations. For the sake of potentially saving time if there is a reservation in place for the benchmark executions, it is important to not extract any results at this point. The result extraction step can take a lot of time since a lot of small files need to

be opened and analysed. High performance parallel file systems are not optimized to do such operations quickly.

After having executed the benchmarks it will likely happen, especially for a prorotype, that errors are occurring. This leads to some complexity for the benchmark execution workflow and needs to be taken into account. Depending on the benchmark suite integration and the goals some errors can be handled in a clear way. One recommendation concluded from this project is to perform some automatized reporting like writing a mail to application or benchmark developer that the benchmark did not succeed. This was especially useful since the complexity of the benchmark suite was growing immensely. Every automatizable step needed to be automatized very aggressively to have time left for further developments. At this point, when one chooses to write automatic mails, one should also take into account to somehow summarize the occurring errors into a small amount of mails. Otherwise every parameter permutation could lead to one mail.

In conclusion, outside of the reservation, if it is available, the results can be extracted and collected. The collection can be performed by use of a database. Here it needs to be analysed which type of results are expected and which type of database would be useful for such a case. This database would then need to be archived. Since data loss happens, especially for prototype systems, this step can easily be forgotten, but is as important as the steps before.

As a last step the results need to be postprocessed. This can be defined depending on the needs of the project. It just can mean the preparation of the results for a visualisation tool. It can also be some sort of complex and elaborated data analysis. Since a lot can be done with the data, this is a whole research topic by itself.

2.2. Benchmarking Suite

A benchmark suite should be considered as a solution to a given problem under given conditions. This is due to the fact that depending on the goals, one would like to achieve, different workflows will be needed to be implemented into the benchmark suite. Therefore, the benchmark suite delivered here within the DEEP-EST project is exactly this. There were specific boundary conditions which needed to be fulfilled. The reservation timeframe, the functionality of cron jobs and the availability of a user specifically created for performing the benchmarks are a subset of the boundary conditions being relevant for this benchmark suite.

2.2.1. Benchmarking Strategy

The benchmarking strategy is mostly the same as stated in teh deliverable D2.1, see [2], of the DEEP-EST project. The extensions and alternations due to technical reasons are communicated within this section.

The frequencies of the benchmarks, which are the numbers of runs per week, are depending on the benchmark by itself. Initially, it was foreseen to run the synthetic benchmarks on a daily basis while the application benchmarks are meant to be running on a weekly basis every Saturday to avoid disturbances of developments on the prototype. As the amount of benchmarks

was growing the frequency of the synthetic benchmarks needed to be reduced to still fit into the daily reservation timeframe. This means that the frequency of the synthetic benchmarks is at least given by one per week depending on the benchmark in mind but in every case not given by 7 times a week.

Due to the amount of benchmarks integrated the compilation time became a significant contribution. In connection with the need for a large availability of the prototype for the early access users and for other developing and testing the compilation time had to be reduced. Therefore, recompilation of the executables was performed on the 1st of every month.

At communication by application developers, software developers, software stack maintainers or system administrators of a potentially interesting new development within the software stack, a change of the loaded modules was initiated within the benchmarks. This is due to the situation of having a default software stack and a development software stack. Both software stacks are needed to offer stable software versions while enabling the usage of current developments on the prototype.

To increase the amount of information gathered for every benchmark run system environment information was gathered for the benchmark runs. For this we developed an extra script. The results produced by JUBE were stored into a database. The files of the benchmark runs with the raw information and the database is backed up every evening.

Some efforts were performed to make the benchmark suite portable to the old JURECA cluster module. But this system was decommissioned end of 2020. Following the benchmark suite and the benchmarking scripts line by line it is easy to port the benchmarks for another system. But no special efforts were made to collect system dependent information within separate files to make it especially comfortable to port from one system to another. When performing a porting of a benchmark to another system still there needs to be a parameter optimization performed. This step is a complex one and should be performed by a human. Since the parameter optimization is also dependent on the goals defined by the benchmark suite this step is very difficult to automatize.

2.2.2. Synthetic Benchmarks

Synthetic benchmarks are synthetizing different hardware usage types of applications running on a system. This has the advantage of being able to perform tests on parameter spaces which are not covered by the applications at hand. Furthermore, they are defining a common basis for comparable benchmarking results for different systems. At last, they are potentially lightweight in the software dependencies and variable in the parameter spaces which can be chosen for execution. These characteristics make them ideal for regular execution on systems to assess the current status and to detect potential problems.

The set of synthetic benchmarks publicly available have a tendency to focus on monolithic and common systems. There is a need for developing modular synthetic benchmarks for non-monolithic system structures when MSA is going to be the reference for future high performance clusters. Furthermore, synthetic benchmarks for novel hardware like NAM and FPGA cards need to be developed and maintained. Here the field of available software is very sparse since modular systems with complex architectures for the modules are not yet common.

The synthetic benchmarks being part of the benchmark suite are given as follows.

- **h5perf:** An IO benchmark measuring file system performance and allowing for different data write and read patterns, see [38, 2]
- **hpcc:** A collection of benchmarks including hpl, DGEMM, stream, PTRANS, RandomAccess, see [39]
- **hpcg:** Performance benchmark based on solving a dirichlet boundary condition problem, see [40]
- **hpl:** Performance benchmark for the solution of a system of equations by use of an LU-Decomposition, see [41]
- **hpl4cuda:** Performance benchmark like hpl including GPUs and using CUDA to harness their compute power, see [42]
- **ior:** IO benchmark for writing and reading files and measuring the performance of these operations, see [43, 2]
- **mdtest:** Benchmark for measuring metadata performance of a file system, see [43, 2]
- **mpiLinkTest:** Measuring network bandwidths when sending messages between tasks, see [44, 2]
- **mpiLinkTestCrossPartition:** A flavor of mpiLinkTest enabling inter partition communication, see [44]
- **mpiLinkTestGPU:** A flavor of mpiLinkTest measuring bandwidth when sending data from one GPU to another GPU of another node by use of CUDA, see [44]
- **stream:** Measurement of the performance of operations on data on memory of a node, see [45]

The chosen parameters for all the benchmarks are stated within the appendix. As an example for the parameters chosen the mpiLinkTest benchmark is discussed here. The mpiLinkTest benchmark measures bandwidth performances when sending data between two tasks by use of MPI. This benchmark is performed on every partition.

Due to partitions and other technical considerations the parameter permutations can take complex dependencies. Therefore, a superset of the potential parameter permutations is printed into the tables of potential parameters. To clarify this: For every parameter value of one parameter within the parameter tables there exists a parameter permutation being part of the benchmark. This also means that all arbitrary parameter combinations are not necessarily integrated into the benchmark suite. When one needs to know whether the concrete parameter permutation in mind is being measured one needs to have a look into the results of the benchmarks.

The parameters of the mpiLinkTest can take the following values.

For example, on the cluster module the number of nodes is ranging from 1 to 32 in the given discretization as stated within table 1 whereas on the DAM partition the number of nodes participating in the benchmark cannot be 16 or 32 since there are only 8 nodes on this partition connected to Extoll or Fabri3.

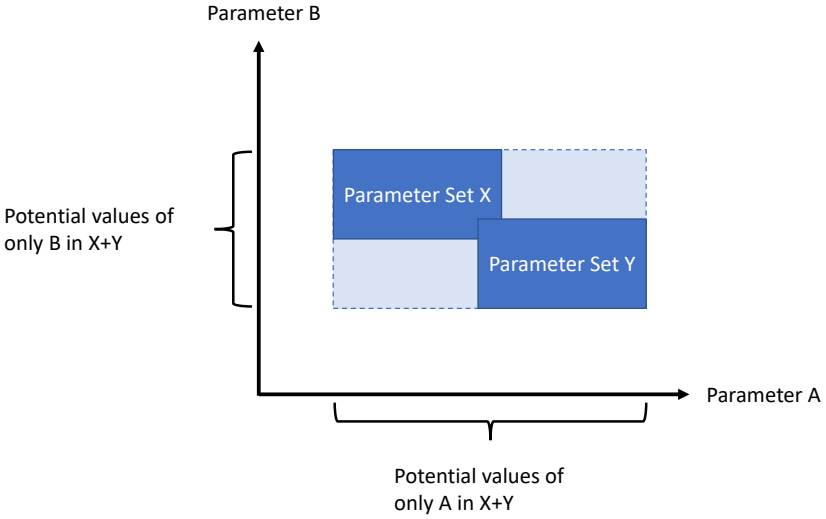


Figure 3.: Visualisation of the parameter sets and how they are mapped into the parameter tables of this deliverable.

nodes	1,4,8,16,32
taskspernode	2
AllToAll	0
Warmup	2
Randomized	0
Size	1,16384,4194304
Iterations	1000
Serialized	0,1

Table 1.: Potential Parameter values of the mpiLinkTest Benchmarks

The message size is another important parameter here. The value of 1 is optimized for measuring the pure latency of the benchmark as exact as possible since the amount of data communicated is almost vanishing. On the other side when increasing the message size as much as possible the effect of the latency becomes more and more negligible leading to the measurement of the pure bandwidth of the network connection.

As an example for benchmarking results, the results of the mpiLinkTest benchmark are given by the runtime in seconds and the maximal, minimal and average bandwidth in MB/s.

2.2.3. Application Benchmarks

The application benchmarks are defining a representative set of software and their demands to the hardware of future MSAs needed.

- **ASTRON/Correlator:** Correlation of tens of hundreds of receivers measuring signals from space, see [2]
- **ASTRON/Imager:** Creating sky images from the data created by the correlator, see [2]
- **CERN/CMSSW:** An event data model and services needed by simulation, calibration, alignment and reconstruction modules that process event data of particle-particle collisions within particle accelerators, see [2]
- **KUL/xPic:** Particle in cell code for the detailed simulation of the plasma environment of the planets with the aim of predicting space weather, see [2]
- **NCSA/GROMACS:** Simulation of multi body systems by use of molecular dynamics, see [11, 2]
- **NMBU/NEST:** Modelling of brain tissue as an abstract collection of neurons, see [8, 2]
- **NMBU/Arbor:** Compartmental neuron simulations through input from NEST, see [9, 2]
- **UoI/piSVM:** An implementation of a parallel support vector machine used to classify hyper-spectral data of natural and man-made land covers via supervised learning, see [2]
- **UoI/HPDBSCAN:** A highly parallel implementation of the established DBSCAN clustering algorithm, see [12, 2]

As an example for the parameters of an application benchmark the GROMACS application for simulating multi body systems by use of molecular dynamics can be discussed. Throughout the project duration new GROMACS versions were published. These were integrated into the regular benchmarking procedure. Correspondingly the topology files were updated. Furthermore, there were three multi body system being benchmarked through the project duration which are given by Magainin, Bombinin and Ribosome.

The amount of compute nodes and MPI tasks per node were varied. To avoid an oversubscription of the sockets the maximal amount of MPI tasks times the number of threads per MPI task was below 48 for the cluster module. This is an example for the boundary conditions restricting the set of possible parameter permutations.

The result of a benchmark execution is defined by the performance in nanoseconds simulated for the molecule per day of simulation time and by the simulation time in seconds.

2.2.4. Benchmark Integration Status

The benchmark integration progression status is described by the benchmarks to be integrated and the partition permutations to be integrated for. This is best described by two tables stating all the status for the synthetic and the application benchmarks (Table 3 and 4).

The status are defined as follows.

Molecule	Magainin, Bombinin, Ribosome
GROMACS Version	2018.4, 2019.3, 2019.6, 2020.1
# Compute Nodes	1,2,4,8
# MPI Tasks per Node	1,4,8,12,24,48
# Threads per MPI Task	1,2,3,4,6,8,12,24,48
tprfile	magainin.tpr,magainin-2019.tpr, bombinin.bombinin-2019.tpr, ribosome.tpr,ribosome-2019.tpr

Table 2.: Potential Parameter values of the GROMACS Benchmarks

Benchmark \ Concurrent Partitions	CM	DAM	ESB	CM & DAM & ESB
mpiLinkTest	+	+	+	/
mpiLinkTestGPU	/	+	+	/
mpiLinkTestCrossPartition	/	/	/	+
ior	+	+	+	/
h5perf	+	+	+	/
hpl	+	+	+	/
hpl4cuda	/	+	+	/
hpcg	+	+	+	/
stream	+	+	+	/
mdtest	+	+	+	/
hpcc	+	+	+	/

Table 3.: Status of the synthetic benchmark integrations. Multiple partitions stated in the header of a column are representing a benchmark execution on multiple partitions.

- + means integrated: The partition permutation for the benchmark was integrated into the benchmark suite.
- / means unplanned: The partition permutation for the benchmark was not planned.

The curse of dimensionality is clearly coming into play here leading to quite some effort to integrate all the benchmarks into the suite. The options for the benchmark driver add even more complexity to this merging and adapting process. Another source of complexity comes from the different workflows and different styles of writing benchmarking scripts such that unified JUBE scripts could not be collected which again does not allow for fully automatized integration workflows.

Concurrent Partitions / Benchmark	CM	DAM	ESB	CM & ESB
NEST	+	unplanned	/	/
Arbor	/	/	/	+
GROMACS	+	+	+	/
Correlator	/	+	+	/
Imager	/	+	+	/
xPic	+	/	/	+
piSVM	+	/	/	/
HPDBSCAN	+	/	/	/
CMSSW	/	+	+	/

Table 4.: Status of the application benchmark integrations. Multiple partitions stated in the header of a column are representing a benchmark execution on multiple partitions.

2.3. Benchmarking Evaluation

Right at the beginning of the project benchmarks were integrated to monitor the system for a potential performance change. It was not clear which benchmarks would deliver that. So, there are a lot of plots just showing statistical variations.

Furthermore, the sheer amount of benchmarks being performed makes it very difficult to have a look into all plots and to discuss them all. Within this chapter the interesting benchmarking results, which were found at a first analysis, are visualised and discussed.

For future projects a benchmarking task of this complexity could be accompanied by a data analysis task developing methods to extract interesting behaviour within the data. Automatically extracting useful information from a large amount of data generated is a field of research which is growing immensely.

2.3.1. mpiLinkTest

The average bandwidths for the mpiLinkTest benchmark for a serialised execution strategy and for a message size of 4096 KB with two tasks per node is given in figure 4.

Before May 10 2020 the average bandwidth is maximal for one node and decreases for increasing number of nodes participating the benchmark execution. After June 28 2020 the average is minimal for the number of nodes equal to 1, which are participating the benchmark execution. For increasing number of nodes the average bandwidth has a tendency to grow. Clearly something happened on the prototype between May 10 2020 and June 28 2020. Both averages of the bandwidths, before and after the step, have a tendency to converge to the same value for a high number of nodes.

Having a look into the kernel information files of both mpiLinkTest runs on these both days the

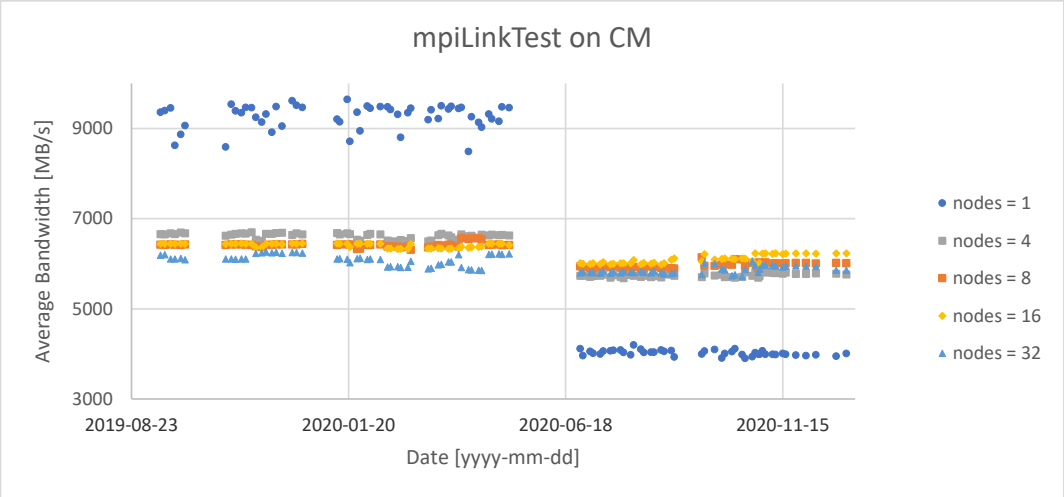


Figure 4.: mpiLinkTest benchmarking results with messagesize_KB = 4096 and serialized = 0

following different system configurations can be found.

On May 10 2020 there were the following environment variables defined and system software installed.

- task 0: __PINNING__=000000000000000000000000000000000000000000000001
- task 1: __PINNING__=000000000000000000000000000000000000000000000010
- psmgmt version: 5.1.29

On June 28 2020 there were the following environment variables defined and system software installed.

- task 0: __PINNING__=000000000000000000000000000000000000000000000001
- task 1: __PINNING__=000000000000000000000000000000001000000000000
- psmgmt version: 5.1.30

The default pinning changed between May and June 2020 when a new psmgmt version was installed on the prototype. The software psmgmt is the process management software being installed on the DEEP-EST prototype. The pinning defines the default positions of tasks being allocated on the compute nodes. The environment variable __PINNING__ is a bit mask with 48 positions. Every position is representing a hardware-thread of the compute node. A 0 within

the bit mask is representing no task being allocated on this thread position. A 1 within the bit mask marks the process having an affinity to the hardware-thread.

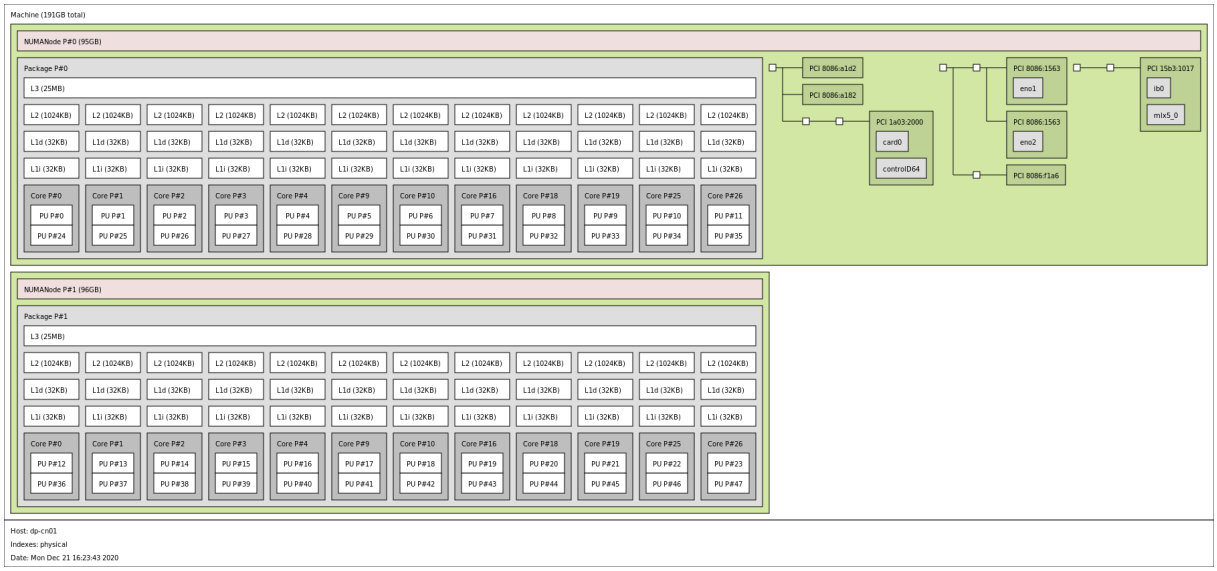


Figure 5.: Cluster module node architecture

Using the software lstopo one can visualise the node architecture on the cluster module nodes, see figure 5. There are two sockets on the node and two threads per socket core being numbered from PU P#0 to PU P#47. The position PU P#0 is the outermost right position of the __PINNING__ environment variable. The position PU P#47 is the outermost left position of the __PINNING__ environment variable. The positions PU P#1 to PU P#46 can be linearly extrapolated.

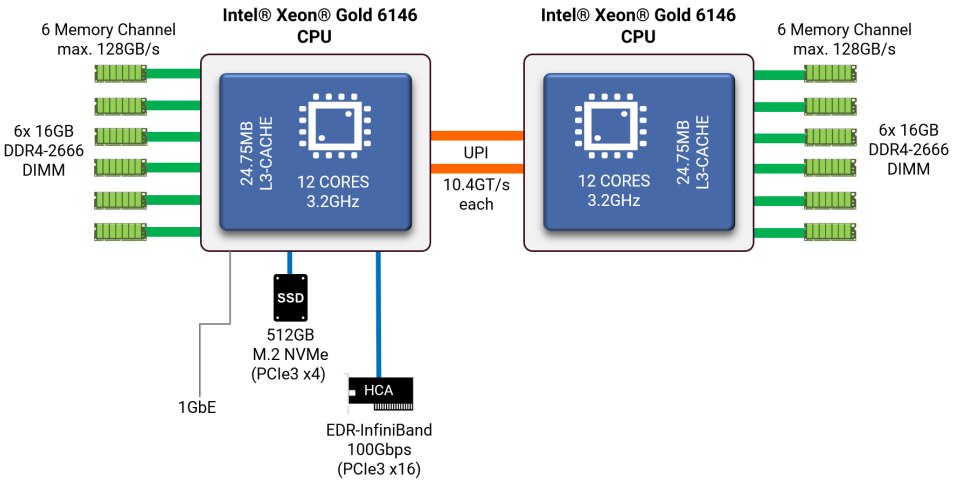


Figure 6.: CM node hardware

Before May 2020 the mpiLinkTest tasks on one node were allocated on the same socket but on different physical cores. This lead to a higher average bandwidth measurement, when the

number of nodes is small. With increasing number of nodes the average bandwidth measurement is tending to the average bandwidth of the network. For intra-node communication the communication uses the same cache on the same socket. For inter-node communication the sockets at the same position compared to the HCAs are used, see figure 6.

After June 2020 the mpiLinkTest tasks on one node were allocated to cores of different sockets of the same node. This reduced the contribution to the average bandwidth coming from the intra node communication. Again, for increasing number of nodes the average bandwidth tends to the average bandwidth of the network. In conclusion, the communication of data between two sockets of one node is slower than the communication between two socket of different nodes through the network connection. For intra-node communication different sockets were used for the communicating processes. Therefore, the data needed to go through the UPI between the both sockets. For inter-node communication one process is located on the socket not being directly connected to the HCA. Therefore, here the data still needed to first be transferred through the UPI to the other socket where the HCA is located, see figure 6.

For verification single benchmarks were performed with one node, two tasks per node, a message size of 4096 KB, 1000 iterations and a pinning of the first task to the 0th thread and a pinning of the second task to the threads 1,12 and 24 showing the influence of the pinning to the bandwidth measurement of the mpiLinkTest benchmark, see table 5.

pinning of the second task	Maximal Bandwidth in MB/s
1	7756.01
12	4097.18
24	10900.04

Table 5.: mpiLinkTest bandwidth verifications for one node, two tasks per node, a message size of 4096 KB, 1000 iterations and a pinning of the first task to the 0th thread.

The bandwidth for the communication of tasks assigned to hardware-threads sharing the same core is the largest. The bandwidth of the case for two tasks cores within the same socket is intermediate. The bandwidth for two tasks sitting on different sockets of the same node is the smallest.

2.3.2. mpiLinkTestGPU

In a functional sense ParaStationMPI was CUDA aware before the DEEP-EST project. It was possible to pass a pointer to the data on the GPU to an MPI_Send directive. But this version of ParaStationMPI was not using the GPUdirect functionality. The sending process of data from the origin GPU to target GPU on another node was initiated by first copying the data from origin GPU to origin node memory. From this point the data was copied from origin nodes memory to target nodes memory. At last the data is copied from the target nodes memory to the memory of the target GPU.

Within the DEEP-EST project the CUDA awareness of ParaStationMPI was combined with the usage of GPUdirect technology. This functionality was made available on the prototype and activated within the benchmark suite during the regular bandwidth measurements. There is a clear step function within the benchmarking results, see figure 7. Before the step in the

average bandwidth happened it was comparable for the message size of 1024 kB and 4096 kB. Afterwards the average bandwidths were different. The larger message size showed a higher average bandwidth.

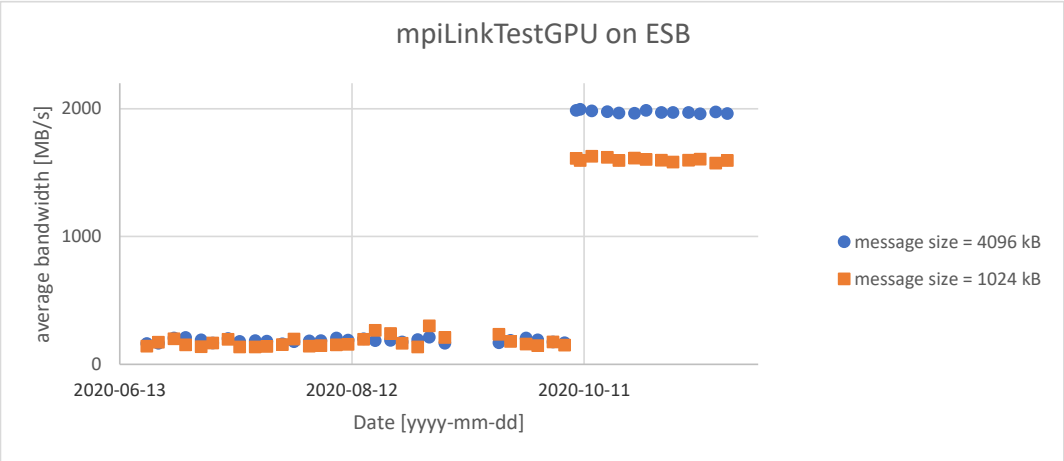


Figure 7.: mpiLinkTestGPU benchmarking results with node unit = GPU, network = IB and serialized = 0

Analysis of the detailed system kernel infos from August 29 2020 and November 17 2020 are leading to different usage of software stages versions and different settings for the PSP_UCP environment variable. From the system kernel information data from August it can be extracted that the environment variable PSP_UCP=0 was defined. For the version in November the environment variable PSP_UCP=1 was defined. Setting the environment variable for a separate and single test with the newer software stage again to PSP_UCP=0 validates the drop of the bandwidth. This is an example for the complexity of the benchmark suite. The benchmark suite needs to be maintained on a daily basis to ensure the usage of the most recent functionalities.

For increased bandwidth the effect of latency to create an MPI connection is becoming more relevant. This is due to decreased time needed for sending data from origin node to target node. Based on this consideration the effective bandwidth measured by two different message sizes should differ more for higher bandwidths achieved. In fact this is the case here for the message size of 1024 kB and 4096 kB, like measured, see figure 7. This is another example for the need of adaptations of parameter spaces of benchmark suites through project duration.

2.3.3. NEST

For the NEST benchmarks scaling tests were performed on a regular basis. The runtime is increasing when doubling the number of tasks and the network size. But the increase is significantly lower than the factor of 2 showing scalability of NEST, see figure 8.

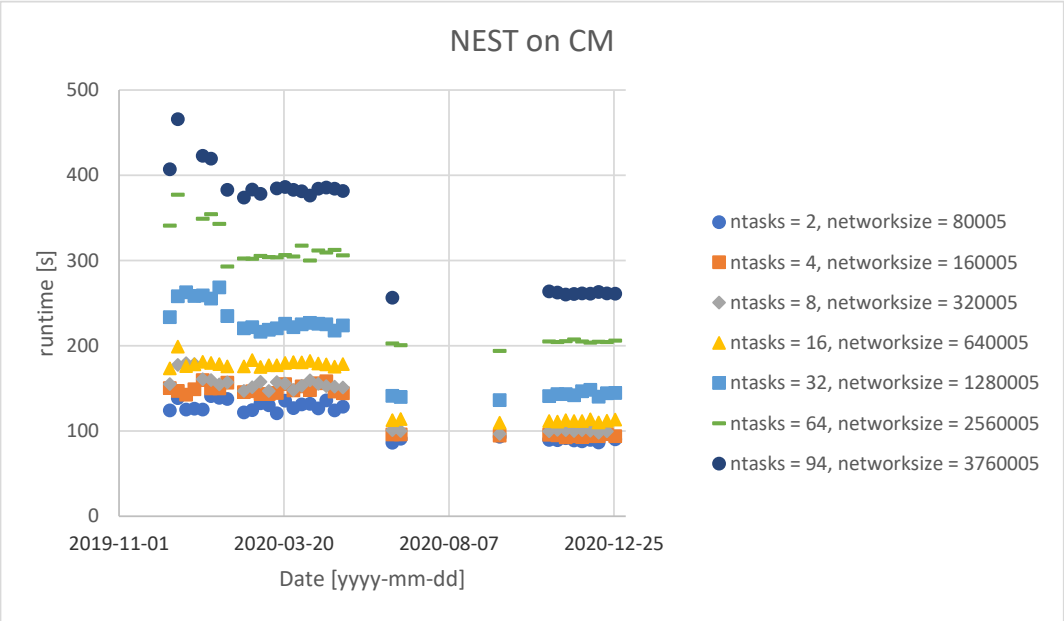


Figure 8.: NEST benchmarking results with commitid = b84c9ba

There are some large gaps for missing measurement points. This was due to an error in the NEST commit and an unexpected error in the benchmark suite workflow such that the missing data points were not communicated to the benchmark suite developers in an automatized way. In a benchmark suite of this complexity there are a lot of benchmarks undertaken on a daily basis. The error handling is for the case of such a diverse combination of software and programming languages not trivial. Easily situations can occur not recognized by the automatized error handling strategy as an error. Due to the amount of data produced it is also a time consuming task to have a manual look at all data points when there are more benchmark needed to be integrated into the benchmark suite in parallel. Therefore, special care needs to be invested in the error handling strategy. This will avoid the occurence of large gaps of measured data like it is shown here in figure 8.

Within all parameter permutations the runtime of the NEST benchmark shows a clear step function. The runtime after the step is reduced. Having a look into the kernel information files on May 9 2020 and on June 20 2020 the software versions used are exactly the same.

Nevertheless, environment variables are indicating a different pinning of tasks on the node sockets.

On May 9 2020 for the case of 1 node with 2 tasks and 24 threads per task the environment variables and system information were given as follows.

- task 0: __PINNING__=000000000000000000000000111111111111111111111111
- psmgmt version: 5.1.29

On June 20 2020 for the case of 1 node with 2 tasks and 24 threads per task the environment variables and system information were given as follows.

- task 0: __PINNING__=000000000000111111111111000000000000111111111111
- psmgmt version: 5.1.30

Before May 9 2020 the threads of task 0 were distributed between two sockets. After June 20 2020 the threads of one task were executed on one socket. This potentially speeds up communications between these threads. So for the way the benchmark system was set up for NEST the threads of one task seem to perform a lot of data exchange between each other. In this case the change of the default pinning lead to a performance increase.

2.3.4. GROMACS

For the GROMACS benchmark the strong scaling behaviour can be verified, see figure 9. Furthermore, there is a dropdown in the performance of the GROMACS performances around end of May 2020 and beginning of June 2020.

Before performance dropdown around end of May 2020 and beginning of June 2020 the following configuration can be extracted from the kernel information logs.

- SBATCH_CPU_BIND_TYPE=ranks
- psmgmt version: 5.1.29

After performance dropdown around end of May 2020 and beginning of June 2020 the following configuration can be extracted from the kernel information logs.

- SBATCH_CPU_BIND_TYPE=threads
- psmgmt version: 5.1.30

Comparing samples for 24 threads per task and 48 threads per task the case for 48 threads per task is not revealing the performance drop, see figure 10. In the first case the number of threads was chosen to fill every physical core with one thread. In the second case multithreading is activated to fill every physical core with two threads. A change of the default pinning scheme makes no difference in the second case. It can be extracted that the communication between the threads of one task is small compared to the compute workload.

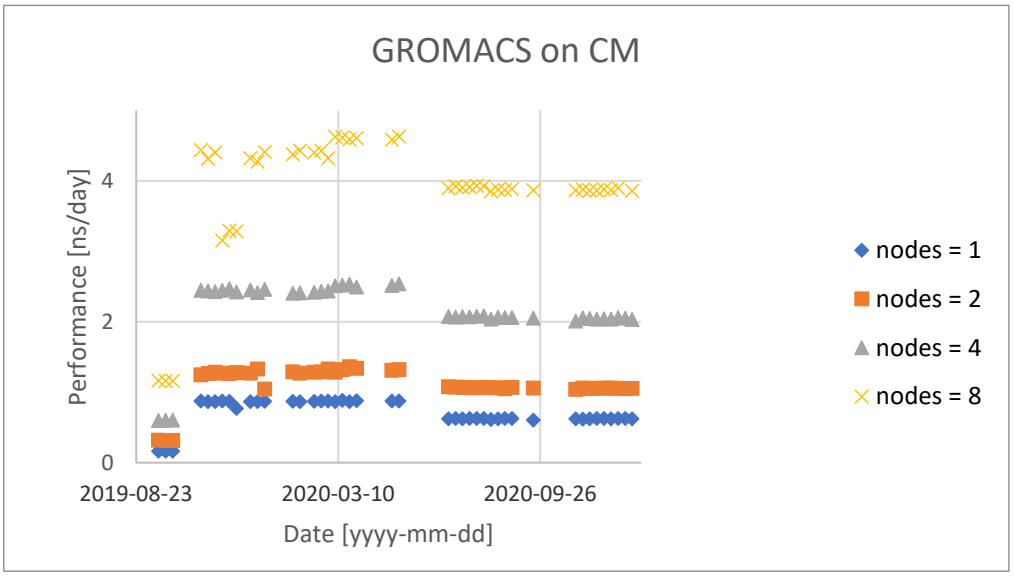


Figure 9.: GROMACS scaling benchmarking results with benchmark_system = ribosome, MPIsperNode = 1 and ThreadsPerMPI = 24

2.4. Conclusion

Benchmarks have typically a lot of parameters to be defined in before. Furthermore, the number of benchmarks is potentially huge. As a conclusion it needs experience and special care to define the parameters of a benchmark for regular benchmark runs.

Besides other aspects, the usage of the system, the software and the hardware environment is constantly changing. With these changes of the system environment also the optimal set of parameters is changing. Therefore, the process of benchmarking does not only need initial in depth considerations and thorough preparations, it also needs constant and iterative maintenance. Visualisation of the results, automatized communication and error handling strategies and automatized result analysis are strong supporters for the benchmark developers.

The first conclusion of this task is a guide of how to write a benchmarking suite. This guide is not able to prepare for all potentially possible benchmark suites. But it gives a valuable first starting point. This is of special need since a thorough preparation of a benchmark suite integration is the key to the success of the very same. The accurate and goal-oriented data collection should be achieved as soon as possible to be able to cover as many interesting effects as possible.

A second conclusion is that you will never know in before what you are going to measure with a

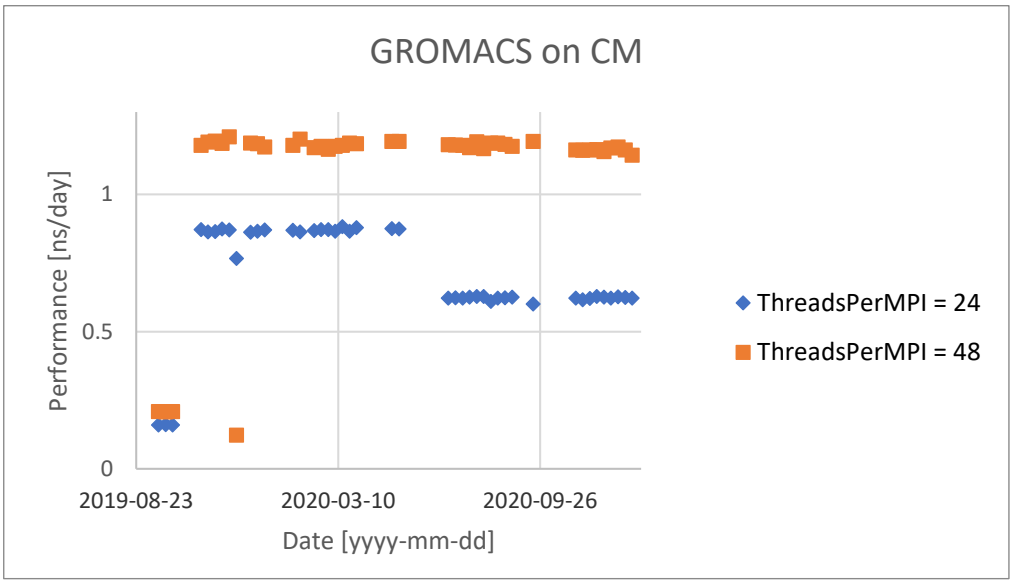


Figure 10.: GROMACS benchmarking results with variation of number of threads per MPI task with benchmark_system = ribosome, nodes = 1 and MPIsperNode = 1

benchmark suite. As an example, the case of the change of the default pinning on the prototype, which was decided within the project duration, was very disruptive to the bencharking results. Therefore, it is very difficult to prepare a suited parameter set in before right at the beginning of a project which is prepared for all the things that can happen with the prototype system.

A close and automatised monitoring of the benchmarking results needs to be implemented. At least, warnings need to be communicated automatically if a jump in the benchmarking results can be extracted. For this point, algorithms need to be developed and used.

Another result of this task is the evaluation of the integration of CUDA-aware MPI and its conclusion to a performance increase when communicating data between two nodes equipped with a GPU.

At last, the unmeasurable but also very important contribution of regular sanity checking of the prototype should be mentioned here. Through the regular benchmark suite execution a sort of unit testing of the prototype was performed. This was a proactive methodology to ensure a basic level of quality of the prototype. Some hidden errors, like diverging directory structures between different nodes, were found multiple times and probably saved a lot of hassle for the early access programmers.

3. Modelling and validation of scheduling policies

The evaluation of scheduling policies or system configurations must be done by executing, simulating (or both) a set of applications confirming a workload. Evaluating a workload doing real executions is unfavorable because for a valid evaluation lots of runs must be done with various configurations. In the project we have done real execution for a functional validation and we have used the BSC-Slurm Simulator for evaluation.

Figure 11 shows the main components of the execution environment when doing simulations. Each experiments receive two inputs: A workload trace file and the system configuration.

Modular trace file describes the set of applications to be executed. As we presented in D2.1[2] and D2.2 [3], a workload trace file includes, per application, details such as submission time, number of nodes requested, execution time, requested time, etc. All the details needed for the simulation describing the information provided at submission time and the real values such as execution time that typically differs from submission values.

The system configuration in this project is included in the slurm.conf file. The Slurm configuration file includes the hardware description (number of modules and module description), and all the other system features such as the scheduling policy and the scheduling policy arguments.

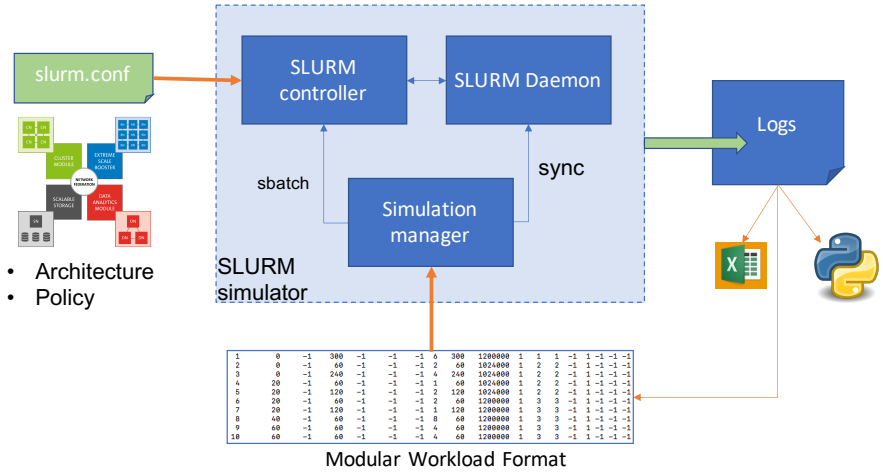


Figure 11.: Execution environment for scheduling evaluation: The BSC-Slurm simulator

The following sections include: Section 3.1 presents the extensions to the Modular trace format presented in D2.1 required by the energy aware policies proposed. Section 3.2 describes the methodology used in the project to create modular trace files. Section 3.3 presents the evaluation of a modular system compared with a conventional homogeneous system. Section 3.4 presents the evaluation of the workflows with dynamic dependencies management.

3.1. The Modular Workload Trace file format

The Modular Workload Format (MWF) is an extended version of the Standard Workflow Format (SWF) to support modular architectures and complex workloads. The format, originally defined in D2.1, was extended to support new use cases of the project. The included fields are:

- **Reference Power** – Input average power in Watts. Input by user or a power model.
- **Average Power** – measured average power in Watts
- **API_call_time** – in seconds. It models events called by the job that impact the job scheduling. It is used to model the API call for the workflow policy, but it can be extended to a list of comma-separated key:value elements, with key representing the event type (ID or keyword), and value the number of seconds passed from the start of the job until the event. E.g.: ”WF_API:650,extend_job:850”.

For completeness we present the list of the MWF fields in its entirety in Appendix 7.

3.2. Modular trace file generation

One of the simulation inputs, the trace files, can be traces shared by supercomputing centers or based on traces generated via workload models. The utilization of traces has the benefit they are real, however, they are specific to the center that provided it and can not be easily modified to adapt them to the size of the system, the load or the number of jobs without significantly affecting the workload characteristics. Because of that reason, we selected the second approach and we have evaluated the project contributions using traces generated with the Cirne model [16]. However, the Cirne model generates trace assuming a single module. In the project we have designed a methodology to merge multiple single module traces into a single modular trace file and to incorporate the characteristics to be evaluated in the project such as the module flexibility of the dynamic management of dependencies.

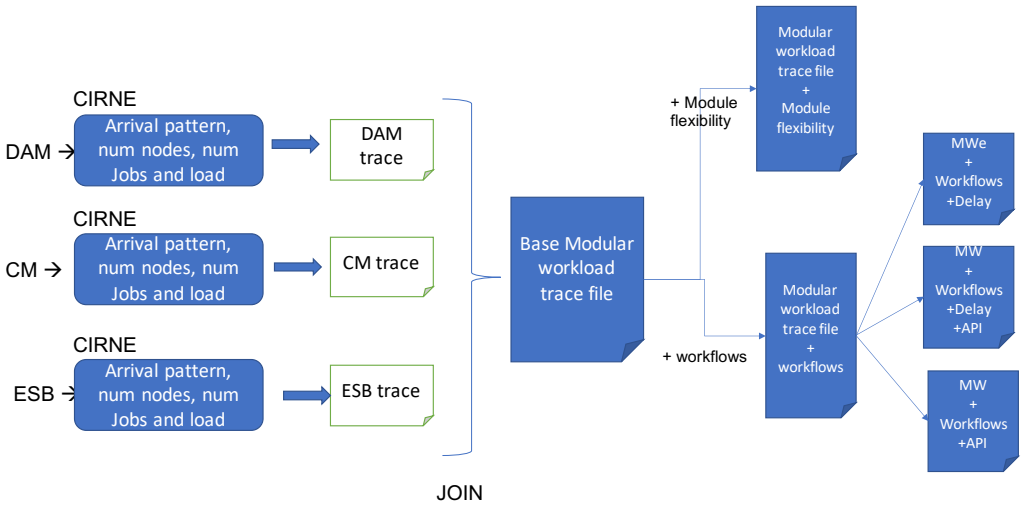


Figure 12.: Modular trace file generation methodology

Figure 12 summarizes the main stages of our methodology to generate modular trace files and to include the characteristics to be evaluated. The first stage generates N single module traces. Cirne model accepts the following inputs:

- Number of nodes: available nodes in the module, it also affects job's arrival pattern.
- Arrival pattern: arrival time model, different models are available, extrapolated from traces released by four supercomputing centers: ANL, CTC, KTH, SCDC.
- Number of jobs.
- Load multiplier: it multiplies the requested time, and influences the arrival time.
- Scale factor: scale job requested nodes, it can be 1 or 2.

Cirne also sets a number of internal parameters:

- n_ul1: parameter for the uniform-log distribution used to calculate the job number of requested nodes.
- n_ul2: constant parameter for the uniform-log distribution used to calculate the job requested number of nodes.
- n_p2_frac: the fraction of power of two jobs over the total.
- tr_ul1: same as n_ul1, but for calculating the job requested time.
- tr_ul2: same as n_ul2, but for calculating the job requested time.

We initially left these parameters at their default values.

The N module traces are merged in a single BaseMWT file **base modular workload trace file**. This trace file has the jobs in the right order to be submitted for evaluation and is the baseline to which we apply the different system characteristics to be later evaluated.

To generate the module flexibility, the scripts use as input the percentage of jobs per module with module flexibility and the list of configurations to be generated. Figure 13 is an example of the combinations generated. For instance, the 100% of the jobs running on the CM were allowed to run in the DAM and ESB given they don't have special requirements. However, only the 50% of the jobs having as preferred module the ESB would present the module flexibility and they will be modified to ask for ESB and CM. These combinations where generated based on the questionnaire we did some months ago to WP1.

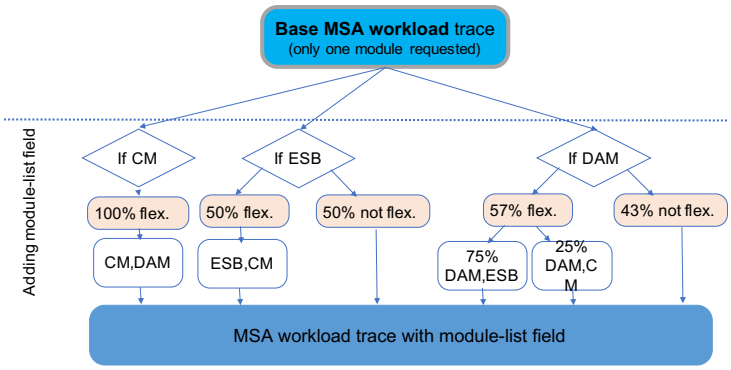


Figure 13.: Modular list example

Figure 14 shows an example of input to generate workloads with workflows. In that case, the percentage of jobs to be part of a workflow and the maximum number of components is also an input for the script. The script modifies the workload adding dependencies to generate traditional jobs or packing jobs in heterogeneous jobs and adapting the submission time and requirements to fit in what Slurm would expect for a job or an heterogeneous job. Moreover, in the scenarios where the delay and the WF-API is used (see 3.4.2), the trace file already includes the time for the delay flag and the time where the WF-API is invoked. These two values are generated using the requested time and runtime and using an uniform distribution between 50% and 90% of the job's requested time for the delay flag and and uniform distribution between 50% and 90% of the job's runtime for the WF-API.

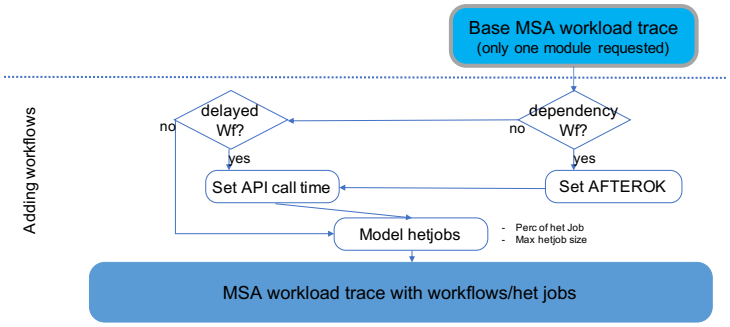


Figure 14.: Workflow generation example

3.3. Evaluating conventional vs Modular systems

This section will compare the execution of a given workload in a conventional homogeneous system and in a modular system. To be fair, we have selected three different configurations for the homogeneous use case, each one equipped with different CPUs and GPUs, based on CM, DAM and ESB architectures. The modular use case is based on CM+ESB modules. To compare modular and conventional systems the presented evaluation does not include the contributions done in this project concerning modular scheduling or dynamic workflows, as they only work on modular systems and they require different workloads that cannot run on the homogeneous case.

3.3.1. Hardware configuration

To dimension the system we fixed a budget and we used it to build the compared systems. We used the cost of the DEEP-EST components as a reference. In particular:

1. The cost of a non-accelerated node (CM) is 1unit.
2. The cost of the accelerated 2-CPUS node (DAM) is 2.5units.
3. The cost of the accelerated 1-CPU node (ESB) is 1 unit.

Table 6 shows the details per module.

Module	Num nodes	Cost	Total module cost	Percentage
CM	50	1	50	30%
DAM	16	2,5	32	25%
ESB	75	1	75	45%

Table 6.: DEEP-EST prototype costs distribution

Following the DEEP-EST prototype system distribution of nodes, for the modeled modular system we have allocated 30% of the budget to the non-accelerated nodes, based on CM, and 70% to the accelerated nodes, based on ESB. We came up with the following clusters:

HOM1-CM Homogeneous system with 165 nodes, equipped with CM nodes.

HOM2-DAM Homogeneous system with 66nodes, equipped with DAM_light nodes (CPU+GPU+Memory).

HOM3-ESB Homogeneous system with 165 nodes, equipped with ESB nodes.

MOD-CM&ESB Modular system with two modules: 50 nodes equipped with CM nodes, and 115 nodes equipped with ESB nodes, for a total of 165 nodes.

3.3.2. Workload

The workload for this evaluation has been generated using the previously proposed methodology with Cirne model with default parameters, ANL arrival pattern, and 3000 jobs asking at maximum 32 nodes, about half of the smallest system size. The BaseMWT file generated is used as the input for the HOM1-CM evaluation.

To evaluate the other scenarios, we have followed the same approach as with the module-list or the energy aware scheduling: a list of *applications types* is provided with their runtimes characteristics, as show in Table 7. The list of application types is based on WP1 applications. The table shows the 5 application types used. CM, DAM and ESB columns shows the execution time ratios (where CM is the reference). As an example, CMS (from CERN) is 2.25 times faster in DAM than in CM and 0.5 time slower in ESB than in CM. On the other side, Correlator (from ASTROM) is 24 times faster in DAM than in CM and 19 times faster in ESB than in CM. Column *Percentage* shows the percentage of each type in the workload.

When running simulations, each job is mapped in a deterministic way[1] with an application type, being applied the runtime settings defined in the *application types* list. It is worth to mention this types are applied to jobs with different runtimes and job size since the job classification is applied to jobs in the trace file. For the modular system jobs have been submitted their preferred partition, i.e. the one with showing runtime.

3.3.3. Evaluation

The four use cases have been evaluated with the BSC Slurm Simulator. Figure 15 shows the average wait time, average response time and completion time (max end time) for the

[1] taking random samples from a uniform distribution

Application	CM	DAM	ESB	Percentage
CMS	1	2.25	0.5	17.5%
Correlator	1	24	19	17.5%
GROMACS	1	3	2	17.5%
xPIC	1	8	8	17.5%
Non-ACC	1	1.5	0.3	30%

Table 7.: Application types used for modular vs homogeneous evaluation. Columns CM, DAM, and ESB show runtime ratios, e.g. CMS runs 2.25 times faster on DAM with respect to CM.

four scenarios. As we can observe, the main benefit comes for the utilization of accelerated nodes given 50% of the applications are significantly accelerated when running on CPU+GPU nodes. Focusing on homogeneous accelerated scenarios, DAM and ESB, we can see ESB architecture is also significantly better than DAM architecture because a significant amount of jobs reports similar performance on ESB and DAM and the other are compensated with the fact ESB is cheaper and then has more nodes. This can clearly observed on the reduction in average wait time, where DAM reports 3.2 times more wait time than ESB.

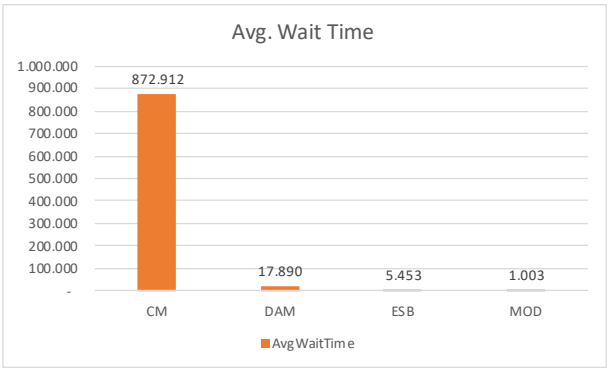
Figure 15d focuses on DAM, ESB and Modular system for the two main metrics used in job scheduling evaluation: wait time and response time. We can observe that the modular benefits in terms of response time and wait time by combining a higher number of accelerated nodes compared to DAM, and more powerful CPUs compared to ESB. The modular system reduces by a factor of 17.8 the wait time compared with DAM and by 5.4 compared with ESB. And concerning the response time the modular approach reduces by a factor of 2.4 compared with the DAM and 2.7 compared with the ESB. At the same time, the completion time, even if not relevant as response time and wait time, is comparable to DAM case, which reports the best value.

3.4. Workflows with dynamic dependencies

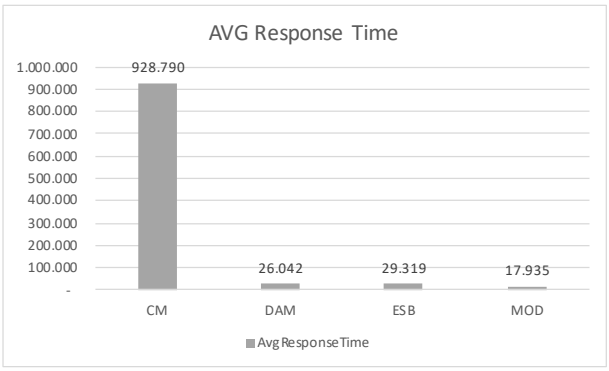
It was defined a use case in which two dependent jobs need to overlap to directly exchange information between them.

To address this use case, a new clause for heterogeneous jobs was defined in D5.3 [6], the *delay* clause. The delay clause is defined as the number of seconds between the start of the first heterogeneous job component and the start of the following component. The job scheduler was edited to accept this clause and create reservations for the heterogeneous job components, and by transforming them in normal jobs.

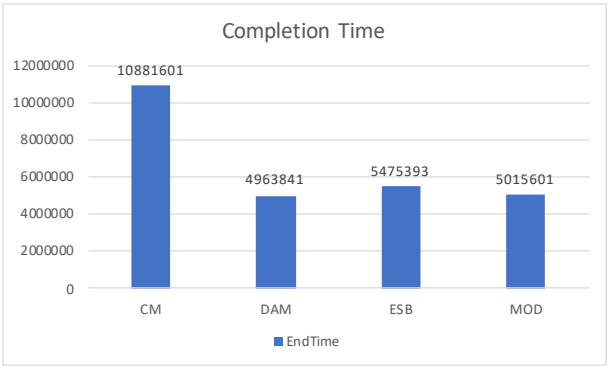
A second solution was developed using classic dependencies between jobs [7]. In this case an API was designed to change the type of dependency at runtime, whenever the time in which the two jobs need to exchange data comes. The dependent jobs are submitted with a AFTEROK dependency, which means the job will run after the one on which it depends terminates. The API transforms this dependency to an AFTER dependency, in which the job will run after the



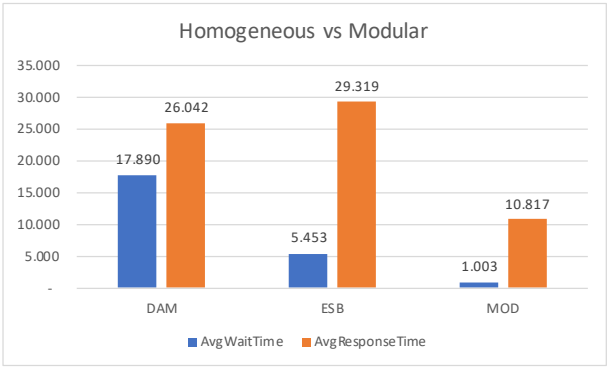
(a) Average Wait time



(b) Average Response time



(c) Completion time



(d) Homogeneous vs Modular summary

Figure 15.: Modular system performance vs Homogeneous clusters

one on which it depends starts.

An API was also developed for the delay clause, in this case the API tries to move the job's reservation backward in the case the delay specified was too long.

To resume:

1. delay clause: it supports overlapping heterogeneous jobs. It grants the overlapping if the user requested time for the job corresponds to the actual job's runtime.
2. dependency+WF-API: it uses a simple technique to overlap dependent jobs. It does not grant that the jobs will overlap.
3. delay clause+WF-API: like point 1, but it tries to fix the problem of actual job's runtime less than requested time. It also helps when the delay value is not known with precision.

3.4.1. Performance metrics for workflow evaluation

In this section we are evaluating the impact of having a dynamic management of dependencies between jobs (both traditional and heterogeneous). Given these jobs or components are part of the same *work*, we have extended the traditional metrics applied to individual jobs to take into account N jobs can be part of a workflow or an heterogeneous job and must be considered as a whole. Simple jobs are considered workflows with one component. Metrics used in scheduling evaluation are the average WaitTime, the average ResponseTime (EndTime - SubmissionTime) and the average Slowdown (the ResponseTime normalized by the RunTime).

Given a workflow WF made up of *n* components *c*, we define the following metrics for WF:

$$WF\_StartTime = \min_{\forall c \in WF} StartTime_c \tag{3.1}$$

$$WF\_SubmissionTime = \min_{\forall c \in WF} SubmissionTime_c \tag{3.2}$$

$$WF\_EndTime = \max_{\forall c \in WF} EndTime_c \tag{3.3}$$

$$WF\_Runtime = WF\_EndTime - WF\_StartTime \tag{3.4}$$

$$WF\_ResponseTime = WF\_EndTime - WF\_SubmissionTime \tag{3.5}$$

$$WF\_Components\_Runtime = \sum_{c=1}^{n} Runtime_c \tag{3.6}$$

$$WF\_Internal\_WaitTime = \sum_{c=2}^{n} (StartTime_c - EndTime_{c-1}) \tag{3.7}$$

$$WF\_Slowdown = \frac{WF\_ResponseTime}{WF\_Components\_Runtime} \tag{3.8}$$

$$WF\_Normalized\_Runtime = \frac{WF\_Runtime}{WF\_Components\_Runtime} \tag{3.9}$$

Regarding $WF\_Internal\_WaitTime$, $(StartTime_c - EndTime_{c-1})$ will be added to the summation only if its value is greater than 0.

3.4.2. Experiments

We have compared the following scenarios:

DEP Jobs use the traditional way to specify dependencies (AFTEROK) when the job with the dependency starts one the previous finishes. In that case, there is no reservation of resources so it can happen the dependent job has to wait until there are enough resources to start.

DEP+API Jobs use the traditional way to specify dependencies (AFTEROK) but once the dependency is ready the job will notify the scheduler about that using the new WF-API and the dependency type will be changed to AFTER, becoming ready to be executed. We would expect from this scenario a reduction in the WaitTime of dependent jobs given the dependency is broken early on time.

HET In this case, jobs with dependencies are packed in heterogeneous jobs. This is a significant difference in the way Slurm internally manage heterogeneous jobs compared with traditional jobs. In that case, resources for all the components are allocated since the start time of the first component. This implementation guarantees resources are ready to be used by any component but in case of components with dependencies this strategy result in a significant waste of resources.

HET+DELAY In this scenario the users specifies a delay time to mark the time at which resources must be ready for the dependent job. This implementation minimizes the waste of resources compared with the standard approach but is still affected by miss-predictions in the runtime of jobs.

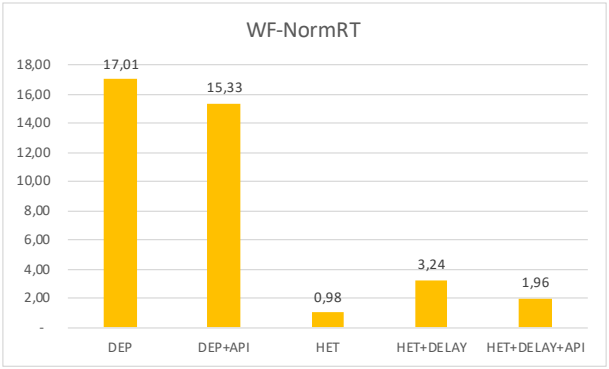
HET+DELAY+API This last scenario is the most complex and powerful. In this case, the user specifies a delay time with the new flag and it notifies at runtime the scheduler about the readiness of the work. It allows the system to start with the resource reservation and to dynamically adapt to changes in the runtime behaviour.

Each scenario has been evaluated with workloads containing 24% or 33% of jobs grouped in workflows. It is worth to mention the goal of this contribution is to improve the execution of jobs in workflows without penalizing the other jobs, so the benefit in global metrics is limited to the total number of jobs in workflows and the weight of these jobs in the total workload CPU time requested.

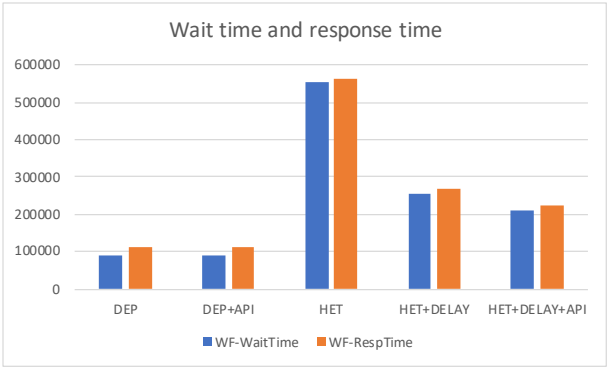
In all the cases, workloads include 3000 jobs, 1000 jobs submitted to each module. We used the default values of load per module, ANL arrival pattern, and we set the reductionfactor to two only for DAM, given its reduced size. A minimum of 10% of jobs request power of two nodes. The maximum job size for the CM is 50, for the ESB is 75 nodes, and 8 for DAM.

3.4.3. Dynamic workflows management evaluation: Workloads with 24% of workflows

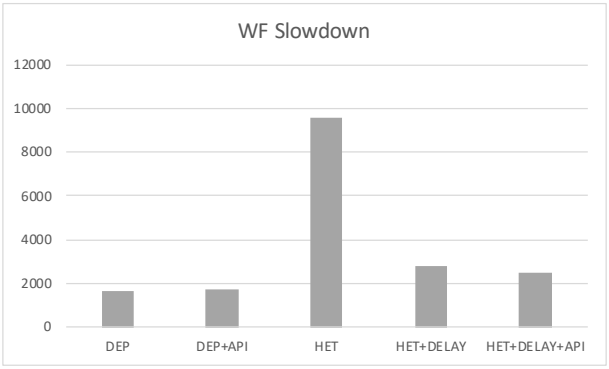
This section presents evaluation results for an use case where 24% of the jobs belong to a workflow. Given our goal is support the overlapping of workflows as much as possible without penalizing jobs not in a workflow, we don't expect to improve generic metrics such as average wait time or response time for DEP scenarios where jobs release their allocations once finishes. In the case of HET scenarios the WF-API can help to improve generic metrics since job allocation includes all the components and the WF-API helps to reduce the pressure in the system by releasing faster the job allocation.



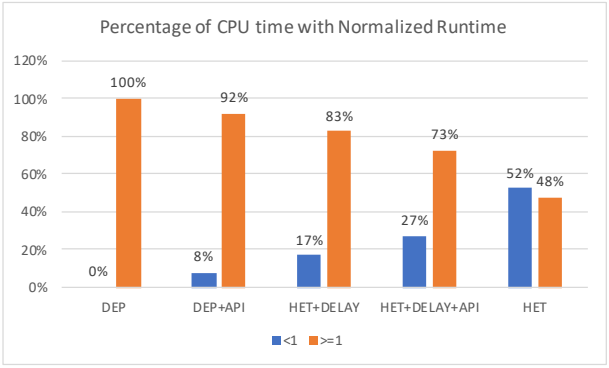
(a) Average WF Normalized Runtime



(b) Average WF wait time and WF response time



(c) Average WF Slowdown



(d) Percentage of CPU time with WF normalized runtime.

Figure 16.: Workflow performance metrics when having a 24% of the jobs being part of workflows

Figure 16a shows the workflow normalized runtime for all the jobs. For jobs not in a workflow the normalized runtime is always 1. For jobs in a workflow, this metric quantifies the overlapping between components, for this reason, in the DEP scenario it's always greater or equal to 1 and in the HET scenario is always lower than 1 because heterogeneous jobs by default are executed only when all the resources are guaranteed. We can observe how the WF-API helps to significantly reduce this value in DEP scenarios and on the other side is not too much incremented in HET scenario. The fact standard heterogeneous jobs are only executed when resources are available negatively affects other metrics such as the average wait time or aver-

age response time as it is shown in Figure 16b. Figure 16c shows the average workflow time and again we can see how the WF-API improves the HET scenario even in generic metrics because the flexibility in the system compared with the standard use case. Finally, Figure 16d shows the percentage of the CPU time executed by workflows that is overlapped (normalized runtime less than 1) and not overlapped (greater than 1). Of course, in terms of normalized time heterogeneous jobs are the best one but at the cost of very bad global results. With the WF-API we don't get the same percentage of overlapping since resource are allocated dynamically but the overall metrics are much better than the basic case.

3.4.4. Dynamic workflows management evaluation: Workloads with 33% of workflows

This section presents evaluation results for a use case where 33% of the jobs belongs to a workflow. Having more workflows introduces even more pressure to the system. The HET case is even worse when comparing traditional metrics such as wait time, response time or slowdown. We can see how the percentages for specific workflow metrics such as the normalized runtime varies but are still able to manage the dynamics of the system. We can improve the normalized runtime in the DEP scenario when using the WF-API (figure 17a) and 8% of the CPU time of workflows are overlapped when using the WF-API (figure 17d).

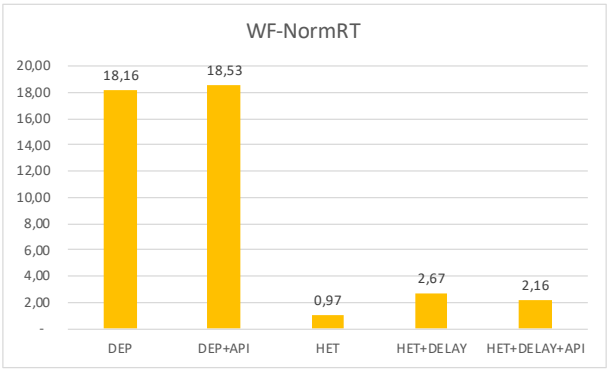
As in the previous workload, the WF-API significantly reduces the penalty on wait time, response time and slowdown suffered by heterogeneous jobs (figures 17b and 17c) by offering a reasonable percentage of job overlapping 27% of the CPU time when using the delay flag and the WF-API together. Figures 17a and 17d also show the benefit of using the WF-API together with the delay flag in order to react dynamically to runtime conditions.

3.5. Conclusions

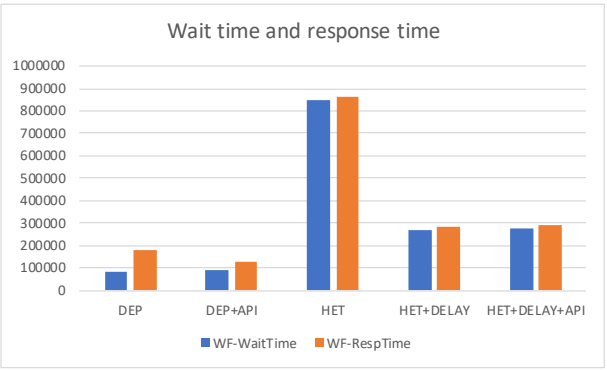
This section has presented, using simulations, two performance evaluations: the comparison of modular compared with homogeneous systems and the comparison of the utilization of the WF-API and delay flag to offer a dynamic management of the workflows.

Simulations uses the same Slurm code installed in the prototype and the only part of these results that is simulated is the execution time of jobs. Because of that, we can guarantee these results could be extrapolated to a real workload with the only requirement execution times used in trace files corresponds with real ones. Given the information to replicate a real experiment was not available in both evaluations, we created a set of trace files and a complete methodology to consider different scenarios with different types of applications. In the first analysis the list of applications and their characteristics was modelling a subset of WP1 applications. We have demonstrated numerically the strengths of the modular system as concept and of the ESB proposal as a key component in this modular system. The fact that it is a cheaper component makes it possible to have a high number of this node type. Together with a given percentage of powerful nodes to support applications with traditional CPU requirements makes this solution a very good solution.

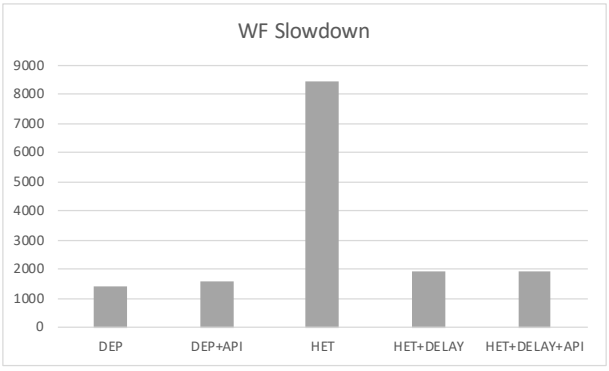
In the second analysis the ratio between components was not relevant for the experiment since



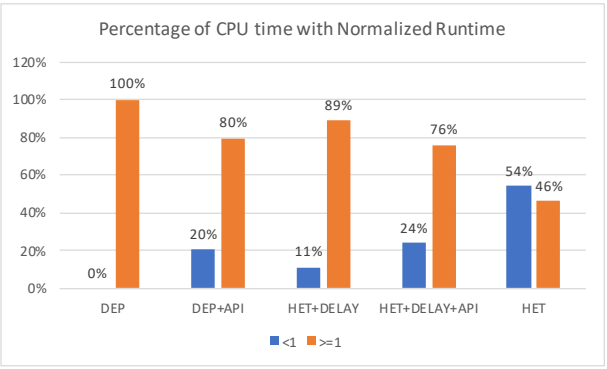
(a) WF Normalized Runtime



(b) WF wait time and response time



(c) WF Slowdown



(d) Precentage of CPU time with normalize WF runtime

Figure 17.: Workflow performance metrics when having a 33% of the jobs being part of workflows

jobs were requesting a predefined module. In that case, we model the dependencies between components in two scenarios: dependencies between jobs and between components. Results have demonstrated it is possible to reduce the strain heterogeneous jobs introduces in the system by defining a given delay and even better when using the WF-API to notify the scheduler about runtimes modifications. To introduce this WF-API we have extended the MWF previously proposed with a new field. The Slurm Simulator has also been extended to support this dynamic event and of course the WF-API feature.

The main motivation to provide job overlapping was to enable data communication between jobs using the network rather than the filesystem, as in the case of traditional jobs (DEP scenarios). The modeling of the extra runtime (overhead) required to perform the IO when the WF-API is not used (DEP) and the extra runtime for communication when using it (DEP+API) have not been included in the simulation because of the lack of real use cases. It was demonstrated with some synthetic experiments that using communications is more efficient than using IO. Introducing this extra parameter in the simulations without following a real pattern or model would just potentially introduce noise on the conclusions. Finally, the absence of the simulation of the IO bottleneck in the evaluation represents a worst-case scenario, and considering it would improve the analyzed metrics.

4. Performance modelling and extrapolation

The goal of the performance modelling and extrapolation task is twofold. First, to analyse the performance and study the scalability of the applications. Second, to model the DEEP-EST architecture to evaluate efficiency, predict executions at larger scale, and to study the impact of different mappings. For this task we employ the performance analysis tools developed at BSC [55]. Extrae and Paraver are used to obtain and visualise traces of the project's applications, and Dimemas to simulate different system configurations. Continuing the work presented in D2.2 [3] we report new case studies for GROMACS, xPIC, NEST+Arbor, NextDBSCAN and EC-EARTH [48].

4.1. BSC modelling tools

The performance modelling approach used in DEEP-EST is based on BSC's efficiency model, which characterizes the performance of the applications using fundamental factors that can be applied to understand and predict application's scalability. The factors are measured as values between 0 and 1, the higher the better. An efficiency of 0.8 indicates that the use of the resources was 80%, so 20% of the corresponding resources allocated to the execution are unused. In the general case, this value can be considered a boundary between good and bad performance.

This is a multiplicative model, where the global efficiency of a parallel application can be decomposed into two main factors:

Parallel efficiency represents the percentage of time spent on the computation (useful work) with respect to the total execution time.

Computation scalability reports the scaling of the computation itself. For instance, an application that suffers code replication when scaling will report degradation in the computation scalability.

Similarly, the parallel efficiency can be decomposed into three main factors:

Load balance (LB) measures the efficiency loss due to differences on the computing time between processes. If some processes take more time in computation, the other processes have to wait for them in subsequent synchronizations, for instance, in MPI collective operations.

Transfer (Trf) measures the efficiency loss due to the transference of data between processes. If the application is dominated by communications, *Transfer* efficiency would be low.

Serialization (μLB) measures the efficiency loss due to dependencies during the execution. This factor also reflects load imbalances that can be compensated along time. If on the even iterations half of the processes do more work than the other half, but on the odd iterations the behaviour is just the opposite, *Load balance* would report good efficiency while *Serialization* would degrade.

Load balance can be directly measured from an instrumented execution. In order to compute the *Transfer* and *Serialization* factors, it is necessary to isolate the application execution from the network characteristics, which can be done using Dimemas to predict the behaviour running on an instantaneous network.

Hybrid applications that use multiple runtimes, for instance MPI+OpenMP, can be likewise characterised with BSC's efficiency model by decomposing the metrics one level further. Hybrid metrics are expressed as the product of the efficiencies of their composing runtimes. Similarly, modular applications comprised of multiple binaries can be characterised by applying the efficiency model globally to the whole application, and individually to each of their modules.

The BSC efficiency model characterizes the execution of a given application on a given platform (the platform component includes not only the hardware, but also the software stack, the process mapping, etc.). Additionally, if the application is very sensitive to the input data, different models may be generated from various inputs. As the model is based on a trace of timestamped calls to the parallel runtime and on the ratio between the computing time and the communication/synchronization time per process, relevant modifications in the code structure or in the MPI calls used, have a high impact on the resulting model and may render a model generated before these modifications useless.

Once computed, the efficiency model can be used to extrapolate the behaviour at larger scales. The input for the extrapolation is a set of traces for at least four or five executions increasing in scale. The model set-up is adjusted to analyse the collected traces by looking for behavioural trends, while increasing the scale. By default, the extrapolations are based on the following Amdahl's fit (Equation 4.1) which reflects the contention/serialisation on a given resource, where $metric$ is the efficiency that we are modelling, f is an adjustment function (*linear*, *cubic* or *log*) that alters the interpretation of the number of processes for each one of the fundamental factors depending on how they interact, and P the number of processes for a given run.

$$Amdahl_{fit} = \frac{metric_0}{f_{metric} + (1 - f_{metric}) * P} \tag{4.1}$$

If the collected traces allow identifying an underlying physical phenomenon that follows a more specific law for a given efficiency factor, the parameter P in the previous formula can be substituted by the corresponding/approximate function based on the number of processes.

If the goal is to predict the behaviour at several orders of magnitude bigger than the instrumented executions, it is important to validate the model with non-instrumented runs with increasing scale, as far as possible from the baseline. Validating the scale at least half way from the largest run used for modelling and the target prediction seems a safe and reasonable approach.

Even though the model is developed for a given platform, Dimemas can be used to estimate the execution on a different platform. Before extrapolating the simulated results, it may be recommendable to check whether we have to readjust the model looking at the traces generated from the simulations. With this approach, we can predict the execution of a given application on architectures that are not available, or we can study the sensitivity of a given code to the key factors that characterize the Dimemas network: bandwidth, communication latency, the number of messages coming in and out for a given node (input/output links), and the number of messages that can use concurrently the network (number of buses). The main target for

Dimemas is modelling the network, but can also be used to simulate the porting to a processor twice as fast, or the impact of improving the computations from specific parts of the code. It is also possible to adapt the mapping of the processes to the available resources; but there is neither modelling of the memory hierarchy, nor multi-core sharing effects.

Using Dimemas simulations to feed the BSC efficiency model, we can build new models of an application for desired architectures and configurations. The results of the models generated are providing us with very interesting insights about constraints, requirements, and potential of a given application.

4.2. Analysis methodology

In the following sections we present performance analyses for several of the project's applications. In general, we have conducted an efficiency analysis, scalability extrapolation, in-depth trace analysis, and provided suggestions to improve towards the exascale. However, each application is different, so the analysis for every case is adjusted to their own idiosyncrasies.

As at the beginning of the project the DEEP-EST prototype was not yet available, the first analyses for GROMACS and xPIC were conducted on JURECA with abstract simulations assuming improved network and CPU characteristics, without relying on a specific parameterization for the DEEP-EST architecture. This was useful to evaluate whether potential improvements would actually contribute to increased performance.

Once the DEEP-EST prototype became available, the analysis tools were ported to this system, and subsequent analyses for NEST+Arbor, NextDBSCAN and EC-EARTH have been conducted for the target architecture. For these studies, we gathered specific system's parameters that have been taken into account to perform more finely-tuned simulations. Table 8 collects relevant parameters that have been used to configure the Dimemas simulator, regarding the dedicated network connections, and inter-/intra-node network parameters such as bandwidth, latency and contention for the CM, DAM and ESB clusters.

These parameters have been collected from a combination of the theoretical system's specifications, measurements provided by other workpackages, and effective measurements taken from the sysbench benchmark tool [47] measuring single-core CPU performance, as well as traces obtained on the target machines for the LULESH benchmark [46]. The validation of the simulation parameters was performed by comparing real execution traces against the resulting simulations. More precisely, we obtained real execution traces of LULESH in the CM, DAM and ESB, and also on JURECA. Using as input the CM trace, we simulated all three DEEP-EST modules (nominal CM, DAM and ESB); and we did the same using as input the JURECA trace. Comparing the cross-module and cross-machine simulations against the real runs, we observed small deviations that mostly originate in increased variability in the duration of the computations on the DAM and ESB clusters, larger on the latter, as shown in Figure 18, that the simulator does not predict.

Nevertheless, the simulation error reflected in the total execution time is $\leq 2\%$ for the CM and DAM, and $\leq 6\%$ for the ESB. All simulations keep real run trends, with an error of less than 2 percentage points for all the efficiency metrics in the computed models between the real and simulated traces.

	Ethernet	CM ↔ ESB	ESB ↔ DAM
Bandwidth	40 Gb/s	100 Gb/s	100 Gb/s
Max messages on-the-fly	∞	∞	∞

(a) WAN settings

	CM	DAM	ESB
Bandwidth	100 Gb/s	100 Gb/s (EXTOLL) 40Gb/s (Ethernet)	100 Gb/s
Max messages on-the-fly	∞	∞	∞

(b) Inter-node settings

	CM	DAM	ESB
Nodes	50	16	75
Cores/node	2x12	2x24	1x8
JURECA CPU ratio	1.83	1.70 CPU 7.30 GPU	1.33 CPU 7.30 GPU
CM CPU ratio	1.00	0.93 CPU 4.00 GPU	0.73 CPU 4.00 GPU
Bandwidth	83.2 Gb/s	83.2 Gb/s	83.2 Gb/s
Number of buses	3	3	2
Local latency	8.79 us	3.60 us	3.60 us
Remote latency (from CM)	9.01 us	3.52 us	3.55 us
Max messages on-the-fly	∞	∞	∞
Network devices	1	2	1

(c) Intra-node settings

Table 8.: Dimemas simulation parameters for the DEEP-EST prototype

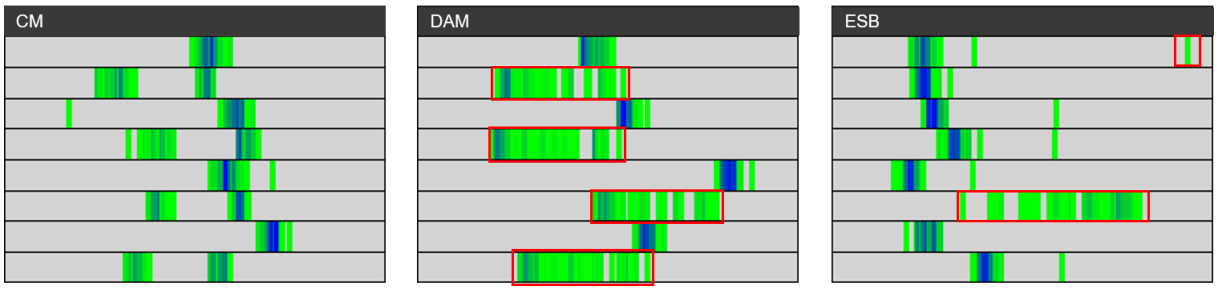


Figure 18.: Histogram of computation duration of LULESH benchmark on CM, DAM and ESB, showing increasingly larger computations from left to right (x-axis).
Light green to dark blue gradient indicates the most frequent behavior.
More spread data (red boxes) indicates higher variability

4.3. GROMACS

GROMACS is a molecular dynamics package designed for simulations of proteins, lipids, and nucleic acids that have complicated bonded interactions. The code is structured into two types of processes —Particle-Mesh Ewald (PME) and Particle-Particle (PP)— that are tighly coupled and synchronise frequently during the execution, with very different granularity and behaviour.

Two different use cases have been studied for GROMACS. In D2.2 [3], an extensive analysis for a problem of 325 thousand atoms in strong scaling mode from 1 to 16 nodes was already presented. In this first analysis, we decided to maintain a fixed ratio between PME and PP processes when scaling in order to minimise variability between executions, but this generated more imbalance in certain scales than the configurations typically tuned by the users.

The second study removes the restriction that we imposed on the PME:PP ratio, enabling the user to finely tune the ratio for each scale, which is the scenario that we consider most relevant for the project, and this is the setup that the users would typically use. Also, a larger input case was selected of 20 million atoms that allowed to execute with a higher number of cores. We obtained traces up to 1536 cores (64 nodes), and execution timings for two additional scales up to 6144 cores (256 nodes), on the JURECA cluster. For this case we extended the previous analysis with brief comparison between both configurations. The efficiency model for the 20M atoms case with variable PME:PP, ratio is presented in Table 9.

	Processes 192	384	768	1536
Global efficiency	92.19	92.55	91.86	81.55
-- Parallel efficiency	92.19	90.74	89.62	80.27
-- Load Balance	93.89	93.12	93.91	92.44
-- Communication efficiency	98.20	97.44	95.43	86.83
-- Serialization efficiency	99.38	98.63	98.55	91.52
-- Transfer efficiency	98.81	98.79	96.83	94.87
-- Computation scalability	100.00	102.04	102.50	101.60
-- IPC scalability	100.00	102.04	103.30	103.27
-- Instructions scalability	100.00	100.24	99.48	98.71
-- Frequency scalability	100.00	99.72	99.74	99.67

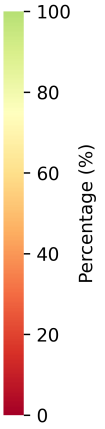


Table 9.: Efficiency model for GROMACS (20 million atoms, variable PP-PME ratio)

While the *Parallel efficiency* only sees a steady decline in the first scales, it suffers a sudden drop of 10 percentage points at 1536 processes, contrary to the previous case, where *Parallel efficiency* sees a steady decrease from $\approx 90\%$ when running on 1 node, to $\approx 60\%$ on 16. The reasons for this degradation in both studies differ. In the 325k atoms with fixed PME:PP ratio, it was mainly caused by *Load balance* issues that aggravate in larger scales (6% *ParEff*, 15% *LB* on 256 nodes). On the other hand, *Load balance* on the 20M atoms case with variable PME:PP ratio, now remains practically steady at $92 - 93\%$, and the problem originates from the *Communication efficiency* that degrades mostly due to the *Serialization* factor, that also sees an abrupt drop of 7 percentage points at the largest run.

Extrapolation analysis

The main difference between both GROMACS studies resides in the setting for the PME:PP ratio. When it was artificially fixed, the model clearly pointed at a problem with *Load balance*, as shown in Figure 19 that depicts the efficiency (y-axis) for each fundamental factor as the scale increases (x-axis, expressed as the scaling factor with respect to the initial run). On the contrary, with process ratios variably adjusted to select the most balanced setting for each scale, *Serialization efficiency* degrades faster and becomes the most limiting factor to the scalability of the application. This is shown in Figure 20a, that depicts the result for an extrapolation with 4 points (from 192 up to 1536 cores, an 8$\times$ increase), validated with 6 points (up to 32$\times$, 6144 cores) with a prediction deviation of ± 1 second of the application's execution time (Figure 20b).

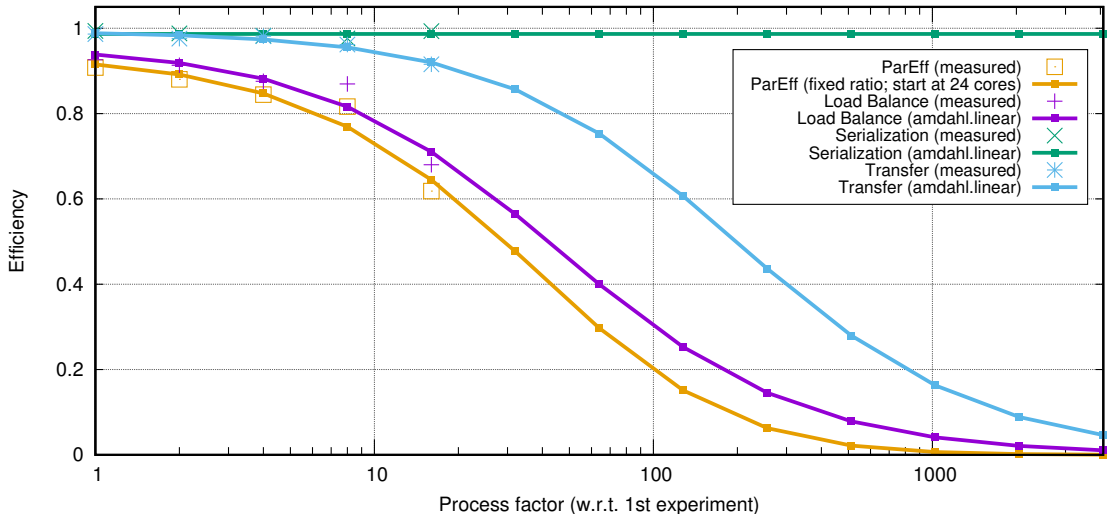
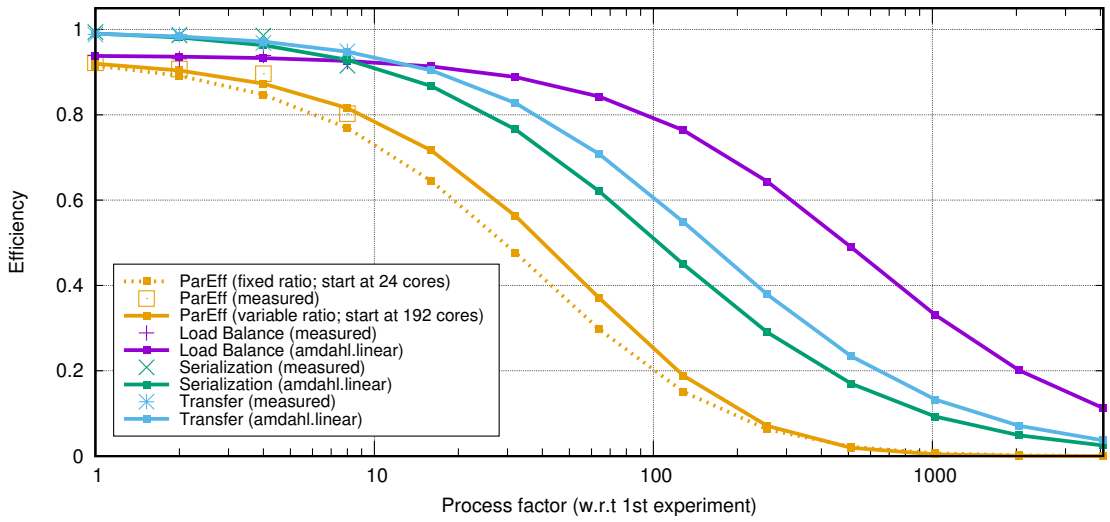


Figure 19.: Extrapolation model for GROMACS (325 thousand atoms, fixed PP-PME ratio)

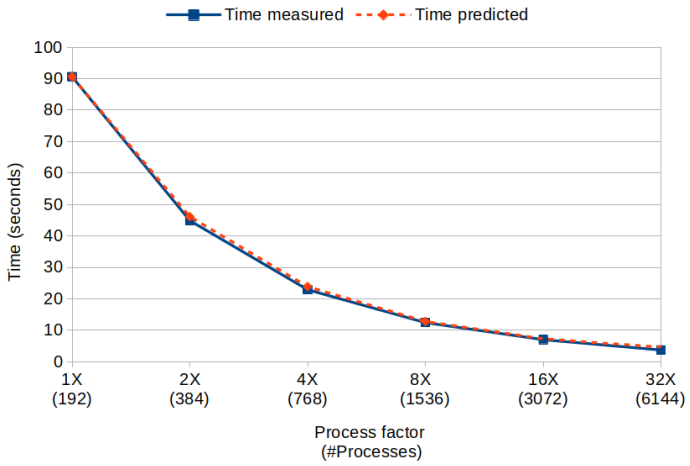
After *Serialization*, *Transfer* is the second component that degrades the most. Since the application is running in strong-scaling mode, this behaviour is expected as the amount of work per process keeps decreasing with the scale, and at some point the execution becomes dominated by communications. Still, the projected *Transfer* trend shows that the application could scale up to 64$\times$ from the base case ($\approx 12k$ processes), with reasonably good efficiency ($\approx 70\%$) for the current input, at which point the problem size probably becomes too small for so many processors, and therefore it would be desirable to increase the input to maintain a favorable computation-to-communication ratio. Contrary to the previous case, *Load balance* now shows a much slower degradation rate, indicating that this factor is hardly an issue until the scale of 128$\times$ from the base case ($\approx 25k$ processes), assuming the variable PME:PP ratios will maintain good balance at larger scales.

Allowing to finely tune the ratio between PME and PP processes as the user suggested, certainly improves the balancing between them. This results in better *ParEff* values (solid yellow) than those on the fixed setup (dotted yellow) and are plotted together for easy comparison in Figure 20a with respect to their scaling factor, independently to the resources they used. The application exhibits the same order of good scalability in both cases, and maintains good effi-

ciency values up to 16×, and despite the scaling trend is similar, the *ParEff* (solid yellow) for the experiment with variable ratios achieves improvements up to 9 percentage points in the measured scales.



(a) Extrapolation with 4 points (runs from 192 to 1536 cores)



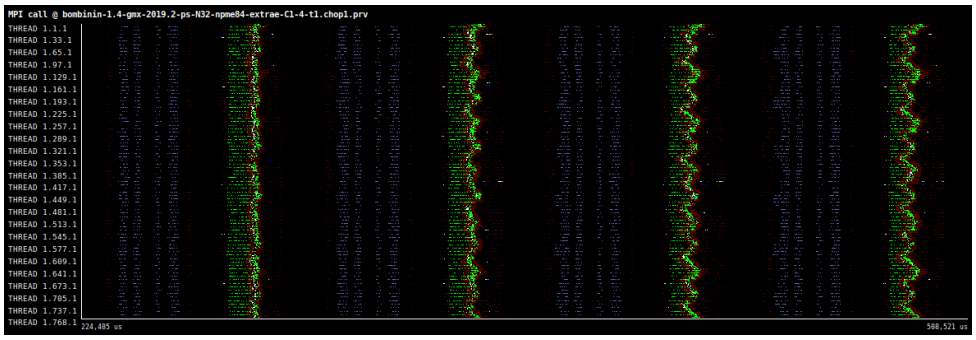
(b) Validation with 6 points (runs up to 6144 processes, fitting error ±1 second)

Figure 20.: Extrapolation model for GROMACS (20 million atoms, variable PP-PME ratio)

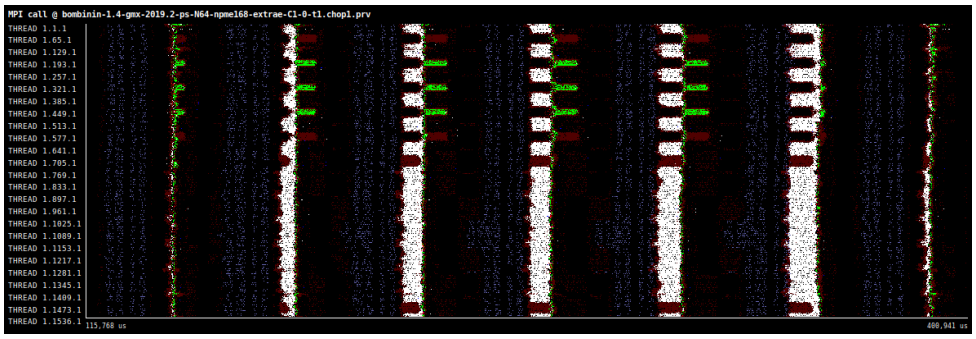
Potential improvements towards exascale

To better understand the serialization issue pointed by the efficiency analysis, we compare the two largest traced executions for the variable ratio case running on 32 nodes (84 PMEs, 684 PPs), and on 64 nodes (168 PMEs, 1368 PPs), that present the same ratio between the two types of processes.

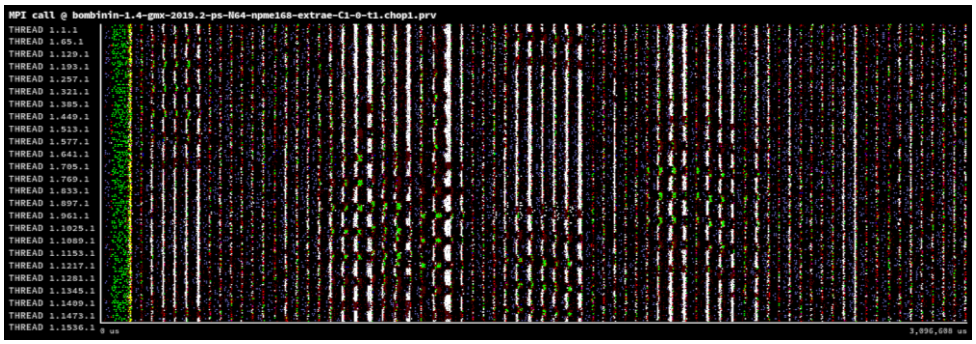
Figure 21 shows the same time lapse starting with the first subiteration of the main loop, where we can see that on 32 nodes (21a) the application is able to complete 4 iterations, while on 64 nodes (21b) only achieves 7 instead of 8 if the scaling had been perfect. The reason for this is the increased time in `MPI_Recv`, `MPI_Waitall` and `MPI_Sendrecv` (white, green and brown regions) that translates in subsequent computations starting delayed, which propagates the effect until it is absorbed, but the problem does not stop here, as it keeps appearing throughout the execution and at different processes, as shown in Figure 21c. Even in the iterations where there is no significant perturbation, PPs usually arrive before the PMEs, and the former stall waiting for the latter in the `MPI_Recv`.



(a) First subiterations in 32-nodes run (84 PMEs and 684 PPs)

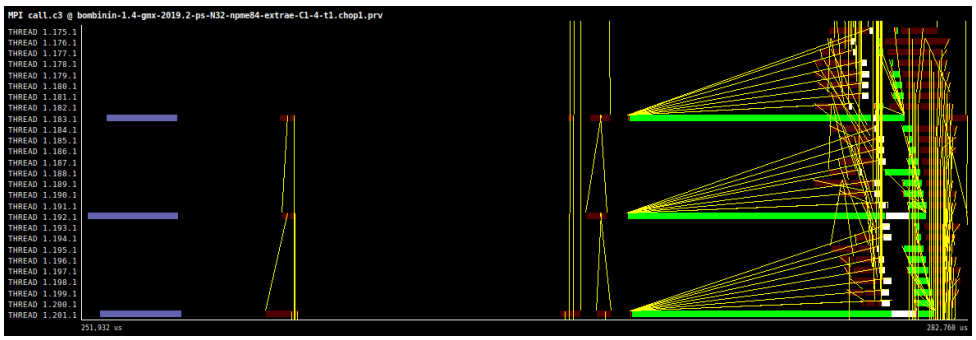


(b) First subiterations in 64-nodes run (168 PMEs and 1368 PPs)

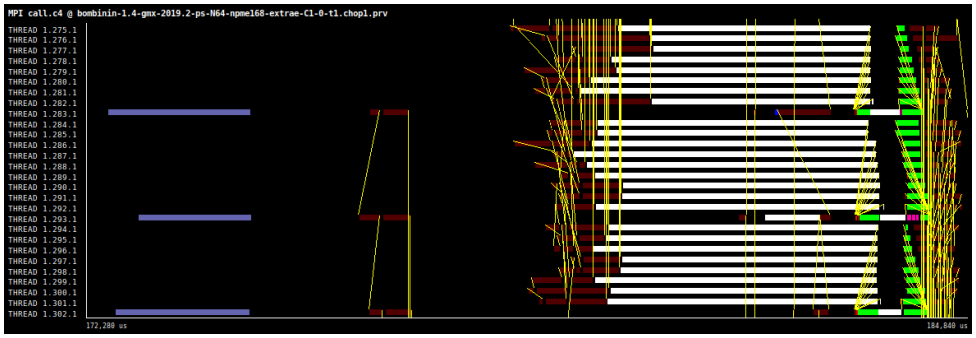


(c) Full iteration in 64-nodes run showing multiple perturbed communication regions

Figure 21.: GROMACS timelines comparing 32- and 64-nodes runs, showing perturbations in MPI calls



(a) Early PMEs stall in `MPI_Waitall` in the 32-nodes execution



(b) Delayed PME results in PPs stalled in `MPI_Recv` in the 64-nodes execution

Figure 22.: Zoomed view of GROMACS showing the distinct wait patterns in the communication phase

Figure 22 shows a zoom with a subset of processes where we can clearly appreciate the difference. While on 32 nodes, the PMEs (processes calling the purple `MPI_Allreduce`) finish their computations sooner and are able to start sending data to the PPs, the latter are still computing and when they are ready to receive, data is already there and can complete their computations very fast. This results in the PMEs stalling in `MPI_Waitall` (green) until the PPs are ready to communicate again.

When the application scales to 64 nodes, the PMEs are unable to reach the communication phase before the PPs as they did on the previous case. The PPs that finish their computations sooner are now stalled in `MPI_Recv` (white), waiting for PMEs to send data. As there are more PP than PME processes, in the 64-nodes case we now see many more processes blocked in MPI communications, about 8 times more with the ratios selected for these cases.

Nevertheless, the most perturbed iterations are caused by delays on the PP's, correlated with unexpected drops in IPC that extend the duration of the computations. Due to the tightly coupled communication pattern, the delay of a single PP is paid both in multiple PPs and also some of the PMEs. This is shown in Figure 23, where process 331 is systematically delayed, causing a cascade of waits in all its direct and indirect communication partners (enlarged white, green and brown).

Both effects are reflected as a drop in the efficieny factors previously seen in the last column of Table 9. Overall, what the model points as *Serialization* is the effect of delays in the compu-

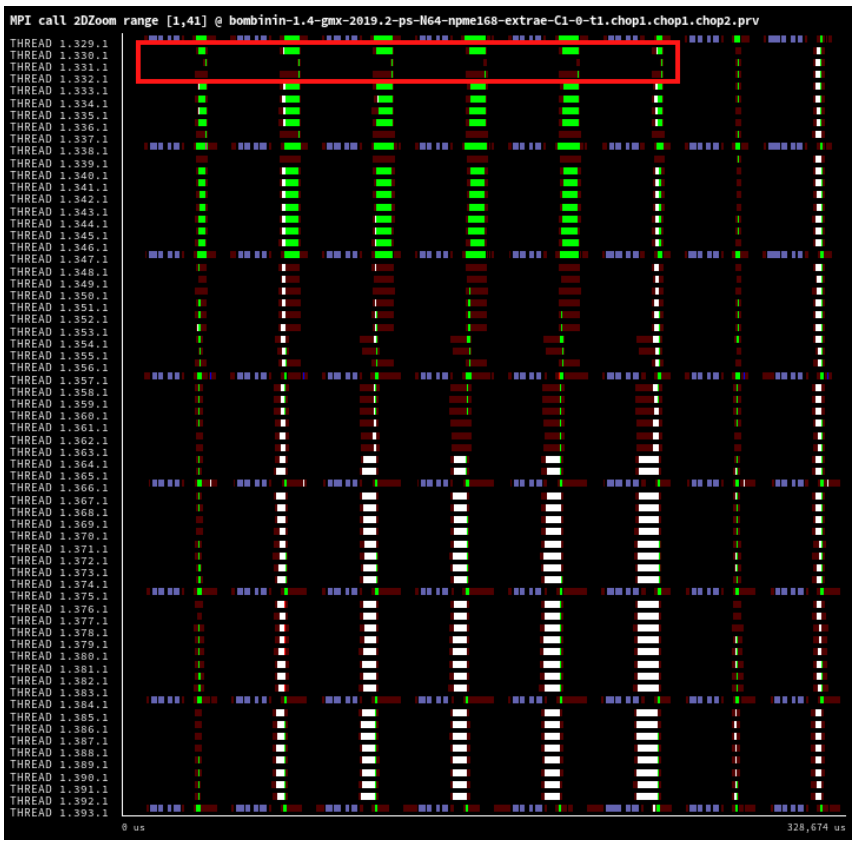


Figure 23.: Zoomed view in 64-nodes run showing process 331 of GROMACS systematically delayed, causing a cascade of increased MPI wait times in partners

tations of a single process, that become amplified by the dependencies in the communication pattern impacting many others.

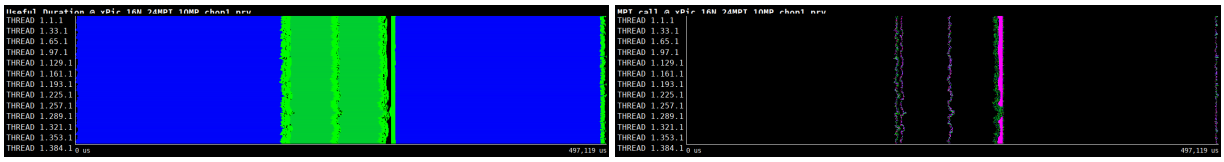
One recommendation would be to reconsider the communication pattern and the sequence of dependencies between PPs, PMEs, and PPs between PMEs, with the objective of minimizing them and increasing the asynchronism of the application, so that it is less sensitive to variability or system noise, that will be very frequent in exascale platforms.

Another consideration to avoid the waits in MPI would be to balance the PME:PP processes in a way that ensures the timely arrival of the PMEs to the communication phase, as many PPs rely on this. After discussing the analysis results with the user, they pointed out that GROMACS has a dynamic auto-tuning mechanism that does adjust the domain sizes and balance the computational load of the MPI ranks, but requires thousands of steps to optimize the particles distribution along the nodes. As a next step, it would be interesting to analyse the effects of this load balancing procedure in the application's efficiencies, and whether it helps to mitigate the *Serialization* issues.

4.4. xPIC

xPic is a particle-in-cell code used in the study of space plasma physics. It uses a first-principles algorithm, iteratively solving Maxwell's equations of electromagnetism and Newton's equations of motion. The code is composed of three main phases distributed in two solvers: a) the field solver calculates the electromagnetic fields in Cartesian grid; b) the particle solver transports billions of electrons and ions in a mesh free 3D space, and extracts the moment distribution functions from the particles to the Cartesian grid in order to couple them to the field solver. The code has multiple layers of memory management and parallelization and it uses MPI for inter-node communications and OpenMP for intra-node processing and offloading.

Two executions of the IMM TestONE case running 100 iterations with 768 cells per core of the code were run in weak scaling mode from 1 to 256 nodes on the JURECA cluster, one pure MPI with 24 ranks/node; and another hybrid MPI+OpenMP with 1 rank/node and 24 threads/rank. Starting from the pure MPI version, the structure of the application for one step of the iterative loop is depicted in Figure 24, showing two long computing phases at the beginning and at the end of the iteration (dark blue in 24a), separated by three waves of fast `MPI Ssend/Irecv` peer-to-peer communications (24b) ending in an `MPI_Allreduce`; and shorter computations (light green in 24a). The very high computation-to-communication ratio is also reflected by very high efficiency values, above 96%, for all the instrumented runs and efficiency factors, reported in Table 10. While the *Computation scalability* factor remains constant, we can observe a slight degradation in *Parallel efficiency* mainly caused by *Communication efficiency*. Actually, this degradation accelerates with scale, and just by looking at the execution times up to 256 nodes we can observe in Figure 25 a $\approx 40\%$ drop in the speed-up, that should remain constant for this weak scaling experiment.



(a) Computation duration (b) MPI calls

Figure 24.: xPIC application structure for 1 iteration of the pure MPI version

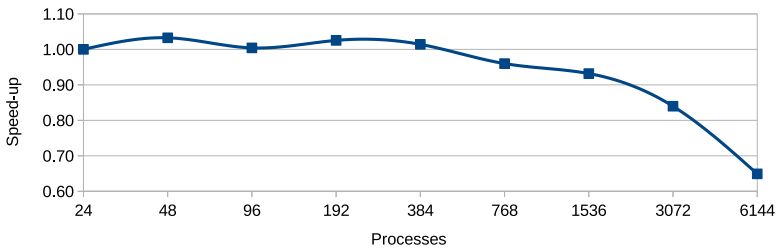


Figure 25.: xPIC measured speed-up for runs from 1 to 256 nodes

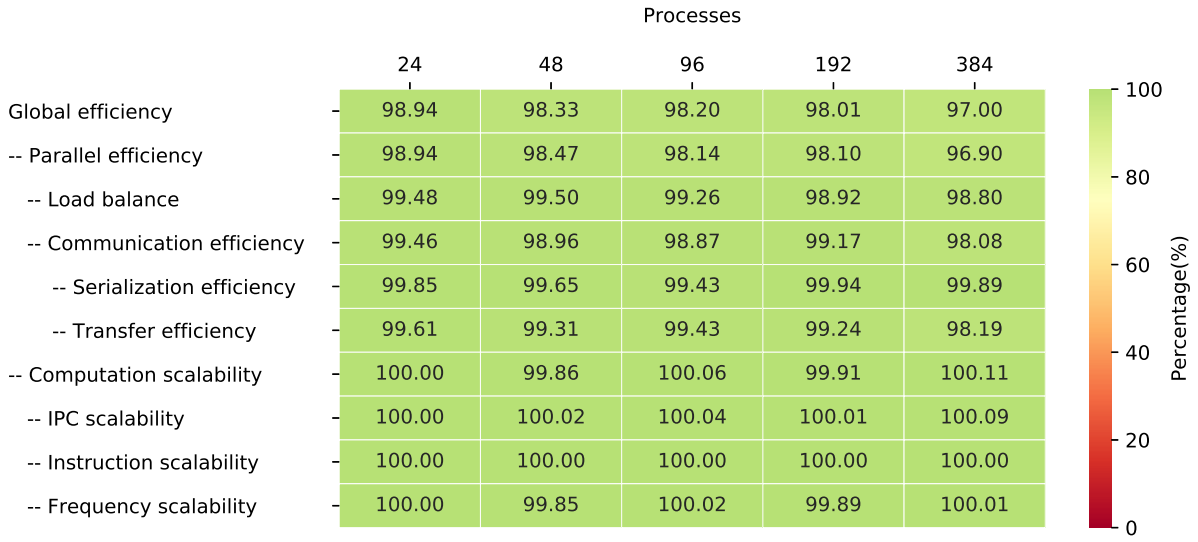


Table 10.: Efficiency model for xPIC (MPI version)

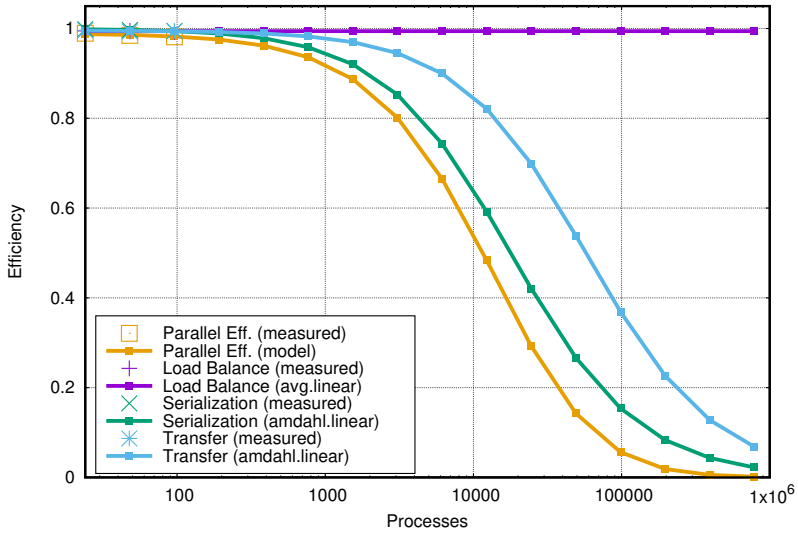
Extrapolation analysis

In previous reports we already presented an extrapolation model projected from the measurements of the first 5 experiments (from 1 to 16 nodes), and assuming an Amdahl fit for all their components, which resulted in a good fit up to the largest measured experiment with 256 nodes. However, the model was optimistic and predicted lower than real execution timings, with a $\approx 10\%$ deviation beyond this scale, as shown in Figure 26b for 6144 processes between real (blue) and predicted (green) timings.

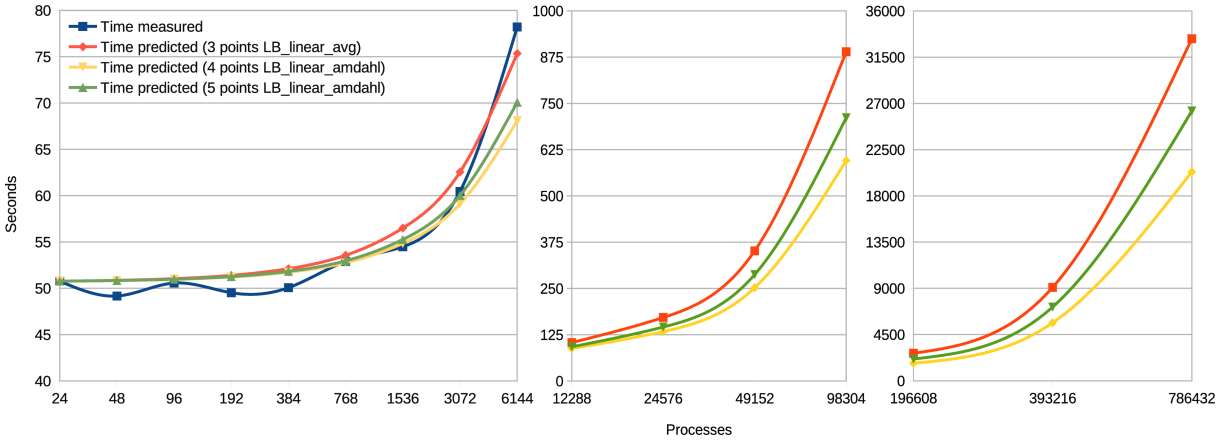
Focusing on the real execution timings, we observe variability mostly at the smaller scales (24 to 384 processes), reflected as ups and downs in the *Time measured* series. Since we work with a small set of measurements, the number of points selected to compute the extrapolation and their trend, specially when the last point falls into a peak or a valley, highly influence the shape of the projection. To this end, we extend this analysis to extrapolate with 3, 4 and 5 points and evaluate whether the projected trend improves or degrades based on the points used.

Figure 26b (left) shows the best fit for each selection of points, where the extrapolation with 3 points results in the most accurate prediction under the reasonable assumption that the *Load balance* factor would not degrade in a weak-scaling setup where the workload per process remains constant, and reduces deviation to $\approx 3\%$.

This is the best adjustment within the validation scale up to 6144 processes, just an order of magnitude higher than the data used for the extrapolation. However, the middle and right plots in Figure 26b show the three models diverge with the scale. To be certain of which of the three models would fit better at the largest scale (up to 3 orders of magnitude higher than the extrapolation data), it would be convenient to obtain more validation measurements not so far away from the intended predictions, so as to confirm which of the curves is closer to reality. Anyway, the observed trends already give us a clear intuition about the evolution of the application's behavior.



(a) Extrapolation with 3 points (runs from 24 to 96 processes)



(b) Validation with 9 points (runs up to 6144 processes; fitting error $< 4\%$)

Figure 26.: Extrapolation model for xPIC (MPI version)

The improved extrapolated model, shown in Figure 26a, shows a continuous degradation in *Parallel efficiency*, and on the basis of these figures we conclude this xPIC configuration should not run with more than ≈6 thousand processes as the efficiency falls way below 80%, which we consider the threshold for good efficiency. This degradation is caused by low *Communication efficiency*, originated by *Serialization* effects that will be studied in detail in the following section.

Potential improvements towards exascale

As we have seen the scalalability of xPIC is topped up well before the exascale. In this section we analyse the traces in detail to understand the causes for degradation and how to circumvent them.

Figure 27 shows 5 timelines for consecutive time intervals within a single iteration of the algorithm, zooming in few processes of a 16-node execution. Starting from the initial computation (1), each timeline (2-5) focuses on the communication region at the end of each phase. The communication pattern consists in each rank talking to their two immediate neighbours, with the last rank communicating with the first, and viceversa. All processes send the same total amount of messages and bytes, but one-third more messages to the upper ranks.

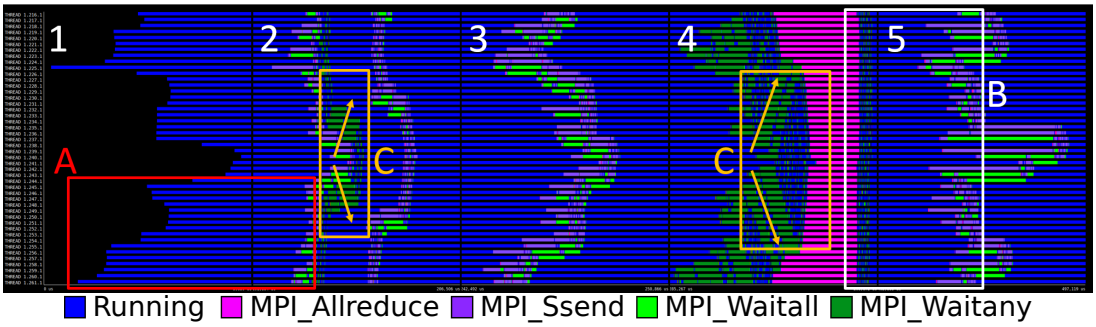


Figure 27.: xPIC collapsed timeline for the beginning and the four communication phases of the main iterative step (from left to right)

When the efficiency metrics are computed at the level of these internal phases, where the ratio of communication-to-computation is much higher, the observed *Parallel efficiency* is significantly lower, as shown in the `Real` row of Table 11. This is partly due to computation imbalance, that can be clearly observed in 27A, where processes not only arrive delayed from the previous iteration, but also take variable amount of time to reach the next communication phase 2. Or in 27B, where all processes exit synchronized from a global operation, but again arrive staggered to the next communication phase 5. Histogram in Figure 28 shows a variability of $\approx 10\%$ on the duration of the computations in phases 2 to 4, where the different processes (rows) present increasingly larger computations (from left to right), mainly due to lower IPC (green) for the larger computations (right). The same behaviour is exhibited in all phases but the slower processes vary over phases and iterations, leading to staggered arrivals to the subsequent communications. On the other hand, the communication pattern aggravates this problem by introducing serializations, seen clearly in 27C where the imbalance is propagated directionally to the immediate upper and lower MPI ranks, creating triangular shapes that expand from a central process. When this process happens to be a slow one, this generates a chain of dependencies through the neighbourhood that is ultimately paid in overly large waiting times in the collectives (Figure 29a). Furthermore, the more processes that participate in this chained communication pattern, the larger the waiting times will become.

With Dimemas we can easily simulate what would be the effect of eliminating the variability observed in the the computations. To this end we modified the duration of every computing phase to be the average of all processes in that same phase, resulting in an up to 90% reduction of communication waiting times, exemplified in Figure 29b for the collective phase, although all other phases show the same degree of improvement. At the efficiency level, the `Balanced` row in Table 11 reflects improvements between 15 to 53 percentage points in *Parallel efficiency* for each phase, gained by balancing the computations. This demonstrates the waiting times in MPI originate from dynamic imbalances in the computations, confirming the extrapolation prediction that serializations are the root cause of the performance issues of the application.

	Phase 2	Phase 3	Phase 4	Phase 5
ParEff Real	81.56	80.08	41.45	77.60
ParEff Balanced	97.06	97.42	94.81	97.22

100 50 0 Pct (%)

Table 11.: xPIC efficiency metrics per phase

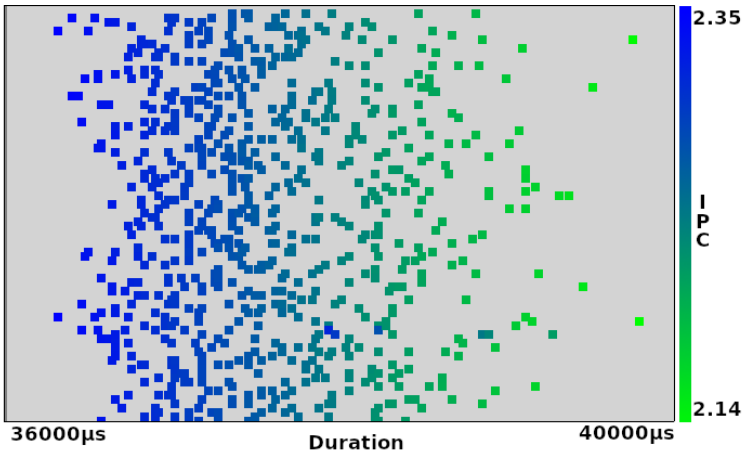
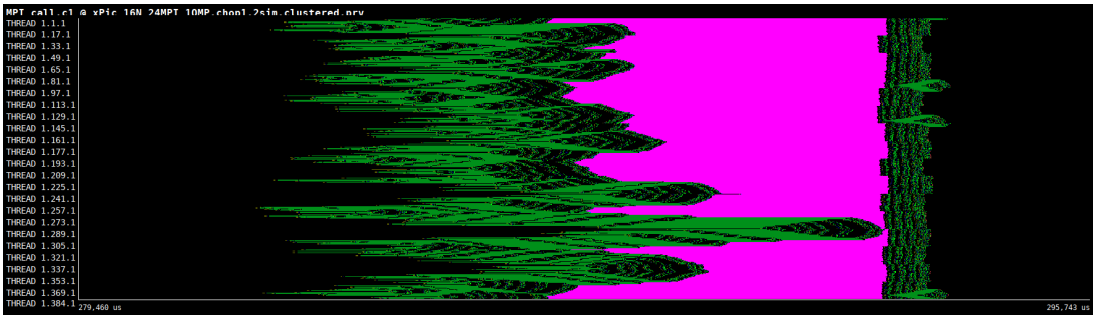


Figure 28.: xPIC computation duration histogram correlated with IPC



(a) MPI communication phase 4 showing serialized `MPI_Waitany` (green) delayed due to communication pattern and consequently delaying `MPI_Allreduce` (pink)

(b) Simulation with balanced computations showing improved MPI times for the same phase as 29a

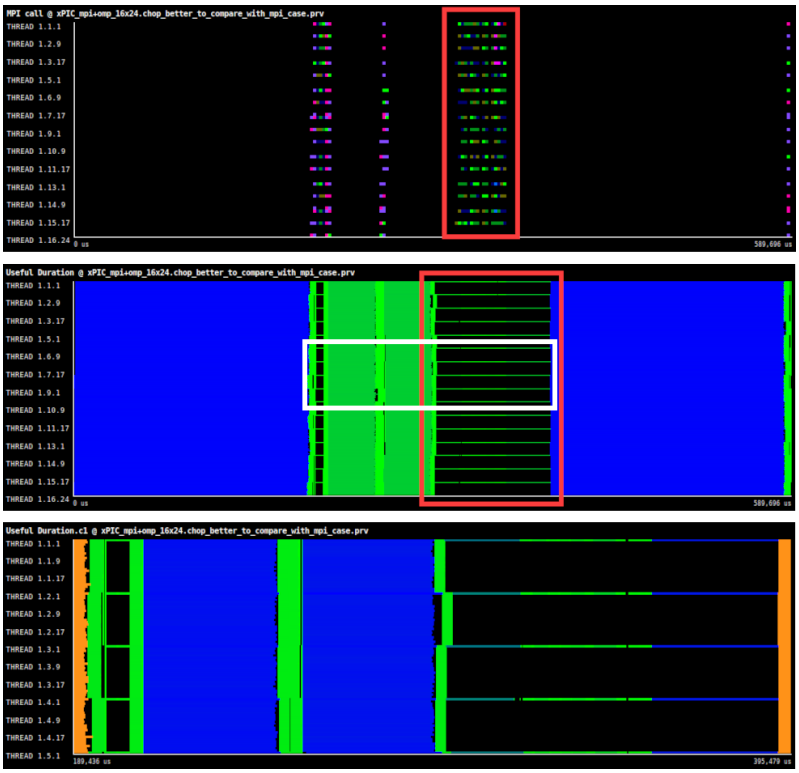
Figure 29.: xPIC detailed views comparing real and simulated runs

As perfect computation balance may be difficult to achieve, another approach to improve the efficiencies, that Dimemas is not able to simulate, would consist in redisigning the communications of the application to a more flexible pattern that is able to break the dependency chains between processes enabling a higher number of simultaneous communications that would reduce those waiting times due to serialized partners.

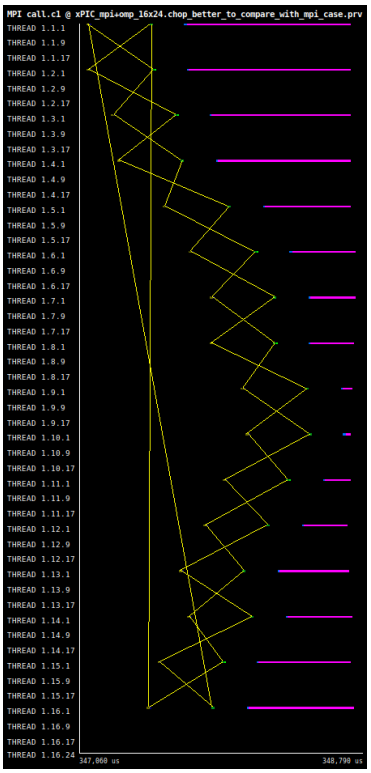
Comparison with hybrid MPI+OpenMP version

xPIC can also be run with a hybrid configuration of MPI+OpenMP. Adding OpenMP has the advantage of increasing parallelism without scaling up the number of MPI ranks, which limits network traffic and reduces potential risk of congestion. Furthermore, as the iteration time is dominated by long computations, thread-level parallelism is a logical solution to accelerate these regions by splitting the work among multiple threads. The following analysis studies this scenario with an execution of a hybrid MPI+OpenMP version of xPIC ran on 16 nodes of the JURECA cluster, with 1 MPI rank per node and 24 OpenMP threads per rank.

The computation structure for one full step of the main iterative loop, as well as a zoomed view for a subset of processes in the middle area, are depicted in mid and bottom timelines in Figure 30a, respectively. The structure is very similar to the MPI version of the code previously shown



(a) Detailed view for MPI communications (top), computations (mid), and zoomed (white box) computations (bottom)



(b) Serialized communication pattern

Figure 30.: xPIC hybrid MPI+OpenMP structure

in Figure 24a, but introduces a new issue: a large sequential phase that represents roughly the 17% of the total iteration time, where only the MPI ranks are computing while all OpenMP threads are sitting idle (black gaps), resulting in an inefficient use of the available resources. Most MPI communications occur during this sequential phase, as shown in Figure 30a (top), which suggests a limitation in the algorithm that requires data exchange to progress, preventing further parallelism.

The communications outside the sequential phase follow the same pattern observed before, with the difference that up and downwards messages now originate from two threads of the same MPI process (both the master and the last worker threads). Having MPI communications inside the OpenMP parallel region is a good attempt to overlap computation and communication. However, with the current configuration of 24 threads per process, the assigned chunk of work is too small and the computation is completed much earlier than the communications take place, so the overlap is not achieved as shown in Figure 31.

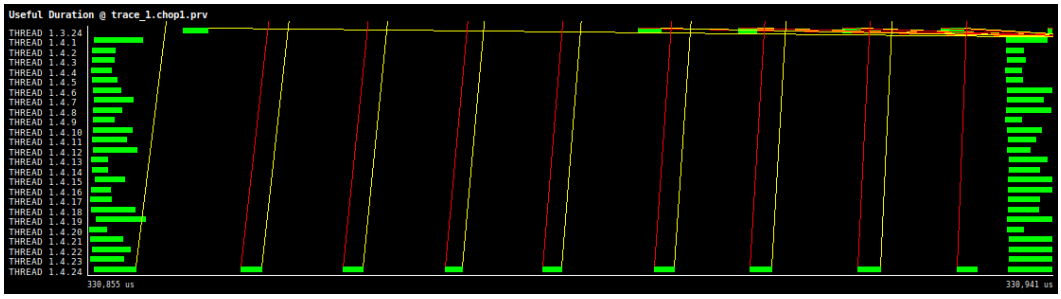


Figure 31.: Hybrid xPIC: MPI communications (red and yellow lines) inside OpenMP parallel regions not overlapping with computations (green phases) of other threads

Serialization effects are still present due to previously unbalanced computations (same effect observed in the pure-MPI case), as shown in Figure 30b, where yellow lines that represent peer-to-peer communications between partners are clearly staggered through a dependency chain delaying the `MPI_Allreduce` collective (pink) that follows.

Threads (MPI ranks x OpenMP threads)

	24(1x24)	48(2x24)	96(4x24)	192(8x24)	384(16x24)
-- Hybrid Parallel efficiency	82.91	82.48	82.17	81.83	81.05
-- MPI Parallel efficiency	100.00	99.57	99.59	99.39	98.93
-- MPI Load balance	100.00	99.65	99.69	99.48	99.04
-- MPI Communication efficiency	100.00	99.91	99.90	99.90	99.89
-- Serialization efficiency	100.00	99.99	99.98	99.93	99.99
-- Transfer efficiency	100.00	99.92	99.92	99.97	99.89
-- OpenMP Parallel efficiency	82.91	82.83	82.51	82.33	81.93
-- OpenMP Load Balance	83.23	83.30	83.06	83.02	83.38
-- OpenMP Communication efficiency	99.62	99.45	99.33	99.17	98.26

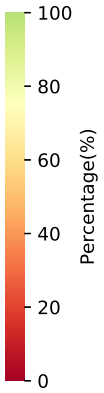
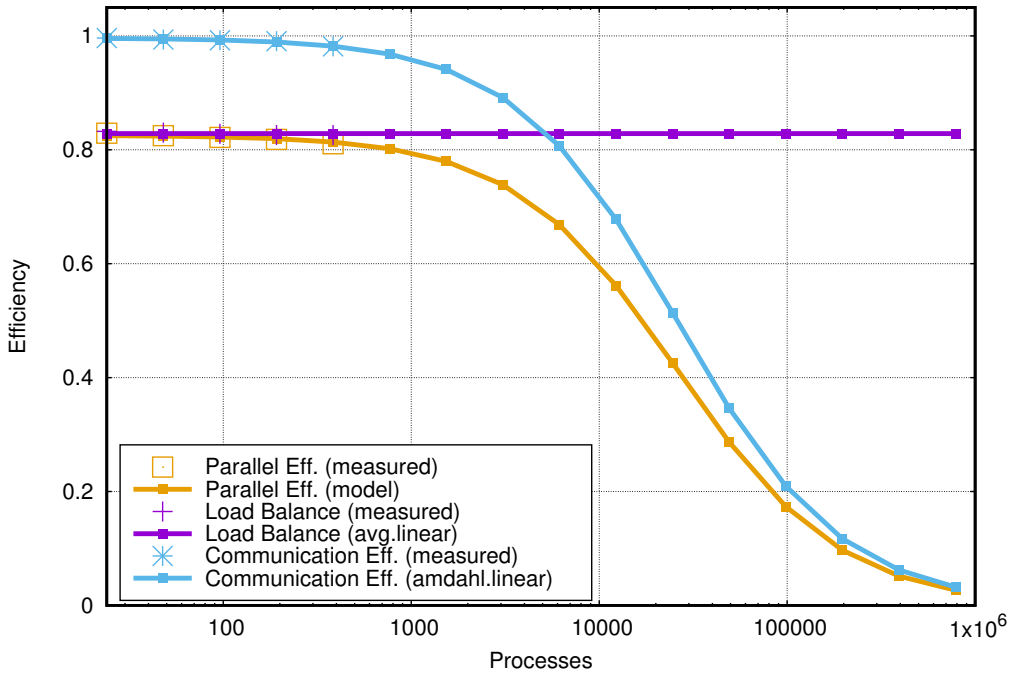
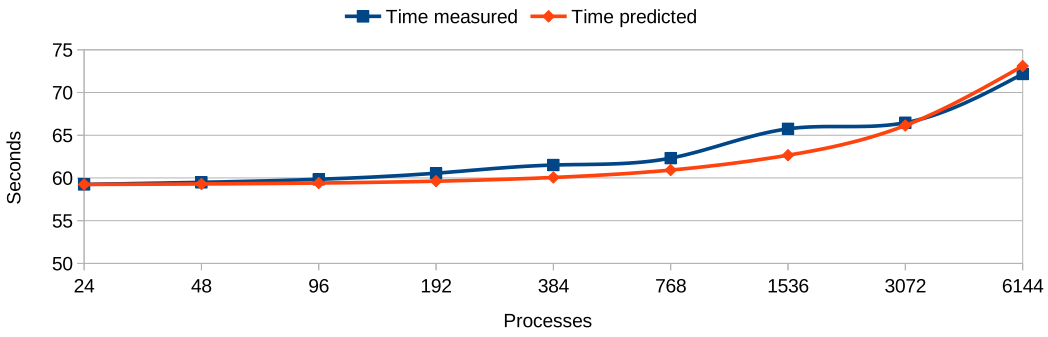


Table 12.: xPIC MPI+OMP efficiency metrics.

The hybrid efficiency model in Table 12 reflects the loss of efficiency in the OpenMP component, with *Parallel efficiency* mainly limited by *OpenMP Load balance*, which drops 17 percentage points down due to the sequential phase where only the master thread works. When scaling up, the extrapolation in Figure 32 shows that the hybrid MPI+OpenMP version presents the same range of good scalability as the MPI case up to ≈6 thousand processes. Yet in comparison, employing equal amount of resources, the pure MPI version performs more efficiently and scales slightly better.



(a) Extrapolation with 5 points (runs from 1x24 to 16x24 threads)



(b) Validation with 9 points (runs up to 256x24 threads; fitting error $< 5\%$)

Figure 32.: Extrapolation model for xPIC (MPI+OpenMP version)

4.5. NEST+Arbor

NEST+Arbor is a co-simulation application where the simulators operate at different levels of description: NEST at the resolution of single neurons allowing for simulations of large-scale neural network models, and Arbor at the resolution of neuronal compartments taking into account cell morphologies. NEST uses a hybrid parallelization scheme with OpenMP and MPI, while Arbor uses CUDA for GPU acceleration and MPI for communication between nodes. MPI is used for communication between the simulators.

NEST was run on the CM, scaling from 1 to 45 MPI processes, and Arbor on the ESB, also scaling from 1 to 45 processes, using one GPU device per process and mapping 1 MPI rank per node. The application was executed in weak-scaling mode, with a constant workload of 56250 neurons per node.

With the first traces obtained, a preliminary analysis detected a very fine-grain level of OpenMP parallelization with trace data that seemed to be produced from additional uninstrumented threads. This is usually the result of having nested parallel OpenMP regions, an structure that is not supported by the BSC tools at the time of writing. It is typically found when an OpenMP application uses lower-level numerical libraries which are in turn also parallelized with OpenMP. This was reported to the user, and to circumvent this limitation, the decision to capture only the MPI activity was taken, and new traces were obtained for this configuration. This approach assumes a perfect OpenMP parallelization, which ultimately results in optimistic efficiencies in the scalability analysis. But even with this simplification the analysis still provides useful insight about the application performance. Recently, the user reported back that a bug was discovered which made NEST to oversubscribe multiple threads on a single core, confirming the effects that were first detected with the analysis.

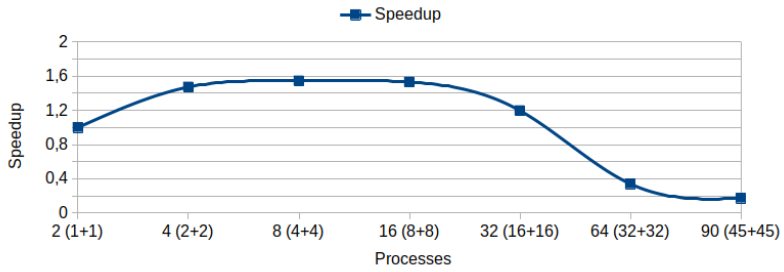
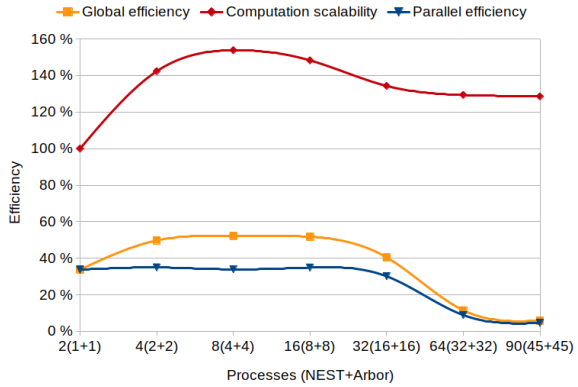


Figure 33.: NEST+Arbor measured speed-up for runs from 1 to 45 processes per module

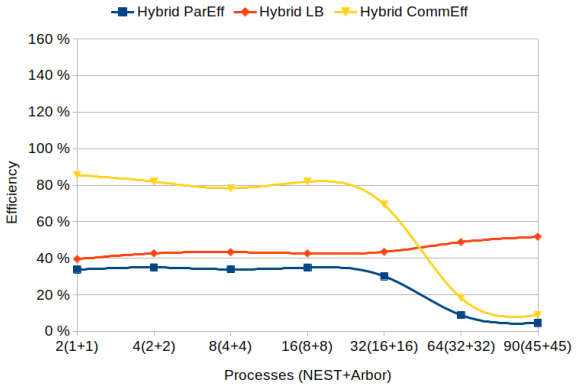
Just by looking at the execution times of the first experiments, we can already anticipate a severe performance problem that arises when scaling beyond 32 nodes, as shown in Figure 33 by the speedup sharply decreasing to less than 0.4 at 64 nodes, and less than 0.2 at 90. To understand this sudden drop, we can inspect the efficiency model for these scales, presented in Table 13 and plotted in Figure 34. The *hybrid Parallel efficiency*, which starts as low as $\approx 34\%$, drops down to less than 9% at 64 nodes, being the MPI component the main responsible for the low baseline, and more specifically, the *MPI Communicacion efficiency* for the abrupt decrease when scaling, pointing out a problem with MPI communications. The second hybrid component shows *CUDA Parallel efficiency* values bordering the $\approx 60\%$, but in CUDA codes these should still be interpreted as good. When parallelizing with CUDA, the host

	Processes (NEST + Arbor)						
	2(1+1)	4(2+2)	8(4+4)	16(8+8)	32(16+16)	64(32+32)	90(45+45)
Global efficiency	33.79	49.76	52.20	51.74	40.45	11.50	5.85
-- Hybrid Parallel efficiency	33.79	34.96	33.93	34.89	30.12	8.89	4.55
-- MPI Parallel efficiency	47.19	54.56	56.76	57.06	49.35	14.91	7.72
-- MPI Load balance	54.90	66.15	71.61	69.06	70.19	81.13	86.92
-- MPI Communication efficiency	85.96	82.47	79.26	82.63	70.31	18.37	8.88
-- CUDA Parallel efficiency	71.60	64.09	59.77	61.15	61.02	59.67	58.88
-- CUDA Load Balance	71.94	64.52	60.49	61.72	61.94	60.22	59.52
-- CUDA Communication efficiency	99.52	99.33	98.82	99.07	98.52	99.08	98.91
-- Computation scalability	100.00	142.35	153.83	148.29	134.30	129.38	128.57

Table 13.: Efficiency model for NEST+ARBOR (small scale optimization enabled)



(a) Global efficiencies: *ParEff* drops at 64 nodes. *CompSca* partially compensates low *ParEff*.



(b) Hybrid efficiencies: *LB* does not degrade. *CommEff* crashes at 64 nodes.

Figure 34.: Global & hybrid efficiencies for NEST+ARBOR (small scale optimization enabled)

process typically delegates most of the work to the GPU and simply waits for results. Hence, the amount of work performed by the GPU and the host becomes unbalanced, which accounts for the low *CUDA Load balance* measured by our model. Yet, this is inherent to the CUDA programming paradigm, and efficiency values up to $\approx 50\%$ are to be considered good. The *Global efficiency* slightly improves over the hybrid components, benefitting from a reduction of the computational complexity at all scales with respect to the base case, as pointed by the *Computation scalability* values over 100%.

In order to investigate further the problem detected in the MPI communications, we compute the efficiency model for both components of the modular application. Tables 14 and 15, along with the corresponding plots in Figures 35a and 35b, show the efficiency model and the decomposition of *ParEff* for NEST and Arbor, respectively. Focusing on NEST, its *CommEff* suffers a huge drop from 70% to less than 20% at 32 cores (64-nodes run), and less than 10% at 45 (90-nodes run), revealing that the NEST module is the main responsible for the *Hybrid CommEff* crash at this scale when the binaries are coupled. Focusing on Arbor, although we can also see a significant drop of *CommEff* at the same scales, the *CommEff* was already low right from the start. Thus, the Arbor module is the main responsible for the low baseline of the *Hybrid ParEff*.

	Processes (NEST) 1	2	4	8	16	32	45
ParEff	85.96	82.35	78.49	80.32	68.61	18.02	8.60
LB	100.00	100.00	99.00	97.00	98.00	98.00	97.00
CommEff	85.96	82.35	79.26	82.63	70.31	18.37	8.88

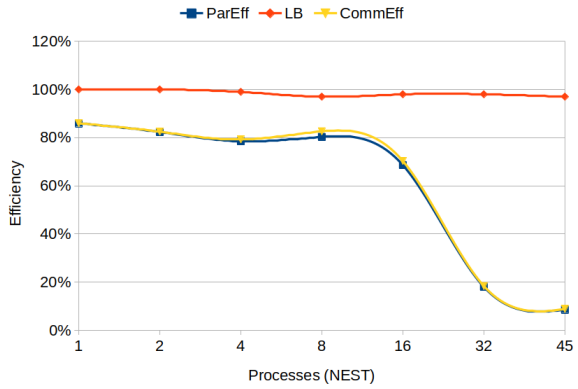
Pct (%): 100, 50, 0

Table 14.: Efficiency model for NEST (small scale optimization enabled)

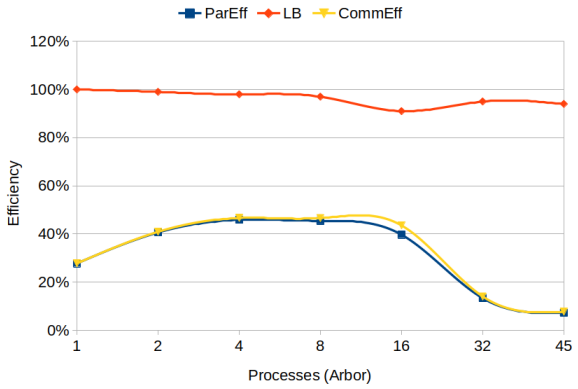
	Processes (Arbor) 1	2	4	8	16	32	45
ParEff	27.81	40.66	45.90	45.43	39.72	13.35	7.28
LB	100.00	99.00	98.00	97.00	91.00	95.00	94.00
CommEff	27.81	40.93	46.65	46.62	43.56	14.06	7.74

Pct (%): 100, 50, 0

Table 15.: Efficiency model for Arbor (small scale optimization enabled)



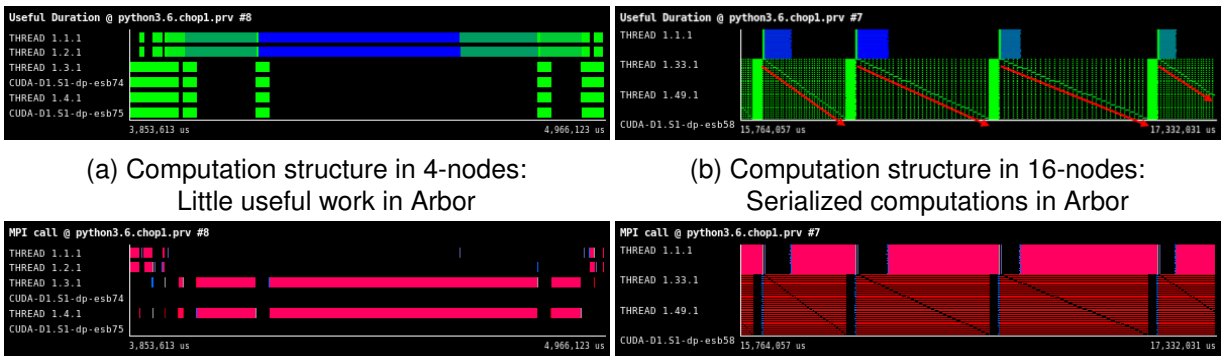
(a) NEST *ParEff* factors:
NEST *CommEff* drop at 32 cores (64-node run)



(b) Arbor *ParEff* factors:
Arbor *CommEff* keeps *ParEff* low from the start

Figure 35.: *Parallel efficiency* decomposition calculated independently for NEST and Arbor (small scale optimization enabled)

An in-depth analysis of the traces help us understand where the MPI inefficiencies originate. Timelines in Figure 36 show views for computations (top) and MPI calls (bottom) for a small scale execution with 4 nodes (left), and for the execution with 64 nodes (right). Top rows represent NEST processes, while bottom rows represent Arbor processes plus their GPU stream. Looking at the timelines for the 4-nodes case, the whole iteration interval is filled with computations for NEST (green and blue in top rows of 36a). The Arbor host processes are stalling most of the time in `MPI_Allgather` (pink in bottom rows of 36c) waiting for NEST to finish its computations, effectively limiting Arbor's *CommEff* and refraining the global *ParEff* from the start. For the 64-nodes case, Arbor introduces a large serialized computation phase (green diagonal in 36b) that skyrockets not only the MPI time in Arbor itself, but also drags NEST into large synchronization stalls waiting for Arbor to finish, causing the sharp drop observed in this scale in *CommEff*. Nevertheless, this effect is not exclusive to the larger runs, as the phase with serialized computations is present in all scales, only that when there are fewer processes



(a) Computation structure in 4-nodes: Little useful work in Arbor

(b) Computation structure in 16-nodes: Serialized computations in Arbor

(c) MPI calls in 4-nodes: High `MPI_Allgather` time (pink) in Arbor

(d) MPI calls in 16-nodes: High MPI synchronization time in NEST & Arbor

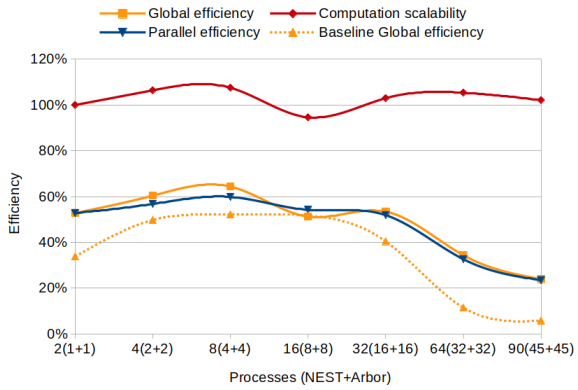
Figure 36.: NEST+ARBOR computation (top) and communication (bottom) structure for 4-nodes (left) and 16-nodes (right) runs

involved, the dependency chain is shorter and it is not until it grows that its impact becomes readily perceived.

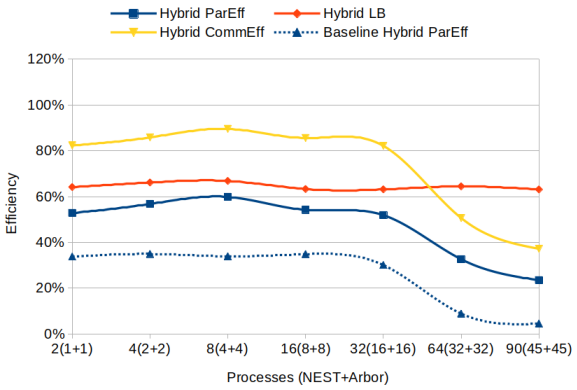
With the reported efficiencies at the largest runs, it would not be reasonable to keep scaling up the current setup of the application any further, thus the extrapolation analysis was not considered. Instead, feedback of the analysis was given to the user, and they were able to identify that a certain optimization targeted to improve the application performance in small scales was activated, which in turn degrades the performance of larger scales. Therefore, a second batch of traces was produced disabling such optimization, in order to study what difference does it make and the potential scalability gains.

Figure 37 shows the same plots as before with the efficiency factors for the new use case. In each plot, the top-level efficiency metric for the previous experiments is also drawn as a baseline (dotted series) for easy comparison. Figure 37a depicts the *Global efficiency* that improves up to 23 percentage points in the larger runs, mainly due to the *Hybrid ParEff* (Figure 37b) that sees improvements in its factors, the *Hybrid LB* going up to $\approx 60\%$ (from $\approx 40\%$), and a more moderate fall of *Hybrid CommEff* down to $\approx 40\%$ (previously 20%). With the new setup, one of the two problems initially detected of having a low baseline *Hybrid Parallel efficiency* is significantly improved, by 20 percentage points on average.

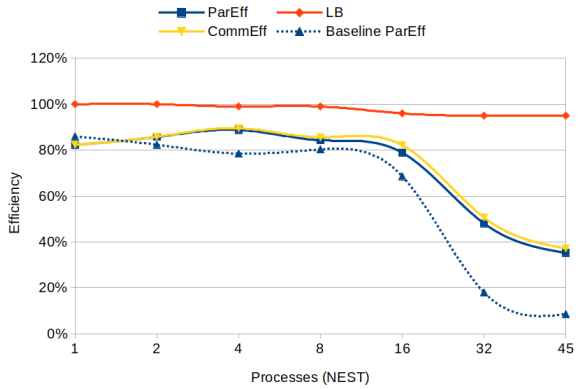
Despite the improvement, the second problem regarding the serialized computations in Arbor is still present in the new setup, with very similar projected trends. We can still appreciate the decrease in *Hybrid CommEff* at 64 nodes, which is still more pronounced for the NEST component (Figure 37c) than Arbor (37d), as the former is heavily delayed by the latter as discussed before.



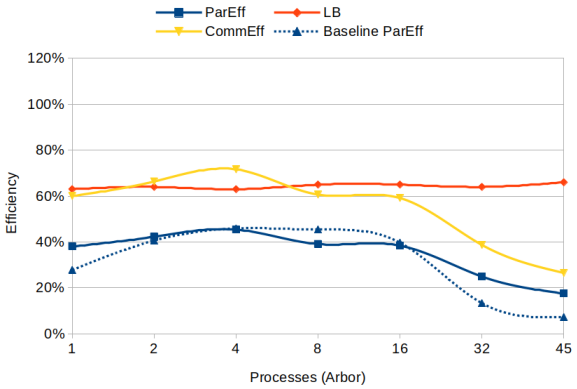
(a) Global efficiencies: Improvements up to 23 percentage points in large runs



(b) Hybrid efficiencies: *LB* & *CommEff* contribute to higher *ParEff*



(c) NEST *ParEff* factors: *CommEff* drop still present at 32 cores (64-nodes run)



(d) Arbor *ParEff* factors: Trades *LB* for *CommEff*, still impacted by serializations at 32 cores

Figure 37.: Efficiencies for NEST+Arbor coupled and per-component (small scale optimization disabled)

Potential improvements towards exascale

In the previous section we have concluded that the biggest performance bottleneck is a large serialization phase in Arbor that also affects NEST, but we can inspect in the traces how this effect happens in detail. Figure 38 shows two iterations of the main loop where, simplifying, both modules have two main phases in each iteration: NEST has a computing phase (A), followed by a synchronization point (B). Simultaneously, Arbor presents the detected serialized computations in the host process (C), followed by a CUDA accelerated region with a large number of memory transfers between the host and the accelerator (D).

The only point of synchronization between NEST and Arbor is at the end of B and D, and B is elongated because it is waiting for Arbor to finish D. However, if C didn't exist in the first place, D would start sooner and run simultaneously with A. Given the amount of work in A and D is similar (take the same amount of time), both would reach the synchronization phase B at the same time, effectively eliminating most of NEST's waiting time. Overall, eliminating C, which is an improvement for Arbor, would also eliminate B, which is a collateral improvement for NEST.

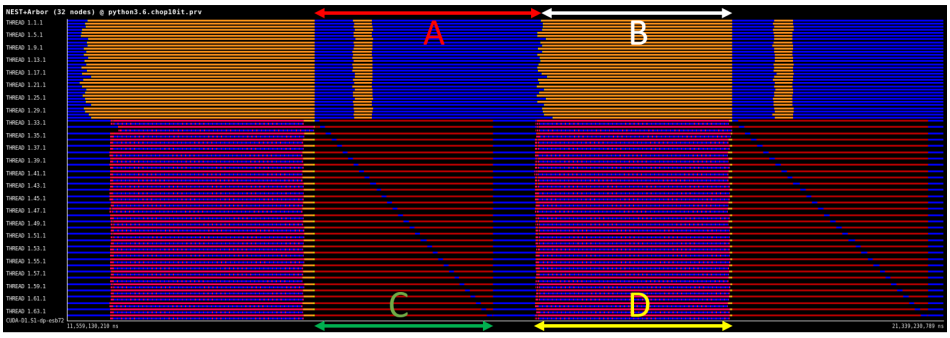
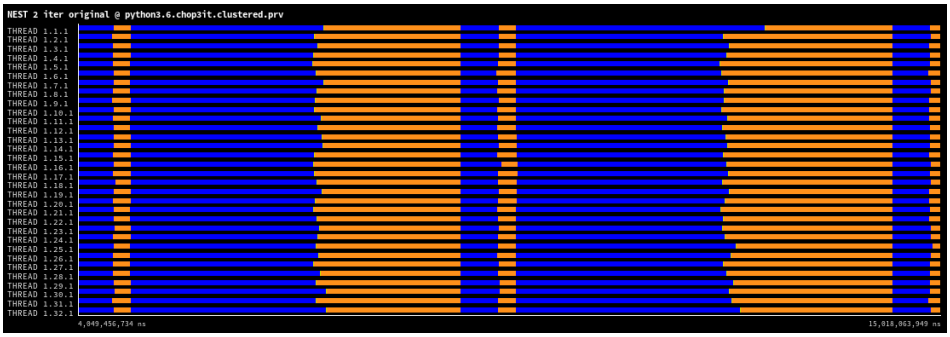
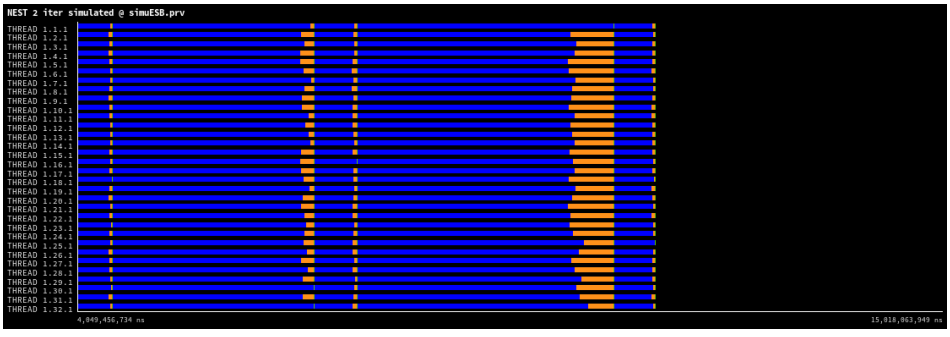


Figure 38.: 2 iterations of the 64 node run showing Arbor's serialization at large scales

This scenario was simulated with Dimemas by setting the length of computations in phase C to 0, which completely eliminates these computations assuming them to be negligible, and makes this phase disappear. The study was performed for the two experiments that present this effect more severely, at 64 and 90 nodes. Timelines in Figure 39 show the impact on NEST, where we see a reduction on the simulated execution of $\approx 90\%$ of the MPI time (orange), (39b) and of $\approx 33\%$ of the total execution time. As we see in the image, the synchronization phase B cannot completely disappear and is more significant in some iterations than others due to small imbalances between the computing phases A and D.

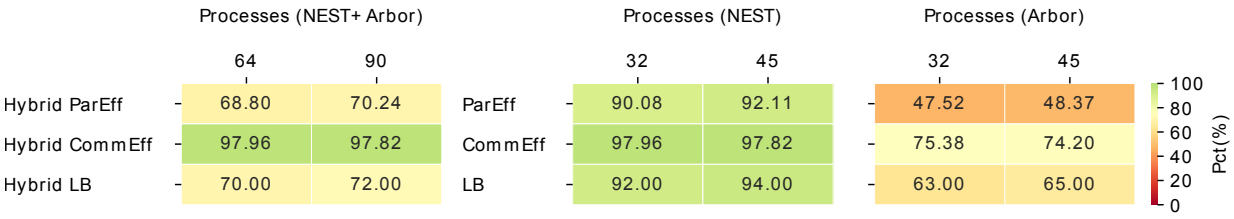


(a) 2 iterations of the main loop of NEST in the original 64-nodes run



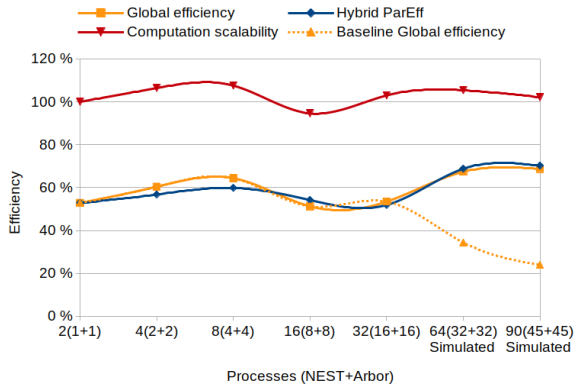
(b) MPI_Allgather (orange) reduced by $\approx 90\%$ for an iteration time improvement of $\approx 30\%$

Figure 39.: Comparison of original and simulated run at 64 nodes showing the effect on NEST of removing Arbor's serialization

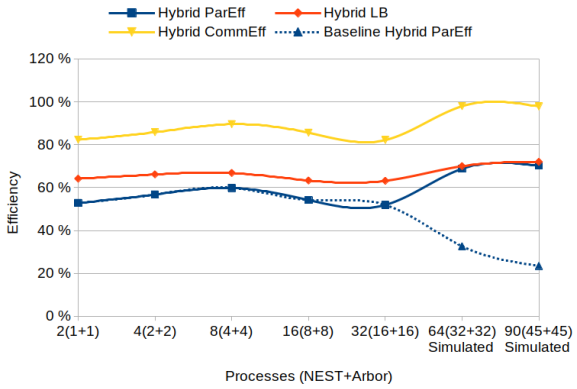


(a) NEST+Arbor (b) NEST (c) Arbor

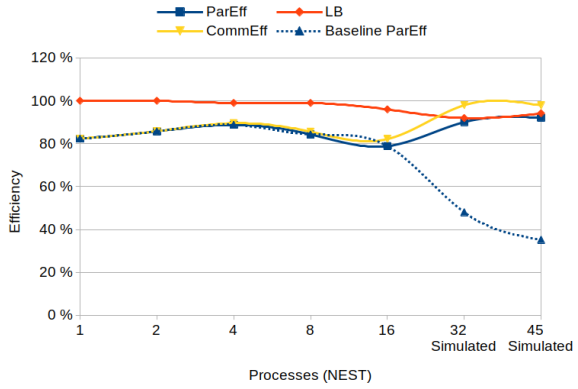
Figure 40.: Efficiencies for NEST+Arbor simulating non-existent Arbor serialization in 2 last experiments (64 and 90 nodes)



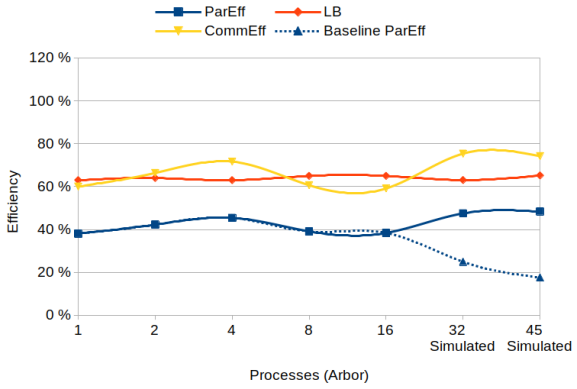
(a) Global efficiencies: Up to 45 percentage points improvement due to *Hybrid ParEff*



(b) Hybrid efficiencies: Better *Hybrid ParEff* due to *CommEff* no longer dropping and improved *LB*



(c) NEST *ParEff* factors: Up to ≈ 60 percentage points improvement



(d) Arbor *ParEff* factors: Up to ≈ 30 percentage points improvement

Figure 41.: Efficiencies for NEST+Arbor coupled and per-component simulating non-existent serialization in 2 last experiments (64 and 90 nodes)

All in all, efficiencies are greatly improved at all levels of the coupled application as shown in Table 40 for the global efficiency metrics, and Tables 40b and 40c for NEST and Arbor respectively. Compared to the initial use case, Arbor sees an improvement of ≈ 40 percentage points in *ParEff*, but the major beneficiary is NEST with twice as much improvement up to ≈ 80 percentage points. The plots in 41 show the efficiency metrics compared to the second

use case with the optimizations for small scale disabled, and present ≈ 30 percentage points improvement in *ParEff* for Arbor, and again twice as much for NEST.

With the serialization problem resolved, it would now be reasonable to try scaling up to a larger number of nodes. Figure 42 presents a very preliminary extrapolation analysis of the projected efficiencies that we would potentially find at larger scales. This has to be very carefully considered, as the extrapolation is computed from the two simulated experiments (2 points) only, and further validation points at larger scales to ensure the correctness of the projection are not available, thus there is no actual guarantee that the application will certainly exhibit the depicted trends. However, under the pessimistic assumption that all efficiency factors would degrade following an Amdahl's fit, the extrapolation highlights *CommEff* as the main limiting factor in the exascale. In this weak-scaling scenario, the number of communications per process remains constant. But with the increasing scale, the total number of communications also increases, as so slightly does the total volume of transferred data, until at some point network contention may become an issue. Starting from the current scale, the application would maintain good efficiency values at least for two additional orders of magnitude, before the impact of the network would start to be noticeable.

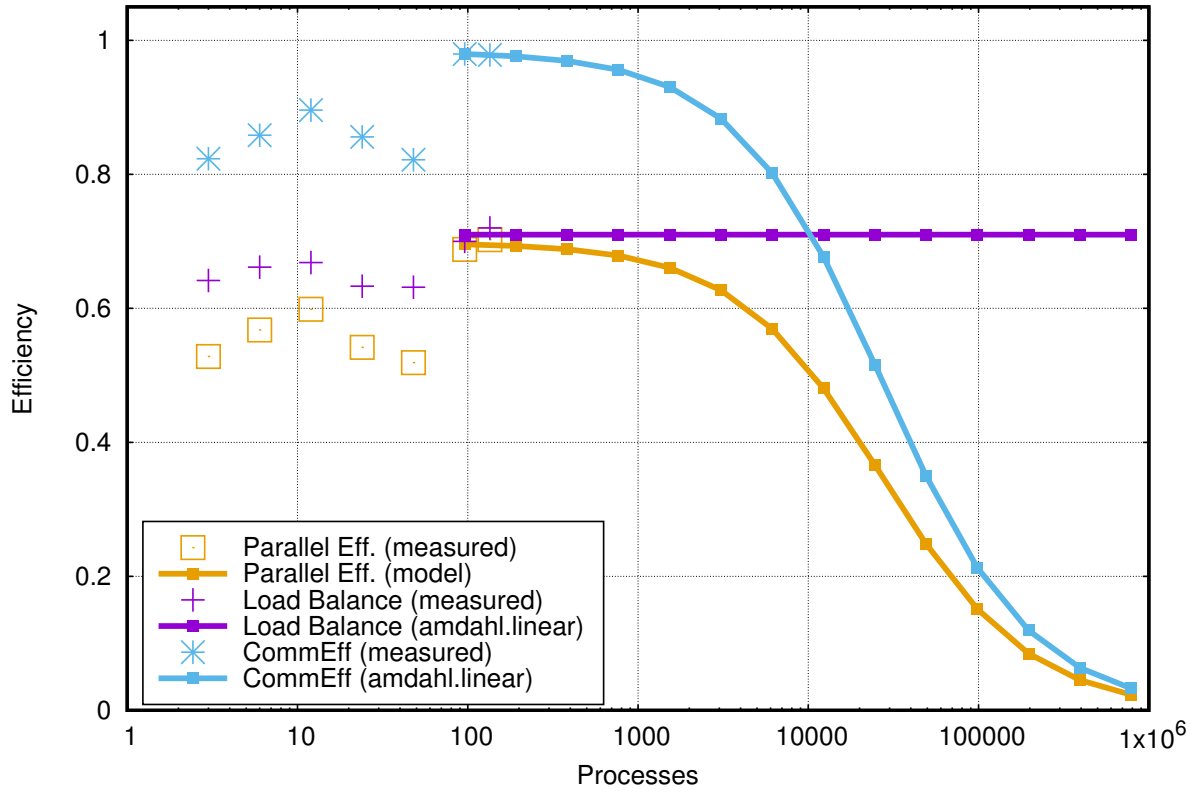


Figure 42.: Extrapolation model for NEST+Arbor with 2 points (simulated runs at 64 & 90 nodes) predicting good scalability up to 10k processes

4.6. NextDBSCAN

NextDBSCAN is a new implementation of the DBSCAN clustering algorithm, which uses a novel data-structure suitable for all types of parallel processing, optionally including GPGPU accelerators. It is capable to perform very fast data partitioning and comparison in parallel.

Table 16 shows the global efficiencies measured for a hybrid MPI+OpenMP version running on the CM from 1 to 44 nodes, mapping 1 MPI rank per node. It stands out that the application has a sudden change of behavior at the scale of 4 nodes, as the *Computation scalability* drops by 25 percentage points, being indicative of 25% more instructions than in previous scales that stabilizes from this point forward. This effect is related to an iterative algorithm that repeats until reaching the convergence of cluster labels across all nodes. Labels are assigned concurrently but can vary across nodes. The more nodes, the higher the chance of a deeper label dependency which causes more iterations than the same problem with lower cardinality. If the result contains a low number of clusters, the chances of this occurring are greatly reduced, however, the opposite applies for a high number of nodes and clusters. For our study, we assumed that this replication would not reappear in larger scales.

Threads (MPI ranks x OpenMP threads)

	24(2x12)	48(4x12)	96(8x12)	192(16x12)	384(32x12)	768(64x12)	1056(88x12)
Global efficiency	99.46	99.37	73.80	73.73	73.65	73.66	73.53
-- Parallel efficiency	99.46	99.74	99.67	99.57	99.47	99.42	99.23
-- Load balance	99.47	99.84	99.81	99.73	99.69	99.62	99.50
-- Communication efficiency	100.00	99.90	99.86	99.84	99.78	99.80	99.73
-- Computation scalability	100.00	99.63	74.05	74.04	74.04	74.08	74.10
-- IPC scalability	100.00	99.63	99.96	99.93	99.94	99.99	100.01
-- Instruction scalability	100.00	100.00	74.11	74.10	74.10	74.10	74.11
-- Frequency scalability	100.00	99.99	99.97	99.99	99.98	99.98	99.97

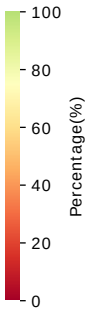


Table 16.: Global efficiency model for NextDBSCAN

Except for this increase of instructions, the model reports very high values for all efficiency factors with just a very slight degradation, mostly perceptible in the *Load balance* and *Serialization* of the MPI component, as shown in Table 17.

Threads (MPI ranks x OpenMP threads)

	24(2x12)	48(4x12)	96(8x12)	192(16x12)	384(32x12)	768(64x12)	1056(88x12)
-- Hybrid Parallel efficiency	99.46	99.74	99.67	99.57	99.47	99.42	99.23
-- MPI Parallel efficiency	99.52	99.82	99.75	99.66	99.57	99.55	99.37
-- MPI Load balance	99.52	99.92	99.89	99.81	99.77	99.74	99.62
-- MPI Communication efficiency	100.00	99.90	99.86	99.85	99.79	99.81	99.74
-- Serialization efficiency	100.00	99.91	99.86	99.86	99.80	99.83	99.78
-- Transfer efficiency	100.00	100.00	100.00	100.00	99.99	99.98	99.96
-- OpenMP Parallel efficiency	99.94	99.92	99.92	99.92	99.90	99.87	99.86
-- OpenMP Load Balance	99.94	99.92	99.92	99.92	99.92	99.88	99.87
-- OpenMP Communication efficiency	100.00	100.00	100.00	99.99	99.99	99.99	99.99

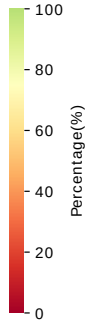
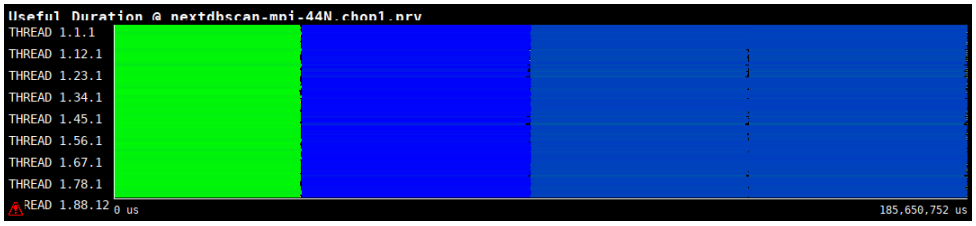


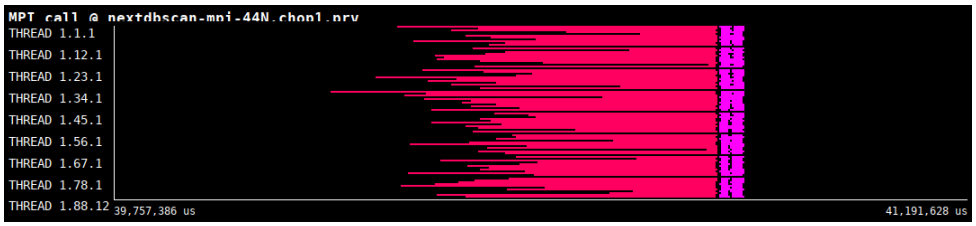
Table 17.: Hybrid efficiency model for NextDBSCAN

The very high efficiencies can be understood by looking at the structure of the application in Figure 43a, where we see four computing steps that dominate the $> 99\%$ of the execution, with communications taking $< 1\%$ of the time. In this situation, the small inefficiencies produced by communication stalls and transfers are almost negligible, hence the high reported efficiencies. The communication phases between computation regions employ collective operations only, with `MPI_Allreduce` called twice in each phase, and a preceding call to `MPI_Allgather` in the first phase only, as shown in Figures 43b and 43c. We can easily see how the first elongated collective is absorbing imbalances in the previous computations, while the subsequent is very fast, yet it is not very significant as the percentage of imbalance is $< 1\%$.



(a) Four computing phases with a 20% variation between the shorter (green) and the larger (dark blue)

(b) `MPI_Allreduce` (pink) phase at the end of each the computing step

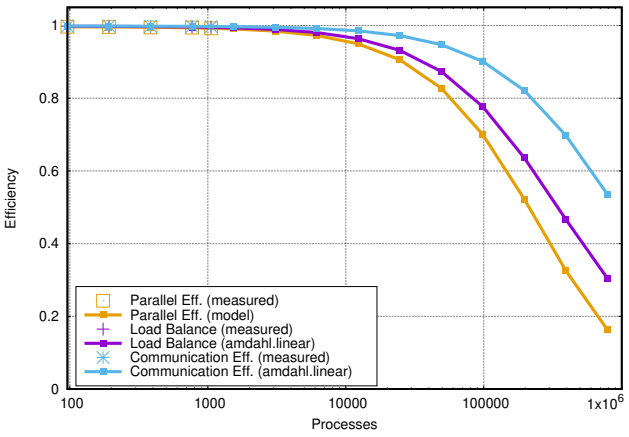


(c) Additional `MPI_Allgather` (red) phase at the end of the first computing step

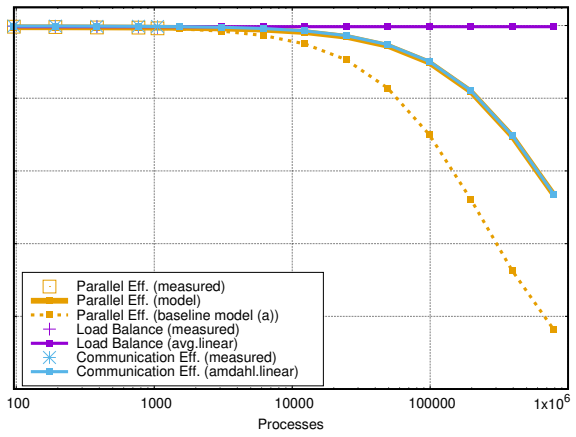
Figure 43.: NextDBSCAN structure for a 44-node run

Extrapolation analysis

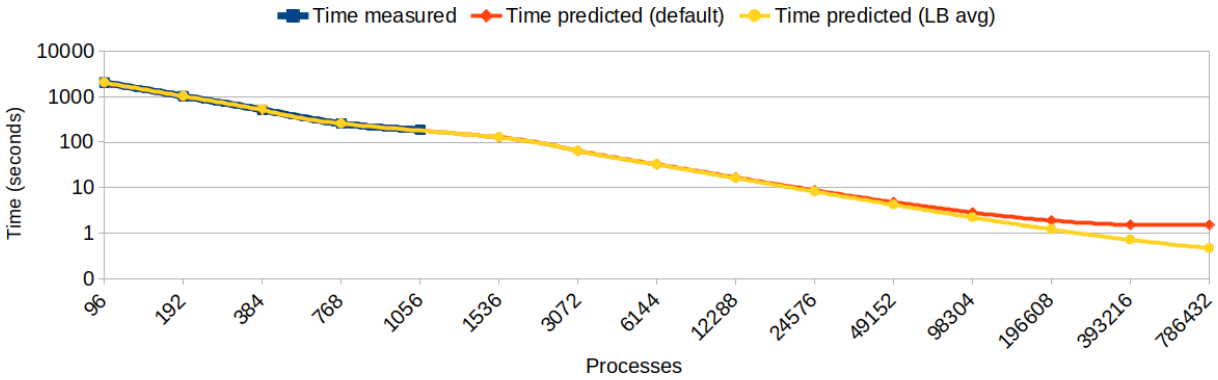
We run the extrapolation model considering two possible evolutions for the scaling of the application, the first hypothesis considers *Load balance* will keep degrading slowly, continuing the trend observed in Table 16. This supposition, plotted in Figure 44a, results in the application being able to scale three orders of magnitude from the baseline, up to ≈ 100000 processes, with high efficiencies (above 70%).



(a) Extrapolation with 5 points (runs from 96 to 1056 cores) with default fit



(b) Extrapolation with 5 points (runs from 96 to 1056 cores) with constant LB



(c) Validation with 5 points (runs up to 1056 processes, fitting error $< 0.2\%$)

Figure 44.: Extrapolation model for NextDBSCAN

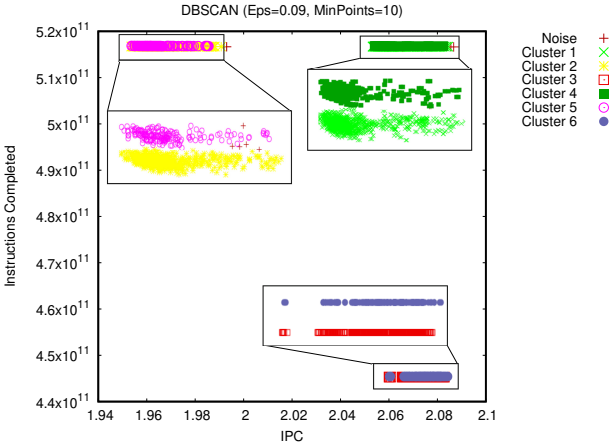
However, with the minimum variability that *Load balance* presents at the measured scales ($< 0.5\%$), this assumption might be too pessimistic, therefore we consider a second scenario supposing *Load balance* will not degrade any further with the scale. This hypothesis, plotted in Figure 44b, results in the application scaling up to ≈ 100000 processes with even higher efficiencies (above 90%), and reaching almost 500000 processes with efficiencies higher than 65%.

Both models validate with an error $< 0.2\%$ (Figure 44c) with the available data, but to confirm the real evolution the application will exhibit would require to obtain at least one more measurement at larger scale. Nevertheless, these two projections, that diverge no more than 2 seconds, can be seen as lower and upper boundaries for the actual trend, which will most likely fall somewhere in the middle.

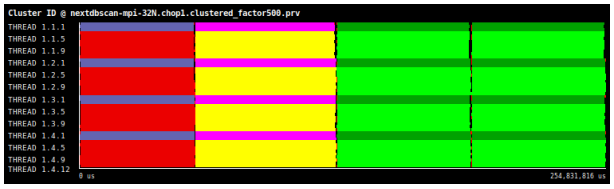
Potential improvements towards the exascale

With the projected efficiencies in the large scale in the range of $70 - 90\%$ after increasing the number of processes by a factor of $1000\times$ (100k processes), and also bearing in mind the application was run in strong scaling mode, which means the ratio of computations/communications will keep inevitably decreasing and the *Communication efficiency* with it, we can consider that the application is already scaling very well and is ready for the exascale.

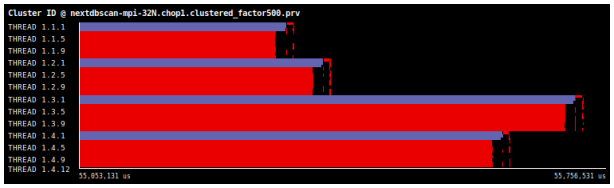
Nevertheless, we analyze the traces to understand the main limiting factors highlighted by the models, and suggest potential improvements. The plots in Figures 45a and 45d show a cluster analysis [53] of the computing regions (running phases) in the instrumented traces for 32 and 44 nodes. Computations are grouped according to the amount of instructions executed (y-axis) and at which speed (x-axis).



(a) 3 main behaviors and 3 deviated trends showing more instructions and IPC variability

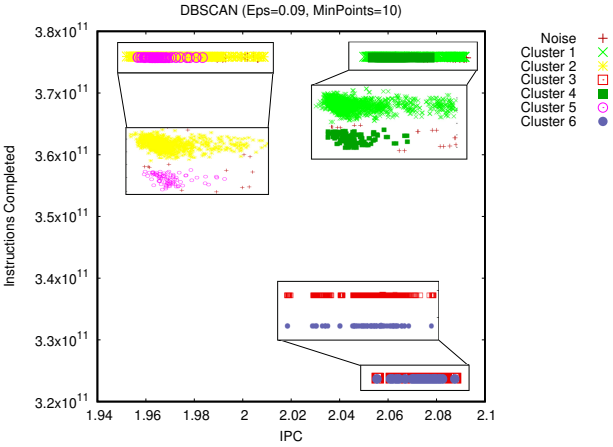


(b) Distribution of clusters over time showing 3 threads doing more work

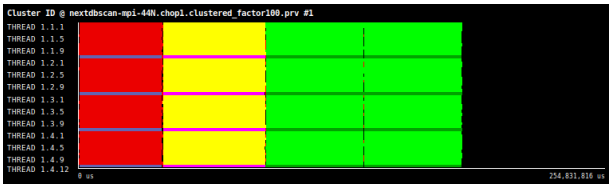


(c) Zoom at the end of a computation region showing a detail of work imbalance

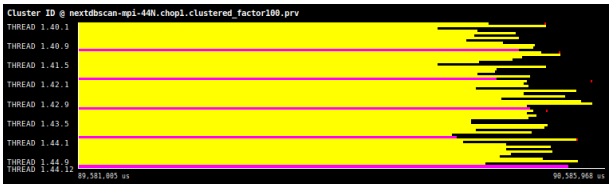
(32-nodes)



(d) 3 main behaviors and 3 deviated trends showing less instructions and IPC variability



(e) Distribution of clusters over time showing 1 thread doing less work



(f) Zoom at the end of a computation region showing a detail of IPC variability

(44-nodes)

Figure 45.: Clustering analysis for NextDBSCAN showing the structure of computations

The application exhibits 3 main behaviors (clusters 1 to 3), that differ by $\pm 18\%$ on instructions and by $\pm 8\%$ on IPC. Cluster 1 (top-right), represents computations that do a high amount of work at high speed. Similarly, cluster 3 (bottom-right), represents phases with lower amount of work at the same speed. The less efficient computing phases are those represented by cluster 2 (top-left), with the same amount of work as cluster 1, but lower speed. More interestingly, the clusters present an elongated shape in the x-axis, meaning that all computations in the program present IPC variability. A closer inspection of these 3 behaviors, reveals that there are actually two trends in the amount of instructions (clusters 4 to 6). The distribution over time in Figures 45b and 45e shows multiple clusters executing simultaneously, hinting at potential imbalances due to variable characteristics on different threads.

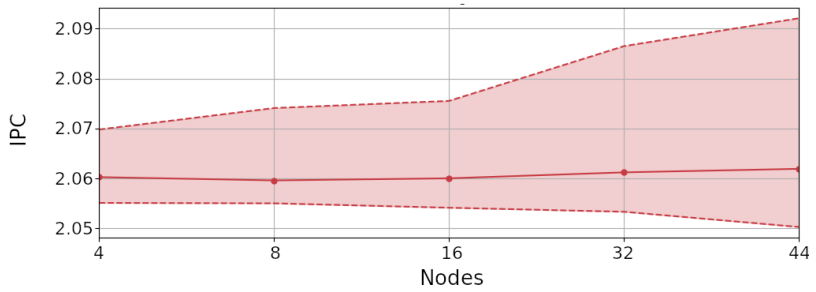
The instructions variability is structured, as the threads that execute more instructions are always the same in all computing phases, but they vary depending on the scale. For instance, in the execution with 32 nodes, the first 3 threads of every process have more work, resulting in cluster 6 systematically finishing later than cluster 3, as seen in Figure 45c. Contrarily, in 44 nodes the last thread has less work than all others. This change in work balancing suggests the load distribution algorithm is not able to split the problem into more equitative chunks.

However, with variabilities both in instructions and IPC, we also see cases where threads with less work are one of the last to finish because they achieved lower IPC, as shown in Figure 45f, where some instances of cluster 5 take longer than cluster 2. Both effects combined result in the inflated collectives in the communication phases initially detected (Figures 43b and 43c), and contribute to the slow degradation of the *Parallel efficiency*.

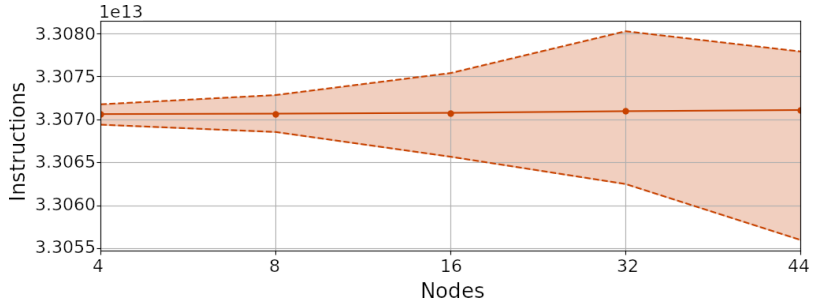
The variabilities are particularly meaningful because they grow with the scale. This is shown in Figure 46 that depicts the average, minimum and maximum IPC and instructions for cluster 1 computations, where it is clearly seen that the distance between the extremes keeps growing. Although the dispersion will not keep increasing forever, this evolution suggests that the variabilities will have a higher impact in larger scales before stabilizing.

The increase in variability affects the whole program as shown in Figure 47. This plot is the result of a tracking analysis [54] and shows how the clusters evolve with scale. Clusters are represented by their centroids (circles) and their perimeter (rectangles). The trajectory the clusters follow from top to bottom, represents the increments in the scale. The movement downwards (instructions y-axis), indicates that the number of instructions per process is reduced proportionally and scales perfectly. The fact that perimeters get wider indicates that the variability grows, and all parts of the program show the same behavior.

The next step would consist in analyzing the source of the variability in IPC, both within clusters, to achieve better balanced computations at higher scales; and between clusters, ideally to accelerate cluster 2 to the same level as clusters 1 and 3, as this easily represents one quarter of the total execution time and even a modest increase in speed would have a positive effect. It is important to ensure that the number of instructions does not increase on higher node counts due to the labels convergence algorithm, as this results in a whole phase replicated which drastically reduces the *Computation scalability* efficiency. As long as it is not affected by this instruction increase, this code exhibits close to perfect scaling up to $1000\times$.



(a) Minimum, average and maximum IPC per scale



(b) Minimum, average and maximum instructions per scale

Figure 46.: NextDBSCAN Cluster 1 computation variability increasing with the scale

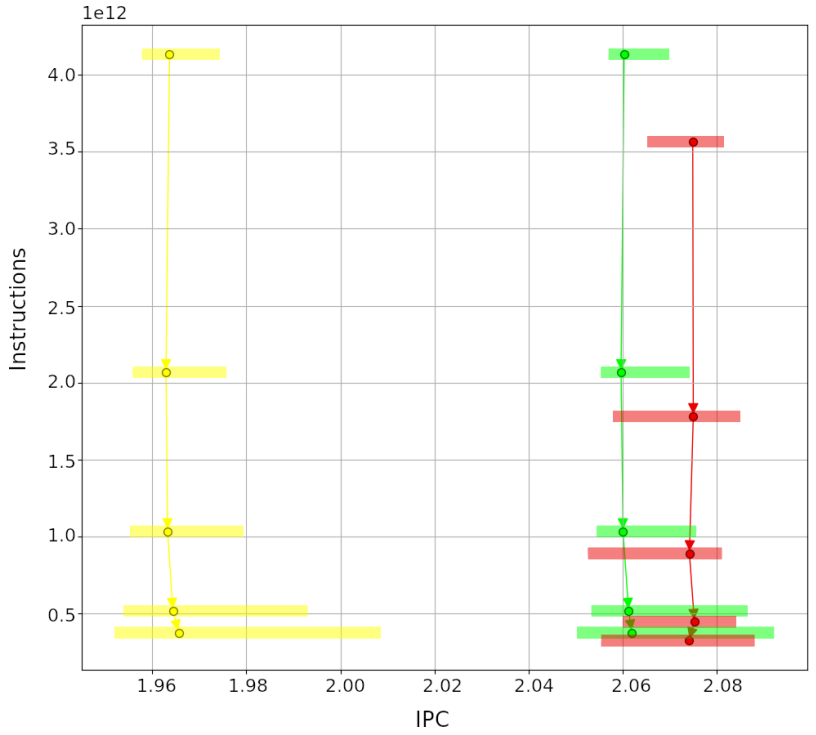


Figure 47.: Tracking analysis for NextDBSCAN showing the evolution of all computing regions from 4-nodes (top) to 44-nodes (bottom) with increasing variability

4.7. EC-EARTH

Albeit not being part of the DEEP-EST project, the EC-EARTH model presents as an excellent candidate to study the scalability of the modular prototype. EC-EARTH is a global, coupled climate model that consists of the separate components IFS for the atmosphere and NEMO for the ocean that are coupled and can be used with a number of optional modules.

This structure fits perfectly with the design philosophy of the DEEP-EST system, enabling a very flexible configuration where application modules with different compute requirements can employ the most suitable system modules to maximise performance. Moreover, this study demonstrates how external applications can benefit from the DEEP-EST modular architecture.

For the analysis the user suggested a use case running 4 pure MPI binaries, comprising the NEMO ocean simulator and the IFS atmosphere simulator, coupled through the Runoff mapper component and using XIOS I/O servers. Due to limitations during the deployment of the prototype at the time of the experiment, all binaries were run homogeneously on CM nodes, although it would have been interesting to place IFS on DAM nodes due to lower computation but higher memory footprint requirements. Instead, we circumvent this limitation by simulating the execution of IFS on DAM to study the potential benefits.

The details of the experiment follow:

- NEMO run with ORCA1L75_LIM3 input, scaling from 24 to 384 processes
- IFS run with T256L91 input, scaling from 24 to 576 processes
- XIOS I/O servers run with 24 processes on a dedicated node
- RNF run with 1 process on a dedicated node
- Coupling frequency set to 2,700 seconds and simulating 12 hours from 1990-01-01T0000

The user informed that with the selected configuration the application runs in strong scaling mode, up to a maximum of few thousand processes distributed either in a ratio of 1 NEMO to 1 IFS for small use cases, or in 2 NEMO to 3 IFS for large use cases. We took into account this insight into our study and computed the efficiency model for both process ratios.

The efficiency models for the series of experiments with ratio 1:1 and ratio 2:3 presented in Tables 18 and 19 report low *Parallel efficiency* values, mostly affected by *Load balance*. This gets partially compensated by *Computation scalability* efficiencies over 100%, caused by an over-proportional reduction of the total executed instructions when the scale is increased. For an easier comparison, Figure 48 plots the *Parallel efficiency* components, showing better *Load balance* with ratio 2:3 in small scale, which does not result in a better *Parallel efficiency* than in ratio 1:1 due to the effect of communications.

For the in-depth analysis we selected the use case with ratio 1:1 and 313 processes (144 for both NEMO and IFS), as it is the case where *Parallel efficiency* first drops below 50%.

	73	121	165	213	313
Global efficiency	60.78	68.45	61.06	63.83	54.75
-- Parallel efficiency	60.78	63.88	55.13	56.65	49.95
-- Load Balance	64.77	70.78	74.76	73.75	63.03
-- Communication efficiency	93.84	90.25	73.75	76.81	79.24
-- Serialization efficiency	96.65	95.69	85.14	93.35	98.72
-- Transfer efficiency	97.09	94.32	86.62	82.29	80.27
-- Computation scalability	100.00	107.16	110.76	112.69	109.61
-- IPC scalability	100.00	97.28	96.70	97.57	97.44
-- Instruction scalability	100.00	110.20	114.92	117.25	115.73
-- Frequency scalability	100.00	99.96	99.67	98.50	97.20

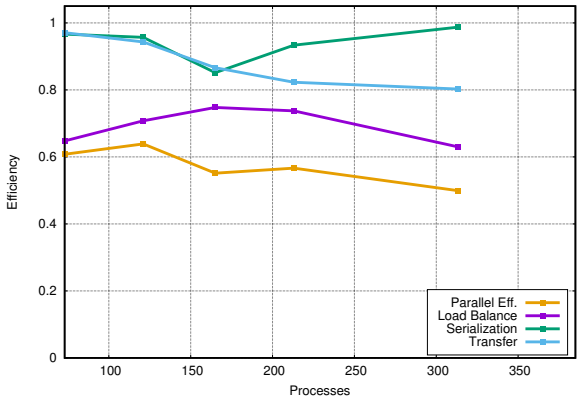
Percentage (%): 100, 80, 60, 40, 20, 0

Table 18.: Efficiency model for EC-EARTH (ratio 1:1)

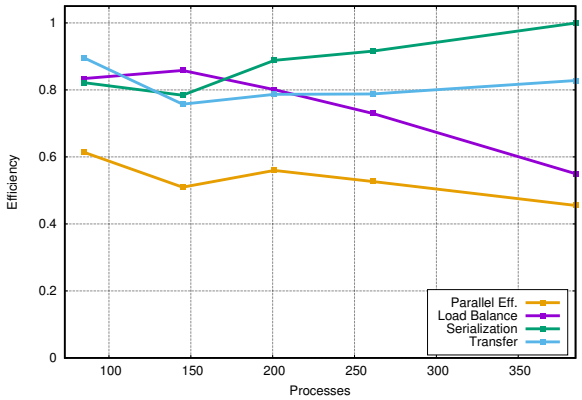
	85	145	201	261	385
Global efficiency	61.34	53.83	59.36	55.67	46.35
-- Parallel efficiency	61.34	50.96	55.96	52.66	45.51
-- Load Balance	83.40	85.81	80.09	72.99	54.99
-- Communication efficiency	73.54	59.39	69.89	72.14	82.76
-- Serialization efficiency	82.15	78.41	88.82	91.56	99.95
-- Transfer efficiency	89.52	75.74	78.67	78.79	82.80
-- Computation scalability	100.00	105.63	106.08	105.73	101.85
-- IPC scalability	100.00	95.39	96.24	95.18	95.55
-- Instruction scalability	100.00	111.10	111.70	114.10	112.05
-- Frequency scalability	100.00	99.67	98.68	97.35	95.14

Percentage (%): 100, 80, 60, 40, 20, 0

Table 19.: Efficiency model for EC-EARTH (ratio 2:3)



(a) Ratio 1:1, runs from 73 to 313 processes



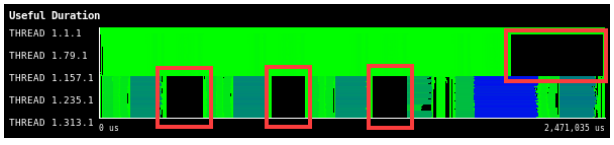
(b) Ratio 2:3, runs from 85 to 385 processes

Figure 48.: Comparison of measured *Parallel efficiency* factors

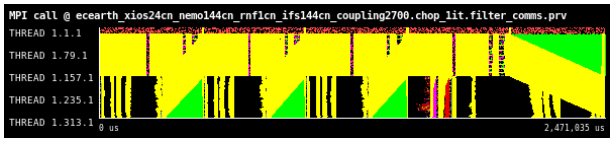
Per-component efficiency analysis

To understand the source of the low *Load balance* values it is essential to discern whether they originate from problems regarding the coupling of the application modules, or they come from imbalance issues affecting one, or more, of the components internally. To this end the efficiency model can be calculated not only globally (considering all components together), but also to each of the components individually. Table 49c shows the per-binary decomposition of the model efficiencies. While global *Load balance* is low (63%), the individual values reflect very good $(89-96\%)$ internal balancing. This indicates that the low global *Load balance* comes not from any particular component, but from the global coupling between them. The Runoff mapper stalls waiting for NEMO and IFS to synchronize in every step thus, the very low *Parallel efficiency* in this module is actually positive, meaning that it needs to do very little work once synchronized and does not hinder NEMO and IFS from progressing. However, we can observe that both IFS, and specially NEMO, present low *Communication efficiency* at the internal level.

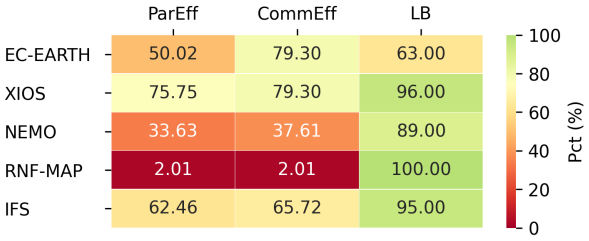
Focusing on these two compute intensive modules, timelines in Figure 49 show the structure of one step of the execution. Figure 49a shows the computation phases for the 4 binaries, with NEMO having a large black gap at the end of the step, and IFS having 3 black gaps throughout the iteration (red boxes). Black gaps correspond to MPI waiting times generated by one binary finishing its computations before the other, as can be seen in Figure 49b that shows the communication pattern where communication lines between NEMO and IFS are pictured in yellow.



(a) Structure of computations



(b) MPI communications



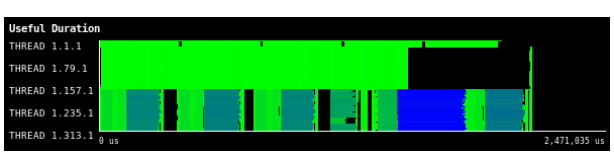
(c) Decomposed efficiencies per component

Figure 49.: One iteration of EC-EARTH run with 144 NEMO processes and 144 IFS processes

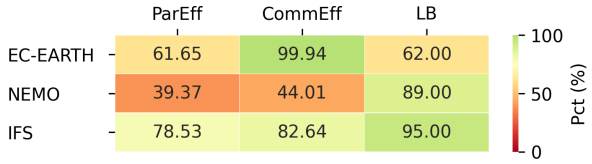
Given that IFS has large waiting times for NEMO, it follows that, if computations on IFS were run on a slower CPU, it would reach the communication phase later and thus, waiting times would be reduced. Additionally, we would benefit from the potential reduction in power consumption from using a slower, more power efficient CPU. This scenario fits in the DEEP-EST architecture as the DAM CPU is slower than the one in the CM. We used Dimemas to simulate this configuration, which in turn also affects communications from NEMO to IFS as we move from CM-to-CM, to CM-to-DAM communications with reduced latencies (refer to Table 8).

The simulation results in reduced communication times thanks to the DAM network (50a), as well as a decrease in the total execution time by roughly a 14%. IFS waiting times (black gaps) are also reduced as expected. Consequently, *Communication efficiency* improves both for NEMO and IFS as shown in Table 51b, and impacts positively in *Parallel efficiency*.

However, NEMO waiting time at the end of its computation phase is still present and even aggravated due to the slower CPU running IFS. This suggests to accelerate key kernels such as the long computing region colored in dark blue at the end of the IFS execution. The simulation of this scenario with Dimemas reports reduced total execution time by an aggregated 19% (Figure 51a) as well as better *Parallel efficiency* and *Communication efficiency* for NEMO (51b]). Albeit NEMO still waits for IFS, accelerating its long computations further would not result in any benefit as the execution is now bound by XIOS.

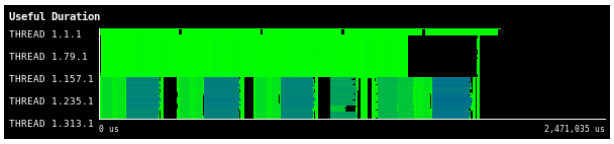


(a) Computation structure

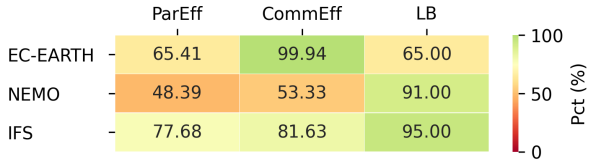


(b) Decomposed efficiencies

Figure 50.: One iteration of EC-EARTH simulating IFS running on DAM



(a) Computation structure



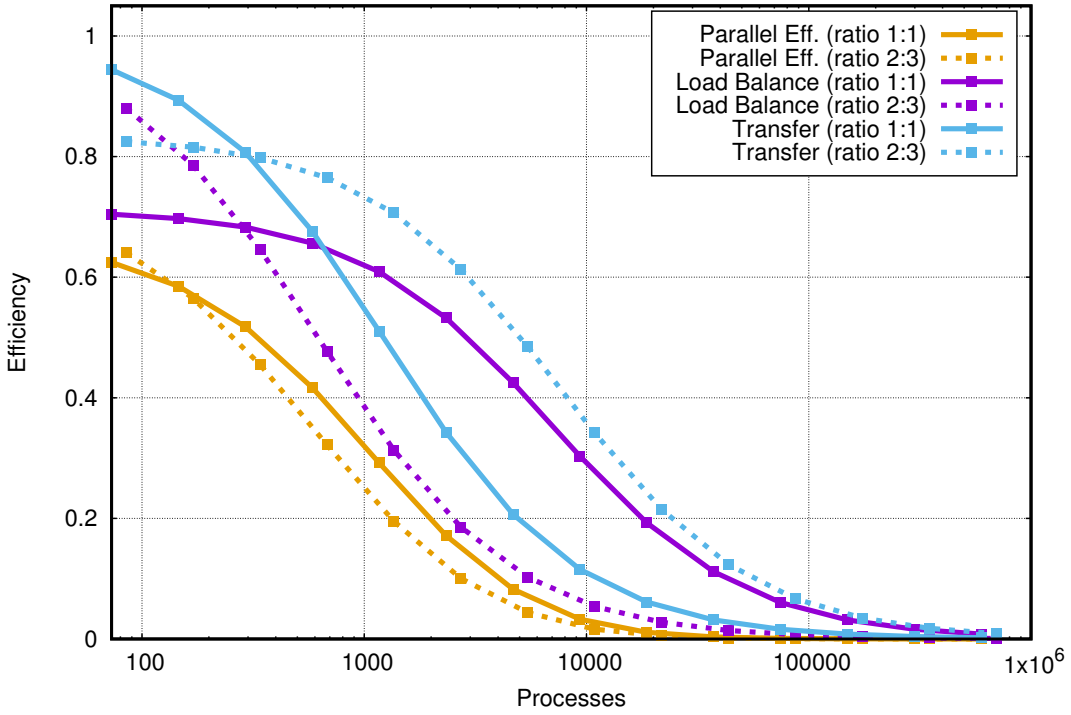
(b) Decomposed efficiencies

Figure 51.: One iteration of EC-EARTH simulating a key computing kernel accelerated

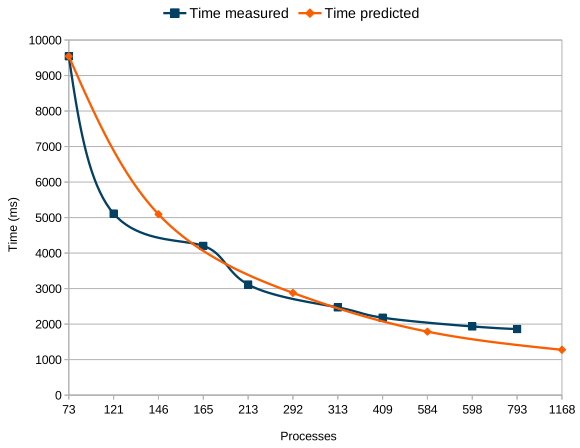
Extrapolation analysis

One concern of the users regarding global *Load balance* is related to the fact that the ratio between NEMO and IFS processes may affect the results and hinder scalability. To study this we ran the extrapolation analysis considering both NEMO:IFS ratios (1:1 and 2:3).

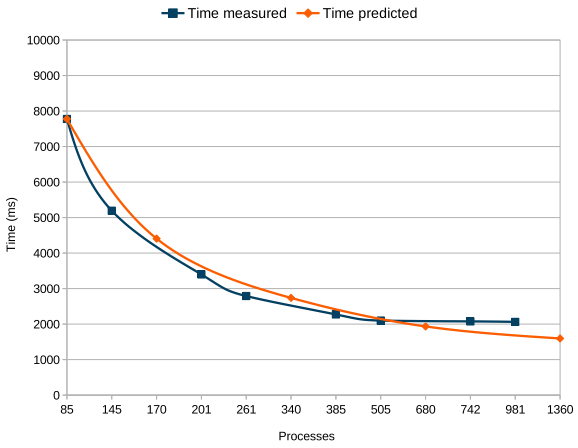
The extrapolation was computed with 5 points, up to 313 processes for the 1:1 ratio, and up to 385 for the 2:3 ratio; and validated with 8 points, up to 1168 / 1360 processes respectively and are shown in Figure 52. The projected trends for the relevant metrics are depicted in Figure 52a in which we can observe changes in behavior starting at a few hundred processes, where the most limiting factor to the *Parallel efficiency* alternates between *Load balance* and *Transfer*, depending both on the scale and the NEMO:IFS ratio. Overall, *Load balance* and *Transfer* compensate each other and the resulting *Parallel efficiency* for the two different ratios is very similar, although ratio 2:3 achieves marginally better *Parallel efficiency* at small scales, while ratio 1:1 does so at larger scales, enabling to scale efficiently slightly further.



(a) Extrapolation with 5 points (from 73-313 processes for ratio 1:1, and 85-385 processes for ratio 2:3)



(b) Ratio 1:1 validation with 8 points (runs up to 793 processes)



(c) Ratio 2:3 validation with 8 points (runs up to 981 processes)

Figure 52.: Extrapolation model for EC-EARTH comparing *Parallel efficiency* and *Load balance* between 2 different ratios of NEMO:IFS

4.8. Conclusions

In the context of performance models, Task 2.3 has contributed with the analysis of the project's applications based on the BSC efficiency model, also extending its applicability to hybrid and modular codes as the main target of the DEEP-EST architecture. The model evaluates whether parallel applications make an optimal use of the resources, characterizing the performance using multiplicative fundamental factors. These reflect the main sources of inefficiencies that hinder an application's scalability stemming from the use of a parallel runtime, namely load balance, data transfer and serialization issues. It also considers those derived from the scaling of the computations, such as variabilities in the amount of work, or the speed at which the work is done. These factors are expressed in values between $0 - 100\%$, where 80% is usually the boundary considered for good efficiency, and are measured from traces of real executions.

With the objective of gaining insight of the application's performance in the exascale, the proposed methodology extends the analysis of the efficiency factors with extrapolation models. The extrapolation technique consists in projecting to the large scale the evolution of the efficiency factors, based on the trends exhibited at small scale. To accurately predict the behavior at several orders of magnitude higher than the measurements, it is important to validate the model with non-instrumented runs, ideally halfway to the target scale. However, due to the size of the DEEP-EST prototype, we have been limited to take measurements in the range of few hundreds, and conduct validations up to few thousand processes only, which is still very far from the exascale. For this reason, we complemented the extrapolation models with in-depth analyses of the traces to better understand the model's projections and corroborate that the observations for the large scale are reasonable.

Another important contribution has been to hint application developers with potential improvements to overcome the inefficiencies that would hinder performance at larger scales. To this end, we characterized the DEEP-EST architecture and simulated scenarios that estimate the benefits of some of the suggested changes, mostly targeted at removing imbalances, reducing dependency chains and accelerating key kernels. These simulations do not consider the complexity of implementing the necessary changes, but rather provide quick estimates and overall trends to assess the potential for improvement beforehand.

Well aligned with the project's objectives, NEST+Arbor is the main application we had access to that clearly presents a modular structure. While other codes have implemented their own modular versions during the project, these were not considered as they were still undergoing major development at the time of analysis. To gain a more solid understanding of the advantages and complexities of the modular architecture, we ported to the prototype the EC-EARTH climate model to have another good example of modular code, demonstrating the benefits of the DEEP-EST architecture for external applications.

From the study of modular applications, a lesson learned is that combining modules with very different structures makes it difficult to achieve a good *Load balance* between them. As each module usually operates on very different workloads, we mostly observe that balancing is not so equitative, and typically one module does more useful work than the others, generating dependencies between them. Moreover, these become critical when a single module experiences any kind of delay as it gets propagated to all processes of the dependent modules. This effect could be seen in Arbor, where a serialization in a computing region misaligns the communica-

tion phases with NEST, which suffers a side effect of inflated stalls that can only be fixed by solving the original issue, for a reduction of the total iteration time of up to $\approx 33\%$. This is a good example of how an issue on a single module can impact other's efficiencies, and eliminating the problem on one side benefits all parties involved.

These effects are likely to be magnified in applications composed of even more modules, for example in EC-EARTH the main computing modules present circular dependencies, and they rely in turn on the I/O server, escalating the common problem of serializations between individual processes, to a bigger problem of serializations between whole modules. To this extent, the flexibility of the DEEP-EST modular architecture can help to mitigate these dependencies by redistributing the application's components across the different system modules according to their computational needs. A proper placement of EC-EARTH components, that seeks to map the more compute intensive processes to faster CPUs, was able to reduce the execution time by 19%, and increase efficiency from 50.02% to 65.41%. Furthermore, the ability to run key kernels on clusters with GPUs proved accelerators to be a powerful tool not only to speed up computations and improve time to solution, but also to achieve better balance between modules by reducing stalls in the critical path. The nature of modular codes aligns very well with the DEEP-EST architecture, providing high flexibility to adapt the executions to the specific needs of the codes.

GROMACS does not strictly fall within the modular category even though it is an MPMD application combining two types of processes, because they are highly coupled and communicate very frequently. For this case, we studied the impact of different process balancing configurations, and the analysis showed that fine-tuning the process ratios can improve *Parallel efficiency* up to 9 percentage points for the $32\times$ case, from 47% to 56%. However, we also identified that it is precisely because the frequent coupling that the application is very sensitive to small delays in the computations due to small IPC variations that most likely respond to system noise, which hugely impact the waiting times in the MPI phases as the communication pattern is highly connected and any single delayed partner affects many others.

The communication pattern plays an important role in propagating or absorbing this kind of variabilities that are often dynamic and reported as *Serialization* issues. Both in the pure-MPI and the hybrid MPI+OpenMP versions of xPIC, once again we identified sporadic delays in the computations that snowball through a chained communication pattern, where every process needs to wait for their predecessor before sending to the next partner, thus the later in the chain, the higher waiting times. This problem can be mitigated either by improving the balance of the computations, or by redesigning the communication pattern. For the former we estimated a benefit up to 53 percentage points, from 41.45% to 94.81% in *Parallel efficiency* at one particular phase that is heavily dominated by communications. However, this approach might not be easy depending on the source of the imbalances. It is an interesting reflection that these effects of punctual delayed computations due to slightly reduced IPC, cycles or frequency will grow in the exascale. The increasing use of technologies like CPU power saving features, dynamic frequency scaling, multithreading and resource sharing; the increasing use of system and monitoring daemons in HPC centers; and even the random hardware failures in increasingly larger clusters; are all examples that can result in one or few processes underperforming. If these effects are unavoidable and difficult to control by the users, then is important to mitigate them and prevent that they impact many other processes. To this extend, changing the structure of communications can break the dependency chains, or at least minimize their

repercussion. Design patterns like overdecomposition, non-blocking communications or speculation [51], aiming at increasing asynchronicity, can be useful to absorb the accidental delays that will be more harmful on exascale platforms.

Incorporating thread-level parallelism into MPI applications is a natural approach to accelerate regions bound by long computations, but it is important to remark that faster does not necessarily imply more efficient. We find an example of hybrid MPI+OpenMP in xPIC, where the addition of OpenMP leaves a serial region unparallelized at the thread level that covers 17% of the iteration time. Even if the number of threads available per node is small, the large number of nodes that would be required for exascale systems will result in a very high number of resources wasted.

The design trend towards exascale platforms mostly consists in machines with a massively higher number of nodes than cores within a node. For instance, the Fugaku pre-exascale system, current leader in Top500 [52], is built with 48 cores per node, but more than 150 thousand nodes, a difference of four orders of magnitude. As the typical process mapping will place 1 or 2 MPI ranks per node and fill the rest of the cores with threads, the prevalence of high node counts reduces to some extent the impact of thread-level parallelism performance, and puts the spotlight on having efficient communication between nodes, guiding where optimization efforts should be mainly directed.

In our experiments the *Transfer efficiency* attained most significance on larger scales. On the analyzed weak-scaling scenarios, where the workload per process remains constant, the number of communications and even the volume of transferred data increases proportionally with the scale. Thus, a low *Transfer* usually becomes an indicator of at which point the application may start experiencing network congestion issues. On strong-scaling scenarios, the proportion of computation per process keeps decreasing with the scale, eventually becoming negligible and communications dominate the execution. A low *Transfer* in this case usually highlights at which point we should stop scaling, or quite the opposite, increase the problem size to make good use of additional resouces. Generally speaking, keeping a high ratio of computations over communications is a safe approach to scale efficiently so as to downplay the relevance of network performance. We find an example of this in NextDBSCAN, an embarrassingly parallel problem where little communication is needed between parallel tasks, and the small variabilities in the computations that dominate over 99% of the execution time have a negligible impact.

In order to fully exploit the potential of the platform, it is crucial to understand the structure of the applications. To this end, performance analysis tools are very useful to gain detailed insight on the application's behaviour, and the main factors that hinder efficient scaling and how to improve them. Continuous hardware improvements in the architecture, either in the form of faster CPUs, memories or network, will always report benefits, but will not necessarily result in making a more efficient use of the resources. With the experienced gained from the analysis of multiple use cases, we have provided users with numerous hints on what are the most frequent problems they will encounter to run their codes efficiently as we move to increasingly larger platforms. This information will help to make more educated choices on how to design the algorithms, select proper configurations and suitable resources to maximize the efficiency of their experiments, as there is a lot to gain just from tuning the software.

5. Energy modeling

The main idea about obtaining a model consists of trying to reproduce a certain behaviour given a set of variables, that is, it is a question of comparing a behaviour (in our case the real curve of energy consumed by an application) with the result obtained after reconstructing this behaviour with a smaller set of variables. Depending on the way we choose this smaller set of variables, we can obtain a better approximation between the real energy consumption curve and the curve projected by the model.

In the past deliverable D2.2 ([3]) the micro benchmarking to be used, the data acquisition to feed and test the model, the possible variables on which the energy models would be based were introduced and the first theoretical concepts on the methodology of variable reduction on which the mathematical models should be based were defined and finally different sets of variables (and therefore different mathematical models) were presented.

Auweter et al. [30] described a model for energy aware scheduling optimizations for Intel Sandy-Bridge architecture based on a base frequency selected for an entire application. The model implicitely assumes that the energy can always be described as a linear combination of hadware perfomance components. That equation 5.1 is applied for a fixed frequency n and predicts the power consumption for any frequency from the available ones.

$$\frac{PWR(f_n)}{PWR(f_0)} = M_{F,C} * P_{C,1} \tag{5.1}$$

The expression can be rewritten in a matrix multiplication notation as follows:

$$M = \begin{pmatrix} A_1 & B_1 & C_1 & D_1 & E_1 & F_1 & G_1 & H_1 \\ A_2 & B_2 & C_2 & D_2 & E_2 & F_2 & G_2 & H_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_i & B_i & C_i & D_i & E_i & F_i & G_i & H_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_F & B_F & C_F & D_F & E_F n & F_F & G_F & H_F \end{pmatrix}_{F \times C} \quad P = \begin{pmatrix} GIPS(f_0) \\ \frac{1}{GIPS(f_0)} \\ CPI(f_0) \\ \frac{1}{CPI(f_0)} \\ GL3PS(f_0) \\ \frac{GL3PS(f_0)}{GIPS(f_0)} \\ GL2PS(f_0) \\ \frac{GL2PS(f_0)}{GIPS(f_0)} \end{pmatrix}_{C \times 1} \tag{5.2}$$

where,

- f_0 is the nominal/default frequency,
- f_n is the frequency on study,
- F is the number of frequencies (=16 in our current study),
- C is the number of hardware events (=8 in our current study, from A_i until H_i), and
- $M_{F,C}$ is the coefficients matrix result of a linear regression procedure,
- P is the vector of the values of the elected components measured at nominal/default frequency.

The idea behind this model is to run the application at reference frequency measuring the

value of the hardware counters which will be represented under the P vector. This vector P remains constant for the application at reference frequency for the system. The matrix M is machine dependent and defines the contribution of each component to the total energy use regarding the energy consumption. The expression allows us to project the energy consumed at any available frequency of the system from a reference frequency, in which the coefficient were calculated. Of course, the model could fit differently by choosing a different reference frequency, since the software behaviour is also different.

From the writting of the deliverable D2.2 ([3]) until today a lot of effort and progress has been made within the calculation of the energy models proposed for the project in the framework of the DEEP-EST project.

5.1. Data input

The micro benchmark consists of different test cases which are encapsulated between PAPI (Performance Application Programming Interface) library function calls. This implies, that only events during the runtime of the test case in particular are measured. No other processes contributes to the energy consumption of the application.

The PAPI events set, which will be measured for every run of the benchmark, consists of:

1. Computational intensity related events:
    - Cycles: number of cycles elapsed during the execution of the test case.
    - Instructions: number of instructions executed by the test case.
    - Double precision scalar flops: number of flops related to 64 bit registers measured during the run of the test case.
    - Double precision 128-bit flops: number of flops related to 128 bit registers measured during the run of the test case.
    - Double precision 256-bit flops: number of flops related to 256 bit registers measured during the run of the test case.
    - Double precision 512-bit flops: number of flops related to 512 bit registers measured during the run of the test case.
2. Memory usage related events:
    - L1: L1 cache misses count.
    - L2: L2 cache misses count.
    - L3: L3 cache misses count.

The idea is to, following a certain methodology, choose the components of the vector P of the expression shown in 5.2.

In order to compare the energy and time projected by the model, it is also necessary to measure the time and energy (or power) consumed by each test case during its execution:

- Package power/energy: power/energy consumed/used by each of the processor sockets

during the execution of the test case.

- DRAM power/energy: power/energy consumed/used by each of the DDRAM modules during the execution of the test case.
- Runtime: Time elapsed between the beginning and end of the test case.

In total 18 events were measured per test case; each of the measured components can be considered to represent a dimension/axis in space. The union of all these dimensions results in the energy curve used by the application as a function of each of the components. The measured energy of every register is a point in this multidimensional space.

Since there is the possibility that several components are related to each other (e.g. cycles, instructions and frequency, or memory bandwidth, and last level cache misses), we must look for the set of linearly independent dimensions that define the same space, while losing as little information as possible. This is exactly the motivation of the Principal Component Analysis (PCA) methodology.

PCA is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set. Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity.

5.2. PCA methodology

As discussed in deliverable D2.2, for the data adquisition we used a modified version of the APEX-MAP benchmark that generates artificial calculations and memory accesses. The initial idea of the Apex project is the assumption that the performance behaviour of any scientific application can be modelled by a set of specific performance factors. Thus, combining these factors, applications that avoid hardware specific models can be designed to simulate typical application performance. Taking into account that the two most dominant performance factors are memory accesses and computational intensity, this benchmark simulates typical memory access patterns of scientific applications. The final goal of this benchmark is to simulate compute and memory bound applications. The modified version of the APEX-MAP includes different test cases whose behaviour simulates different memory accesses and computational intensity patterns depending on the input parameters of the application.

PCA is one of the most popular multivariate analysis method. The goal of PCA is to summarize the information contained in a multivariate data by reducing the dimensionality of the data without loosing important information. It allows to summarize and to visualize the information in a data set containing individuals/observations described by multiple intercorrelated quantitative variables. Each variable could be considered as a different dimension. Having more than 3 variables in a data set, make it very difficult to visualize them in a multi-dimensional hyperspace.

PCA is used to extract the important information from a multivariate data table and to express this information as a set of few new variables called principal components. These new variables correspond to a linear combination of the originals. The number of principal components is

less than or equal to the number of original variables. The information in a given data set corresponds to the total variation it contains. The goal of PCA is to identify directions (or principal components) along which the variation in the data is maximal.

The dimension reduction is achieved by identifying the principal directions, called principal components, in which the data varies. PCA assumes that the directions with the largest variances are the most "important" (i.e, the most principal). In the end the eigenspaces of M has to be identifyed and then select the ones with the largest eigenvalues.

In the figure below, the PC1 axis is the first principal direction along which the samples show the largest variation. The PC2 axis is the second most important direction and it is orthogonal to the PC1 axis. The dimensionality of the two-dimensional data can be reduced to a single dimension by projecting each sample onto the first principal component (Plot 1B). So the principal components algorithm can be seen as a space change formed by orthogonal axis which are redefined as the ones which have a largest correlation with the original data.

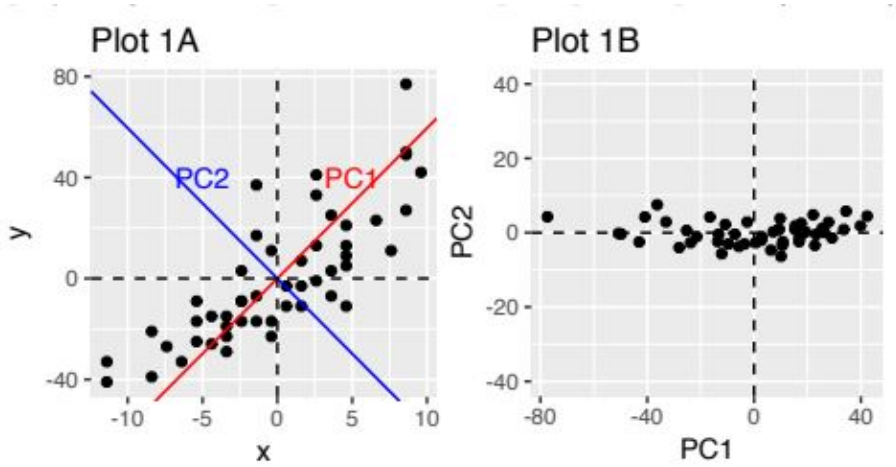


Figure 53.: Representation of a certain information in the original base and its transformation into the new space.

The amount of variance retained by each principal component is measured by eigenvalues of the data. Seeing the data as a matrix in which the observed events and the frequency are related, the eigenvalues determines if certain columns of this matrix are correlated; if the eigenvalue is zero (or close to a low threshold), we can assure that this column (hardware events) is related to the other ones and herefore, the information provided by the hardware event is duplicated. Thus, this column (axis) can be eliminated from the matrix without loosing a lot of information. This is exactly the dimension reduction associated to the PCA methodology. By repeating this process with all axis pairs, the redundant information of the entire data space will be known and therefor the components that can not be eliminated from the model expression.

But, how many components must be removed so that not too much information is lost? That will depend on how correlated the information in the data space is; of course, every dimension provides information to the global system and there is no way to reproduce the original information with 100% accuracy. In order to depict the concept, PCA was applied to face images shown in figure 54. Later, the information of the faces was reconstructed using more or less components. The result can be seen in the image 54.

Figure 54.: Example of PCA methodology applied to face images.

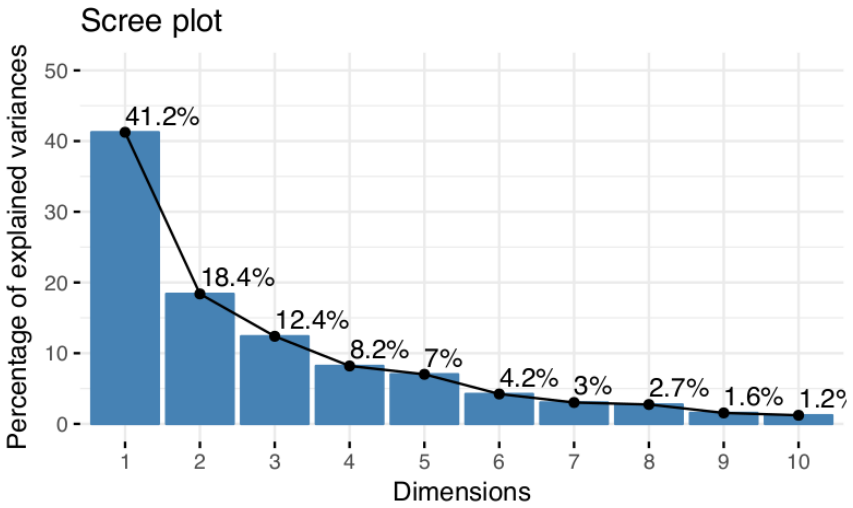


Figure 55.: Contribution of each principal component to the global information explanation.

We can apply the same procedure to the methodology for calculating the energy models. The example depicted in figure 55 shows the contribution of each principal component to the model, which means, the information that can be additionally explained by adding new principal components. This was made for a particular data information collected for the calculation of the energy model but serves as an example that every new component contributes less to the information explanation. In theory, in that particular example, if we want to stay below the 3% error, we should select the first eight principal components, since in that case, the accumulated percentage of information that can be explained reaches 97.1% of the total.

Applying this ideas to the CPU model calculation methodology, after the components reduction, we got that the PCA set should contain these 5 counters:

- Instructions: number of CPU instructions executed.
- L2 cache misses: amount of CPU L2 cache misses.
- L3 cache misses: amount of CPU L3 cache misses.
- Double precision flops: number of double precision floating point operations.
- Vectorization per cycle ratio: ratio of number of double precision floating point operations using 128-bit, 256-bit and 512-bit vectorization per CPU cycle.

Similary, the GPU model will be based on the following counters:

- Instructions: number of GPU instructions executed.
- Cycles: number of GPU cycles performed.
- L1 cache misses: amount of L1 cache misses.
- L2 cache misses: amount of L2 cache misses.
- Bandwidth: memory bandwidth.
- Double precision flops: umber of double precision floating point operations.

5.3. Model validation

As explained in previous deliverables, we used a modified version of the APEX-MAP benchmark, since this microbenchmark includes different test cases whose behaviour generates different memory accesses and computational intensity patterns depending on the input parameters of the application. The complete input set for the modelling procedure corresponds to:

- Five different binaries according to the five different sets of compilation flags (O0, O3+msse4.2, O3+xAVX, O3+AVX2, O3+AVX512). [1]
- All system available frequencies (fifteen or sixteen depending if turbo enabled or not).
- Forty different experiments per test case.
- Eighteen hardware counters and events.

[1] The model calculation and model validation has been performed on intel machines. For non-intel/AMD machines, the correspondent compile flags should be used.

In total there are 6480 records (experiments) each consisting on 25 parameters (compilation flags set, configured fixed frequency, elapsed runtime, input parameters for the patterns simulation and the values of the monitored hardware events). From this amount of experiments, the information of half of them will be used as data input for the model calculation (data group), whereas the other half is reserved to validate the model (test group). The reason for splitting the records into two different groups is that we can not use the same records to calculate the coefficients of the model and to test its behaviour since, in that case, the real energy consumed by the experiment and the energy projected will be the same; having a lot of experiments in the data group implies having less opportunities to check the accuracy of the model and, the other way round, having less experiments to calculate the model will lead into an uncompleted model.

The records belonging to each group are selected arbitrary: the records are firstly shuffled and afterwards splitted in two groups.

The coefficient matrix of the energy model is calculated with the records belonging to the data group using the PCA methodology explained above. This matrix will be used to project the energy of each of the registers belonging to the test group.

The validation of the model consists in the comparison of the real energy consumed by the application with the energy projected by the model; both energy values, the projected and the consumed one, can be found in the output of the experiment. In order to give this comparison a quantitative value and so that different models can be compared, the Root Mean Square Error (RMSE) expression has been used. RMSE is a standard way to measure the error of a model in predicting quantitative data. Formally it is defined as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (\hat{y_i} - y_i)^2}{n}} \tag{5.3}$$

where $\hat{y_i}$ represents the real energy consumption values; y_i represents the projected energy consumption values and n is the number of experiments performed.

A normalized percentage can be therefor calculated as follows:

$$RMSE(\%) = 100 * \frac{\sqrt{\frac{\sum_{i=1}^{n} (\hat{y_i} - y_i)^2}{n}}}{max(\hat{y}, y)} \tag{5.4}$$

5.3.1. Microbenchmark results

In order to check if a certain energy model fits accurately enough the energy values, we will compare the real energy consumed by the application with the energy projected for the same run. That means, the value obtained by the multiplication between the energy coefficient matrix and the events values, as described in the formula 5.2. We repeated the process for each of the modules and compared the RMSE values obtained.

We firstly tested the SuperMUC-NG compute nodes which are very similar with to the ones of the CM module. The microbenchmark was run full node (48 threads) at a reference frequency of 2.3GHz. The results are depicted at figure 56. In the figure the experiments are ordered

by energy consumption. The black line depicts the energy measured while runing each of the application belonging to the test group; the red line represents the predicted energy by the model. The RMSE value for this certain frequency is 2.26% which will be aproximately 85J.

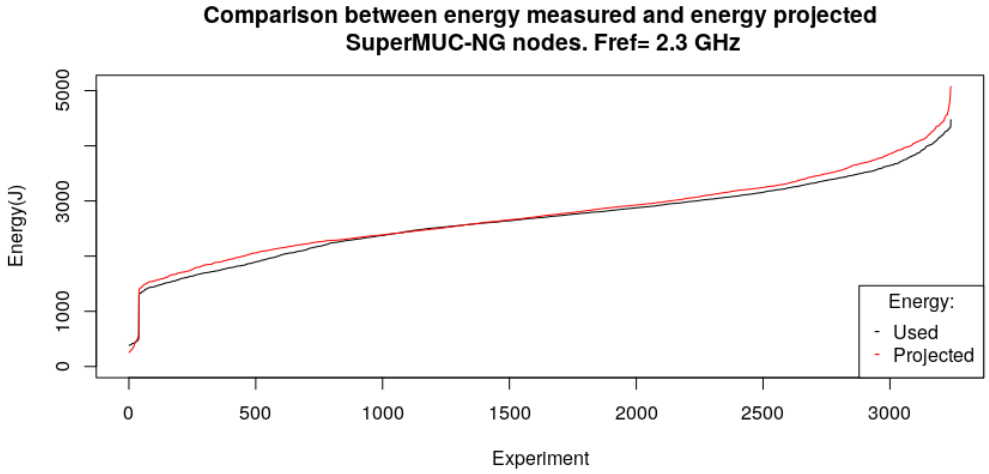


Figure 56.: Energy model for the SuperMUC-NG compute nodes using a reference frequency of 2.3GHz.

We can see a similar behaviour with the CM nodes. In this case, as in the previous one, the microbenchmark was run with 48 threads and a reference frequency of 2.3 GHz. The results can be seen in figure 57. The black line depicts the energy measured while runing each of the application belonging to the test group; the red line represents the predicted energy by the model. The RMSE value in this particular case is 1.39% which is aproximately 49J.

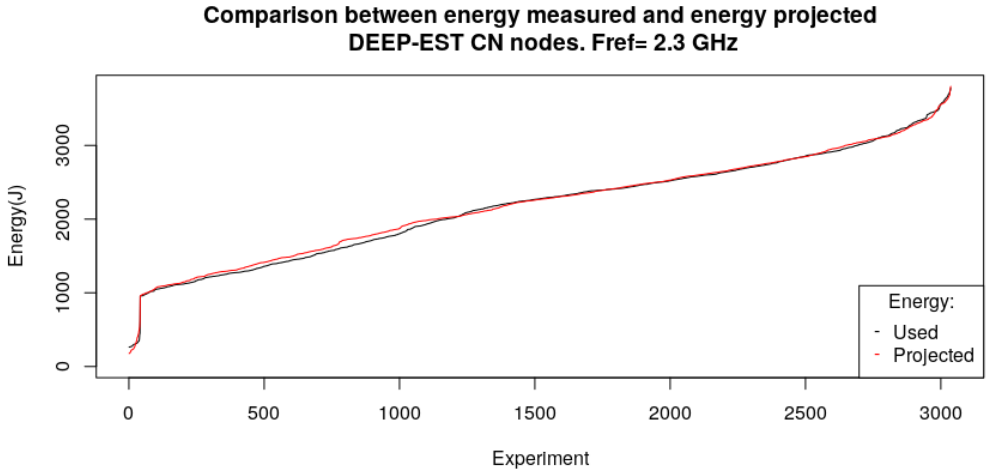


Figure 57.: Energy model for the CM nodes using a reference frequency of 2.3GHz.

Regarding the DAM nodes, the microbenchmark was executed in full node with 96 threads at a reference frequency of 2.3GHz. The results are depicted in figure 58. The black line depicts

the energy measured while runing each of the application belonging to the test group; the red line represents the predicted energy by the model. The RMSE error in this case is, for that particular case, 1.70% which implies a cuadratic error of aproximately 82J.

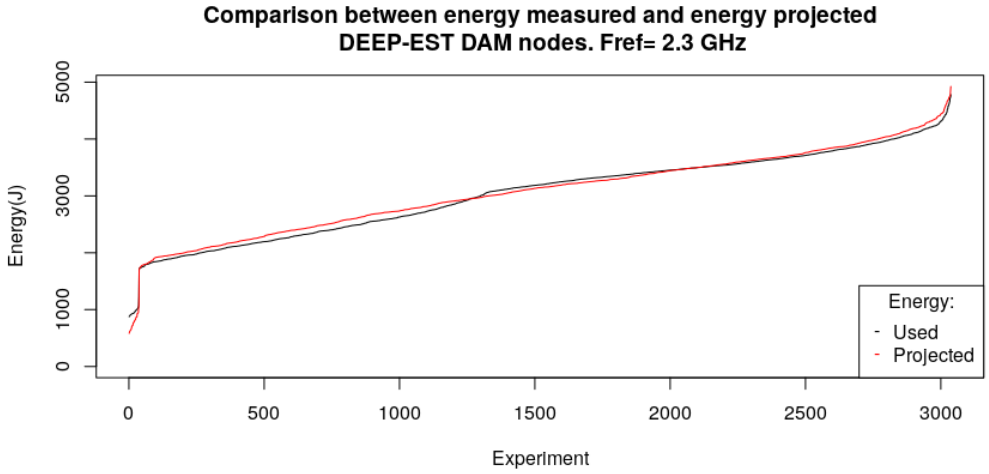


Figure 58.: Energy model for the DAM nodes. The reference frequency is 2.3GHz.

The last model calculated belongs to the ESB nodes depicted in figure 59. In that case, the microbenchmark runs were performed at 1.9GHz in full node with 16 threads, which implies a cuadratic error or aproximately 5J.

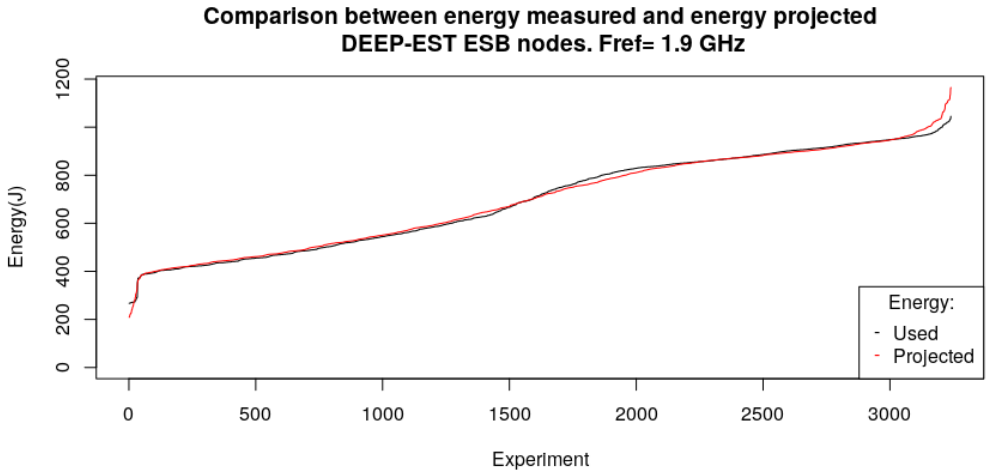


Figure 59.: Energy model for the ESB nodes. The reference frequency is 1.9GHz.

In the same way that we calculated the models for the CPUs, we also calculated the energy model for the GPU accelerators. As a clarification, the models mentioned for the DAM and ESB nodes exclude GPU energy. We ran the microbenchmark and of the output registers obtained, we reserved half for the calculation of the coefficient matrix and the other half to check the fit of the calculated model. In this case, as we saw earlier, the model is very similar. GPUs have a

completely different frequency range and therefore the adjustment of the reference frequency selection will be more complex. The result of the GPU power consumption model can be seen in the figure 60. There the black line depicts the energy measured while runing each of the application belonging to the test group; the red line represents the predicted energy by the model.

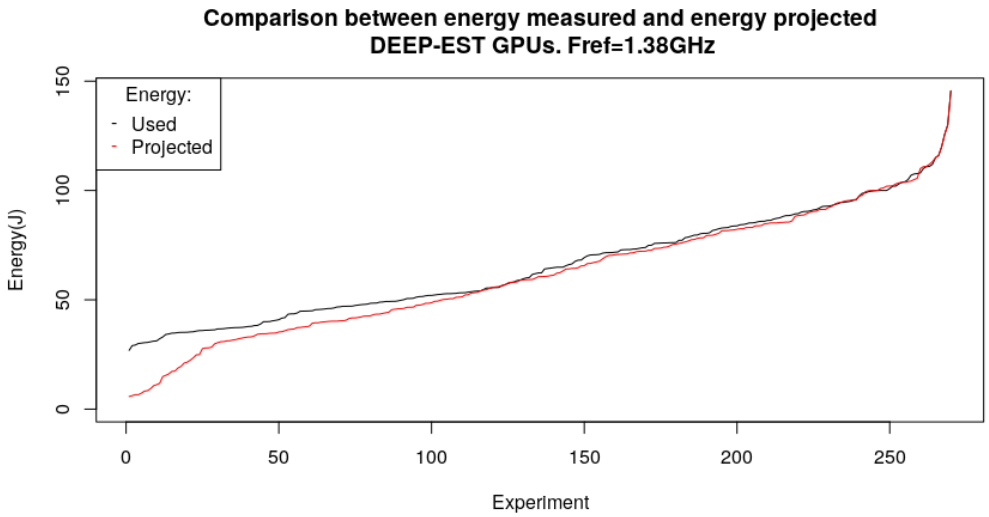


Figure 60.: Energy model for the GPU accelerators. The reference frequency is 1.38GHz

The behaviour of an application, the value of hardware counters such as cycles, instructions or flops is highly dependent on the frequency at which that application is executed. For this reason, before being able to define the frequency at which the energy consumption by a certain application (in our case microbenchmark) in a certain cluster is optimal, it will be necessary to know the RMSE when projecting the energy consumed from any frequency of the system to any other.

For the visualisation of this RMSE distribution per cluster partition, we have prepared two different graphs:

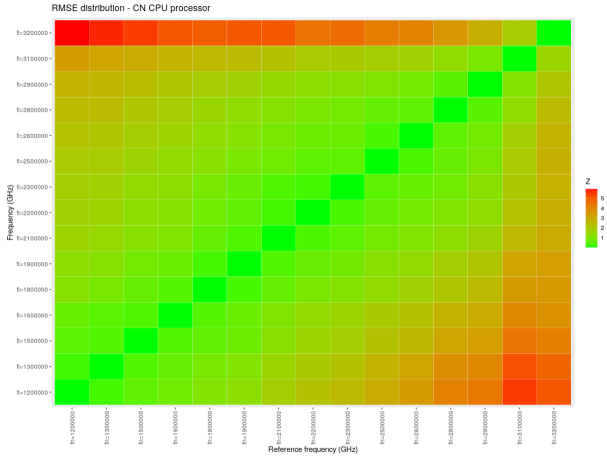
- The first one, a heatmap type plot, represents the RMSE error when projecting the energy from any frequency to any other frequency available in the cluster. Red cells indicate a higher error while as the cells change colour towards green, the error decreases. The diagonal of such a matrix shall always be zero, since the projected and measured energy shall be the same. For these plots, the X-axis depicts the reference frequencies and the Y-axis, the frequency to which the projection has to be calculated.
- The second graph represents the average of the errors of projecting the energy according to the model from any frequency to all available frequencies of the system.

Both plots are related in such a way that the second plot represents the average of the RMSEs per column of the first plot. This allows us to have a particular and a global vision of the RMSE distributions in relation with the system frequencies.

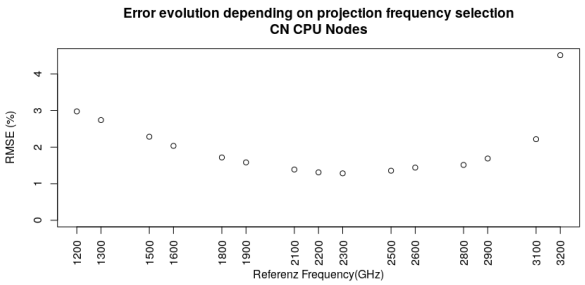
As it can be induced from those graphs:

- Projecting an energy value from frequencies further and further away from the reference frequency causes the RMSE value to increase. This is completely logical since the model being applied is a linear model.
- On the other hand, it is also observed that projecting the energy from the turbo frequency is much worse than from any other frequency. This is because the power is much higher but does not always time scale with frequency.

This study has been performed for the three available modules on the DEEP-EST prototype (see figures 61, 62 and 63).

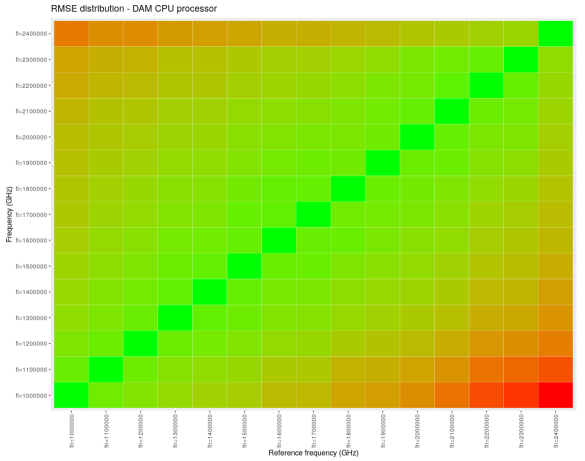


(a) CM Heatmap: RMSE of the model using each possible frequency as a reference frequency and projecting to all available frequencies.

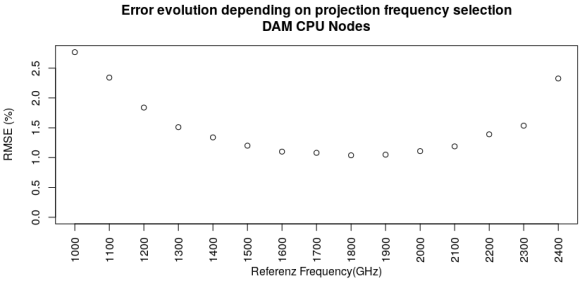


(b) CM Errors: Distribution of RMSE values when projecting the energy from a certain reference frequency.

Figure 61.: On the CPUs of the CM nodes the best frequency to proyect energies is around 2.2GHz, 2.3GHz



(a) DAM Heatmap: RMSE of the model using each possible frequency as a reference frequency and projecting to all available frequencies.



(b) DAM Errors: Distribution of RMSE values when projecting the energy from a certain reference frequency.

Figure 62.: On the CPUs of the DAM nodes the best frequency to proyect energies is located around 1.8GHz and 1.9GHz

5.3.2. Real applications

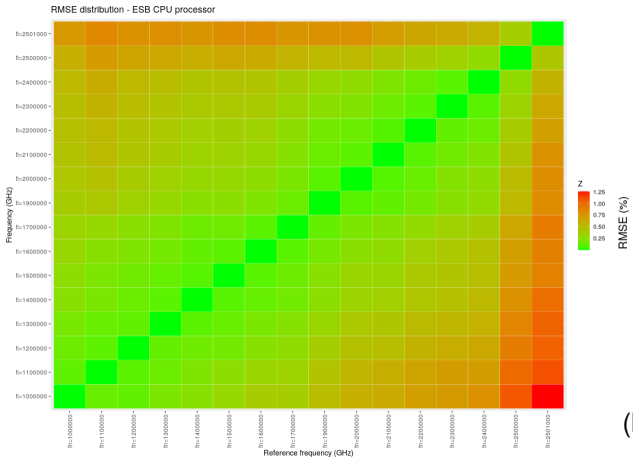
In the previous section, the model has been validated with microbench experiments very similar to those used to create the energy consumption model itself. This is valid as a first approximation of the accuracy of the model, but does not provide us with reliable information about its results in real applications.

To validate the energy model of the microbenchmark experiments, the complete set of results was divided into two groups, of which one was reserved for model computation and the other was used to check the accuracy of the model. On this occasion, the group for the calculation of the model must not be recalculated, since we want to check if that group is complete enough to differ the energy consumption of these real applications. On the other hand, in order to validate the model against real applications, the hardware counter values obtained during the execution of the application in question will be used as a test set.

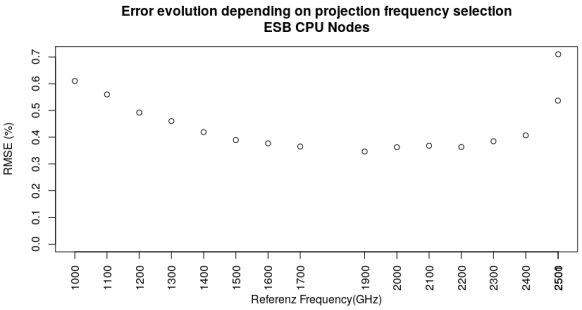
As a real application the NAS Parallel Benchmarks (NPB) were chosen. NPB are a set of benchmarks targeting performance evaluation of highly parallel supercomputers. NPB consists of 11 benchmarks of which 3 have been selected for the validation of the models. These 3 benchmarks are:

- “Block Trigiagonal” (NPB-BT, or simply BT)
- “Scalar Pentadiagonal” (NPB-SP, or simply SP)
- “Lower-Upper Symmetric Gauss-Seidel Decomposition” (NPB-LU, or simply LU)

which solve a synthetic system of nonlinear partial differentation equations (PDE) using three different algorithms involving block tridiagonal, scalar pentadiagonal and symmetric successive over-relaxation (SSOR) solver kernels, respectively.



(a) ESB Heatmap: RMSE of the model using each possible frequency as a reference frequency and projecting to all available frequencies.



(b) ESB Errors: Distribution of RMSE values when projecting the energy from a certain reference frequency.

Figure 63.: On the CPUs of the ESB nodes the best frequency to proyect energies is located around 1.9GHz and 2.0GHz

To create the set of experiments of the test group, the value of the hardware counters involved in the model is needed. In the case of the microbenchmark, these values were obtained by calls to the PAPI library injected into the microbenchmark source code. In the case of real applications, the administrator, developer or user is not expected to profile the source code of the application to obtain the desired counter values at the end of the execution but, is delegated to a system application that monitors the clusters and applications continuously in the background. At the end of the execution of the application, only one query must be made to this monitor to obtain the value of the hardware events in question. In our particular case, this system monitor is the Data Center Data Base (DCDB), from which the values of the counters can be reached by means of queries. DCDB collects and aggregates all the hardware performance counters every second. Having longer latencies, i.e. 10 seconds, if an application takes for example 39 seconds, we will be loosing the information of 9 seconds. This 9 seconds difference will increase the difference between the projected energy and the consumed one, increasing therefor the RMSE value for the test. So, with a latency of 1 second, there is no much information which could be missed and which could increase the global RMSE error of the models.

The model was tested with the three NPB bencharks commented above each of them on the three modules of the DEEP-EST prototype. All tests were done in an unique node running in full mode (one process per core) and with all possible frequencies of the chip as reference frequency for the model.

In order to make it easier to visually compare the results of the different benchmarks, a graph has been produced for each cluster. But, since every testbench has a different runtime, in this case a normalized energy value has been depicted. The normalized energy values corresponds to the power dimension. Each graph shows a comparison of the real energy consumed by the benchmark in that cluster and the energy projected by the model fed with the values of the necessary hardware meters. Both, the value of the real energy used and the values of the hardware counters have been obtained by means of queries from the DCDB monitor.

The first module in which the model has been validated is CM (see figure 64). In this case, 2.3 GHz has been used as the reference frequency, since this is the optimal reference frequency according to previous studies. We can also see that, at 2.3GHz both energy measurements are in accordance with each other. Using this frequency and the RMSE formula (see 5.4), the error for the test NPB-BT is 5.12% on power, while for the test NPB-SP it is 3.59% on power. As seen above, the model does not fit well for the turbo frequency . Much of the RMSE error accumulates at the latter frequency in the plot.

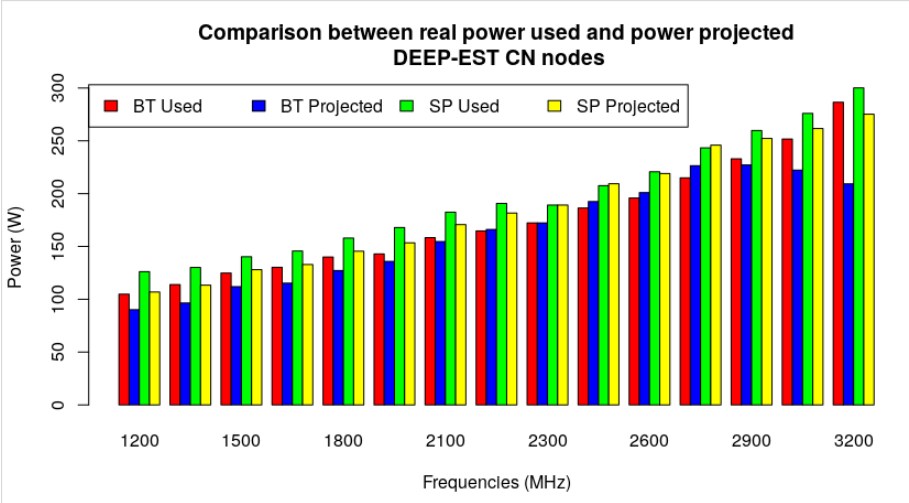


Figure 64.: Comparison of the power used by a node in the CM module while runing the NPB benchmarks.

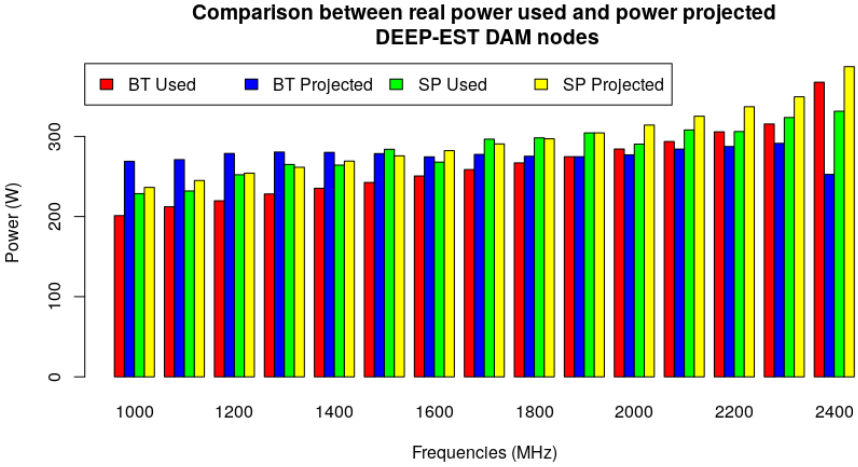


Figure 65.: Comparison of the power used by a node in the DAM module while runing the NPB benchmarks.

As a second example, the model was validated on the DAM module (see figure 65). As expected from what was learned from the behaviour of the model in the CM module, the model fails in the calculation of the energy projection at the maximum frequency. Even so, the RMSE

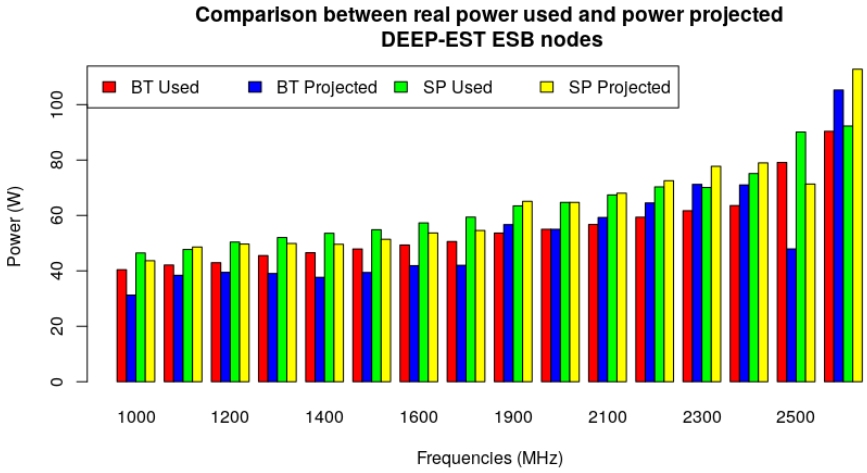
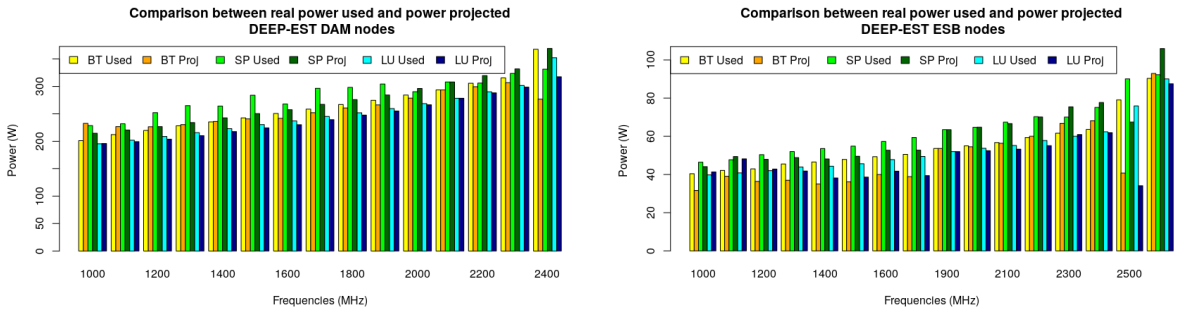


Figure 66.: Comparison of the power used by a node in the ESB module while runing the NPB benchmarks.

error for NPB-BT is 3.61% on power, while the RMSE error in the NPB-SP case is 4.42% on power. In this particular case, the reference frequency used is 2.1GHz.

As a third example, the model was validated on the CPU processors of the ESB module (see figure 66). As it could be expected from what was learned from the behaviour of the model in the CM and DAM modules, the model fails much more at higher frequencies. The reference frequency used in this example is 2.0GHz as it was observed, it could be the optimal one for the model computation. In this case, using this reference frequency the RMSE error for the NPB-BT testbench is 7.72% on power and for the NPB-SP experiment is 4.31% on power.

The model calculated was validated also with the NPB-LU benchmark. The NPB-LU benchmark is even more compute bound as the NPB-SP or NPB-BT testbenches and it has the characteristic that also OMP threads are involved in the computation. The results of the model fit can be show in figure 67. In that case, only the fit of the DAM and ESB models are depicted, since the behaviour of the CM and DAM processors are really similar.



(a) DAM Partition

(b) ESB Partition

Figure 67.: Comparison of the power used by a node in the DAM and ESB cluster partitions while runing the NPB-LU benchmark.

Regarding the validation of the GPU model, at the time of writing, the proposed model has not yet been able to validate the NPB bechmark in an acceptable way. The RMSE error is currently around 25%. This failure to validate the model may be due to several things:

- That the set of hardware counters (components of the model) is not adequate to reproduce the behaviour of the total energy consumption of the application. Of course, the behaviour of a CPU is different from the behaviour of a GPU and therefore it is possible that the selected set of counters is not correct. One component that might make sense for the model would be the use of the GPU as we believe this would very effectively modulate the power consumption of the GPU. In this case it would be necessary to re-profile the microbenchmark by including more or different hardware counters and see if after the PCA analysis these components are relevant or not.
- The microbenchmarks used to generate the model coefficients are not suitable for GPU application. It could be, that they do not really make an efficient use of the GPU. In this case it would be necessary to find a testbench or application that is better suited to the characteristics of the GPUs in use.

5.4. Results

In this chapter on the calculation of energy models, a summary of the state of the art has been made from the point of view of the previous deliverables. In addition, the studies and results carried out in this last phase of the project have been presented, in which, in addition to studying the contribution of the selection of the reference frequency on the accuracy of the model, the different models have been compared with each other.

Finally, the energy models have been validated in each of the possible modules of the DEEP-EST project prototypes for different applications such as the NPB. For this purpose, the functionalities of the DCDB monitoring tool, also developed within the framework of this project, have also been used.

It is observed that the maximum RMSE error in the microbenchmark experiments is less than 2%, 1.5% and 0.5% for the CM, DAM and ESB modules respectively. Applying a similar methodology applied to the NPB testbenchs, it is observed that the errors are now less than 5.2%, 4.5% and 7.8% on power.

Something that is also observed in all the tests is, that the model error is always larger when projecting onto the higher node frequencies. This might be due to the fact that:

- The runtime does not scale linearly with frequency. Therefore, as a linear model is being considered, there will always be a greater error in cases where time no longer scales. This occurs at the highest frequency of the system, as there are always components external to the CPU whose access time is independent of the CPU frequency.
- The power consumption of the chip is given by the formula $P = V * f^2$, where P refers to the power consumption, V means voltage and f is the frequency. So, the power scales with the squared frequency and therefor, the error predicting the model also increases with the square increase in frequency.

It is also worth noting that, in relation to the validation of the models in real applications, the RMSE error of the NPB-SP test is always lower than the NPB-BT error. This might be due to the fact that the SP benchmark is more compute bound than the BT experiment. So, the model fit improves with the compute bound feature of the application; the higher the compute bound ratio is, the better the energy model fits. This is reasonable since a memory bound application can not be accelerated by changing the CPU frequency so, the model have much energy variances with the frequency changes.

To conclude, it has been proven that the energy can be modeled in base of certain performance counters measured at a reference frequency with a low RMSE value. In out case, this set of performance counters consists of level two cache misses, level three cache misses, the vectorization per cycle rate and the number of instructions performed. Of course it depends on the characteristics and behaviour of the application running on the system.

6. Summary

This document presents the main contributions and conclusions from the four tasks in WP2. Tk2.1 has delivered a benchmark suite for application and system evaluation. Nine applications and eleven synthetic kernels have been integrated into the JUBE benchmarking environment an executed regularly. We have described in this document the lessons learned and the experience of these years regarding application integration, experiments configuration, data management, etc, which is even more valuable than the files composing the benchmark suite itself. System monitoring has been shown very valuable in order to identify unexpected changes in the system performance when installing new or updating components. Benchmark monitoring is targeted to identify changes in the application performance because of new releases, new library versions or different use cases. The experience has demonstrated this is doable and very valuable, but also that a more flexible tool would be needed to make it portable to other environments.

In a similar way, the contribution of Tk2.2 has been not only the evaluation the WP5 contributions but also the definition of the methodologies to perform such evaluation given the characteristics of a modular system. The lack of standard inputs for these architectures has forced us to define the methodology for workload generation and the metrics for the workflows evaluation in modular systems. Using the Cirne model as baseline, our methodology mixes N workloads into a single modular workload using the MWF proposed in D2.1 to be able to ask for specific characteristics of our systems such as different modules, components dependencies, etc. Apart from describing the execution environment in detail, we have introduced a set of specific metrics for workflow evaluation. Two new evaluations have been done: The comparison between modular system and three different homogeneous clusters, each one using one of the module architectures of the DEEP-EST prototype. The results have provided very interesting conclusions showing the ESB architecture by itself is a very good approach since it accelerates lots of applications and the fact it is a less expensive solution makes it possible to have more nodes. Moreover, the modular system is the best design because it includes CPU only nodes (CM) and applications that don't take benefit of GPUs can be efficiently executed in the CM. So, the modular system combines the benefits of the two approaches as expected. Finally, we have also evaluated the WF-API proposal for dynamic workflow management. This API is an innovative strategy to coordinate the application and the job scheduler. Results presented, even they are a worse case for us since we are not introducing any additional runtime for jobs communicating without our proposal, shows promising results in terms of job overlapping. In the case with 24% of jobs being part of a workflow, between 8% and 27% of the CPU time has been able to be overlapped.

In Tk2.3 we contributed with an analysis methodology to evaluate the application's efficiency and scalability for homogeneous, hybrid and modular applications. Furthermore, we conducted analyses on 5 different applications with several interesting configurations. The scales of the analyses range from few hundreds to few thousand processes, which is still far from the exascale. Therefore, we generated tentative extrapolation models projecting the efficiency trends up to 1 million cores to anticipate which efficiency factors would become more limiting to the application's scalability. Due to the limited size of the deployed cluster, we were unable to validate the predictions with real executions at larger scales. Hence, we have studied the traces in de-

tail to validate the observations and provide ideas on how to improve the applications towards the exascale. Some of these suggestions, mostly targeted at improving small imbalances and reducing dependency chains that accumulate and propagate, have been simulated to estimate their impact on larger scales. We reported potential improvements in the efficiency of the parallelization of up to 9 percentage points for GROMACS; up to 15—50 for the different phases of xPIC; up to 40, 80 and 65 percentage points for Arbor, NEST and their coupling, respectively; and up to 15 on EC-EARTH.

Tk2.4 has created CPU and GPU energy models for CM, DAM and ESB. These models predict the power and time of the different modules and devices (CPU and GPU) individually and can be used for energy optimization. As part of Tk2.4 we created an API to be able to load different modulfor different architectures. This API was tested in a collaboration with WP5 and was demonstrated to be usable to load different models and coefficients for different modules. The CPU energy models showed a very good average error (1,5% and 3%) when evaluated with samples collected during the creation of the model. The GPU energy model was showing worse results, in average close to 10%. This document includes a full description of the methodology used to create the models as well as the final analysis of these models including the average error when projecting from the different CPU frequencies and the validation using additional benchmarks. The analysis of the average error when projecting from different frequencies shows us the error is not constant and depends of the reference.

Appendices

A. Benchmarking

This appendix contains technical information about the benchmarking suite integrated within the DEEP-EST project.

A.1. Benchmark Suite

The git repository of the benchmark suite has the following directory structure.

```
deep-est_benchmarks/
├── README.md
├── applications/
├── bokeh_server/
├── support_scripts/
├── synthetic/
├── application_benchmark_driver
├── benchmark_driver
└── synthetic_benchmark_driver
```

The directory applications/ contains the application benchmarks delivered by the collaboration partners. The directory bokeh_server/ contains infrastructure to plot the benchmarking results by use of a local bokeh server, see reference [1]. The directory support_scripts/ contains the infrastructure needed for regular benchmarking. The directory synthetic/ contains the synthetic benchmarks. The script application_benchmark_driver enables initiation of all application benchmarks, the script benchmark_driver enables initiation of all benchmarks and the script synthetic_benchmark_driver enables initation of all synthetic benchmarks.

With increasing project duration also more and more functionalities needed for the infrastructure of the benchmark suite were integrated into the support scripts and the visualisation logics. They became an integral part of the functionality of the benchmark suite. These infrastructure functionalities needed to be integrated since the complexity of the benchmark suite was growing immensely because of the amount of benchmarks and partitions on the prototype.

The benchmark driver script can be used as a first general starting point for initialising a benchmark. The help can be printed and shows all user options available.

```
$ ./benchmark_driver -h
Help for benchmark_driver:

This script calls benchmark scripts according to the options set up by the user.

-r    'R'un the benchmarks in 'parallel' xor 'serial' batch submission mode.
-p    'P'erform the benchmarks 'mpiLinkTest', 'ior', 'h5perf', 'stream', 'hpl',
      'hpcg', 'mdtest', 'hpcc', 'mpiLinkTestGPU', 'hpl4cuda', 'synthetic',
```

```
      'UoI-piSVM', 'UoI-HPDBSCAN', 'UoI-NEXTDBSCAN', 'UoI-DeepLearning',
      'NMBU-NEST', 'NMBU-Arbor', 'NMBU-Elephant', 'KUL-xPic', 'KUL-DLMOS',
      'NCSA-GROMACS', 'CERN-CMSSW', 'ASTRON-Correlator',
      'ASTRON-Imager', 'UoI', 'NMBU', 'KUL', 'NCSA', 'CERN', 'ASTRON',
      'application' or 'all'.
-t    'T'ype of reservation can be of the form 'daily', 'weekly' xor 'none'.
-o    'O'n 'deepest_cm', 'deepest_dam', 'deepest_esb',  'deepest_cm_esb',
       'deepest_cm_dam',  'deepest_dam_esb',  'deepest_cm_dam_esb' (xor).
-s    'S'teps to perform: 'compile', 'benchmark', 'analysis', 'packtodb' or
      'all'.
```

This script is calling the driver scripts for the synthetic and the application benchmarks. They have mostly the same parameter options only with a subset of the performable benchmarks. These scripts are again calling the benchmark driver scripts of the corresponding benchmarks having the same options again besides the parameter -p. So, performing a single synthetic or application benchmark can also be performed by just moving into the directory of this benchmark and calling the corresponding driver script.

The parameter -r describes the way how the sbatch scripts are executed. 'parallel' is leading to sbatch jobs which are independent from each other and executed in parallel if enough resources are available. The parameter -t describes the reservation used for the execution of this benchmark. 'daily' and 'weekly' are using the corresponding reservations which were allocated daily and weekly for the sake of running theses benchmarks as independent as possible from the rest of the users. The parameter '-o' defines the module permutation being used for this benchmark. The parameter -s states which step of the benchmark is going to be performed. 'compile' just performs the compilation of the benchmarking software. This parameter was introduced since the execution for FPGA software could take hours. 'benchmark' initiates the benchmark itself. 'analysis' triggers the result extraction from the data produced by the benchmarks. 'packtodb' initiates the storing of the results into a database to make them visualisable at a later point.

6.2. Synthetic Benchmarks

The synthetic benchmarks which could be integrated given the resource constraints are stated in the following file hierarchy.

```
synthetic/
├── h5perf/
├── hpcc/
├── hpcg/
├── hpl/
├── hpl4cuda/
├── ior/
└── mdtest/
```

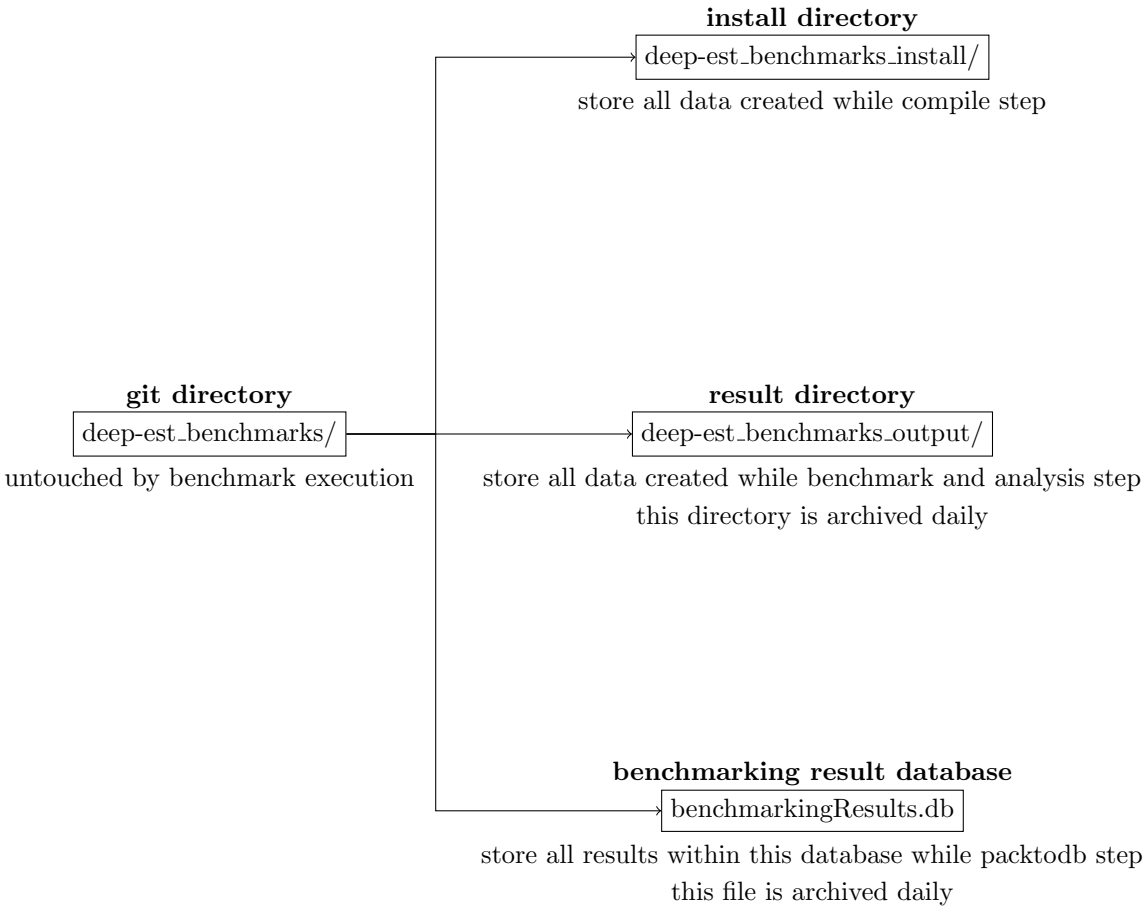


Figure 68.: Files created by benchmarking steps

```
mpiLinkTest/
mpiLinkTestCrossPartition/
mpiLinkTestGPU/
stream/
```

Each directory contains a driver script, JUBE scripts and further software needed for the execution of a benchmark.

h5perf

The parameter values, which can be chosen, are explained within deliverable 2.1. The potential parameter values used are given like follows.

The results are given by average write in MB/s, average read in MB/s and runtime in seconds.

nodes	10
taskspernode	4
api	phdf5
nBytes	2G
iterations	1
collective	true,false
chunked	true,false
interleaved	true,false
minimal transfer size	1G
maximal transfer size	1G

Table 20.: Potential Parameter values of the h5perf Benchmarks

hpcc

HPCC by itself again is a small benchmark suite containing several benchmarks.

- HPL: software package that solves a (random) dense linear system in double precision (64 bits) arithmetic on distributed-memory computers
- DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication
- stream: Sustainable Memory Bandwidth in High Performance Computers
- PTRANS: exercises the communcations where pairs of processors communicate with each other simultaneously
- RandomAccess: measures the rate of integer random updates of memory (GUPS)
- FFT: a set of benchmarks to measure communcation characteristics based on the b_eff benchmark which is the effective bandwidth benchmark

nodes	1
taskspernode	8,24

Table 21.: Potential Parameter values of the hpcc Benchmarks

Here we are only using the results from the RandomAccess benchmark which are given by MPIRandomAccess_LCG_GUPs, MPIRandomAccess_GUPs, StarRandomAccess_LCG_GUPs, SingleRandomAccess_LCG_GUPs, StarRandomAccess_GUPs and SingleRandomAccess_GUPs. GUPs is representative for 10^9 updates per second.

hpcg

This performance benchmark algorithm solves a dirichlet boundary condition problem in 3 dimensions with a 27 point stencil. It is designed to define a benchmarking metric which focuses on other algorithmic workflows compared to the HPL benchmark.

nodes	1,2,3,4
taskspernode	8,24
nx	256
ny	128
nz	128
seconds	240

Table 22.: Potential Parameter values of the hpcg Benchmarks

The result is given by performance in GFLOP/s and runtime in seconds.

hpl

This benchmark is the Quasistandard for measuring performance of a system. The scalable algorithm solves a linear system of equations by use of an LU-Decomposition. The output is the performance of the system for a particular set of input parameters. This set of input parameters is varied until the maximum performance is yielded.

The result is given by the performance in GFLOP/s.

hpl4cuda

This algorithm, developed by nvidia, expands the hpl benchmark to include the usage of available GPUs. The amount of work is spread between the CPUs and the GPUs.

The result is given by the performance in GFLOP/s.

ior

The ior benchmark was discussed in deliverable 2.1. The parameters for the benchmarks were chosen as in table 25.

The results are given by read and write bandwidth in MB/s and by the runtime in seconds.

mdtest

Being an extension of the ior benchmark, mdtest measures the metadata performance of the file system at hand. The parameters used for this benchmark are given as follows.

The results are given by FileCreation_op/s, FileStat_op/s, FileRemove_op/s, FileRead_op/s, TreeCreation_op/s and TreeRemoval_op/s. The unit op/s seconds is stating the number of operations per second.

nodes	1,4,16
taskspernode	8,24
ns	79872,115584,159744,463104
number_of_nb	1
nbs	192
pmap_processing_map	0
number_of_process_grids	1
ps	2,4,8,16
qs	4,6,12,24
threshold	16
number_of_panel_fact	1
pfacts	2
number_of_recursive_stopping_criterium	1
nbmin	4
number_of_panels_in_recursion	1
ndivs	2
number_of_recursive_panel_fact	1
rfacts	1
number_of_broadcast	1
bcasts	1
number_of_lookahead_depth	1
depths	1
swap	2
swapping_threshold	64
l_in_form	0
u_in_form	0
equilibration	1
memory_alignment_in_double	8
number_of_additional_problem_sizes_for_ptrans	0
values_of_n	1200 10000 30000
number_of_additonal_blocking_sizes_for_ptrans	0
values_of_nb	40 9 8 13 13 20 16 32 64

Table 23.: Potential Parameter values of the hpl Benchmarks

mpiLinkTest

The mpiLinkTest measures bandwidth performances when sending data between two tasks by use of MPI. The parameters were discussed within deliverable 2.1 and take the following values.

The result is given by the runtime in seconds and the maximal, minimal and average bandwidth in MB/s.

nodes	1,2
taskspernode	1
ns	32768
number_of_nb	1
nbs	512
pmap_processing_map	0
number_of_process_grids	1
ps	1,2
qs	1
threshold	16
number_of_panel_fact	1
pfacts	0
number_of_recursive_stopping_criterium	1
nbmin	2
number_of_panels_in_recursion	1
ndivs	2
number_of_recursive_panel_fact	1
rfacts	0
number_of_broadcast	1
bcasts	0
number_of_lookahead_depth	1
depths	1
swap	1
swapping_threshold	192
l_in_form	1
u_in_form	1
equilibration	1
memory_alignment_in_double	8
number_of_additional_problem_sizes_for_ptrans	0
values_of_n	1200 10000 30000
number_of_additonal_blocking_sizes_for_ptrans	0
values_of_nb	40 9 8 13 13 20 16 32 64
cpu_cores_per_gpu	1
cuda_dgemm_split	0.95
cuda_dtrsm_split	0.85

Table 24.: Potential Parameter values of the hpl4cuda Benchmarks

mpiLinkTestCrossPartition

This extension to the mpiLinkTest benchmark allows for intermodular communication.
The result is given by the runtime in seconds and the maximal, minimal and average bandwidth.

nodes	1
taskspernode	2
api	POSIX,MPIIO
blockSize	64g
transferSize	8m
segmentCount	1
repetitions	1
verbose	0
fsync	0,1
collective	0,1
memoryPerNode	0%
storeFileOffset	0
taskPerNodeOffset	0
multiFile	0
reorderTasksConstant	1
reorderTasksRandom	0
writeFile	1
readFile	1
filePerProc	0
useO_DIRECT	0
fsyncPerWrite	0
fsync	0

Table 25.: Potential Parameter values of the ior Benchmarks

nodes	1
taskspernode	2
testFilesOnly	-F
createOnLeafOnly	-L
uniqueDir	-u
create	-C
stat	-T
remove	-r
items	150,250
read	””,-E
writeBytes	””,”-w 3901”
readBytes	””,”-e 3901”

Table 26.: Potential Parameter values of the mdtest Benchmarks

mpiLinkTestGPU

An extension to the mpiLinkTest benchmark includes the usage of GPU buffers for sending messages from one GPU to another GPU on another node. This way the communication is significantly speeded up allowing for a slim acompanying CPU and RAM.

nodes	1,4,8,16,32
taskspernode	2
AllToAll	0
Warmup	2
Randomized	0
Size	1,16384,4194304
Iterations	1000
Serialized	0,1

Table 27.: Potential Parameter values of the mpiLinkTest Benchmarks

nodes	6
tasks	20
AllToAll	0
Warmup	2
Randomized	0
Size	131072
Iterations	50
Serialized	0

Table 28.: Potential Parameter values of the mpiLinkTestCrossPartition Benchmarks

nodes	2
taskspernode	2
AllToAll	0
Warmup	2
Randomized	0
Size	1,16384,1048576,4194304
Iterations	50
Serialized	0,1
network	TCP,Extoll,InfiniBand
node_unit	CPU,GPU

Table 29.: Potential Parameter values of the mpiLinkTestGPU Benchmarks

The result is given by the runtime in seconds and the maximal, minimal and average bandwidth.

stream

This benchmark measures the performance of the memory at hand. The stream benchmark allows for measurement of the cache levels.

The results are given by the performance of different operations on data on the memory. These are defined by add_MB/s, copy_MB/s, scale_MB/s and triad_MB/s.

nodes	1
taskspernode	1
threadspertask	4,8,12,24
o_level	3
size	2^26
ntimes	10
offset	0

Table 30.: Potential Parameter values of the stream Benchmarks

6.3. Application Benchmarks

Further information and more in depth descriptions of the application benchmarks are located within D1.2. The file hierarchy structure representing the application benchmarks within the benchmark suite is given by the following.

```
applications/
├── ASTRON/
│   ├── Correlator/
│   └── Imager/
├── CERN/
│   └── CMSSW/
├── KUL/
│   ├── DLMOS/
│   └── xPic/
├── NCSA/
│   └── GROMACS/
├── NMBU/
│   ├── Arbor/
│   ├── Elephant/
│   └── NEST/
└── UoI/
    ├── DeepLearning/
    ├── HPDBSCAN/
    ├── NEXTDBSCAN/
    └── piSVM/
```

Each directory contains a driver script, JUBE scripts and further software needed for the execution of a benchmark.

ASTRON/Correlator

The correlation step correlates the station signals, which contain information on amplitude and phase of the source.

The parameter for variation for the Correlation benchmark is given by NR_STATIONS. NR_STATIONS defines the number of telescope pairs being used for the algorithm. All parameters used by this

benchmark are given by the table 31.

nodes	1
tasks per node	1
NR_STATIONS	1,16,32,48,64,80,96,112,128,144,160,176,192,208,224,240,256,272,288, 304,320,336,352,368,384,400,416,432,448,464,480,496,512,528,544,560, 576

Table 31.: Correlator Parameter Permutations

The main result of the benchmarks is defined by the total correlation performance in TFLOP/s.

ASTRON/Imager

There are two parameters being varied for the Imager benchmark. NR_CHANNELS is stating the number of pictures being overlapped. The parameter timesteps defines the number of pictures being shot after each other. The parameters are given as in table 32.

nodes	1
tasks per node	1
threads per task	16,96
NR_CHANNELS	1,4,8,16
timesteps	3600,7200,14400

Table 32.: Imager Parameter Permutations

The results of this benchmark are given by gridding_MVis,degridding_MVis,gridding_gflops,degridding_gflops and fft_gflops. Gridding and degridding are represented by normal and inverse fourier transformations of the algorithm. MVis is standing for 10^9 Visibilities.

CERN/CMSSW

For the CMSSW benchmark one parameter permutation was performed.

nodes	1
tasks per node	1
threads per task	16

Table 33.: Potential Parameter values of the CMSSW Benchmarks

The main result is given by eps which is the number of the events per second which can be processed during execution.

KUL/xPic

Main parameters to variate are given by run_ntcx, run_nblockx, run_nppc and job_input. The parameter run_ntcx defines the number of cells per species. The parameter permutations are mainly created by permuting the number of MPI processes job_mpiproc and by varying run_nctx. The parameter run_nblockx defines the number of blocks per MPI process. The parameter run_nppc defines the number of particles per cell per process.

job_mpiproc	1,2,4,8,12,16,24
run_ntcx	16384,32768,65536,131072,196608,262144,393216
run_nblockx	4,8,16,32,48,128,256,512
run_nppc	100,1000
job_input	test_02.inp,test_03.inp,test_04.inp,test_05.inp,test_06.inp,test_07.inp

Table 34.: Potential Parameter values of the xPic Benchmarks

The benchmarking result is given by the runtime in seconds.

NCSA/GROMACS

Three molecules with different sizes were chosen to be benchmarked. The molecular dynamics simulations can be executed on all partitions. CPUs and the GPUs can perform the simulations. Throughout the project duration new GROMACS versions were published. These were integrated into the regular benchmarking procedure. The parameter permutations are given by a tensor product of the following parameters with some restrictions to valid parameter permutations.

Molecule	Magainin, Bombinin, Ribosome
GROMACS Version	2018.4, 2019.3, 2019.6, 2020.1
# Compute Nodes	1,2,4,8
# MPI Tasks per Node	1,4,8,12,24,48
# Threads per MPI Task	1,2,3,4,6,8,12,24,48
tprfile	magainin.tpr,magainin-2019.tpr, bombinin.bombinin-2019.tpr, ribosome.tpr,ribosome-2019.tpr

Table 35.: Potential Parameter values of the GROMACS Benchmarks

See D2.1 , [2], for more details concerning the molecules. Not all potential parameter permutations are being executed. If a set of parameter permutations makes no sense or if the execution fails due to incommensurable values, with respect to the system to be simulated, the simulation is not performed.

The result of a benchmark execution is defined by the performance in nanoseconds simulated for the molecule per day of simulation time.

NMBU/NEST

The neural network simulation NEST's parameter set is defined by the number of MPI tasks, the number of nodes and the commit id of the NEST git repository. Potential values are given in the table 36.

commit id	00e0d3c,3236fe9,b84c9ba,37722d5,c2a153b
ntasks	2,4,8,16,32,64,94
tasks per node	2
nodes	1,2,4,8,16,32,47
threads	24

Table 36.: Potential Parameter values of the NEST Benchmarks

The result from this benchmark is the simulation time in seconds.

NMBU/Arbor

Only one set of parameter permutations is used at this point.

OpenMPthreads	24
MPIProcesses	1
NEST Version	master@a16232066

Table 37.: Potential Parameter values of the Arbor Benchmarks

The result is given by the runtime of the benchmark.

UoI/HPDBSCAN

The input parameters are given by the number of nodes, the threads per task and the input datasets. The potential values are give by the following.

nodes	2,8
threadspertask	2,16
dataset	bremenSmall.h5.h5,bremen.h5.h5

Table 38.: Potential Parameter values of the HPDBSCAN Benchmarks

The main result is given by the runtime.

UoI/piSVM

This application benchmark uses the same input dataset and parameters throughout the project for one parameter permutation.

The result is given by the runtime of the benchmark.

nodes	2
taskspernode	24
train_dataset	indian_processed_test.el
dataset	indian_processed_training.el

Table 39.: Potential Parameter values of the piSVM Benchmarks

6.4. Support Scripts

The support scripts evolved with increasing experience and needs of the benchmark suite. They contain, in summary, the following functionalities.

- logic for updating and checking out repositories
- check the stated user flags and defining defaults for parameters
- cron schedule
- check the status of running jobs
- clean the job queue
- check and installing database software
- pack results to database
- collect system information (environment variables, kernel version, etc.)
- logic for updating/backup of executables
- perform all compilations
- upload of the last execution date of benchmarks
- logic for archiving benchmarking files

The benchmarking results were stored into several sqlite databases. These databases are used to perform the plotting.

Examples for the files and the type of functionality offered herein is given by the script pack_results_to_db.py.

```
$  python ./pack_results_to_db.py -h
Use this script as follows and in this parameter order:

pack_results_to_db.py -r resultPath -d databasePath -t tableRootName
                      -n numberOfResultsToPack -f resultFileName
resultPath           : defines the path where the jube benchmarks are performed
                       and the results can be found which will be integrated
                       into the database. This should be a string starting with
                       a /. The result folder is assumed to contain all jube
                       runs in ascending numeric order.
databasePath         : defines the path of the database in which the results
```

```
                        will be stored. This should be a string starting with a
                        slash.
tableRootName         : defines the root part of the tablename into which the
                        results will be stored within the database.
numberOfResultsToPack: defines the number of the last results which will be
                        considered for integrating into the database. This should
                        be a positive integer or zero. If 0 it will take all
                        results.
resultFileName        : defines the name of the result file to read.
```

By use of this script the regular benchmarking results collected and packed into an sqlite database for later use by a bokeh visualisation server.

Another example for the functionalities of the support scripts is given by the script collect_sys_info.sh. This script is collecting system information of the current exeuction like environment variables and the linux kernel version. It serves as a way to check the state of the environment of the execution of a benchmark. This could be of special interest if one can identify significant changes in the performance of a benchmark execution.

6.5. Visualisation

Visualisation of the benchmarking results allows for receiving an overview. The file hierarchy of the visualisation software is given by the following.

```
bokeh_server/
├── bokeh_driver
├── clean_database.py
├── description.html
├── main.py
└── visualization.py
```

By use of the script bokeh_driver the bokeh server and the visualisation can be performed.

```
$ ./bokeh\_driver -h
Help for bokeh_driver:

This script performs the necessary steps to open a bokeh visualization server.
Once the server is started you have to follow the steps of the bokeh output to
connect to the server. If you are running the server on a the deep cluster and
you would like to visualize the data in your local browser type:
ssh -N -f -L localhost:5006:localhost:5006 <your_user_name>@deep.fz-juelich.de
and use afterwards the link stated by bokeh on your local browser.
To release the connection on your local machine type:
```

```
lsof -ti:5006 | xargs kill -9

Please state all of the following parameters when running this script:

-b    'B'enchmark to visualize.
-d    'D'atabase path.
```

The relevant results are plotted on the vertical axis. The timeline is plotted on the horizontal axis. The user of the visualisation tool can state possible parameter combinations plotting different temporal benchmarking result evolutions. The tool used here is a bokeh server.

The server can be set up on the login node. After connecting with the login node by use of ssh the visualisation can be performed on the local webbrowser.

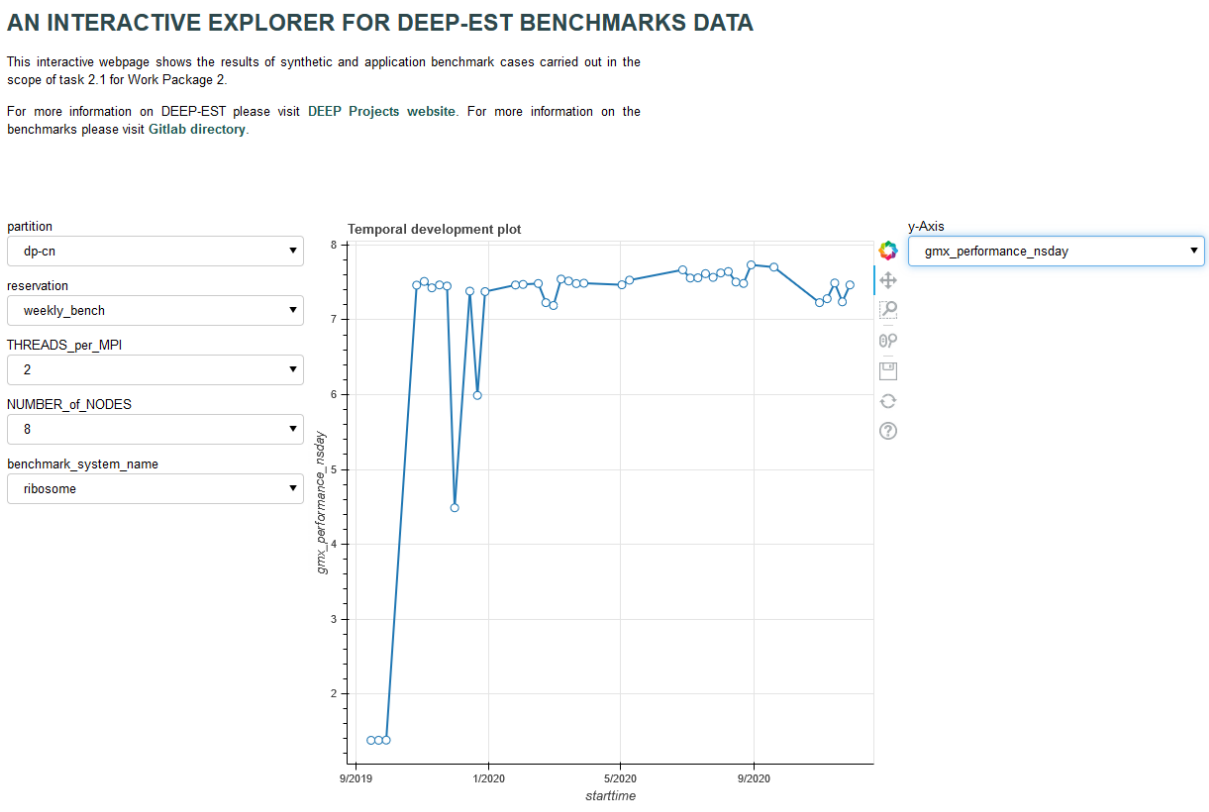


Figure 69.: Sample Webbrowser Visualisation

The server is connecting to one of the databases storing the regular benchmarking results. There is a database for every partition permutation used throughout the measurement.

7. Modular Workload Format

7.1. Modular Workload Format fields

Modular fields

1. **Modular_Job_Id** – An ID common to all the components of the modular job.
2. **Total_Components** – Number of components in the modular job (minimum one)
3. **Modular_Job_Name** – Text
4. **Submit_Modular_Job_Time** – in seconds. Submission time for the first set of components
5. **Wait_Modular_Job_Time** – in seconds. The difference between the job's submit time and the time at which it actually began to run (some of its components). It is not needed for evaluation, only for comparison between results.
6. **Modular_Requested_Time** – in seconds. Limit for the modular job. -1 if this value is not provided. In that case, the partition limit will be used
7. **Num_Components_At_Submit_Time** – Integer. This field is the number of components submitted together at modular submit time

Job Component fields

8. **Component_Job_Id: Modular_Job_Id+ Offset** – This JOB ID is unique. It goes from Modular_Job_Id to Modular_Job_Id+(NumComponents-1).
9. **Component_Job_Name** – text
10. **Wait_Component_Job_Time** – in seconds. The difference between the job's submit time and the time at which it actually began to run. It is not needed for evaluation, only for comparison between results
11. **Component_Run_Time** – in seconds. Integral number of seconds.
12. **Status** – 0 means COMPLETED with success. Values different from 0 will represent errors.

Job Component resource requirements description

The job scheduler receives job component requirements, applies the job scheduler and resource selection policy, and reports a set of resources allocated. Resource allocation is reported for comparison but it is not part of the input. One component will run in a single module. If one job needs more than one module, one component per module will be specified.

HW resources

13. **Executable_Number** – a natural number, between one and the number of different applications appearing in the workload. in some logs, this might represent a script file used to run jobs rather than the executable directly; this should be noted in a header comment
14. **Partition_Name** – Text with the partition name; NA, if no specific partition is requested
15. **Nodes** – an integer. Number of nodes requested
16. **Processes_Per_Node** – an integer
17. **Threads_Per_Process** – an integer
18. **Memory_Per_Node** – In KB
19. **Freq** – frequency in kilohertz, min and max
20. **Reference Power** – Input average power in Watts. Input by user or a power model.
21. **NAM** – in KB. NAM (Network Attached Storage) is a global resource and users will be able to ask for it
22. **Local_Storage** – in MB
23. **Network** – list o network requirements. More than one can be added with AND $\&$ or OR $|$
24. **Constraint** – A set of keywords, potentially with $\&$ or $|$ special characters. These constraints must be specified in the different modules to simplify resource selection. For instance, based on sbatch manual [**?**] $intel\&gpu, intel|amd$
25. **Hint** – at least cpu_intensive and memory_intensive. SLURM supports hints for resource allocation. These hints can be also used, and extended, for energy–performance models

SW resources

Licenses – Text. More than one can be added with AND $\&$ or OR $|$

Component resource allocation description

One component will run in a single Module. If one job needs more than one module, one component per module will be specified.

HW resources

26. **Component_Module_Id** – 0 - Number of Modules (One component will run in a single module). Module ID where this component is executed
27. **Partition_Name** – Text with the partition name selected
28. **Nodes** – Number of allocated nodes
29. **Processes_Per_Node** – an integer
30. **Threads_Per_Process** – an integer
31. **Memory_Per_Node** – In KB (0 if not requested)
32. **Freq** – frequency in kilohertz
33. **Average Power** – measured average power in Watts

34. **NAM** – in KB (0 if not requested)
35. **Local_Storage** – in MB (0 if not requested)
36. **Network** – keywords describing resources allocated or NA if not requested

SW resources

37. **Licenses** – licenses allocated or NA if not requested

Dependencies

38. **After_Component_Job_Id** – This component must start after job ID. -1 if there is no dependency
39. **Dependency_Type** – NA/DYNAMIC/AFTER/AFTERANY/AFTEROK/AFTERNOTOK//SINGLE. This is the list of types of dependencies supported by SLURM. DYNAMIC is an additional type defined here.
    - NA means there is not dependency
    - DYNAMIC means the component must be started N seconds after AFTER_COMPONENT_JOB_ID. The number of seconds is defined in the next field, and in that case it is relative to the dependent job start time.
    - AFTEROK/AFTERNOTOK – This job can begin execution after the specified jobs have successfully/not successfully executed
    - SINGLE is a special case defined by SLURM. This job can begin execution after any previously launched jobs sharing the same job name and user have terminated
40. **Think_Component_Time** – in seconds. When DYNAMIC is selected, it corresponds to the requested delay from the start of the first component to the start of this component. Otherwise it is related to the job finalization overhead

Component-level events

41. **API_call_time** – in seconds. It models events called by the job that impact the job scheduling. It is used to model the API call for the workflow policy, but it can be extended to a list of comma-separated key:value elements, with key represent the event type (ID or keyword), and value the number of seconds passed from the start of the job until the event. E.g.: ”WF_API:650,extend_job:850”.

List of Acronyms and Abbreviations

C

CM Cluster Module: with its Cluster Nodes (CN) containing high-end general-purpose processors and a relatively large amount of memory per core

D

DAM Data Analytics Module: with nodes (DN) based on general-purpose processors, a huge amount of (non-volatile) memory per core, and support for the specific requirements of data-intensive applications

DB Data Base

DCDB Data Centre Data Base (a tool developed in DEEP)

DEEP-EST DEEP - Extreme Scale Technologies

E

ESB Extreme Scale Booster: with highly energy-efficient many-core processors as Booster Nodes (BN), but a reduced amount of memory per core at high bandwidth

FPGA Field-Programmable Gate Array, Integrated circuit to be configured by the customer or designer after manufacturing

H

HCA Host Channel Adapter

HPC High Performance Computing

IFS Integrated Forecasting System

M

MWF Modular Workload Format

MPI Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

MPMD Multiple Program Multiple Data

MSA Modular Supercomputer Architecture

NAM Network Attached Memory

NEMO Nucleus for European Modelling of the Ocean

NPB NAS Parallel Benchmarks

NPB-BT NAS Parallel Benchmarks - Block Trigiagonal
NPB-LU NAS Parallel Benchmarks - Lower-Upper Symmetric Gauss-Seidel Decomposition
NPB-SP NAS Parallel Benchmarks - Scalar Pentadiagonal

P

PCA Principal Component Analysis
PAPI Performance Application Programming Interface
RMSE Root Mean Square Error

S

SLURM Job scheduler that will be used and extended in the DEEP-EST prototype
SPMD Single Program Multiple Data
UPI Ultra Path Interconnect
XIOS XML Input/Output Server

Bibliography

[1] *bokeh: Python library for creating interactive visualizations for modern web browsers* [Online], Available: https://docs.bokeh.org/en/latest/index.html

[2] K. Kulkarni et al.: *DEEP-EST Deliverable 2.1 - DEEP-EST Benchmark Suites*, March 2018

[3] Corbalan et al.: *DEEP-EST Deliverable 2.2 - Initial Application Analysis and Models*, July 2019

[4] A. Kreuzer et al.: *DEEP-EST Deliverable 1.4 - Initial Application Ports*, December 2019

[5] A. Kreuzer et al.: *DEEP-EST Deliverable 1.5- Final report on application experience*, March 2021

[6] *DEEP-EST Deliverable 5.3 - Prototype Software Implementation*, March 2019

[7] N. Eicker et al.: *DEEP-EST Deliverable 5.4 - Complete System-SW Implementation*, December 2019

[8] Marc-Oliver Gewaltig and Markus Diesmann: *NEST (NEural Simulation Tool)*, Scholarpedia, issue 2(4), 2007, http://www.scholarpedia.org/article/nest_(neural_simulation_tool)

[9] *Arbor: The Arbor multi-compartment neural network simulation library* [Online], Available: https://github.com/eth-cscs/arbor

[10] *Elephant: Electrophysiology Analysis Toolkit* [Online], Available: http://elephant.readthedocs.io/en/latest

[11] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen: *GROMACS: A message-passing parallel molecular dynamics implementation*, in Computer Physics Communications, volume 91, 1995, https://doi.org/10.1016/0010-4655(95)00042-E

[12] M. Goetz, C. Bodenstein and M. Riedel: *HPDBSCAN – Highly Parallel DBSCAN* in Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'15), Machine Learning in HPC Environments (MLHPC) Workshop https://dl.acm.org/citation.cfm?id=2834894

[13] Morris A. Jette, Andy B. Yoo and Mark Grondona: *SLURM: Simple Linux Utility for Resource Management*, in Proceedings of the 9th International Workshop Job Scheduling Strategies for Parallel Processing (JSSPP), Springer, Lecture Notes in Computer Science (LNCS), volume 2862, pages 44–60, http://dx.doi.org/10.1007/10968987_3

[14] D. Krause and P. Thörnig: *JURECA: General-purpose supercomputer at Jülich Supercomputing Centre* in Journal of Large-scale Research Facilities (JLSRF), volume 2, article 62, http://dx.doi.org/10.17815/jlsrf-2-121

[15] Stephen Trofinoff and Massimo Benini: *Using and Modifying the BSC Slurm Workload Simulator* in Slurm User Group Meeting, 2015, https://slurm.schedmd.com/SLUG15/BSC_Slurm_Workload_Simulator_Enhancements.pdf

[16] Walfredo Cirne and Francine Berman: *A Comprehensive Model of the Supercomputer Workload*, in Proceedings of the 4th Annual Workshop on Workload Characterization,

2001, https://doi.org/10.1109/WWC.2001.2

[17] *Heterogeneous Job Support in Slurm* [Online], Available: https://slurm.schedmd.com/heterogeneous_jobs.html

[18] *Consumable Resources in Slurm* [Online], Available: https://slurm.schedmd.com/cons_res.html

[19] *Jobs Submitted to Multiple Partitions, p6* [Online], Available: https://slurm.schedmd.com/SUG14/sched_tutorial.pdf

[20] *The RICC log* [Online], Available: http://www.cs.huji.ac.il/labs/parallel/workload/l_ricc/index.html

[21] Weinberg V, Brehm M, Christadler I. 2010. *OMI4papps: Optimisation, Modelling and Implementation for Highly Parallel Applications*.

[22] Strohmaier E, Shan H. 2004. *Architecture independent performance characterization and benchmarking for scientific applications*. International symposium on modeling, analysis and simulation of computer telecommunications systems.

[23] Strohmaier E, Shan H. 2005. *Apex-Map: A global data access benchmark to analyze HPC systems and parallel programming paradigms*. Proceedings of SC2005.

[24] *Extrae instrumentation tool*[Online], Available: https://tools.bsc.es/extrae

[25] *The workload on parallel supercomputers: modeling the characteristics of rigid*. Journal of Parallel and Distributed Computing Year 2003, Volume 63, Number 11, Pages 1105-1122.

[26] *Enabling Continuous Testing of HPC Systems Using ReFrame*. Tools and techniques for high performance computing Year 2020, Springer.

[27] Browne S, Deane C, Ho G, Mucci P. 1999. *PAPI: A Portable Interface to Hardware Performance Counters*. Proceedings of Department of Defense HPCMP Users Group Conference

[28] David H, Gorbato E, Hanebutte U, Khanna R, Le C. 2010. *RAPL: memory power estimation and capping*. Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design. ACM.

[29] Intel Corp. 2012. *Intel 64 and IA-32 Architectures Software Developer Manual*.

[30] Auweter A., Bode A, Brehm M, Brochard L, Hammer N, Huber H, Panda R, Thomas F, Wilde T. 2014. *A Case Study of Energy Aware Scheduling on SuperMUC*. International Supercomputing Conference (ISC).

[31] Luehrs S. et al. 2015. *JUBE – A Flexible, Application- and Platform-Independent Environment for Benchmarking*. International Supercomputing Conference (ISC).

[32] del Moral M.J., Valderrama M.J. 1997 *A principal component approach to dynamic regression models*. Journal of Forecasting (13), 237–244.

[33] *R Studio tool*[Online], Available: http://www.rstudio.com

[34] *Workload logs*[ONLINE] Available: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html

[35] *Workload models*[ONLINE] Available: http://www.cs.huji.ac.il/labs/parallel/

workload/models.html

[36] *List of publications for Lublin model*[ONLINE] Available: http://www.cs.huji.ac.il/labs/parallel/workload/models.html#lublin99

[37] *JURECA system overview*[Online] Available: https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html

[38] *HDF Group Webpage*[Online], Available: https://support.hdfgroup.org/HDF5/doc/index.html

[39] *HPC Challenge Webpage*[Online], Available: https://icl.utk.edu/hpcc/

[40] *HPCG Benchmark Webpage*[Online], Available: https://www.hpcg-benchmark.org/

[41] *HPL Benchmark Webpage*[Online], Available: https://www.netlib.org/benchmark/hpl/

[42] *HPL4CUDA Benchmark Webpage*[Online], Available: https://developer.nvidia.com/computeworks-developer-exclusive-downloads

[43] *ior and mdtest Benchmark Github Repository*[Online], Available: https://github.com/hpc/ior

[44] *mpiLinkTest Webpage*[Online], Available: https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/LinkTest/_node.html

[45] *stream Webpage*[Online], Available: https://www.cs.virginia.edu/stream/

[46] Ian Karlin and Jeff Keasler and Rob Neely. LULESH 2.0 Updates and Changes. August 2013, pages 1-9, LLNL-TR-641973.

[47] *sysbench GitHub repository*[Online], Available: https://github.com/akopytov/sysbench. Last accessed: 2020-09-03.

[48] Prodhomme, C., L. Batté, F. Massonnet, P. Davini, O. Bellprat, V. Guemas & F.J. Doblas-Reyes, 2016b: Benefits of increasing the model resolution for the seasonal forecast quality in EC-Earth. *Journal of Climate*, 29, 9141-9162, doi:10.1175/JCLI-D-16-0117.1.

[49] Rosas, C., Giménez, J. & Labarta, J. (2014). Scalability prediction for fundamental performance factors. *Supercomputing Frontiers And Innovations*, 1(2), 4-19. doi:10.14529/jsfi140201

[50] Casas, Marc; Badia, R. M. & Labarta, Jesus. (2008). Automatic Analysis of Speedup of MPI Applications. *Proc. 22nd Intl. Conf. on Supercomputing*, Pag. 349-358. doi:10.1145/1375527.1375578

[51] Becker, A.; Venkataraman, R.; Kale, L. V. (2009). Patterns for Overlapping Communication and Computation. *Workshop on Parallel Programming Patterns*

[52] *TOP500 List* - November 2020

[53] Gonzalez, Juan; Giménez, Judit; Labarta, Jesús. (2009). Automatic detection of parallel applications computation phases. *IEEE International Symposium on Parallel & Distributed Processing* doi:10.1109/IPDPS.2009.5161027

[54] Llort, Germán; Servat, Harald; González, Juan; Giménez, Judit; Labarta, Jesús. (2013). On the usefulness of object tracking techniques in performance analysis. *SC '13: Pro-*

ceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Article No.: 29, Pag. 1-11. doi:10.1145/2503210.2503267

[55] *Performance Analysis Tools: Details and Intelligence — BSC-Tools*
`https://tools.bsc.es`