SEVENTH FRAMEWORK PROGRAMME

FP7-ICT-2013-10

DEEP-ER

DEEP Extended Reach

Grant Agreement Number: 610476

D6.3

Final report on application experience

Approved

Version: 2.0

Author(s): A. Zitz (JUELICH), J. Morillo (BSC)

Contributor(s): R. Leger (Inria), S. Solbrig (UREG), S. Rodriguez (BSC), M. Petschow (ASTRON), A. Emerson (CINECA), F. Affinito (CINECA), G. Brietzke (BADW-LRZ), J. Amaya (KU Leuven), D. Gonzalez (KU Leuven)

Date: 04.05.2017

Project and Deliverable Information Sheet

DEEP-ER Project	**Project Ref. №:** 610476	
	Project Title: DEEP Extended Reach	
	Project Web Site: http://www.deep-er.eu	
	Deliverable ID: D6.3	
	Deliverable Nature: Report	
	Deliverable Level: PU	**Contractual Date of Delivery**: 31 / March / 2017
		Actual Date of Delivery: 31 / March / 2017
	EC Project Officer: Juan Pelegrín	

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

Document Control Sheet

Document	**Title:** Final report on application experience	
	ID: D6.3	
	Version: 2.0	**Status:** Approved
	Available at: http://www.deep-er.eu	
	Software Tool: Microsoft Word	
	File(s):DEEP-ER_D6.3_Final_report_on_application_experience_v2.0-ECapproved	
Authorship	**Written by:**	A. Zitz (JUELICH), J. Morillo (BSC)
	Contributors:	R. Leger (Inria), S. Solbrig (UREG), S. Rodriguez (BSC), M. Petschow (ASTRON), A. Emerson (CINECA), F. Affinito (CINECA), G. Brietzke (BADW-LRZ), J. Amaya (KU Leuven), D. Gonzalez (KU Leuven)
	Reviewed by:	G.Congiu (Seagate), I.Schmitz (ParTec)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	31.03.2017	Final	EC submission
2.0	04.05.2017	Approved	EC approved

Document Keywords

Keywords:	DEEP-ER, HPC, Exascale, application, porting, benchmarking, lessons learned, best practices.

Table of Contents

Project and Deliverable Information Sheet 1
Document Control Sheet 1
Document Status Sheet 2
Document Keywords 3
Table of Contents 4
List of Figures 7
List of Tables 10
Executive Summary 12
1 Introduction 13
2 Co-design 15
3 Progress of WP6 16
4 Task 6.2: Space weather (Task leader: KU Leuven) 17
4.1 Application overview 17
4.2 Optimisations 18
4.2.1 Work done during the last project year 18
4.2.2 Progress overview 22
4.3 Resiliency 23
4.4 Input/Output 28
4.5 Comparison between architectures 30
4.5.1 Comparison between processor architectures 30
4.5.2 Comparison of the different DEEP(-ER) systems 32
4.5.3 Preliminary results on QPACE3 37
4.6 Conclusion 39
5 Task 6.3: High temperature superconductivity (Task leader: CINECA) 40
5.1 Application overview 40
5.2 Optimisations 41
5.2.1 Work done during the last project year 41
5.2.2 Progress overview 44
5.3 Resiliency 45
5.3.1 TurboRVB with Scalable Checkpoint Restart (SCR) support 45
5.4 Input/Output 47
5.5 Comparison between architectures 47
5.5.1 TurboRVB on Broadwell and KNL (Marconi). 47
5.5.2 KNC and KNL single node comparison for 2degas 48
5.6 Conclusion 49
6 Task 6.4: Assessment of human exposure to electromagnetic fields (Task leader: Inria) 51
6.1 Application overview 51
6.2 Optimisations 52
6.2.1 Work done during the last project year 52
6.2.2 Progress overview 57
6.3 Resiliency 58
6.3.1 Evaluation of the persistent CP/RS overhead 58

6.3.2Checkpointing performance on the DEEP-ER SDV ... *59*
6.4 Input/Output ... **60**
6.4.1SIONlib ... *60*
6.4.2NVMe ... *61*
6.4.3I/O offload with OmpSs ... *61*
6.5 Comparison between architectures ... **62**
6.6 Conclusion ... **64**
7 Task 6.5: Rapid crustal deformation & earthquake source equation (Task leader: BADW-LRZ) ... **65**
7.1 Application overview ... **65**
7.2 Optimisations ... **66**
7.2.1Work done during the last project year ... *66*
7.2.2Progress overview ... *67*
7.3 Resiliency ... **68**
7.4 Input/Output ... **69**
7.5 Comparison between architectures ... **74**
7.5.1Scaling and per node performance measures ... *74*
7.6 Conclusion ... **76**
8 Task 6.6: Radio Astronomy (Task leader: ASTRON) ... **78**
8.1 Application overview ... **78**
8.1.1Correlation ... *78*
8.1.2Imaging ... *78*
8.1.3Mapping onto the DEEP-ER Prototype ... *79*
8.2 Progress overview ... **79**
8.3 The correlator ... **80**
8.3.1Optimisations ... *80*
8.3.2Resiliency ... *80*
8.3.3Input/Output ... *82*
8.3.4Comparison between architectures ... *84*
8.4 The imager ... **85**
8.4.1Optimisations ... *85*
8.4.2Resiliency ... *85*
8.4.3Input/Output ... *88*
8.4.4Comparison between architectures ... *88*
8.5 Conclusion ... **90**
8.5.1Acknowledgment ... *90*
9 Task 6.7: Enhancing oil exploration: Full Waveform Inversion (Task leader: BSC) 91
9.1 Application overview ... **91**
9.2 Optimisations ... **92**
9.2.1Work done during the last project year ... *92*
9.2.2Progress overview ... *93*
9.3 Resiliency ... **94**
9.4 Input/Output ... **96**
9.5 Comparison between architectures ... **97**
9.6 Conclusion ... **99**
10 Task 6.8: Lattice QCD (Task leader: UREG) ... **101**
10.1 Application overview ... **101**

10.2 Optimisations ... **102**
10.2.1 Work done during the last project year ... *102*
10.2.2 Progress overview ... *104*
10.3 Resiliency ... **105**
10.4 Input/Output ... **105**
10.4.1 HDF5 write mockup ... *105*
10.4.2 SIONlib integration ... *106*
10.4.3 E10 integration ... *108*
10.4.4 Full application I/O benchmark ... *108*
10.5 Comparison between architectures ... **109**
10.5.1 Intra node scaling ... *109*
10.5.2 Inter node scaling ... *110*
10.5.3 Full application benchmark ... *112*
10.6 Conclusion ... **113**
11 Global conclusions ... **114**
Annex A ... **116**
A.1 Best practices Guide ... **116**
A.2 Recommendation for improvements ... **126**
Annex B ... **127**
B.1 Characteristics of the SDV ... **127**
B.2 Characteristics of DEEP Cluster ... **128**
B.3 Characteristics of miclogin ... **129**
B.4 Characteristics of Marconi ... **130**
B.5 Characteristics of GALILEO cluster ... **131**
B.6 Characteristics of SuperMUC ... **132**
B.7 Characteristics of QPACE3 ... **133**
List of Acronyms and Abbreviations ... **134**
Bibliography ... **140**

List of Figures

Figure 1: Workflow of xPic (KU Leuven) 18
Figure 2: Runtime comparison between iPic3D and xPic (KU Leuven) 19
Figure 3: Runtime comparisons when using different number of dimensions (KU Leuven) ...22
Figure 4: Writing time in SINGLE checkpointing mode comparing NVMe devices and HDD in a strong scaling test (xPic, KU Leuven) 24
Figure 5: Writing time in PARTNER checkpointing mode comparing EXTOLL and TCP. NVMe devices are used (xPic, KU Leuven) 25
Figure 6: Writing time in PARTNER checkpointing mode comparing NVMe devices and HDD. Extoll is used (xPic, KU Leuven) 25
Figure 7: Resiliency overhead for weak scaling in the 1D electrostatic case (xPic, KU Leuven) 26
Figure 8: Resiliency overhead for strong scaling in the 1D electrostatic case (xPic, KU Leuven) 27
Figure 9: Impact of I/O optimisation for weak scaling (KU Leuven) 28
Figure 10: Impact of I/O optimisation for strong scaling (KU Leuven) 29
Figure 11: Comparison of the 1D mockup code on different architectures (KU Leuven) 30
Figure 12: Relative energy efficiency of the 1D mockup code in different architectures (KU Leuven)) 31
Figure 13: Performance of xPic in different DEEP(-ER) architectures (KU Leuven) 33
Figure 14: Comparison of the total runtime and the field solver runtimes for different numbers of used processes/threads (KU Leuven) 34
Figure 15: Comparison of the particle mover and the moment gathering runtimes for different numbers of used processes/threads (KU Leuven) 35
Figure 16: Weak scaling on the different DEEP(-ER) systems (KU Leuven) 36
Figure 17: Strong scaling on the different DEEP(-ER) systems (KU Leuven) 37
Figure 18: Weak scaling on QPACE3 (xPic, KU Leuven) 38
Figure 19: Workflow of TurboRVB (left) and 2degas (right) (CINECA) 40
Figure 20: Impact of OpenMP parallelisation on 2degas (CINECA) 43
Figure 21: Weak scaling of 2degas for 60-240 threads or tasks (CINECA) 43
Figure 22: Comparison between MCDRAM and main memory (DDR) using 2degas (CINECA) 44
Figure 23: Impact of SCR integration (CINECA) 46
Figure 24: Weak scaling performance of TurboRVB on the Broadwell and KNL partitions of Marconi (CINECA) 48
Figure 25: Performance of 2degas on KNC (Galileo) and KNL (SDV), for both the MPI and OpenMP versions (CINECA) 48
Figure 26: Workflow of GERShWIN (Inria) 51
Figure 27: Strong scaling comparison between MCDRAM and DDR (Inria) 52
Figure 28: OpenMP parallel efficiency comparison between KNC and KNL (Inria) 53

Figure 29: OpenMP runtime comparison between KNC and KNL (Inria) 54
Figure 30: Parallel efficiency comparison between OmpSs and OpenMP on Haswell (Inria) 55
Figure 31: Parallel efficiency comparison between OmpSs and OpenMP on KNL (Inria) 56
Figure 32: Impact of OpenMP optimisation for KNL (Inria) 57
Figure 33: Resiliency overhead (Inria) 58
Figure 34: Checkpointing performance (SCR) on SDV (Inria) 59
Figure 35: SIONlib on KNL + NVMe (Inria) 60
Figure 36: Single node walltime comparison of the time loop (Inria) 62
Figure 37: Percentage of peak performance (Inria) 63
Figure 38: Workflow of SeisSol (BADW-LRZ) 66
Figure 39: Resiliency checkpoint and restart overhead (BADW-LRZ) 68
Figure 40: Tuning of SIONlib, exploring SIONlib-I/O-options (BADWLRZ) 70
Figure 41: I/O backends in comparison on the DEEP-ER SDV (BADW-LRZ) 71
Figure 42: Tuning SIONlib I/O backend on the DEEP-ER SDV (BADW-LRZ) 72
Figure 43: I/O backends in large scale strong scaling I/O tests on SuperMUC (BADW-LRZ) 73
Figure 44: I/O on the NVMe devices using BeeOND (BADW-LRZ) 74
Figure 45: Performance results for the considered Intel CPUs: SNB/HSW/KNL (BADW-LRZ) 75
Figure 46: Workflow of the data processing pipeline (ASTRON) 78
Figure 47: MPI fault tolerance (ASTRON) 81
Figure 48: Performance of the data distribution (ASTRON) 83
Figure 49: Scaling of the corner-turn on DEEP Cluster (ASTRON) 83
Figure 50: Bandwidth comparison of global storage and NVMe devices (ASTRON) 84
Figure 51: Comparison of OmpSs and OpenMP parallelisation in the correlator (ASTRON) 85
Figure 52: Visualisation of test data set RX42 (ASTRON) 86
Figure 53: Overhead of using the SCR library (ASTRON) 87
Figure 54: Imaging for various numbers of clean cycles (ASTRON) 88
Figure 55: Roofline analysis with practical upper bounds (ASTRON) 89
Figure 56: Workflow of the FWI mockup (BSC) 92
Figure 57: Resiliency overhead (BSC) 95
Figure 58: Impact of using the NVMe devices (BSC) 97
Figure 59: Runtime comparison between KNC and KNL (BSC) 98
Figure 60: Parallel efficiency comparison between KNC and KNL (BSC) 99
Figure 61: Workflow of the analysis mode (left) and generation mode (right) (UREG) 102
Figure 62: Impact of thread affinity placement difference (UREG) 103
Figure 63: Scaling of QDP++ on BeeGFS (UREG) 106
Figure 64: Synthetic benchmark on QPACE3 comparing SIONlib and HDF5 writing time (UREG) 106

Figure 65: Synthetic benchmark on QPACE3 comparing SIONlib and HDF5 writing time depending on the number of tasks (UREG) 107

Figure 66: Synthetic benchmark on QPACE3 comparing SIONlib and HDF5 reading time (UREG) 107

Figure 67: Full application benchmark with and without MPIWRAP on the SDV (UREG) ... 108

Figure 68: Full application I/O benchmark (UREG) 109

Figure 69: Intra node scaling speedup and parallel efficiency comparison between KNC and KNL (UREG) 109

Figure 70: Intra node scaling GFlop/s comparison between KNC and KNL (UREG) 110

Figure 71: Inter node scaling speedup and parallel efficiency for KNL (UREG) 111

Figure 72: Inter node scaling walltime comparison between KNC and KNL (UREG) 111

Figure 73: Comparing Chroma on KNC and KNL (UREG) 112

Figure 74: Analysis (Step 1) 116

Figure 75: Cluster-Booster division (Step 2) 117

Figure 76: KNL optimisation – Threading (Step 3a) 119

Figure 77: KNL optimisation – Memory (Step 3b) 119

Figure 78: KNL optimisation – Vectorisation (Step 3c) 120

Figure 79: KNL optimisation – HW modes (Step 3d) 121

Figure 80: I/O optimisation (Step 4) 122

Figure 81: Resiliency support (Step 5) 123

Figure 82: DEEP Cluster at JUELICH 128

Figure 83: KNC 129

Figure 84: GALILEO cluster at CINECA 131

Figure 85: SuperMUC at BADW-LRZ 132

List of Tables

Table 1: Co-Design overview 15
Table 2: Progress overview of WP6 16
Table 3: Experiment setup for comparison between iPic3D and xPic (KU Leuven) 20
Table 4: Progress overview (KU Leuven) 22
Table 5: Experiment setup regarding resiliency (KU Leuven) 27
Table 6: Benchmarking setup regarding I/O optimisation for weak scaling (KU Leuven) 28
Table 7: Benchmarking setup regarding I/O optimisation for strong scaling (KU Leuven) 29
Table 8: Benchmarking setup for the comparison of the xPic performance on the different DEEP(-ER) systems (KU Leuven) 33
Table 9: Weak scaling benchmark setup on the different DEEP(-ER) systems (KU Leuven) 36
Table 10: Strong scaling benchmark setup on the different DEEP(-ER) systems (KU Leuven) 37
Table 11: Weak scaling benchmark setup for measurements on QPACE3 (KU Leuven) 38
Table 12: Experiment setup regarding OpenMP/MPI comparison (CINECA) 44
Table 13: Progress overview (CINECA) 45
Table 14: Experiment setup for the SCR compilation and testing on the SDV (CINECA) 46
Table 15: Benchmarking setup for the TurboRVB runs on Marconi (CINECA) 47
Table 16: Benchmarking setup for 2degas runs on Galileo (KNC) and SDV (KNL) (CINECA) 49
Table 17: Experiment setup regarding MCDRAM tests (Inria) 53
Table 18: Experiment setup regarding the comparison between KNC and KNL (Inria) 54
Table 19: Experiment setup regarding the OmpSs and OpenMP comparison (Inria) 56
Table 20: Progress overview (Inria) 57
Table 21: Experiment setup regarding resiliency overhead (Inria) 59
Table 22: Experiment setup regarding checkpointing performance on the SDV (Inria) 60
Table 23: Experiment setup regarding SIONlib on KNL + NVMe (Inria) 61
Table 24: Benchmarking setup regarding architecture comparison (Inria) 63
Table 25: Progress overview (BADW-LRZ) 67
Table 26: Experiment setup regarding resiliency (BADW-LRZ) 69
Table 27: Benchmarking setup regarding I/O optimisation (BADW-LRZ) 71
Table 28: Benchmarking setup regarding LOH4 run (BADW-LRZ) 75
Table 29: Progress overview (ASTRON) 79
Table 30: General settings for all experiments (ASTRON) 82
Table 31: A single subband of the data set RX42 (ASTRON) 86
Table 32: General settings used for SCR tests (ASTRON) 87
Table 33: Progress overview (BSC) 93
Table 34: Experiment setup regarding resiliency (BSC) 95

Table 35: Benchmarking setup regarding I/O optimisation (BSC) 97
Table 36: Benchmarking setup regarding runtime comparison between KNC and KNL (BSC) 99
Table 37: Experiment setup regarding … (UREG) 103
Table 38: Progress overview (UREG) 104
Table 39: Benchmarking setup regarding SIONlib/HDF5 comparison (UREG) 108
Table 40: Benchmarking setup regarding E10 integration (UREG) 108
Table 41: Benchmarking setup for intra and inter node scaling (UREG) 112
Table 42: Experiment setup for the full application benchmark (UREG) 113
Table 43: System information: SDV 127
Table 44: System information: DEEP Cluster 128
Table 45: System information: miclogin 129
Table 46: Characteristics of the Marconi 130
Table 47: System information: GALILEO cluster 131
Table 48: System information: SuperMUC 132
Table 49: System information: QPACE3 133

Executive Summary

The main goal of Work Package 6's final Deliverable is to summarise the work done during the project and to present the results obtained with respect to the different hardware and software components of the DEEP-ER Prototype. This document describes in detail the efforts done by application developers in the past 12 month, and it also gives an overview on the overall progress of the applications during the entire duration of the project.

Due to the delays in the deployment of the final DEEP-ER Prototype, the results were obtained mostly on the Software Development Vehicle (SDV). The SDV has the same architecture as the final prototype but a reduced number of nodes. All DEEP-ER enabled features were installed in the SDV so it could replicate, although at a smaller scale, the final hardware/software configuration. Applications could use the SDV for test cases up to 16 Cluster and 8 Booster Nodes.

In order to evaluate the DEEP-ER enabled hardware and software features, target applications were modified appropriately. Different applications required different types of optimisations depending on the used DEEP-ER features. Some of the partners focussed more on performance optimisation of their code for the Intel® Xeon Phi™ architecture; others invested more effort in the optimisation of I/O performance. In both cases the outcomes were similar: applications' performance and scalability were greatly improved.

Another achievement of the project is resiliency. Most of the applications didn't handle failures initially, whereas now they can handle and recover from failures thanks to the DEEP-ER resiliency features.

1 Introduction

This document reflects the work done and the lessons learned in Work Package 6 (WP6) during the DEEP-ER Project. In this respect the document first describes the Co-Design efforts in WP6 and gives an overview of the progress made during the whole project duration; it then summarises the changes done to each application, along with the produced results. In order to have a consistent description across the document, every application's section follows the same structure:

- *Application overview*
- *Optimisations*
  - *Work done during the last project year*
  - *Progress overview*
- *Resiliency*
- *Input/Output*
- *Comparison between architectures*
- *Conclusions*

The *"Application overview"* subsection gives a short description of the application. The "*Optimisations"* subsection explains in detail the changes and optimisations performed during the last project year. This subsection also shows the overall progress for each application and describes which features are implemented/used. If some of the features aren't used, the reason is well described. The *"Resiliency"* subsection specifies the resiliency strategy employed by the application. In this case applications use different approaches. Some of them use the application level checkpoint-restart mechanism (e.g. with SCR) while others use the task based resiliency features of OmpSs. The *"Input/Output"* subsection presents some of the I/O benchmarks that were used to evaluate the different I/O features implemented in the codes.

For the resiliency and Input/Output benchmarks WP6 worked closely together with the Work Packages 4, 5 and 7. To get a good overall picture of the resiliency and Input/Output achievements of the DEEP-ER Project each Work Package concentrates on different aspects: The resiliency and Input/Output benchmarks in this document focus on benchmarking the applications I/O performance and the resiliency overhead. A detailed analysis of synthetic benchmarks to compare the different storage devices with bigger data sizes (to avoid the cache) and long-time runs is done by Work Package 4 in Deliverable D4.5 [1]. Work Package 5 focuses on the analysis of resiliency overhead with micro benchmarks, the evaluation for directive-based checkpointing and a detailed investigation for task-based checkpointing with OmpSs and MPI (see Deliverable D5.4 [2]).

The *"Comparison between architectures"* subsection, presents benchmarking results of applications on different architectures. Here, results on different node and storage types are compared, e.g., Xeons, Xeon Phis, GPUs or HDDs and local non-volatile memory (NVMe) devices. Also some measurements on QPACE3 are shown here. The last subsection, *"Conclusions"*, contains the conclusions and also describes the main benefits the application obtained because of the project.

The document continues with a more general conclusion section that highlights, how all the applications have benefited from the project. Here, the innovative DEEP-ER features and the excellent support of the experts from all the other Work Packages are stressed. The annex provides some recommendations for improvements and a Best practices Guide.

Since the DEEP-ER Prototype was not available before the submission deadline for this deliverable, the benchmarking was done on other systems. Most of the measurements were performed on the DEEP-ER SDV Cluster (16 Haswell nodes) and the SDV KNLs (8 KNL nodes). The SDV Cluster is described in D6.2 [3]. Since June 2016, 8 KNLs are installed in the SDV. Each node of the SDV (Cluster and KNLs) has its own local NVMe device installed. The annex contains a complete list of systems used for this document.

The present deliverable summarises all the results collected until the official submission deadline, on the 31st of March 2017. Any results achieved beyond that date will be added to the slides of WP6 for the final project review meeting, which will be held in May 2017.

2 Co-design

Table 1 gives a quick overview of the main co-design achievements from M30 to M42 in which WP6 had an active participation. In addition to that all efforts from M1-M30 are described in Deliverable D6.2 [3]. The table shows that all Work Packages worked closely together to make the project success.

Topic	Participants	Outcome
Booster network topology (required number of EXTOLL links, number of links between Cluster and Booster subsystems)	WP3, WP6 (during teleconferences)	• Evaluating the 7th EXTOLL link would cost lots of additional effort. • Application developers are ok with having only 6 links. • Number of links between Cluster and Booster: Bandwidth should be as high as possible, at least as many links as in the DEEP system.
NAM use cases	WP3,WP4,WP5,WP6 (teleconferences, F2F meetings)	• Concentrate on one use case • Chosen use case: NAM should be used as checkpoint restart memory with SIONlib/SCR
NVMe testing	WP3, WP6	• Multi node tests on NVMe devices of the SDV. • Used xPic and the FWI application.
Resiliency support	WP5, WP6 (resiliency workshop at BSC, F2F meeting, teleconferences)	• Applications are resilient now. • The FORTRAN support of OmpSs task based resiliency was improved. • The SIONlib developers were notified about the problems encountered when trying to use SCR with SIONlib in FORTRAN codes. As result SIONlib will be updated to integrate SCR support for FORTRAN in the future;
I/O optimisation support	WP4, WP6	• Applications were integrated in JUBE, thus simplifying the benchmarking process of WP4 and WP5 software. • Great improvements in I/O performance for all applications. • The experience matured with BeeGFS within DEEP-ER was very helpful for tailoring platforms in other projects. E.g. it was decided to use BeeGFS for QPACE3.

Table 1: Co-Design overview

3 Progress of WP6

This section gives on overview of the progress made within WP6 during the whole project duration. Table 2 shows how the applications evolved year by year until the end of the project. The most important thing is that each feature enabled by the DEEP-ER Project is used by at least two applications. The only exception is the NAM checkpointing. By the time of writing this Deliverable, first synthetic tests demonstrate the feasibility of using the NAM to store redundancy information and restore from it in case one node fails. This preliminary results can be found in Deliverable D3.4 [4]. These developments are too recent, though, to have enough time to test them in actual applications or their mockups.

In case a feature or optimisation was not planned (marked here with a "-") or the integration was not finished (marked with "1" or "0") the application partner explains the situation in the subsection "Progress overview" of the corresponding application.

Application	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
KU Leuven	2	1	1	1	1	1	0	0	0	0	0	0	1	0	0	0	0	0	-	0	0	0	
CINECA	2	1	1	0	0	1	1	1	0	0	1	-	1	0	0	0	0	0	-	0	0	0	
Inria	2	1	2	1	1	1	0	0	0	0	0	-	1	-	0	0	0	0	0	0	0	0	
BADW-LRZ	2	1	2	1	1	1	-	-	0	0	1	0	1	0	0	0	0	0	-	0	0	0	1st year
ASTRON	2	1	1	2	2	1	0	0	0	0	-	-	1	0	0	-	0	-	-	0	0	0	
BSC	2	1	2	1	1	1	1	1	0	0	-	-	1	2	0	-	0	-	0	0	0	0	
UREG	2	1	2	1	1	1	-	-	0	0	0	0	1	-	0	-	0	-	-	0	0	0	
KU Leuven	2	2	2	1	1	1	2	2	0	0	0	0	1	1	0	0	0	0	-	0	0	0	
CINECA	2	2	2	0	0	1	2	2	0	0	2	-	1	0	0	0	0	0	-	1	0	0	
Inria	2	2	2	2	2	1	2	1	0	0	2	-	1	-	0	0	0	2	0	2	0	0	
BADW-LRZ	2	2	2	2	2	1	-	-	0	0	1	0	1	0	0	0	1	0	-	0	0	0	2nd year
ASTRON	2	2	2	2	2	1	1	1	0	0	-	-	1	0	0	-	0	-	-	0	0	0	
BSC	2	2	2	2	2	1	2	2	0	0	-	-	1	2	0	-	0	-	0	0	0	0	
UREG	2	2	2	1	1	1	-	-	0	0	0	0	1	-	0	-	0	-	-	0	0	0	
KU Leuven	2	2	2	2	2	1	2	2	2	1	2	1	1	2	2	0	1	0	-	2	0	0	
CINECA	2	2	2	2	0	1	2	2	2	2	2	-	1	2	2	0	1	0	-	2	0	0	
Inria	2	2	2	2	2	1	2	1	2	2	2	-	1	-	2	0	1	2	1	2	0	0	
BADW-LRZ	2	2	2	2	2	1	-	-	2	1	1	0	1	2	2	0	1	0	-	0	0	0	3rd year
ASTRON	2	2	2	2	2	1	2	2	2	0	-	-	1	2	2	-	1	-	-	0	0	0	
BSC	2	2	2	2	2	1	2	2	2	2	-	-	1	2	2	-	1	-	2	2	0	0	
UREG	2	2	2	2	2	1	-	-	2	2	0	2	1	-	2	-	1	-	-	0	0	0	
KU Leuven	2	2	2	2	2	2	2	2	2	2	2	1	2	2	2	0	2	2	-	2	2	2	
CINECA	2	2	2	2	2	2	2	2	2	2	2	-	2	2	2	0	2	2	-	2	2	2	
Inria	2	2	2	2	2	2	2	1	2	2	2	-	2	-	2	0	2	2	2	2	2	2	
BADW-LRZ	2	2	2	2	2	2	-	-	2	1	2	1	2	2	2	0	2	1	-	0	2	2	End of the project
ASTRON	2	2	2	2	2	2	2	2	2	1	-	-	2	2	2	-	2	-	-	1	2	2	
BSC	2	2	2	2	2	2	2	2	2	2	-	-	2	2	2	-	2	-	2	2	2	2	
UREG	2	2	2	2	2	2	-	-	2	2	1	2	2	-	2	-	2	-	-	0	2	2	

Table 2: Progress overview of WP6

1 = Analysis
2 = Writing D6.1
3 = Porting code to KNC
4 = Threading
5 = Vectorisation
6 = General code optimisation
7 = OmpSs (paralellisation/Offload)
8 = Cluster-Booster division
9 = Porting code to SDV
10 = Use NVM
11 = Integrate SIONlib
12 = Integrate E10
13 = General I/O optimisations
14 = Implementing a mockup
15 = Writing D6.2
16 = NAM checkpointing
17 = Optimise code for KNL
18 = Implementing SCR
19 = Implementing task based resiliency
20 = Integrate application in JUBE
21 = Final benchmarking
22 = Writing D6.3

Legend	
-	Not planned
0	Planned, but not startet yet
1	Work in progress
2	Finished

4 Task 6.2: Space weather (Task leader: KU Leuven)

In deliverable D6.2 [3] a mockup code was introduced to study the performance of the Particle-in-Cell method for the simulation of astrophysical plasma physics. The mockup showed great performance for the basic simulation of astrophysical plasmas, in one compute node, using OpenMP multi-threading and SIMD vectorisation.

Now the code xPic (exa-scale/multi-method Particle-in-Cell code) will be presented. xPic is the new code from KU Leuven that will replace, in the near future, the older iPic3D code. xPic includes all the accomplishments achieved with the mockup and presented in the previous Deliverable and adds new features:

- Three levels of parallelism: vectorisation with SIMD, multi-threading with OpenMP and multi-node execution with MPI.
- Multiple I/O options: ASCII, HDF5 and SIONlib files.
- Resiliency using SCR and SIONlib.
- Dynamic field solver based on PETSc.
- Fully electromagnetic particle mover.
- Symmetric and asymmetric running modes.

In the “asymmetric” running mode the code xPic runs as a classic parallel code with the two main sections of the code (the field solver and the particle solver) running sequentially in the same nodes. In the “symmetric” mode, the code is split in two, one section is run in one array of compute nodes, and the second section is run in a different set of nodes. This splitting, inspired by the Cluster-Booster architecture, allows to commit more resources to the time consuming particle solver, while the communications intensive field solver runs in a different machine. The field solver and the particle solver can run either in the DEEP-ER Cluster (Xeon), in the DEEP-ER Booster (Xeon Phi) or in a combination of both. However, the field solver requires the use of the PETSc library, which has not yet been optimised to take advantage of the vector registers in the Xeon Phi processors.

The new generation of Particle-in-Cell code, xPic, has better performance than the original iPic3D code. Moreover its internal structure provides flexibility for the development of new numerical methods, and for the inclusion of new I/O libraries and resiliency tools. The code can work as a platform to test new I/O and resiliency technology on current Petascale, and future Exascale architectures.

4.1 Application overview

The xPic code was developed to take advantage of the vector registers in the new generation of Intel architectures, including the Intel Xeon and the Intel Xeon Phi processors. The iPic3D code, predecessor of xPic, was tested and analysed prior to the conception of xPic. The idea was to adapt iPic3D to the new architecture. However, two issues prevented us from easily implementing vectorisation and multi-threading in iPic3D: a) the data structure of the particle solver had complex communication patterns, difficult to adapt loops and interconnected gather and mover routines, and b) the field solver and the particle solver were too tightly interconnected, sharing data structures and subroutine calls.

Before moving forward with the project it was decided to study in detail the vectorisation and multi-threading routines in a simple version of the code. A small 1D electrostatic “mockup” code was created that emulates the behaviour of the particle solver of iPic3D. The mockup

was adapted, using basic optimisations, to different architectures, including Xeon, Xeon Phi and GPU. Results are shown in Section 4.5.

The possibility of extending the mockup code was explored. Using the PETSc library a fully coupled field/particle solver system was developed quickly. However, the grand challenges were to include MPI communications for multi-node simulations, and to include a multi-dimensional grid. It was considered that such an effort was worth the investment: more and more supercomputers make use of modern multi and many-core processors with vector registers. To stay in the edge of scientific discovery it is needed to adapt the technology too. The “mockup” evolved into what is called today xPic.

The xPic code has the same algorithmic structure than its predecessor, iPic3D. It is composed of three main phases: the particle mover, the moment gathering and the field solver. Figure 1 shows this general structure. However, there is a clear difference between iPic3D and the new xPic: the particle solver and the field solver have a clear boundary. No information or data structure is shared between the two sections and data exchanges are handled by an interface that only contains basic 1D arrays of floats. This division of work allows for a better distribution of resources (human and computational).

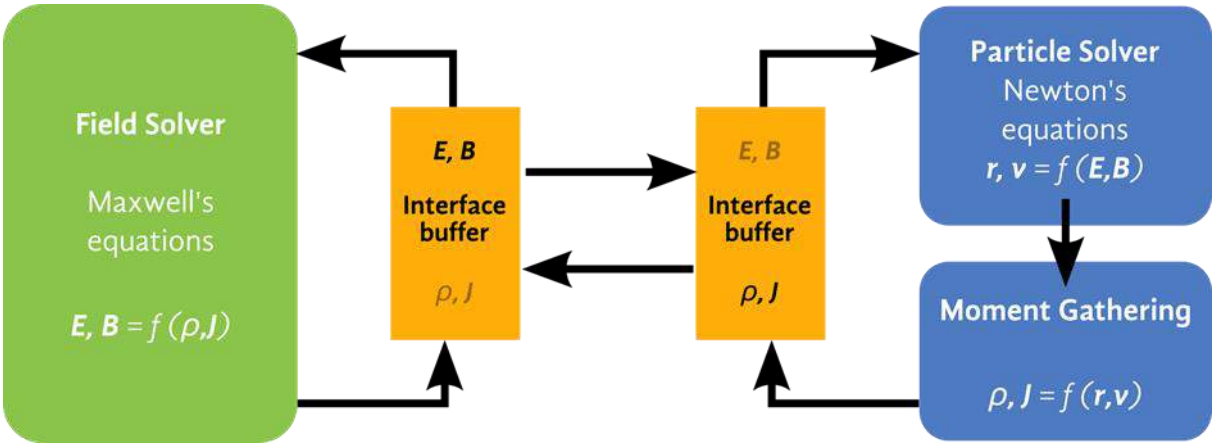


Figure 1: Workflow of xPic (KU Leuven)

4.2 Optimisations

As it was mentioned in the introduction, the focus of the second year of the project was on the optimisation of the algorithms in the code. During the last (third) year the most important improvements of the code involved the introduction of MPI to run on multiple nodes, the division of work for the Cluster-Booster architecture, the extension of the code to full 3D geometries, and the use of resiliency tools.

4.2.1 Work done during the last project year

4.2.1.1 MPI

The xPic code features a three-level parallelisation of the particle data structure: Vectorisation, multi-threading and message-passing are used for multiple-particle, multiple-block and multiple-domain processing respectively. From the top down, all the particles in the domain are divided into multiple MPI subdomains. Each one of these MPI subdomains can be mapped to a number of cores between 1 and `Nmax`, where `Nmax` is the maximum number of cores in one node. In the case of the DEEP-ER Cluster one MPI process can be mapped to up to 24 cores.

Each MPI subdomain is again subdivided into blocks. Each MPI process will be in charge of a list of blocks to compute. The blocks are distributed among the available cores in the MPI partition using OpenMP multi-threading. The processing of data inside each block takes advantage of vectorisation.

MPI communications are performed among blocks and not among MPI subdomains. This means that each OpenMP thread can perform MPI communications independently in a MPI subdomain. This is known as the multi-threaded (mt) MPI. It requires the use of the `MPI_Init_thread` subroutine with the argument `MPI_THREAD_MULTIPLE`.

During initialisation, each block detects for the data transfer mode; some communications have to be performed via memory-copy, others via MPI processing. During runtime, each block at the border of an MPI subdomain performs its own communications with neighbouring blocks in a different MPI sub-domain.

MPI communications in the field solver are handled by the PETSc library. This highly optimised mathematical library does not use OpenMP multi-threading. Instead it uses pure MPI with neighbourhood collectives and shared memory. PETSc argues that this approach gives a better consistency and performance than alternative methods.

Figure 2 shows a simple comparison of the runtimes between the iPic3D code and the new xPic code. In all cases the later gives better performances than the former. However, the speedup presented here has been obtained without additional optimisations of the xPic code. After the latest modifications and upgrades of the xPic code it was necessary to perform additional analysis to get the best performances possible. It should be noticed, that there is still some potential to increase the speedup of the xPic code. In particular, as it will be shown in the next sections, the field solver only uses two cores per node in these tests, which slows down the total runtime. The Cluster-Booster division allows to accelerate the code by efficiently use the available resources. The simulations presented in this figure use the setup provided in Table 3.

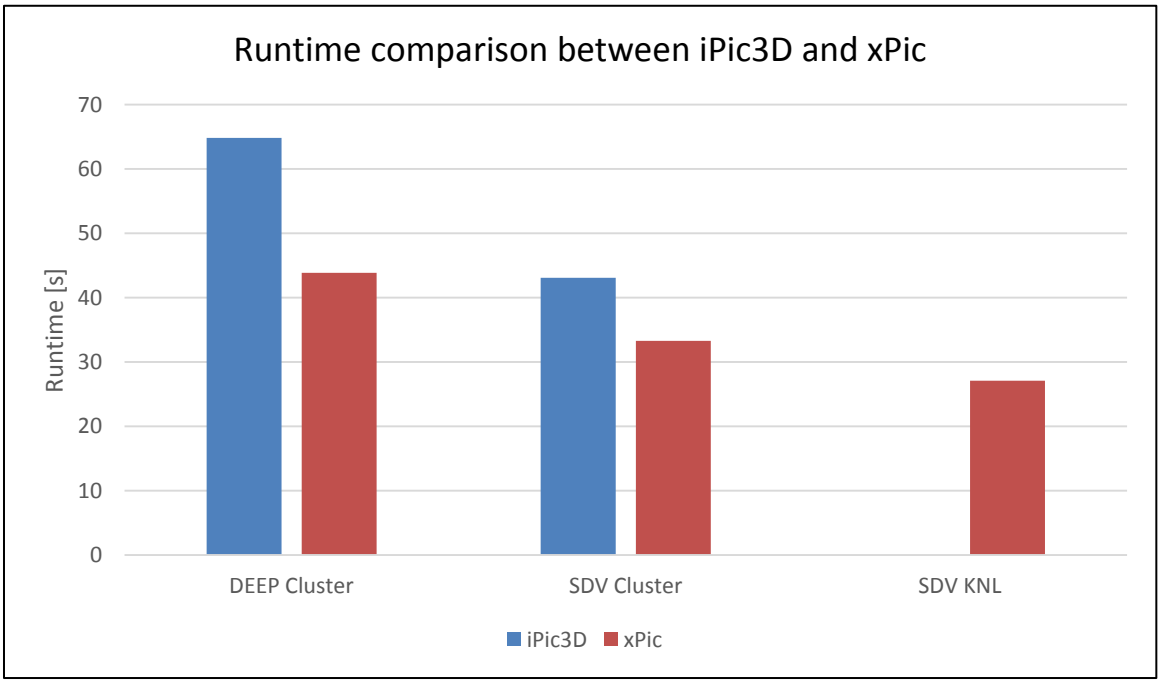


Figure 2: Runtime comparison between iPic3D and xPic (KU Leuven)

Experiment details	65536 total cells 1000 particles per cell
Used system	DEEP systems
Compiler version	Intel/16.3
MPI runtime versions	parastation/intel-mt-5.1.4-1_1_g064e3f7
Compilation flags	Multi-threading: -openmp SIMD vecotrisation: -mavx
MPI processes per node	2
Threads per process	8

Table 3: Experiment setup for comparison between iPic3D and xPic (KU Leuven)

4.2.1.2 Cluster-Booster division

The xPic code has been divided in two separate entities, as shown in Figure 1. Data exchanges are performed across a buffer of basic 1D arrays. This is done to avoid complex data structures and accelerate the data exchange. When the code runs in the “asymmetric” mode, using only one architecture (e.g. in the SDV Cluster), the buffers are allocated on the particle solver side, and the vectors in the field solver side are initialised as pointers to the vectors allocated on the particle side. This way there is no duplication of data among the solvers.

In the “symmetric” mode each solver runs in different nodes. Both solvers perform a local allocation of memory for the interfacing buffer, and data transfer is handled by MPI communications. For synchronisation reasons, all communications are point-to-point, un-buffered, and the receive calls are non-blocking. This allows to overlap procedures that do not need to be synchronous, e.g. the output of files is done in the particles solver, while the field solver is computing.

To activate the Cluster-Booster mode (symmetric) a compilation flag has been added to the code. Pre-processing macros have been included in the sources, and the system only compiles the chunks of code related to the selected mode (symmetric or asymmetric). For the symmetric mode the compilation generates two executables, each one cross-compiled for their target architecture with the appropriate flags. Each executable will run natively in each part of the system (hence the name “symmetric”). It is important to note that the symmetric mode can also be run in the same architecture, i.e. Cluster-Cluster mode or Booster-Booster mode.

The starting sequence of the symmetric mode has the following steps:

1. The particle solver is launched in the Booster.
2. The particle solver calls the `MPI_Comm_spawn` to launch the field solver in the Cluster architecture.
3. The field solver is launched in the Cluster, it detects that it is the child process and performs the data initialisation.
4. The particle solver initialises its data structures and waits for the initial data from the field solver.

5. The main computing loop starts in both solvers and data is exchanged once in each direction at every iteration.

Currently, one MPI process in the particle solver can only be mapped to one MPI process in the field solver. This means that the field solver runs only in as many cores as MPI processes are selected per node. This is caused by the fact that the PETSc library does not run using multi-threading but only MPI. It is clear that one future addition to the code will be to map one MPI process in the particle solver to multiple MPI processes in the field solver. This will allow for an ideal use of the available resources. This solution is not yet implemented.

4.2.1.3 Multi-dimensional runs

From the beginning of the xPic design it was decided to implement all data structures in a simple way in order to easily extend its use to 3D geometries. In the last year the missing auxiliary structures, book-keeping routines and data-transfers required to implement the 3D version of the code were added.

However, running 1D, 2D or 3D cases requires different communication strategies. Moment-gathering is based on the projection of information from the particles to their neighbouring nodes. Nodes at the border of the block domain require then the particle information in the last cell of the neighbouring block. Particles from the last cell are duplicated in the ghost cell of the adjacent block. However in 1D, this procedure is not necessary in the y and z directions. Data movement can then be avoided.

Nevertheless, interpolations from particles to the nodes, and vice versa, require information on both sides of the nodes. 1D and 2D simulations are in fact “flat” 3D simulations, where one (or two) of the dimensions has only one cell in depth. Information about the particles is not transferred among blocks in the flat directions, but a periodicity condition must be enforced to obtain the correct values of the interpolation. In 2D and 1D simulations, after the moments are gathered in each node, the values of the moments must be added in the flat dimensions and shared among the local nodes.

Let’s take the example of a 1D simulation. At the position x_i, the grid has one cell in the y and z direction. Four nodes share the same position xi. Once the moments are gathered in each one of these nodes, due to the periodicity condition in the y and z directions, they must be added and then copied in each one of them. This procedure is currently un-vectorised in the xPic code, and can cause slow memory access. In 2D simulations the problem is similar, but the number of nodes involved is only 2.

Figure 3 shows that the runtime of the application changes slightly depending on the number of dimensions of the problem, even when the data load is exactly the same in all cases. This particular simulation shows two phases with lower performances: the field solver, using the PETSc library, runs faster in 3D simulations, while the particle mover slows down for 1D simulations. More measurements are necessary to explain these results, but it is believed that the constant crossing of particles in the flat dimension causes a high load of memory copies inside a single block. The periodic condition in the flat direction must be studied with more detail.

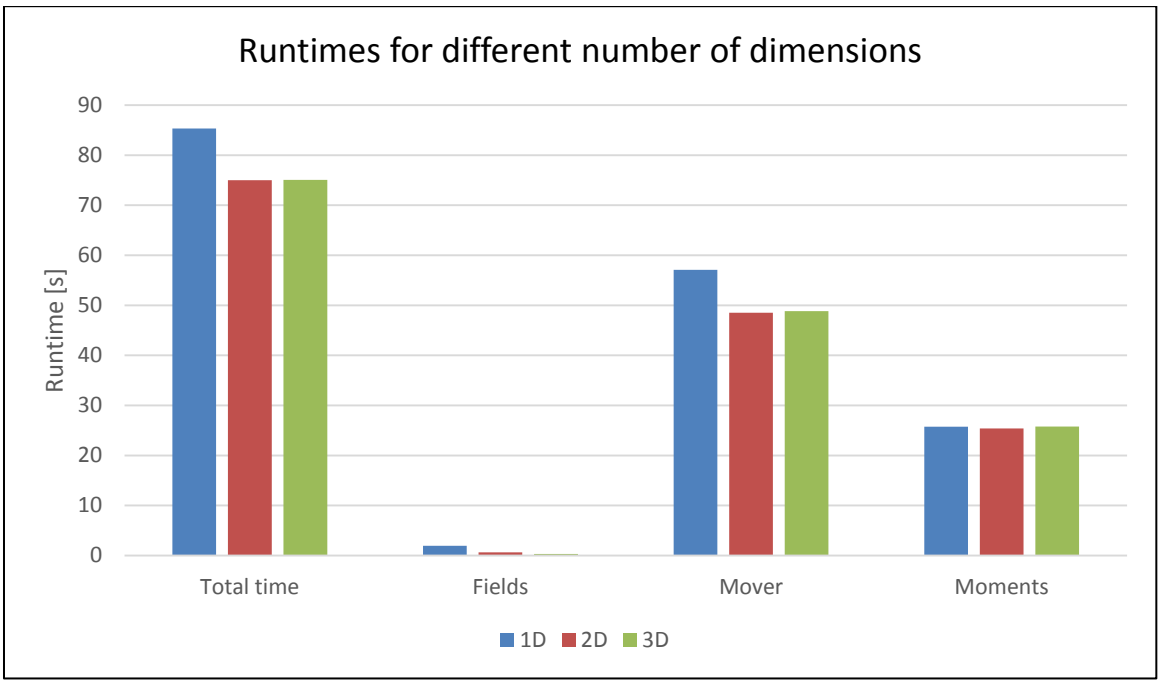


Figure 3: Runtime comparisons when using different number of dimensions (KU Leuven)

4.2.2 Progress overview

Workflow elements	M1-M6	M6-M12	M13-M18	M19-M24	M25-M30	M31-M36	M37-M42
Analysis	1	2	2	2	2	2	2
Writing D6.1	0	1	2	2	2	2	2
Port to KNC	0	1	2	2	2	2	2
Threading	1	1	1	1	2	2	2
Vectorisation	1	1	1	1	1	2	2
General code optimisation	1	1	1	1	1	1	2
Implement OmpSs parallelisation and/or offload	0	0	2	2	2	2	2
Cluster-Booster division	0	0	2	2	2	2	2
Port to SDV	0	0	0	0	2	2	2
Use NVM	0	0	0	0	0	1	2
SIONlib integration	0	0	0	0	1	2	2
E10 integration	0	0	0	0	0	1	1
General I/O optimisation	1	1	1	1	1	1	2
Implementing a mockup	0	0	0	1	1	2	2
Writing D6.2	0	0	0	0	2	2	2
Checkpointing on NAM	0	0	0	0	0	0	0
Optimise for KNL	0	0	0	0	0	1	2
Implement SCR	0	0	0	0	0	0	2
Implement OmpSs task based resiliency	-	-	-	-	-	-	-
JUBE integration	0	0	0	0	0	2	2
Final benchmarking	0	0	0	0	0	0	2
Writing D6.3	0	0	0	0	0	0	2

Table 4: Progress overview (KU Leuven)

Table 4 shows the progress of our work along the duration of the project. Almost 100% of the tasks assigned to this project are completed. In the last months, big advances were made in the specific topic of I/O and resiliency, two of the DEEP-ER main objectives. There were a

strong collaboration with the developers of SIONlib and SCR to integrate the two libraries in the code. There are however three tasks that require special attention.

The libNAM library for accelerating computation was not yet available. One of the original ideas of Task 6.2 was to use the NAM architecture to offload the computation of non-critical values, like the calculation of the total kinetic energy of the particles, or the total magnetic energy of the domain. However, it turned out that the memory requirements for such calculations exceed the memory available in the NAM prototype. Moreover, the operations required for such computations have to be specifically implemented in the FPGA. It was then planned to use the libNAM library as a support for the SIONlib library to speed up the I/O runtimes. Once it will be available, its use should be transparent to the application, and it can be considered that this task is almost completed from the applications point of view.

Initially it was not planned to use SCR for the resiliency of the code: the in-code I/O and restart methods allow performing soft restarts (restart from a previously saved file) to cover in the case of a hardware failure. However, it was decided to include SCR in our code to comply with the resiliency approach of the DEEP-ER Project. The restart is based on an interfacing between SCR and SIONlib. This proved to be easy to deploy and extremely useful for the users of the code: When restarting, the code can automatically detect previous runs and pick up the simulation at the latest checkpoint. Since SIONlib can internally exploit libNAM for checkpointing, the application can use it transparently with SIONlib.

Finally, the application doesn't use the OmpSs programming model. An analysis, carried out at the beginning of the project, compared the OmpSs offloading vs the spawning method, revealing that the later was more suitable for the structure of our application. The particle solver is the core of the xPic code: this section of the code must be micro-managed to extract the best performances from the different processors. The data structure suggested that the best approach was to divide the compute phases in two separate and independent solvers for the fields and the particles. The initialisation and setup of the simulation is done independently by each solver. This simplifies the Cluster-Booster division and the development process. For these reasons the Cluster-Booster division is performed using an `MPI_Comm_spawn` to divide the code in two sections that run natively in each architecture, using standard MPI calls for communication.

4.3 Resiliency

A typical run of xPic uses thousands of iterations, and can take several days to finish. The resiliency strategy in case of an error is to write checkpoints every certain number of iterations, so the simulation can be restarted from these files in case of a failure. Each node requires two files: one with the information from the particles and other with the data from the fields. This means that for each checkpoint two files are written per node.

In the current version of the code the SCR library is used to perform this task. This library supports two different levels of resiliency. In the simplest mode (called SINGLE), the data required for the restart is stored locally in each node. Hence no network communication between nodes is needed. The drawback of this strategy is that if one node fails and cannot be recovered, the restart is not possible. The alternative is called (PARTNER), in this case each node not only writes the data locally but also in another node (its partner). If one node fails and its information is lost, the simulation can restart from the data stored in its partner. The user can choose both methods in the input file. In both cases the data is written using the SIONlib library.

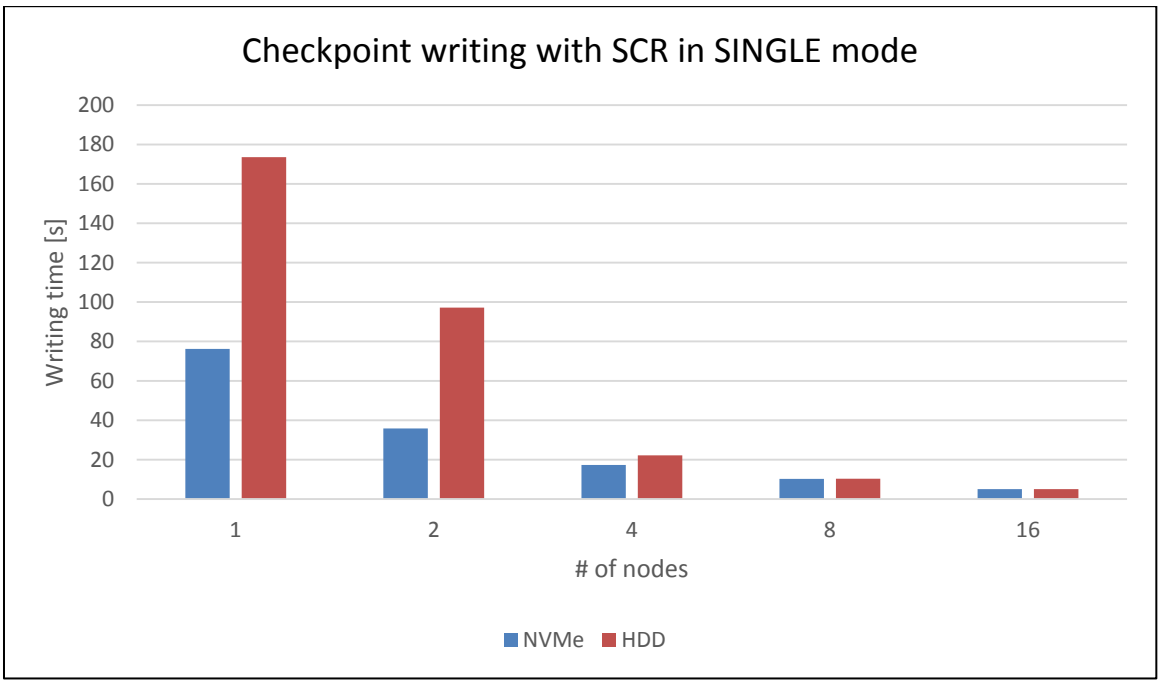


Figure 4: Writing time in SINGLE checkpointing mode comparing NVMe devices and HDD in a strong scaling test (xPic, KU Leuven)

Figure 4 shows the time spent in writing all of the 11 checkpoints to the local storage, using the SINGLE mode in two operational cases: when checkpoints are stored in an NVMe device or in a regular HDD (/tmp). The total size of the written data is 8GB for each checkpoint. As the number of nodes increase the differences between NVMe device and HDD decreases. Since the write operation is done locally (there are no communications involved) two factors may impact the writing time: the size of the checkpoints and the total number of files written per checkpoint. In Figure 4 the benchmark running in 8 nodes shows the same writing time for both devices. In this case each node writes two files, one small file (about 10MB) for the field information, and one big file (about 1GB) for the particles information. For 8 nodes the total writing time for each checkpoint is about 1 second (dividing the value in the figure by 11 checkpoints). It possible that the observed writing times are due to the system calls to open and close the files, but this still needs to be investigated.

Figure 5 shows the time spent in writing the checkpoints in PARTNER mode. In this case, each node needs to communicate data with one other node (his partner). The network plays an important role here. The figure shows the total writing time of the code when using EXTOLL and TCP over Ethernet, in both cases NVMe storage devices are used. It can be seen that EXTOLL provides much faster communications and hence it considerably reduces the checkpointing time.

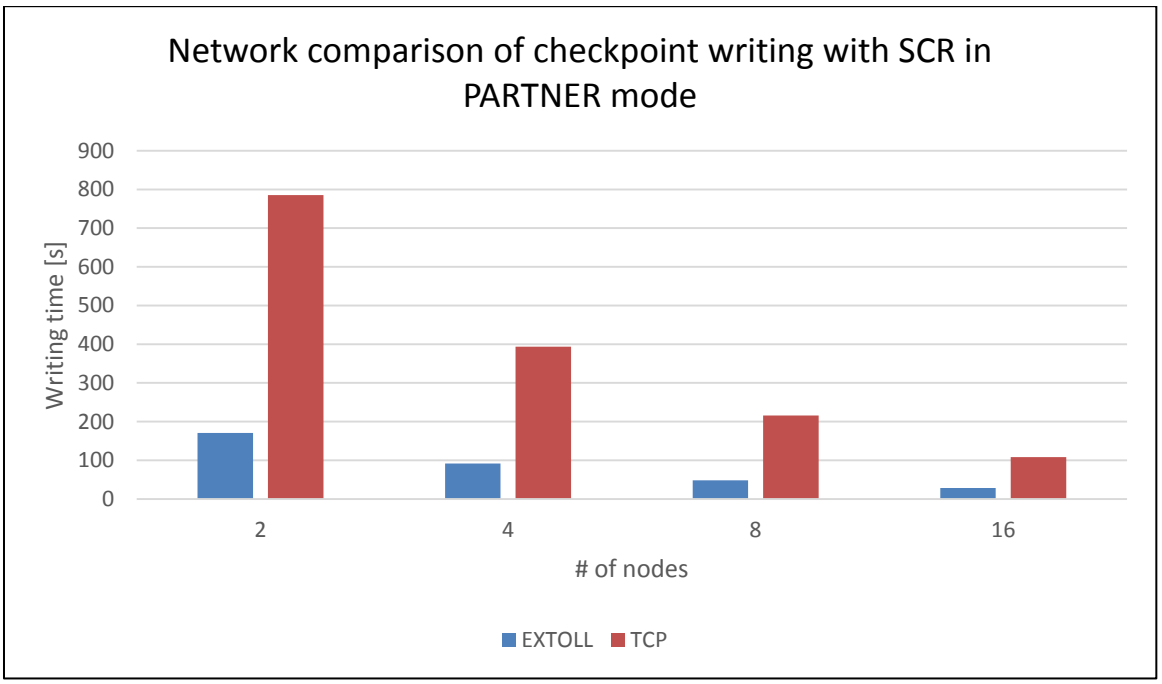


Figure 5: Writing time in PARTNER checkpointing mode comparing EXTOLL and TCP. NVMe devices are used (xPic, KU Leuven)

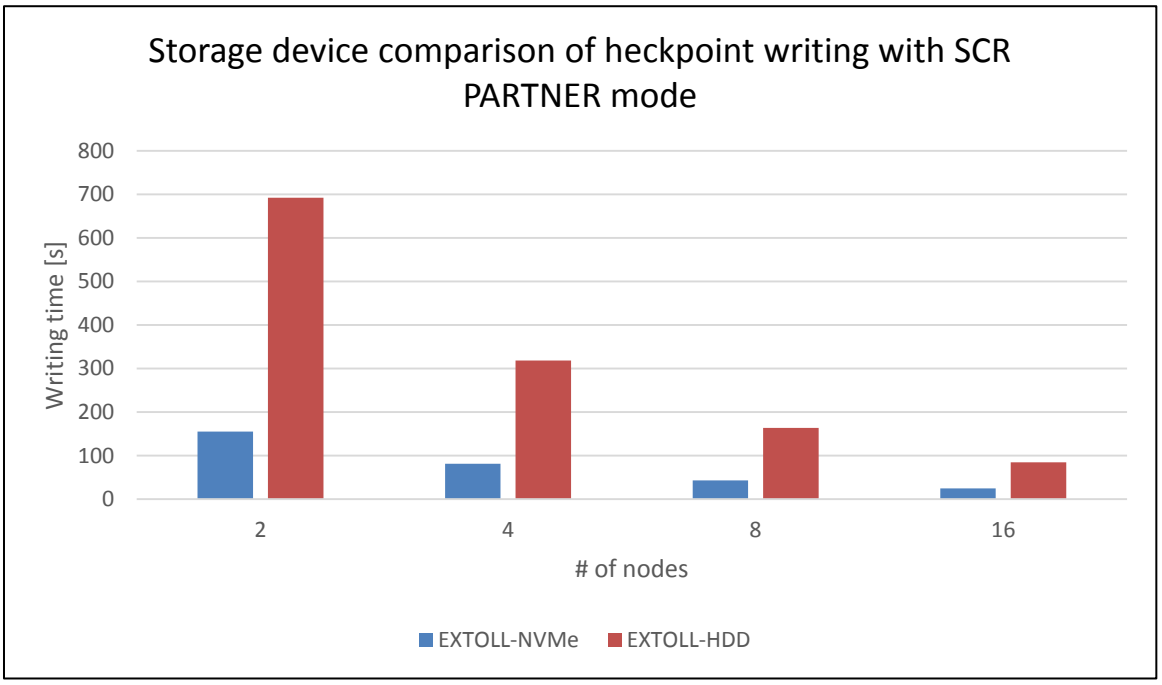


Figure 6: Writing time in PARTNER checkpointing mode comparing NVMe devices and HDD. Extoll is used (xPic, KU Leuven)

Figure 6 shows the writing time of the checkpoints in PARTNER mode when using NVMe devices and the regular HDD. In both cases EXTOLL has been used. In this case, even with the same problem the size each node has to write twice the amount of data (its own checkpoint and the checkpoint of its partner). This could explain the difference between the NVMe devices and the HDD which are not observed when using SINGLE checkpointing (Figure 4). In conclusion the use of EXTOLL together with the NVMe devices considerably speeds up the writing of checkpoints.

The file size for this strong scale testing, with the PARTNER mode, is double the size than those obtained with the SINGLE mode. We expect that writing time should be twice in the former than the later. However, comparing with the first figure of this section an increment of around 2.4x can be observed. Having a deeper look into SCR explains the additional 0.4. SCR only works on the files and doesn't see the communication layer. So for the PARTNER mode SCR has to first write the data to the node, then it reads this data again and writes it on the partner node. So here a factor between 2 and 3 would be expected. The reading will mostly happen in cache, that's why the factor should be a little below 3. And that's exactly what can be seen here. However, at 16 nodes, the NVMe device still has better performance than the HDD. This is a contradiction with the run in the SINGLE mode, where already at 8 nodes, the two devices have the same writing time. The measured times on the NVMe devices in both cases are as expected, but the times measured for the HDD especially in the SINGLE mode plot show cache effects. A detailed analysis of the cache effects is done in D4.5 [1].

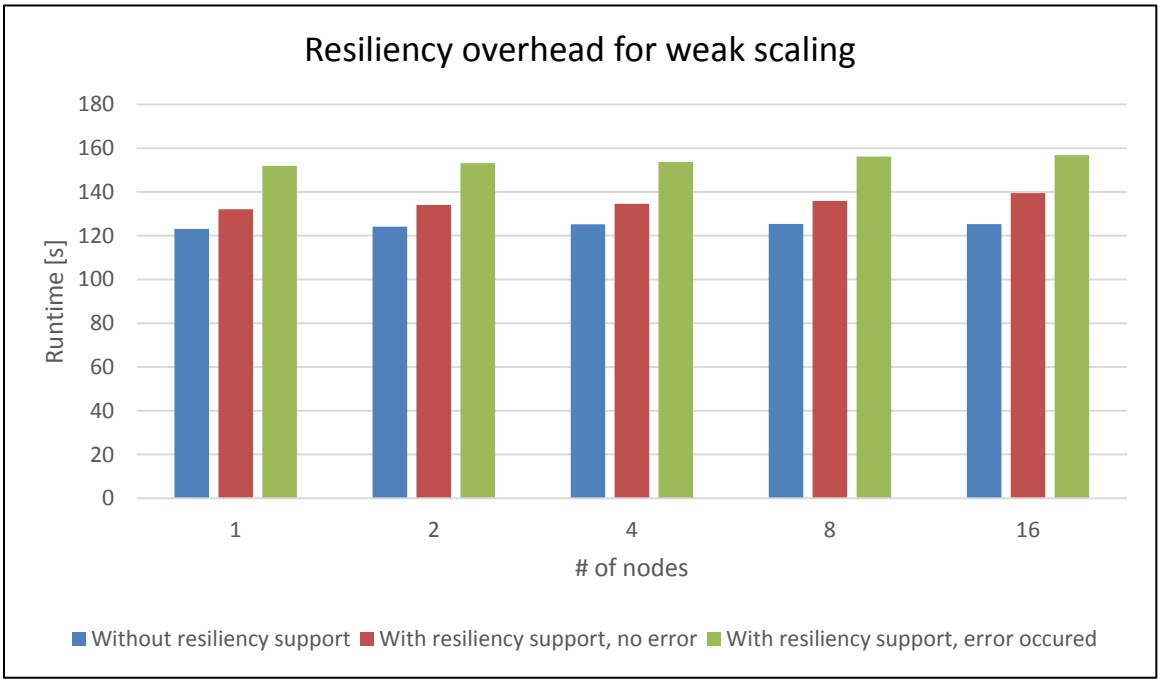


Figure 7: Resiliency overhead for weak scaling in the 1D electrostatic case (xPic, KU Leuven)

Figure 7 and Figure 8 show the resiliency overhead for weak and strong scaling. Here the total runtime for three situations is plotted: a) code without resiliency, b) code with SINGLE resiliency, and c) code with resiliency including a forced error and restart. The error is triggered by the MPI process with rank 0. When it reaches the iteration number 60 it calls the `MPI_Abort()` function. In this benchmark, 100 iterations have been computed and a checkpoint has been written every 25 iterations. This corresponds to a very strong resiliency: in case of failure only 25 iterations will be lost (usually one iteration takes only a few minutes). Even with this high frequency checkpointing, the weak scaling test shows that the overhead is only about 10% of the runtime. The strong scaling test shows only a 5% overhead.

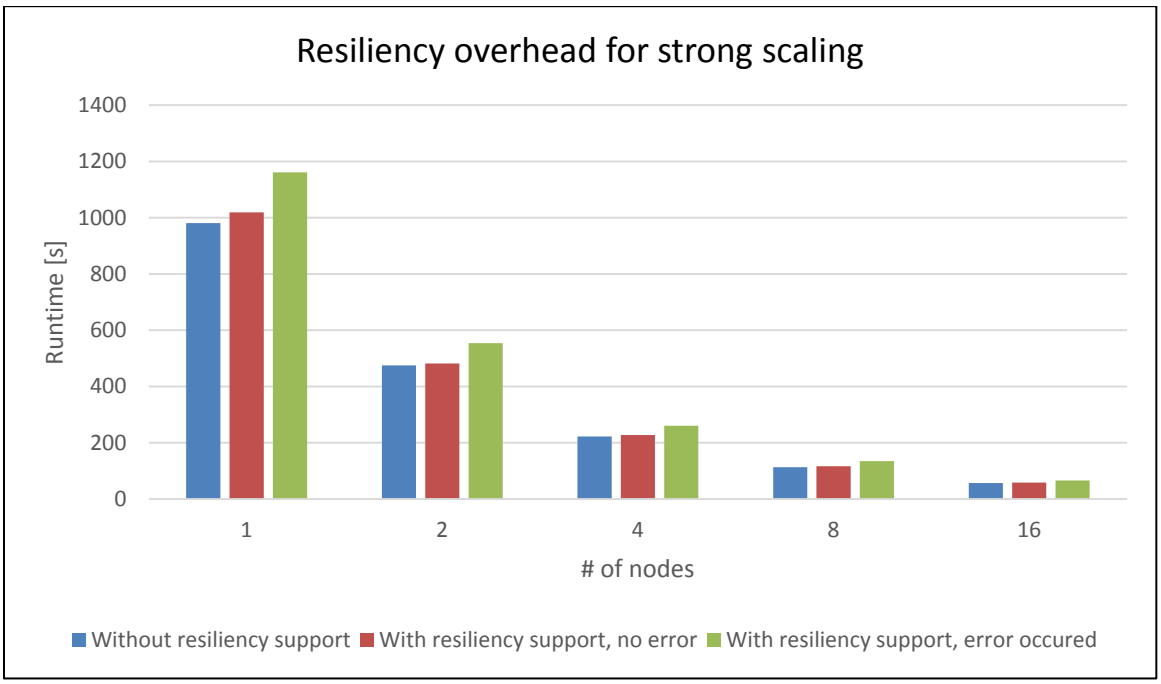


Figure 8: Resiliency overhead for strong scaling in the 1D electrostatic case (xPic, KU Leuven)

Experiment details	Homogeneous plasma in 1D with 2 species Particles per cell: 1024
Number of cells	Figure 4 - Figure 7: 64 x 64 x 64 = 262144 cell (total) Figure 8: 64 x 64 x 32 = 131072 cells per node
Number of checkpoints	Figure 4 - Figure 6: 11 Figure 7 - Figure 8: 4
Number of cycles	Figure 4 - Figure 6: 11 Figure 7 - Figure 8: 100
Used system	DEEP-ER Cluster (SDV)
Compiler version	Intel/2016.2.181
MPI runtime versions	Impi/5.1.3
Compilation flags	OpenMP: -qopenmp Xeon vectorisation: -mavx
MPI processes per node	2
Threads per process	12
Describe the error (location, time, ..)	At cycle 60, the first time the job is launched the process with rank 0 remove all the local checkpoints and call MPI_Abort().

Table 5: Experiment setup regarding resiliency (KU Leuven)

4.4 Input/Output

In the current version of the code (xPic) either SIONlib or HDF5 can be used to write both the fields and the particles. In this section the iPic3D code, the predecessor of xPic, is used as a point of reference for comparisons. In iPic3D the I/O is done with the h5hut library, which uses the HDF5 format. In all cases the output is written to /sdv-work, which implements the BeeGFS file system. The results for weak scaling are shown in Figure 9 while the ones for strong scaling can be found in Figure 10. All the I/O procedures used to do the tests work in parallel, which means that all processes write to a single file.

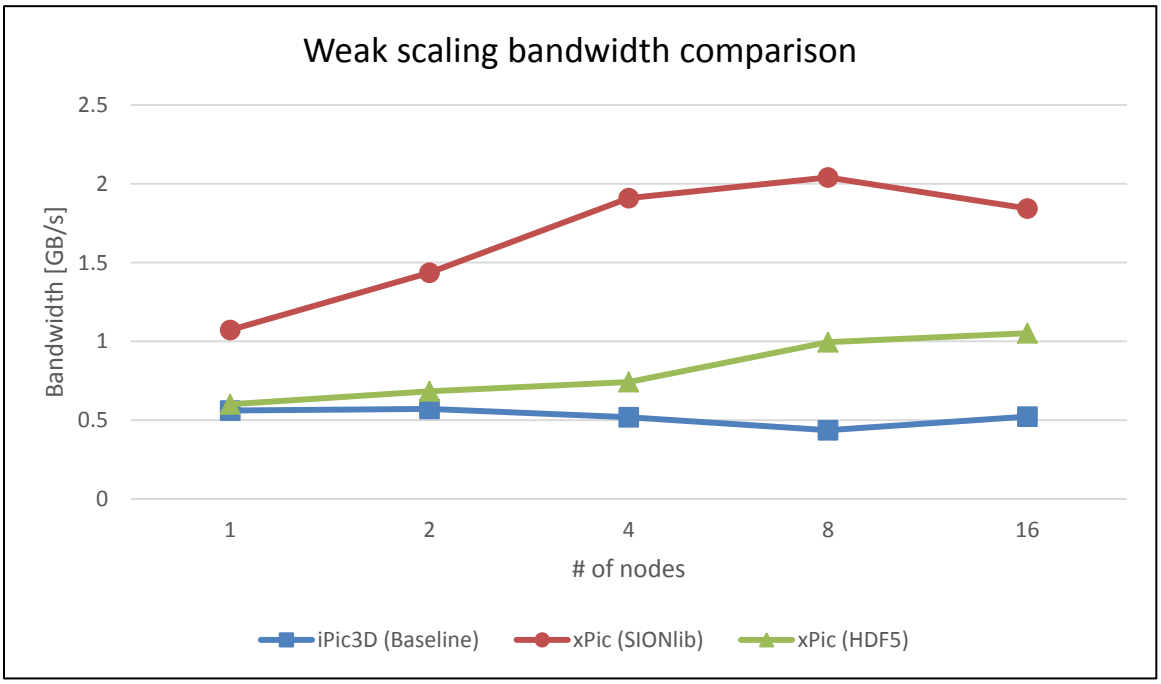


Figure 9: Impact of I/O optimisation for weak scaling (KU Leuven)

Scaling	Weak scaling
Number of cells	64 x 32 x 32 = 65536 cells per node
Other experiment details	Particles per cell: 1024 Number of particles files: 11 Number of field files: 11 Total size of data written: 22 GB/node
Compiler version	Intel/2016.2.181
MPI runtime version	Impi/5.1.3
Compilation flags	OpenMP: -qopenmp Xeon vectorisation: -mavx
MPI processes per Node	2 MPI processes
Threads per process	12 OpenMP threads

Table 6: Benchmarking setup regarding I/O optimisation for weak scaling (KU Leuven)

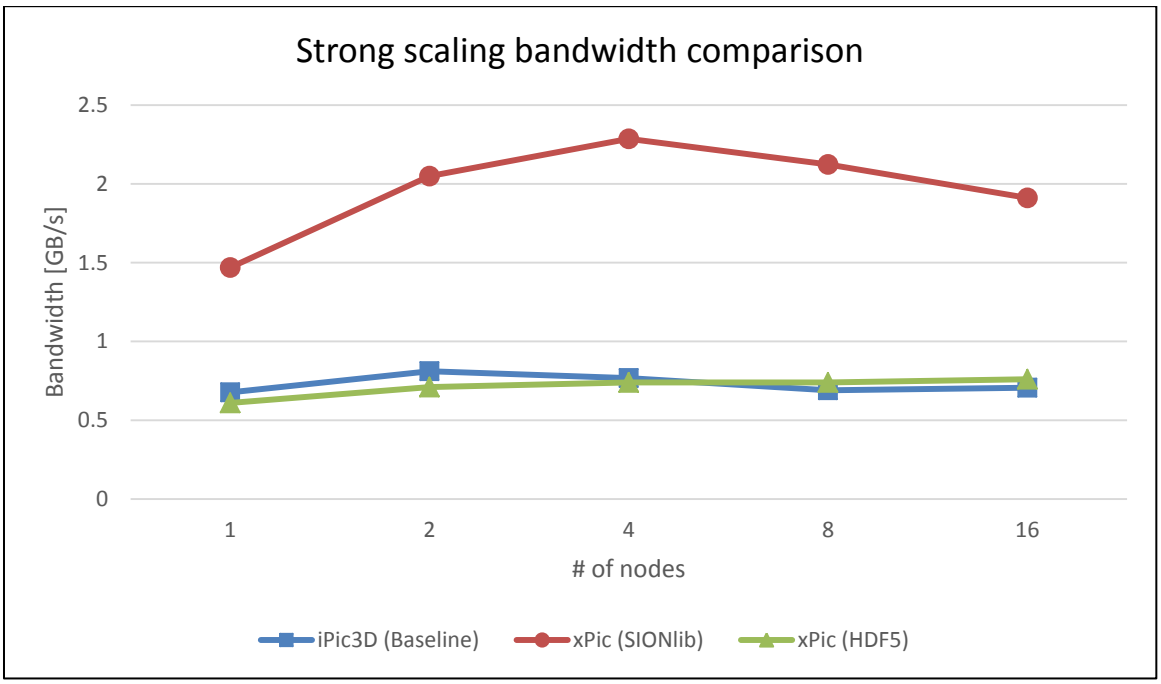


Figure 10: Impact of I/O optimisation for strong scaling (KU Leuven)

Scaling	Strong scaling
Number of cells	64 x 64 x 64 = 262144 cells
Other experiment details	Particles per cell: 1024
Compiler version	Number of particles files: 11
MPI runtime version	Number of field files: 11
Compilation flags	Total size of data written: 90 GB
MPI processes per Node	Intel/2016.2.181
Threads per process	Impi/5.1.3

Table 7: Benchmarking setup regarding I/O optimisation for strong scaling (KU Leuven)

In both cases SIONlib is the fastest method and the HDF5 from xPic has the same performance as the h5hut implemented in iPic3D (baseline). It can be observed that the bandwidth achieved with SIONlib increases with the number of nodes, until a maximum bandwidth is reached (around 2GB/s). It is important to note that SIONlib is particularly efficient when many processes are used. SIONlib enables all processes to efficiently write results in the same file, so that only one file is created. This fact can save a lot of time when many processes are used, since the creation of one file per process can be as expensive as the writing of the data itself. The advantages of SIONlib at very large scale do not manifest in our results, since our testing platform (the SDV) has only 16-nodes. But it should be noticed that in the xPic code the I/O time has been improved up to a factor 4 with respect to the original code iPic3D.

The bandwidth obtained with HDF5 increases with the number of nodes in the weak scaling test. However the bandwidth is always smaller than the one achieved by SIONlib. For the

h5hut library in iPic3D, the bandwidth is almost constant in both cases and always smaller than the other two.

4.5 Comparison between architectures

4.5.1 Comparison between processor architectures

The first tests of the initial code, iPic3D, using multi-threading and vectorisation showed a bad performance on Xeon Phi processors. It was decided to investigate if such performances were caused by the code or by the architecture. In order to gain better insights in the iPic3D code, a “mockup” code of the particle solver was developed. The mockup used an algorithm very similar to iPic3D with two main differences: it solved an electrostatic case (without magnetic field) and it was 1D.

The mockup allowed to perform simulations of the electrostatic two stream instability in plasmas. Figure 11 on the left shows the particles of the simulation in the phase space (position of the particles vs. velocity of the particles) at six different cycles. Particles in blue have an initial positive velocity and particles in red have an initial negative velocity. This counter flow of charged particles creates an electric field that grows and feeds back on the particle motion. Particles gain momentum in the opposite direction and eventually start circling in the domain.

The mockup was adapted, using basic optimisations, to different architectures, including Xeon, Xeon Phi and GPU. The same simulation was launched in multiple different architectures. This code did not use MPI communications so it could only be run in one compute node using multi-threading and SIMD vectorisation for the Intel processors, and vectorisation for the GPU processors.

Figure 11 right shows the runtimes of the mockup in the different architectures. This figure clearly shows that using a simple data structure, the particle solver can feature an almost ideal scale up in all architectures. It can be seen that the worse performing architecture is the Xeon Phi KNC accelerator. The best performance, on the other hand, is obtained using the Xeon Phi KNL cards of the SDV-Booster. Xeon processors and GPU’s are still a competitive choice.

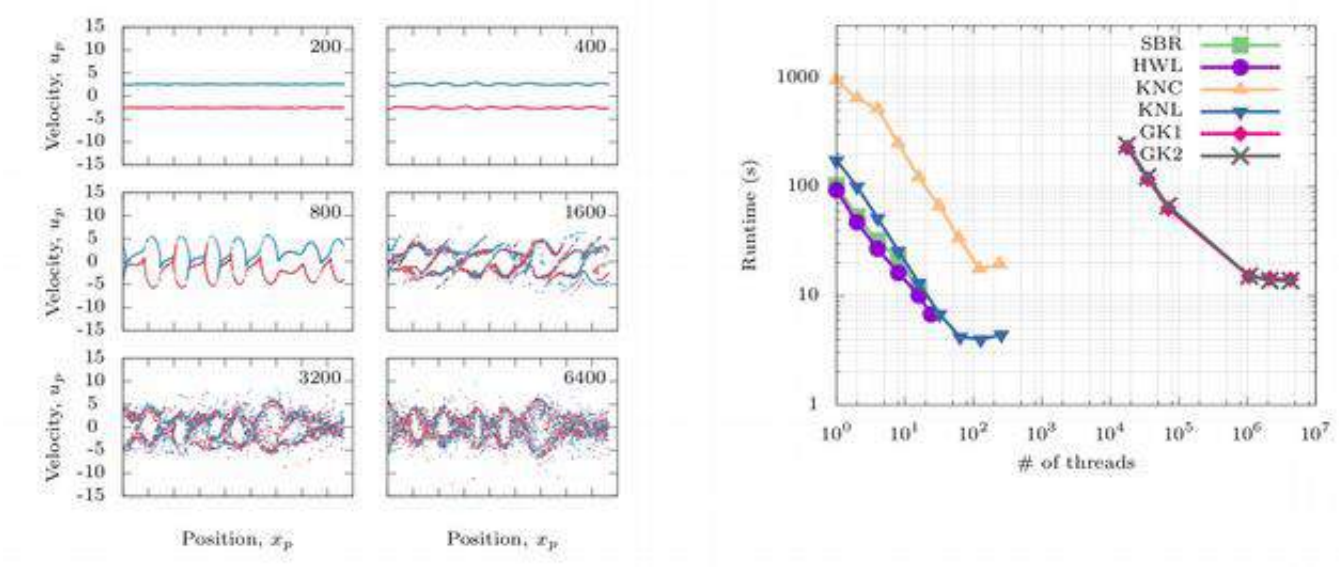


Figure 11: Comparison of the 1D mockup code on different architectures (KU Leuven)

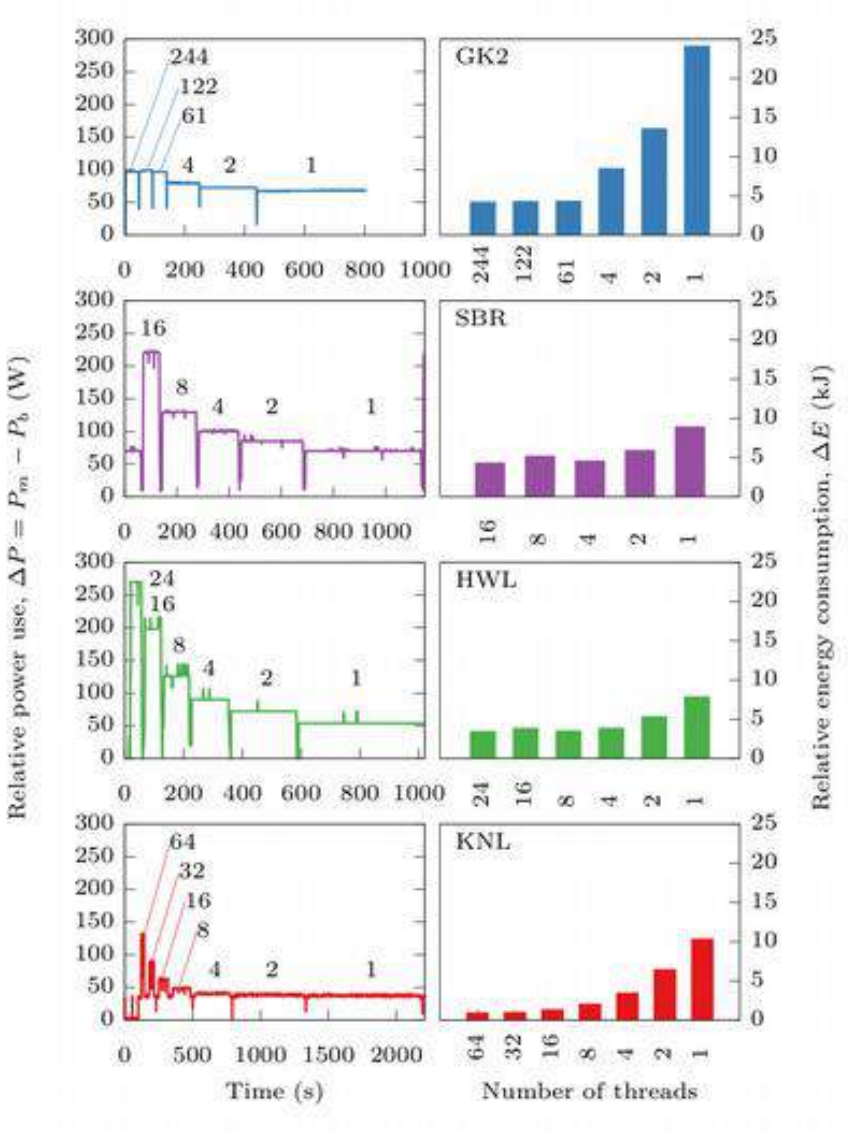


Figure 12: Relative energy efficiency of the 1D mockup code in different architectures (KU Leuven))

In addition to the runtimes, the energy efficiency of each one of the architectures was measured. Figure 12 shows the relative energy consumption of the same simulations shown in the previous figure. The panels on the left show the relative power use, i.e. the instantaneous energy use relative to the background (idle) energy use of the node. The same vertical scale is used in all the panels. These clearly show the moment were the simulations were launched with different number of cores. The panels on the right show the relative energy consumption per simulation. This is equal to the integration in time of the power use (panels on the left). This is the quantity used by electric companies to write the electricity bill.

For the faster runs, when using the maximum computing capacity of the processors, the KNL shows very good energy efficiency, consuming less than 2 kJ per run. The GK2, HWL and SBR processors, on the other hand, are consuming two to three times that amount of energy, around 5 kJ per run. This is an extremely important result: the KNL processor reduced by more than half the power consumption of the mockup code.

The maximum energy consumption in these tests is given by the GK2 running with only one thread per block with 25 kJ. This is followed by the KNL processor running on one thread,

consuming 10 kJ of energy. CPU architectures running on one core show an energy consumption of around 8 kJ. This means that serial code will consume less energy in a CPU processor than in a KNL or GK2 processor. Serial sections should be avoided in accelerators. Surprisingly, the energy consumption of the GK2 is very high, with values comparable or even superior to those measured in the SBR and HWL. Even having a lower instantaneous power use, the GK2 fails to provide good energy consumption because in the performances of the processor are only comparable to those of the KNC.

Energy consumption is almost constant for CPU processors from four cores and up. But in general terms the HWL processor presents better energy efficiency than the SBR. This analysis shows that today the best way to reduce energy consumption is to improve the parallel efficiency of the codes in the different architectures. It is however important to note that these figures present values relative to the background energy use. In the DEEP-ER Cluster, the HSW nodes had a background idle energy use twice as high as the DEEP Cluster.

Three important conclusions arise from this analysis:

1. The mockup shows that the particle solver can give very good performances using vectorisation alone.
2. Energy consumption depends mainly on the background (idle) energy use of the node and the parallel efficiency of the code.
3. Serialised sections of the code must be run on CPU architectures to consume the minimum amount of energy.

4.5.2 Comparison of the different DEEP(-ER) systems

Figure 13 shows the performance of the xPic code in three asymmetric cases: 1) an asymmetric run in the Xeon DEEP Cluster, 2) an asymmetric run in the Xeon DEEP-ER Cluster, and 3) an asymmetric run in the Xeon Phi DEEP-ER Booster.

The code xPic runs faster in the DEEP-ER Booster (KNL), followed by the DEEP-ER Cluster (SDV) and finally the DEEP Cluster. First of all it can be observed that the code runs completely in the KNL processors, using the PETSc library compiled for the CPU. However, the figure also shows that the field solver run much slower in the KNL than in the CPU processors. This is caused by two factors: individual cores in the KNL processors are slower that cores in the two other architectures, and the PETSc library (using the MKL math libraries) are not compiled and optimised for the Xeon Phi architecture.

These three simulations use very similar computational loads, with the conditions presented in Table 8. The difference in the number of cells for the SDV run is due to the total number of cores per node. The total number of cells has to be divisible by the number of processors and by the number of blocks per processor, in order to have a balanced load. The closest possible number of cells were used that fit in the SDV configuration.

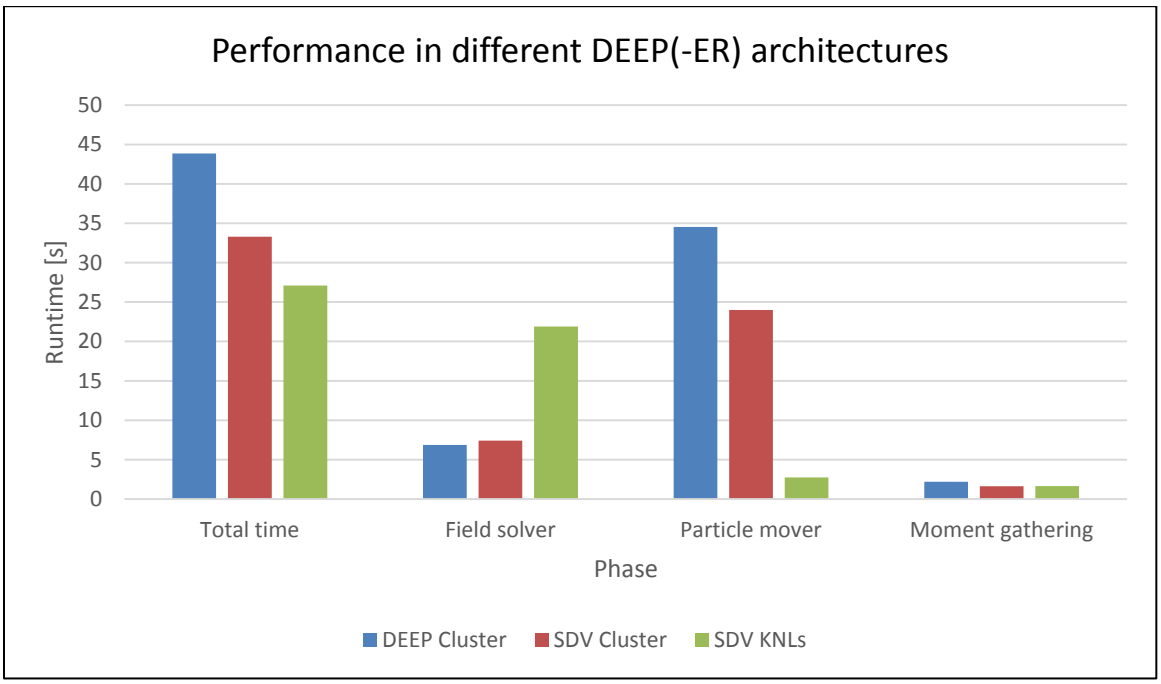


Figure 13: Performance of xPic in different DEEP(-ER) architectures (KU Leuven)

Number of cells	DEEP Cluster: 65536 cells per node DEEP-ER Cluster (SDV): 65472 cells per node DEEP-ER Booster (KNL): 65536 cells per node
Other experiment details	1000 particles per cell DEEP Cluster: 16 blocks per MPI subdomain DEEP-ER Cluster (SDV): 24 blocks per MPI subdomain DEEP-ER Booster (KNL): 64 blocks per MPI subdomain
Compiler version	Intel/17.1
MPI runtime version	parastation/intel-mt-5.1.4-1_1_g064e3f7
Compilation flags	Multi-threading: -openmp SIMD vectorisation: -mavx (Xeon) , -xMIC-AVX512 (KNL)
MPI processes per Node	2
Threads per process	DEEP Cluster: 8 DEEP-ER Cluster (SDV): 12 DEEP-ER Booster (KNL): 32

Table 8: Benchmarking setup for the comparison of the xPic performance on the different DEEP(-ER) systems (KU Leuven)

4.5.2.1 Distribution of threads

These first results also raise the question: What is the best distribution of threads per node? Figure 14 shows the total runtime obtained for the same benchmark using different thread distributions. The code runs in one SDV KNL node, using 1 to 64 MPI processes, with 64 to 1 OpenMP threads respectively. These runtimes depend on the size of the problem, and the

number of blocks per processor selected, but give a helpful indication of the efficiency of each one of the compute phases under different thread distributions.

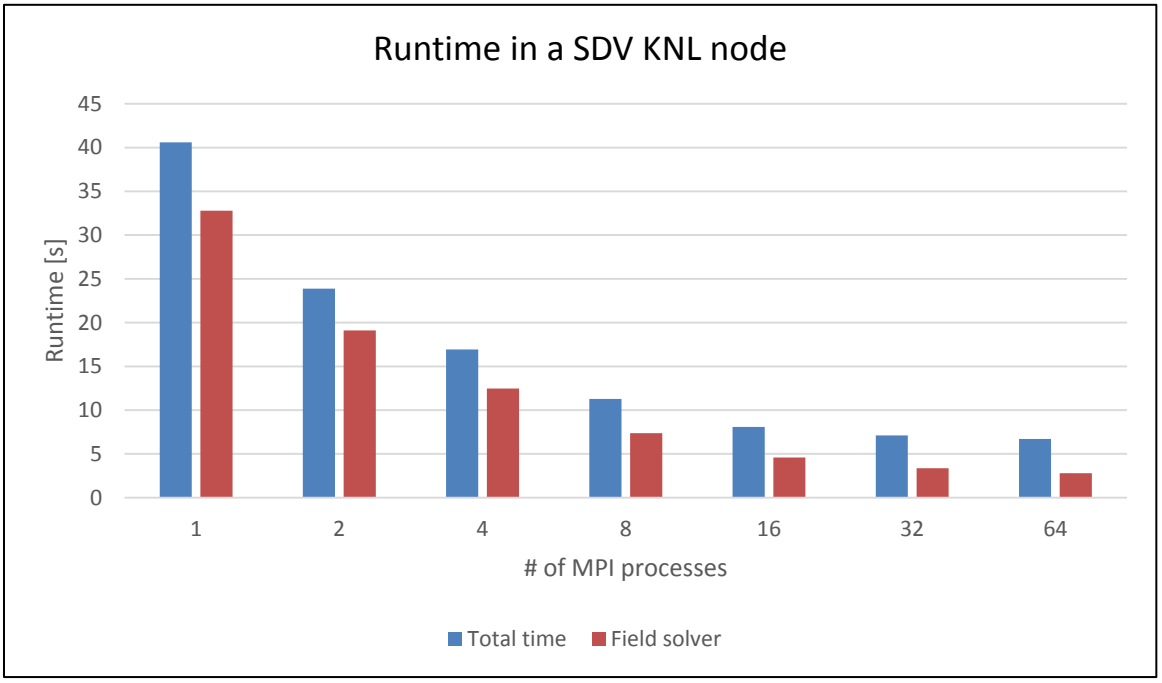


Figure 14: Comparison of the total runtime and the field solver runtimes for different numbers of used processes/threads (KU Leuven)

First it can be noticed that there is indeed a difference in the total runtime of the code. The figure seems to indicate that the best option is to use only MPI processes, instead of using OpenMP multi-threading. This, however, is not a good conclusion.

First let's notice that in the same figure, the partial runtime of the field solver follows the same pattern. The field solver time seems also to encompass the largest part of the total runtime, i.e. the field solver is the slowest phase, in particular when using a small number of MPI processes. Here again the effects of using the PETSc library are shown. In these cases the field solver only uses a number of processes equal to the number of MPI tasks, so it is normal that the smaller the number of MPI processes used, the slower the field solver is.

There are two solutions to this problem: use a mapping from multiple MPI processes in the particle solver to 1 process in the field solver, or use the Cluster-Booster approach to make a one-to-one mapping that better corresponds to the compute intensity of the different phases.

Figure 15 shows the partial runtime of the particle solver phases. It can be observed that for more than 1 MPI process, their runtime is almost constant. In each case all the cores are used, and differences in the execution time depends only on the data exchange method.

In particular it can be seen that the runtime of the field solver is almost equal to the runtime of the particle solver when the former uses all the 64 cores of the KNL card. The best solution would then be to use one KNL card for the field solver and one KNL card for the particle solver, were the distribution of threads in the particle solver would not impact the total runtime of the application.

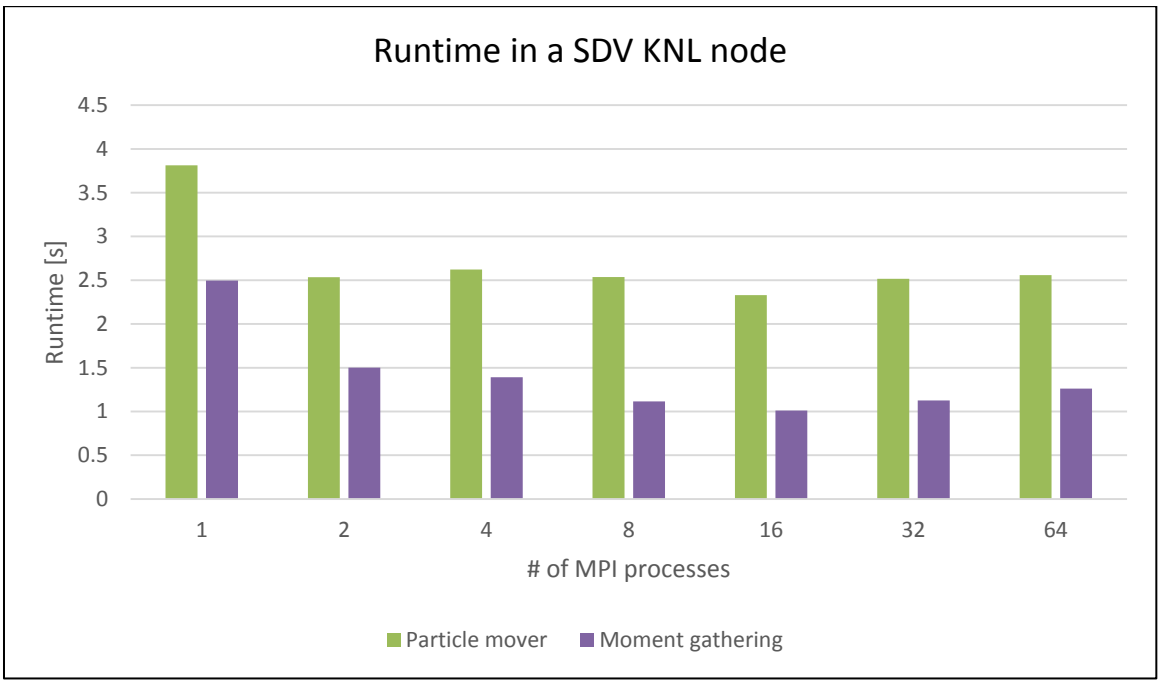


Figure 15: Comparison of the particle mover and the moment gathering runtimes for different numbers of used processes/threads (KU Leuven)

4.5.2.2 Performance of the asymmetric code

Using the asymmetric code weak and strong scaling tests were performed. The three available architectures were used for the asymmetric code. The results presented here are preliminary and require a more detailed analysis and additional optimisations. In the past months additional features were included to the code, and new numerical algorithms. It is again necessary to go over the code and implement additional optimisations. We now have the tools and the know how to achieve the best performance, as with the 1D mockup from which the xPic code emerged.

Nevertheless, encouraging results from the scalability tests were achieved. Figure 16 shows the weak scaling test performed using the setup presented in Table 9. The Intel Xeon architectures show high levels of efficiency, above 0.9 for the maximum number of nodes available in the DEEP and DEEP-ER systems. Unfortunately it was not possible to use more than 2 nodes for the testing of the KNL systems due to inaccessibility to larger resources. An initial drop in efficiency for runs with less than 8 nodes needs to be investigated in more detail. However, the figure shows that the code retains a good efficiency for a large number of nodes. Still, additional modifications are planned to improve even more the code and bring it to truly Exascale performance.

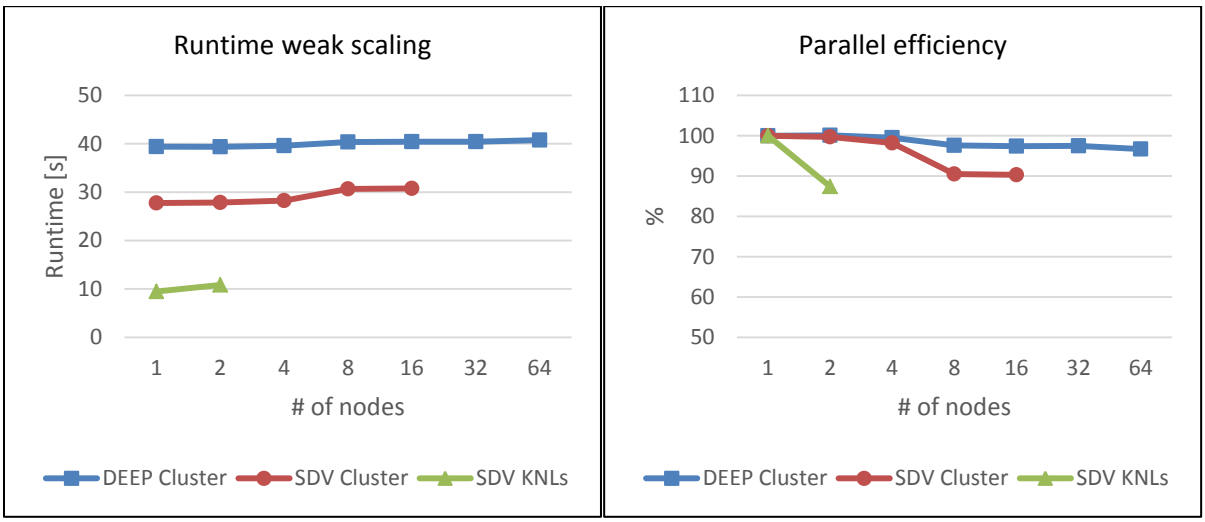


Figure 16: Weak scaling on the different DEEP(-ER) systems (KU Leuven)

Scaling	Weak scaling
Number of cells	DEEP Cluster: From 65536 to 8388608 cells per node SDV Cluster: From 65472 to 1047552 cells per node SDV KNLs: From 65536 to 131072 cells per node
Other experiment details	1000 particles per cell
Compiler version	Intel/17.1
MPI runtime version	parastation/intel-mt-5.1.4-1_1_g064e3f7
Compilation flags	Multi-threading: -openmp SIMD vectorisation: -mavx (Xeon) , -xMIC-AVX512 (KNL)
MPI processes per Node	2
Threads per process	DEEP Cluster: 8 SDV Cluster: 12 SDV KNLs: 32

Table 9: Weak scaling benchmark setup on the different DEEP(-ER) systems (KU Leuven)

The same type of results have been obtained for the strong scaling tests. Figure 17 shows that for the DEEP Cluster and the SDV KNLs the parallel efficiency is supra optimal. This benchmark test has been performed using the setup described in Table 10. However the SDV Cluster presents a drop in performance down to 0.8 for 16 nodes. This is again a result that requires a more detailed analysis, as the size of the problem may influence the performance of the code in different architectures.

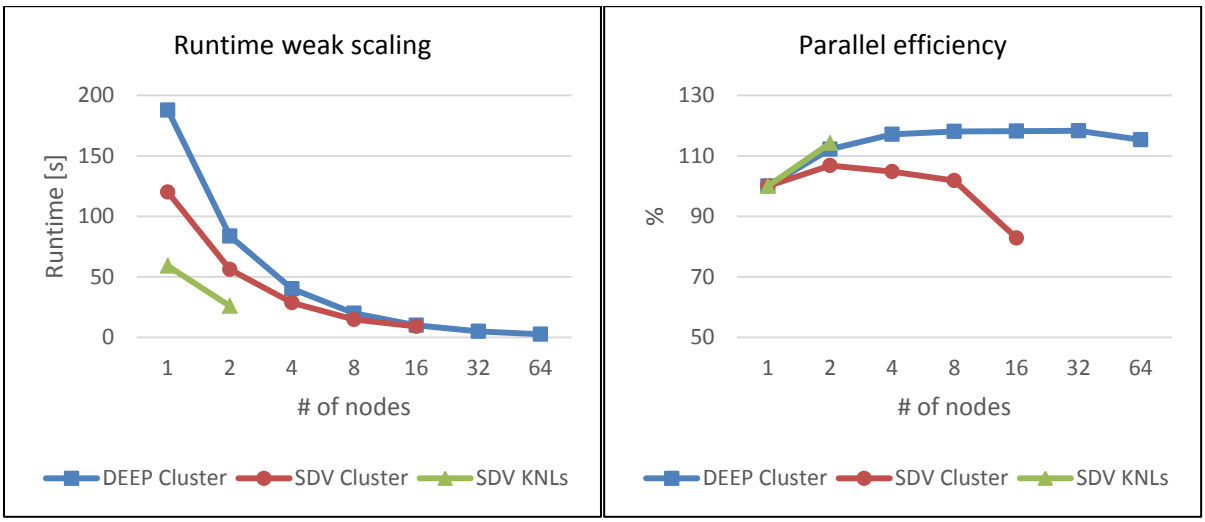


Figure 17: Strong scaling on the different DEEP(-ER) systems (KU Leuven)

Scaling	Strong scaling
Number of cells	DEEP Cluster: 262144 cells per node SDV Cluster: 261888 cells per node SDV KNLs: 262144 cells per node
Other experiment details	1000 particles per cell
Compiler version	Intel/17.1
MPI runtime version	parastation/intel-mt-5.1.4-1_1_g064e3f7
Compilation flags	Multi-threading: -openmp SIMD vectorisation: -mavx (Xeon) , -xMIC-AVX512 (KNL)
MPI processes per Node	2
Threads per process	DEEP Cluster: 8 SDV Cluster: 12 SDV KNLs: 32

Table 10: Strong scaling benchmark setup on the different DEEP(-ER) systems (KU Leuven)

4.5.3 Preliminary results on QPACE3

This section presents some preliminary results gained on the recently installed QPACE3. The system is available for users only since a few weeks and the whole software environment is still being optimised. The results are shown in Figure 18 and the benchmarking setup is described in Table 11.

For the tests 3 scenarios were used:

(1) the application runs for 100 iterations without checkpointing;
(2) the application runs for 100 iterations and writes checkpoints to the global file system every 10 iterations;
(3) the application runs for 100 iterations and writes checkpoints to the local storage (on Ram disc) every 10 iterations.

The amount of data that is written for each checkpoint is 3GB per node.

It can be seen that the difference between global and local checkpointing grows when increasing the number of nodes. For 64 nodes the difference is already quite significant. Since these results were obtained only very recently, a full analysis could not be done yet. However, these preliminary results already serve to verify the assumption that having local storage devices like the NVMe cards in DEEP-ER will improve the I/O for larger scale setups.

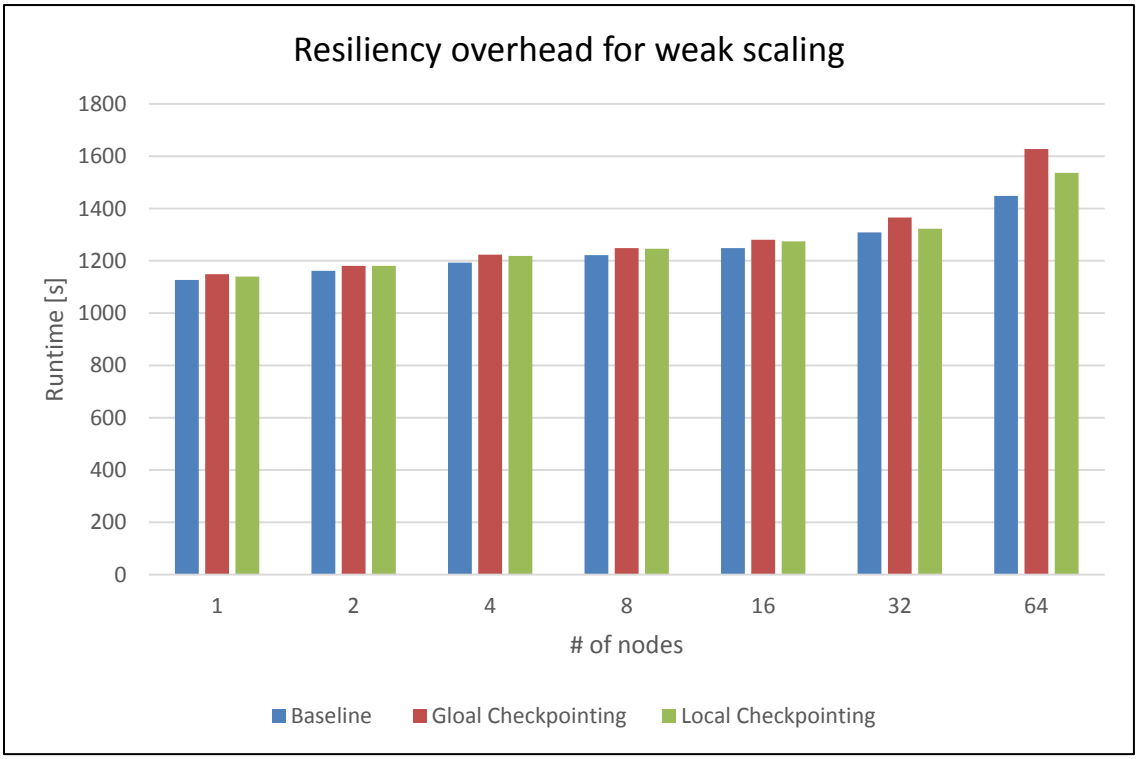


Figure 18: Weak scaling on QPACE3 (xPic, KU Leuven)

Experiment details	Homogeneous plasma in 1D with 2 species Particles per cell: 1024
Number of cells	32768 cells per node
Number of checkpoints	10
Number of cycles	100
Used system	QPACE3
Compiler version	gcc 4.8.5
MPI runtime versions	ParaStation MPI
MPI processes per node	32
Threads per process	2

Table 11: Weak scaling benchmark setup for measurements on QPACE3 (KU Leuven)

4.6 Conclusion

In this section the code xPic was presented. This code is the new generation of Particle-in-Cell plasma codes developed at KU Leuven. It shows a good parallel efficiency and makes use of all the technologies introduced in the DEEP-ER Project, including hybrid architectures and innovative I/O and resiliency tools. It was shown that the new code outperforms the original iPic3D code both in computing and I/O performances. In addition the code is ready to use new technology, like the NAM for accelerated checkpointing.

The xPic code and the KU Leuven team have made extensive use of the following DEEP-ER technologies:

- JUBE: Performing benchmarks using this systems is an incredible experience. The simple XML files used by this system are self-explanatory and easy to understand. JUBE was included in our project in less than one afternoon, and it is currently used for all automated benchmarking tests.
- SIONlib: It was planned to use this library for the storage of large files containing the particle information, but it was found out that including field information would allow to easily implement restart operations of the simulations. Support was received from the development team but also feedback was given for the better use of the library. Now the xPic code can perform fast I/O and quick restarts thanks to the use of this library.
- SCR: It was not planned to use SCR for the checkpointing of the code, however, the installation on top of SIONlib was performed in one afternoon. And it is a pleasure to perform restarts using SCR. Not only does the system keep our files safe in case of failure but restarting the code at the last saved point is done automatically without any intervention from the user. This makes the code extremely simple to use.
- NVMe: As it was shown in the previous sections, the use of the NVMe technology has accelerated the I/O time of the code for the range of file sizes that is normally used.
- EXTOLL: It is clear that the interconnect plays an important role both in the performances of the communications but also in the I/O procedures that require data exchanges among different nodes.

These technologies weren't not only used but it was also given feedback on the experience made to improve them. Also some of the other software and technology features from the DEEP and DEEP-ER Projects were used, such as Extrae/Paraver and BeeGFS, and it is planned to use libNAM and the E10 libraries once they will be available to accelerate and reinforce the I/O routines of our code.

The new code xPic also features a new, and much more flexible, architecture. Thanks to this approach it is are planned to include multiple new models for the study of plasma physics, from the simulation of full planetary environments to small scale energy fusion experiments.

Major advances were achieved in the numerical tool that will have an impact in the scientific findings in the following months and years.

5 Task 6.3: High temperature superconductivity (Task leader: CINECA)

In the final year of the project the main effort was focussed on the OpenMP threading of the 2degas program (a developed mockup to test optimisations that may be included in TurboRVB), and the use of the SCR library for TurboRVB resiliency. Tests and benchmarks were performed on the DEEP Cluster, the DEEP-ER SDV, the Xeon (Broadwell generation) and KNL partitions of the Marconi cluster of Cineca. It should be mentioned that it was not possible to perform some of the tasks foreseen in the previous Deliverable for this period (see Section 4.6.1 “Next Steps” in D6.2 [3]). Thus, for example, due to the OpenMP strategy employed it was not possible to use OmpSs for a Cluster-Booster division of 2degas (see Section 5.2 for more details). In addition, a deeper analysis of the code with profiling tools such as VTune demonstrated that, because of the short loop lengths, there were no significant opportunities for vectorisation.

5.1 Application overview

Details of the applications were given in previous deliverables ([3], [5]) but it is recalled that both TurboRVB and 2degas perform Quantum Monte Carlo sampling using the concept of Monte Carlo (MC) “walkers” which run in parallel and, for most of the program duration, execute independently. Each walker can be simulated by no more than one MPI rank so normally weak scaling tests are performed as benchmarks. Among the WP6 applications TurboRVB is one of the more complex applications, having the so-called Branching phase which consists of MPI point-to-point communications between walkers (see Figure 19) and this phase can influence parallel scaling where MPI communications are not optimal (e.g. when using multiple Xeon Phi KNC processors). 2degas, on the other hand, is very highly scalable even on Xeon Phi devices since communications (e.g. `MPI_Bcast` and `MPI_Reduce`) are only present at the beginning of the program and at intervals during execution for collecting averages. Both programs are released as MPI-only applications written in FORTRAN 77/90 and for most inputs have limited memory and I/O footprints.

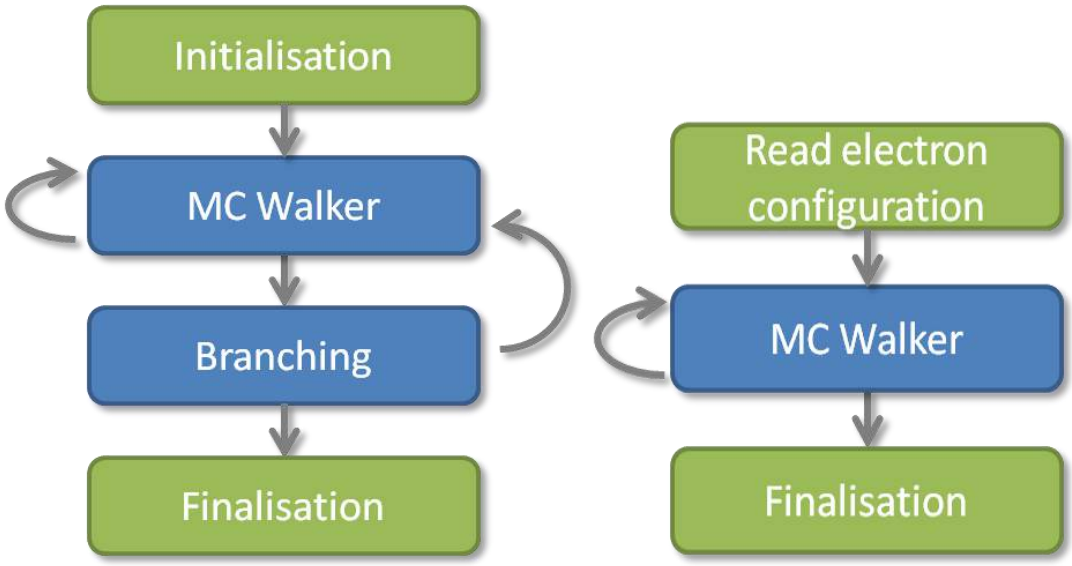


Figure 19: Workflow of TurboRVB (left) and 2degas (right) (CINECA)

5.2 Optimisations

In this section optimisations performed on the 2degas application are described. Since it is known that both KNC and KNL processors are not best suited for programs parallelised with only MPI, it was decided to create a threaded version of the program with OpenMP, with the aim of a further extension to a hybrid MPI/OpenMP parallelisation. Possibilities for vectorisation, another key feature of Intel Xeon Phi technology, were also examined.

5.2.1 Work done during the last project year

5.2.1.1 OpenMP parallelisation of 2degas

The initial strategy for adding thread parallelism to the program was based on the idea of adding `OpenMP DO` directives to the longest loops in the MPI program code, identified by an analysis with Intel VTune. Although some code modifications were attempted, this met with little success because no loops were found to be long enough to justify OpenMP threading. In addition, given the serial nature of each Monte Carlo walker, an OpenMP task parallelisation is not possible. This strategy was therefore abandoned and it was decided instead to convert the MPI-only version to an OpenMP program, i.e. each Monte Carlo walker is represented by an OpenMP thread rather than by an MPI task. Since the communications in the MPI version are low, this conversion was not expected to bring performance gains for small numbers of walkers but it was hoped that it would be more efficient at higher numbers of walkers, particularly on MIC architectures. Of course, with only shared memory parallelism the program can only be run on one node but it was intended at a later stage to introduce MPI code to spawn OpenMP walkers on multiple nodes and thus introduce hybrid parallelism to 2degas.

The OpenMP conversion of 2degas was performed in the following steps:

1. Modifications and refactoring of the FORTRAN 77 source code, including localisation as much as possible of the program's global variables via modules and new subroutines in order to decide which variables needed to be private or shared amongst the OpenMP threads;
2. Replacement of the MPI calls with OpenMP directives or equivalent code in the shared memory environment;
3. Comparison with the MPI-only version.

The next sections go into more detail for each of these steps.

5.2.1.1.1 General re-factoring of the 2degas source code

As described in a previous Deliverable (D6.2), the original source code available before the project started was written in FORTRAN77 and is almost entirely contained in one single file. A partial re-factoring was necessary for example to remove unaligned COMMON blocks and to allow the use of external libraries such as MKL for the linear algebra calculations. Some hard-coded limits and bugs were also removed. The new OpenMP strategy necessitated further code modifications in order to decide which variables needed to be private to each thread and which needed to be shared. In order to limit the refactoring necessary, routines not involved in the calculations were moved into separate files and although compiled into the executable were not altered further. The result of this is of course that the program can be run with OpenMP parallelisation only for the MC algorithm used in this project (i.e. Variational Monte Carlo). For the subroutines and functions required by the benchmark input,

all the code was then converted to FORTRAN90, localising variables when appropriate (virtually all the variables in the program were by default global). The resulting version consisted of a FORTRAN90 kernel with the global variables, private to each thread (i.e. to each MC walker), contained in a separate file.

5.2.1.1.2 OpenMP conversion

With the correct identification of the variables, replacing MPI calls with OpenMP and shared memory constructs was relatively straightforward. Thus, for example, `MPI_Bcast` calls were often replaced with `!$OMP single copyprivate` constructs. It should be noted that the need to use `OMP threadprivate` declarations for the global variables means that OmpSs cannot now be applied to the program since OmpSs does not currently support this `threadprivate` declaration. A more complicated aspect of the conversion involves the file handling since each MC walker creates its own restart and scratch files and this is not thread safe without code modifications. The scratch files were found to be quite small (less than a few tens of MBytes) so they could be removed by writing to memory instead (i.e. using variables). The restart file creation on the other hand was initially wrapped in an `OMP CRITICAL` section. This worked but gave very poor performance. On the other hand, by using distinct FORTRAN I/O units for each thread, it was found possible to remove the CRITICAL directive and obtain reasonable performance and scaling (see below).

5.2.1.1.3 Comparison with MPI version

After it was checked that the results of the OpenMP code gave the same results as the MPI version, weak scaling benchmarks were run on a number of architectures including Xeon (Haswell generation), KNC and KNL processors. In this section only the results for the KNL nodes on the DEEP-ER SDV Cluster are reported (see Table 12 for details). Figure 20 shows the weak scaling results of the OpenMP version of the code compared to the MPI implementation on an SDV KNL. It can be observed that the MPI version is faster than the OpenMP implementation, but the difference diminishes as the number of threads or tasks increases. This is not totally unexpected since the MPI communications footprint is very low (see D6.2 for more details). In Figure 21 a similar graph is shown but for up-to 240 threads or tasks. Here instead it can be seen that the OpenMP version outperforms the MPI implementation for more than 60 threads or tasks.

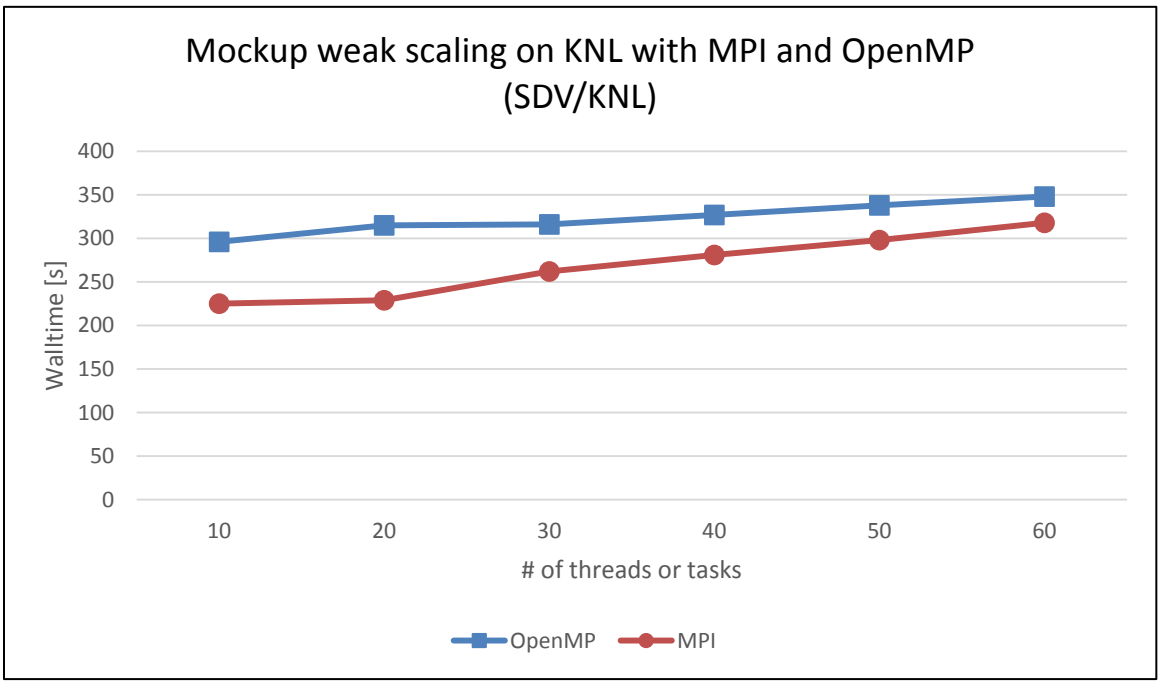


Figure 20: Impact of OpenMP parallelisation on 2degas (CINECA)

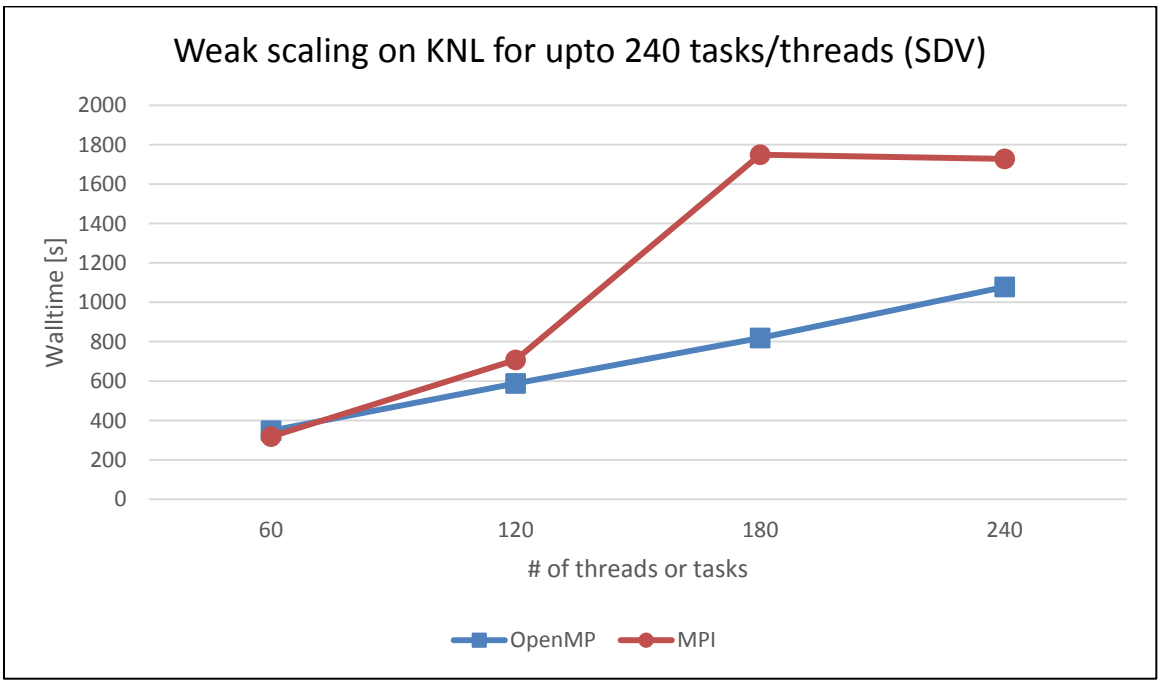


Figure 21: Weak scaling of 2degas for 60-240 threads or tasks (CINECA)

5.2.1.1.4 Performance with MCDRAM on KNL

To assess the possible performance benefits of the KNL MCDRAM, the OpenMP version of 2degas was run with 1, 2 and 4 threads/core with and without the MCDRAM as shown in Figure 22: the MCDRAM was selected by running the program with the `numactl` command. As can be seen from the figure, for low numbers of cores and threads the MCDRAM gives no performance benefit. However, as the number of cores and threads/core increase more

influence from the MCDRAM can be observed. In fact at 60 cores with 4 threads/core, the MCDRAM gives a performance increase of nearly 30%.

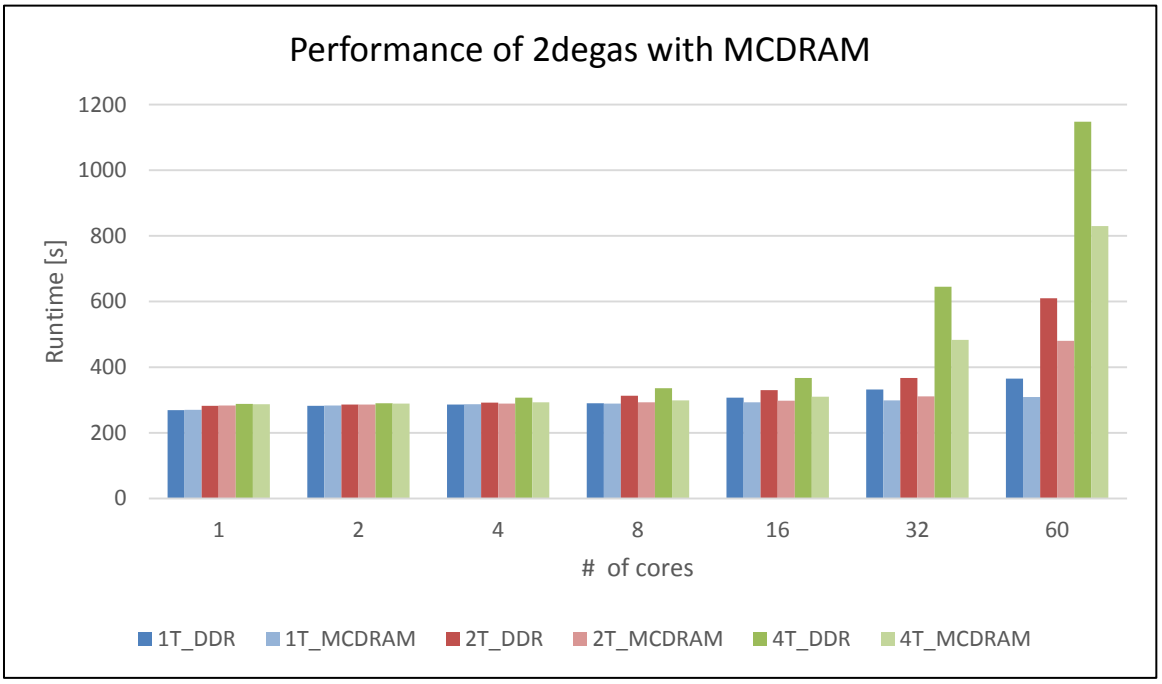


Figure 22: Comparison between MCDRAM and main memory (DDR) using 2degas (CINECA)

Experiment details	1 MC block of 5000 cycles
Used system	SDV/KNL
Compiler version	Intel 16.2
MPI runtime versions	parastation/gcc-5.1.4-1_1_g064e3f7
Compilation flags	-O2
MPI processes per Node (MPI version)	10-240
Threads per node (OpenMP version)	10-240

Table 12: Experiment setup regarding OpenMP/MPI comparison (CINECA)

5.2.2 Progress overview

Workflow elements	M1-M6	M6-M12	M13-M18	M19-M24	M25-M30	M31-M36	M37-M42
Analysis	1	2	2	2	2	2	2
Writing D6.1	0	1	2	2	2	2	2
Port to KNC	1	1	2	2	2	2	2
Threading	0	0	0	0	1	2	2
Vectorisation	0	0	0	0	0	0	2
General code optimisation	1	1	1	1	1	1	2
Implement OmpSs parallelisation and/or offload	0	1	2	2	2	2	2
Cluster-Booster division	0	1	2	2	2	2	2
Port to SDV	0	0	0	0	2	2	2

Use NVM	0	0	0	0	0	2	2
SIONlib integration	0	1	2	2	2	2	2
E10 integration	-	-	-	-	-	-	-
General I/O optimisation	1	1	1	1	1	1	2
Implementing a mockup	0	0	0	0	1	2	2
Writing D6.2	0	0	0	0	2	2	2
Checkpointing on NAM	0	0	0	0	0	0	0
Optimise for KNL	0	0	0	0	0	1	2
Implement SCR	0	0	0	0	0	0	2
Implement OmpSs task based resiliency	-	-	-	-	-	-	-
JUBE integration	0	0	1	1	2	2	2
Final benchmarking	0	0	0	0	0	0	2
Writing D6.3	0	0	0	0	0	0	2

Table 13: Progress overview (CINECA)

This section describes why some DEEP-ER technologies were not implemented for TurboRVB or 2degas.

5.2.2.1 E10 integration

Since the I/O footprints of both TurboRVB and 2degas are low, and the checkpointing of the former has in any case been optimised with the SIONlib library for parallel task local I/O, E10 integration was not expected to bring significant benefits.

5.2.2.2 Implementation of OmpSs task based resiliency

This was not planned for either TurboRVB or for 2degas. For the former it was possible to implement OmpSs only for the offload, whereas for 2degas the use of `threadprivate` variables prevented any OmpSs constructs.

5.3 Resiliency

5.3.1 TurboRVB with Scalable Checkpoint Restart (SCR) support

Before the project both TurboRVB and 2degas had only limited resiliency and checkpointing facilities. In the previous Deliverable D6.2 [3] the implementation of a rank-local and SIONlib-based checkpointing facility for TurboRVB was described, which was tested on various architectures. This section now describes the implementation of the SCR library for TurboRVB.

SCR was implemented by inserting calls from the SCR API (v 1.1.8) into the checkpoint version of the TurboRVB code. The modifications were quite straightforward, requiring only the `SCR_INIT` and `SCR_FINALIZE` calls, at the beginning and end of the program, and the SCR checkpoint commands inserted in the checkpoint module of TurboRVB. Thus, in FORTRAN the code is like this:

```
call scr_start_checkpoint(ierr)
checkpfile_tmp=trim(checkpfile)//".ckpt"
call scr_route_file(checkpfile_tmp,checkpfile,ierr)
open(iunit,file=checkpfile)
```

```
…write data..
close(iunit)
call scr_complete_checkpoint(valid,ierr)
```

The program was compiled with the usual Intel compiler and ParaStation MPI libraries, linking in the SCR library in order to make the SCR API calls available (see Table 14 for details). This implementation was then tested on the SDV Haswell nodes and compared with the non-SCR version of TurboRVB running on the same platform. Figure 23 shows the walltime as a function of the number of nodes for both versions of the code, using the /sdv-work filesystem for storing the checkpoint and temporary files. As it can be seen from the figure the performance of the TurboRVB program is virtually unaffected by the SCR calls, even giving a slight improvement for small numbers of nodes on the SDV. One explanation could be that the I/O footprint of TurboRVB is low. Even with 16 nodes on the SDV the I/O is not stressed much.

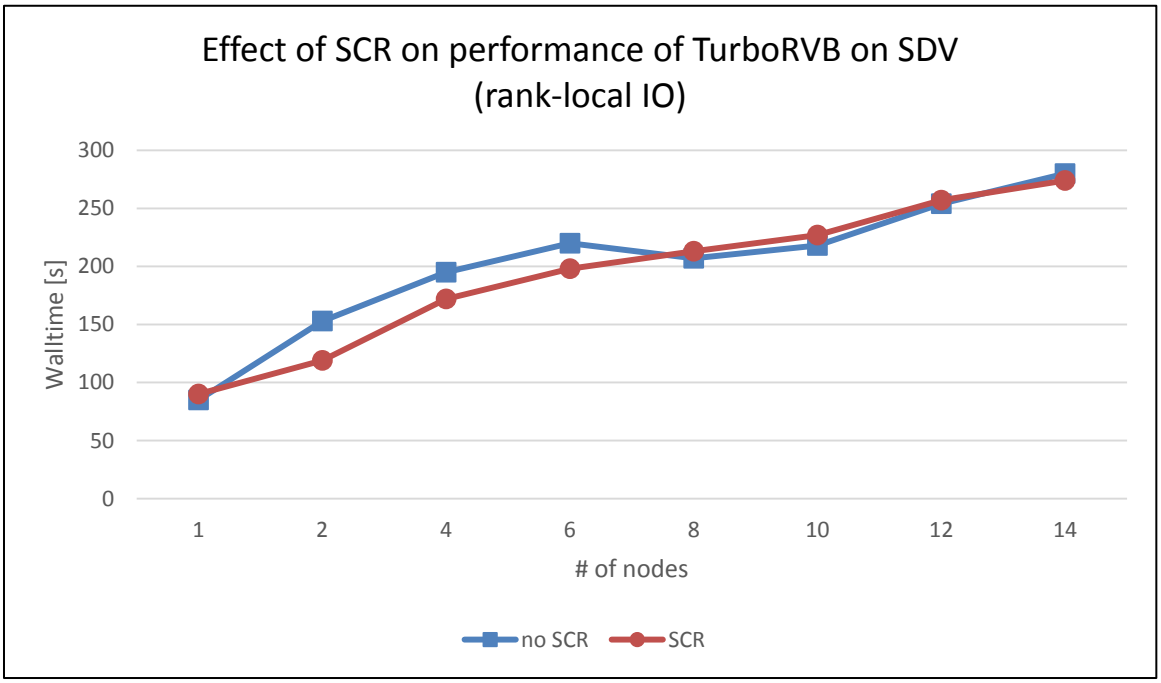


Figure 23: Impact of SCR integration (CINECA)

Experiment details	200 electrons, 1000 MC cycles
Used system	DEEP-ER SDV
Compiler version	Intel 16.2
MPI runtime versions	parastation/gcc-5.1.4-1_1_g064e3f7
Compilation flags	-O3 $(SCRFLAGS)
MPI processes per node	24
Describe the error (location, time, ...)	No error occurred.

Table 14: Experiment setup for the SCR compilation and testing on the SDV (CINECA)

5.4 Input/Output

Neither TurboRVB nor 2degas have a significant I/O footprint, even for large input sizes (i.e. many electrons). However, as it was showen in D6.2 [3], using SIONlib for the restart files greatly reduces the time needed to restart a simulation compared to rank-local restarts (by a factor of 10-100x, in some cases). In addition, by using NVM devices (e.g. SSD disks) it is possible to reduce the start-up time by a further order of magnitude (see D6.2).

5.5 Comparison between architectures

This section shows some sample benchmark results comparing different architectures for both TurboRVB and 2degas.

5.5.1 TurboRVB on Broadwell and KNL (Marconi).

Here a comparison of the performances of TurboRVB on the Xeon (Broadwell generation) and the KNL partitions on the Marconi Supercomputer at Cineca is reported; a summary of the compute node characteristics of the two partitions of the Marconi cluster is given in Table 46 in the annex.

For this runs of the TurboRVB code an input set of 200 electrons (100 up and 100 down) was used for 1000 cycles, storing checkpoint configurations every 100 cycles – for further details of the runs see Table 15.

Scaling	Weak
Number of electrons	200
Other experiment details	1000 MC cycles, checkpoint every 100 cycles
Compiler version	Ifort v 17.x
MPI runtime version	Intel 17.
Compilation flags	-O3 –ax512-AVX
MPI processes per Node	36 (Broadwell), 68 (KNL)
Threads per process	1 (for MKL, i.e. MKL_NUM_THREADS=1)

Table 15: Benchmarking setup for the TurboRVB runs on Marconi (CINECA)

The walltimes and the parallel efficiencies for weak scaling of the program for the two partitions are shown in Figure 24. From the figure it can be observed that the parallel efficiency on KNL becomes quite poor once 50 nodes are reached, whereas the performance on Broadwell is within our expectations. Even allowing for the fact that the Broadwell nodes have fewer cores than the KNL nodes, it is not clear at the moment why this is the case. But it should be emphasised that the A2 (KNL) partition of Marconi has been installed only recently and it is possible that further tuning of the system is needed. In particular, it has been observed with other programs that the OmniPath network is not performing as

expected with the KNL nodes, but since the DEEP-ER Booster won't use OmniPath, this will not be investigated further here.

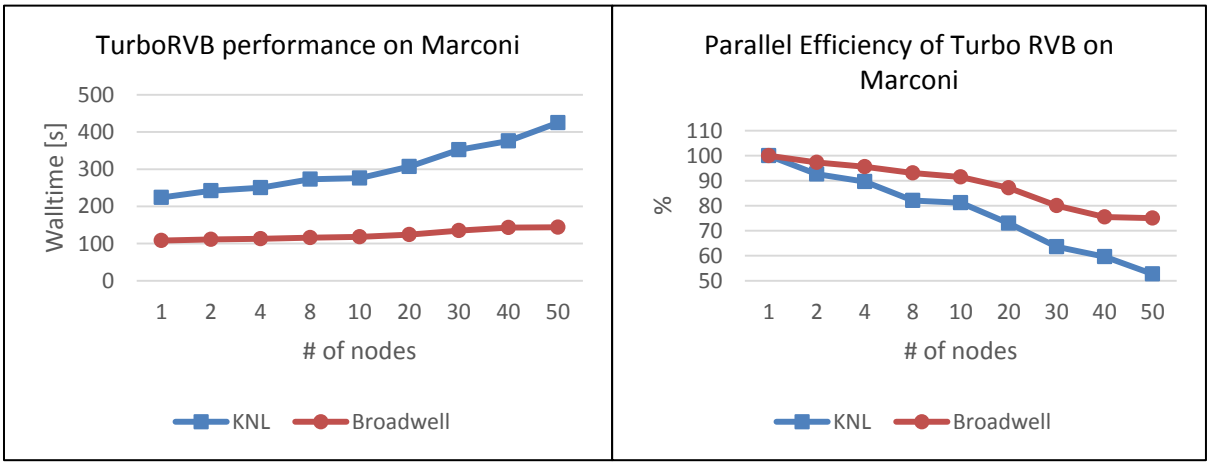


Figure 24: Weak scaling performance of TurboRVB on the Broadwell and KNL partitions of Marconi (CINECA)

5.5.2 KNC and KNL single node comparison for 2degas

For this comparison, the KNC devices present on the Galileo cluster at Cineca and the KNL nodes available on the DEEP-ER SDV were chosen. In Figure 25 the performance is shown in terms of the wall clock time needed to run 500 MC cycles/block for 2degas on a KNC and on a KNL, using both the MPI and OpenMP versions of the program (see Table 16 for more details of the simulation). It can be observed that the program behaves similarly on both devices, showing good weak scalability and with the MPI version slightly faster than the OpenMP code. Presumably this is due to the overheads present in the OpenMP runtime (given the low communication overheads in the MPI code the OpenMP version wasn't expected to perform better). It was also observed that the program on KNL runs approximately 3-4 times faster than it does on a KNC.

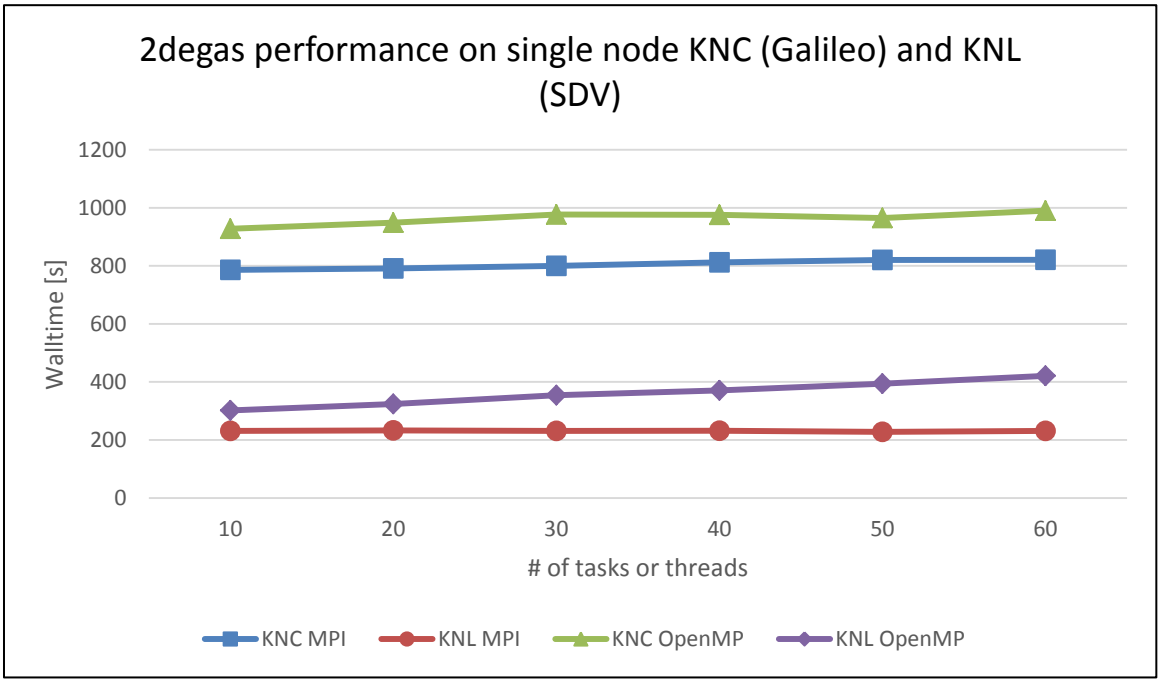


Figure 25: Performance of 2degas on KNC (Galileo) and KNL (SDV), for both the MPI and OpenMP versions (CINECA)

Scaling	Weak
Number of electrons	26
Other experiment details	Variational Monte Carlo, 10 blocks of 5000 cycles
Compiler version	17
MPI runtime version	2017
Compilation flags	-O3 -axMIC-AVX512 (KNL) –mmic (KNC)
MPI processes/OpenMP threads per Node	1-60

Table 16: Benchmarking setup for 2degas runs on Galileo (KNC) and SDV (KNL) (CINECA)

5.6 Conclusion

This project has had a big impact on the TurboRVB and 2degas applications, both in terms of functionality and performance. In addition, if we assume that these applications are reasonably representative of parallel Quantum Monte Carlo simulation codes in general, our experience has given us insights on the possibilities, and also limitations (e.g. the difficulty of applying OpenMP threading and vectorisation), of DEEP-ER hardware and software technologies in this important field of computational science.

The specific improvements in TurboRVB relate to resiliency, in particular the checkpointing facility, which was not present before the project began. The checkpointing facility is particularly flexible in so far as it allows both task-local files and files managed by the SIONlib library (an option available at run time to the user). Because of the low I/O footprint of TurboRVB, SIONlib does not have a significant impact on write times for the checkpointing but by reducing the number of files from potentially many thousands (i.e. one for each MPI task) to just a few, managing jobs for the application user becomes much simpler. An additional advantage is that on restart checkpoint files managed by SIONlib can result in start-up times which are orders of magnitude faster than for task-local restart files (as shown in D6.2 [3]). The project also allowed us to show that this start-up is further accelerated if NVM disks are used for storing the restart files (also shown in D6.2). The use of the SCR library implementation provides additional resilience to the application and does not impact negatively on the program performance.

Improvements in application performance were most noticeable instead in the prototype code 2degas. The decision to study an alternative QMC code was taken, we recall, in order to investigate better features of TurboRVB without dealing with the high complexity of the full code, e.g. its complex communication pattern (consisting partially of MPI point-to-point calls) and its bulky code that prevented the implementation of OpenMP parallelisation. The 2degas program instead has a simpler communication scheme and has fewer lines of code, although as it turned out significant re-factoring was required before important modifications could be made. As described above, since the loops in the code are short when the benchmark input is used, a simple OpenMP threading of the MPI version could not be done, so instead the MPI calls were removed completely and QMC walkers re-implemented as OpenMP threads

with the aim of then adding MPI to allow communications between NUMA nodes. It was found that on KNL, the OpenMP version runs slightly slower than the MPI version (comparing processes with threads), although it still shows excellent weak scaling. The OpenMP version on the other hand becomes more efficient than the MPI code when multiple threads per core were used, demonstrating a clear improvement when compared to the code available before the project. In addition, if MCDRAM is used, a further performance of up to 30% can be achieved.

A hybrid MPI/OpenMP version is currently being developed. We recall that unfortunately OmpSs cannot be used because of the presence of `threadprivate` variables so the `MPI_Comm_spawn` command has to be used.

We found that some of the key features of the KNL, such as the wide 512bit AVX2 vector units could not be exploited by 2degas with the input used since loops are too short to allow an efficient vectorisation and OpenMP threading.

To conclude, if we assume that the two applications are representative of QMC programs in general, i.e. based on parallel MC walkers, then the DEEP-ER hardware and software technologies can greatly improve the scope for checkpointing and resiliency. From the point of view of performance optimisation, an MPI program can benefit from the conversion to OpenMP, but probably only at a “coarse-grain” level, i.e. parallelising MC walkers as single threads. This is due to the presence of short loops which, at least in TurboRVB and 2degas, are simple functions of the number of spins (as low as 26 in the case of the 2degas input). Simulations with larger numbers of spins could be attempted but memory requirements increase dramatically and exceed the resources available long before any benefits from the longer loop counts are likely to be seen. Tasking on the other hand is not possible due to the serial nature of the Monte Carlo algorithm.

As a final comment, we point out that for both programs important code re-factoring was needed in order to make the necessary code changes, even though we endeavoured to keep this to a minimum. These activities could have been reduced, and the porting to the prototype Exascale architecture possibly improved, if the application software had been built with more rigorous software engineering principles. This is however a common feature of “academic” application codes and must be taken into account in similar projects.

6 Task 6.4: Assessment of human exposure to electromagnetic fields (Task leader: Inria)

In the third year of the DEEP-ER Project, our development efforts have been oriented towards the following objectives:

- Extend the previous optimisation efforts on KNC in the second year of the project by porting and benchmarking on KNL.
- Pursue the integration of I/O and resiliency tools, in particular OmpSs for application-level persistent checkpoint/restart, and offload of I/O from Booster Nodes to Cluster Nodes.
- Experiment with taskification of the main routines enabling, if possible, task-based shared memory parallelism and lightweight checkpoint/restart with OmpSs.

Also, the application has been renamed GERShWIN for "discontinuous **G**al**ER**kin **S**olver for micro**W**ave **IN**teraction with biological tissues" in place of MAXW-DGTD.

6.1 Application overview

Inria's application GERShWIN is a time-domain solver of the Maxwell-Debye system of differential equations modelling electromagnetic wave propagation in biological tissues in 3D, which is based on a Discontinuous Galerkin Time-Domain (DGTD) method. The workflow of the application is recalled on Figure 26. It is composed of a pre-processing phase, the time-loop and a post-processing phase. While the pre- and post-processing phases are essentially concerned with reading an unstructured mesh and writing the final solution to disk respectively, the time loop is mostly concerned with processing - except when application-level checkpointing is activated. This processing consists of sequentially updating the electric and magnetic fields, as well as updating the Fourier transforms at the source central frequency on the fly. More details can be found in D6.2 [3].

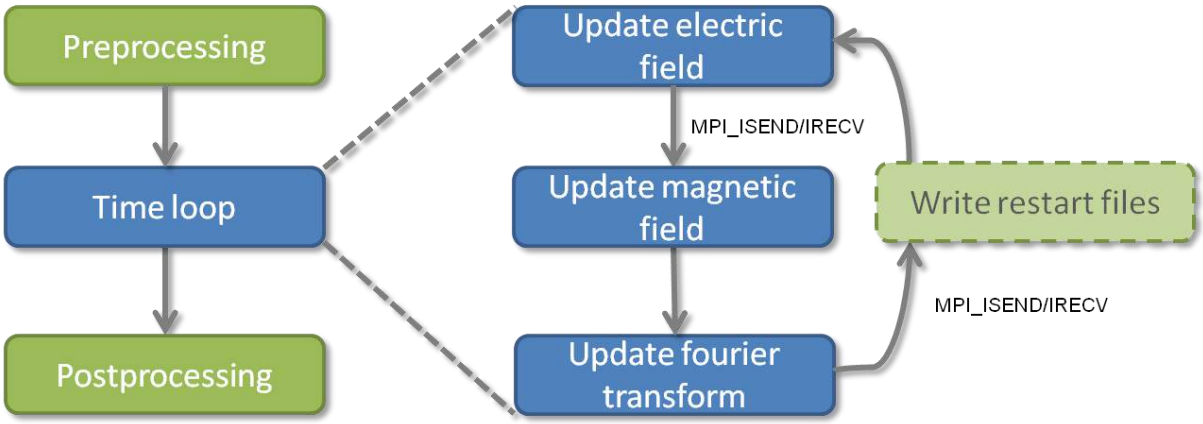


Figure 26: Workflow of GERShWIN (Inria)

6.2 Optimisations

6.2.1 Work done during the last project year

6.2.1.1 Porting and benchmarking on KNL

Porting and benchmarking on the KNL architecture is a task that was greatly facilitated by the work reported in D6.2 [3] in which GERShWIN was ported and optimised to the preceding KNC architecture. Raw performance on KNL will be commented and compared to other architectures in Section 6.5.

One of the opportunities of porting to the KNL is the possibility of taking advantage of the on-package high-bandwidth memory (MCDRAM). For accessing this memory, simply the `numactl` command was used to place all application data either in the DDR or in MCDRAM.

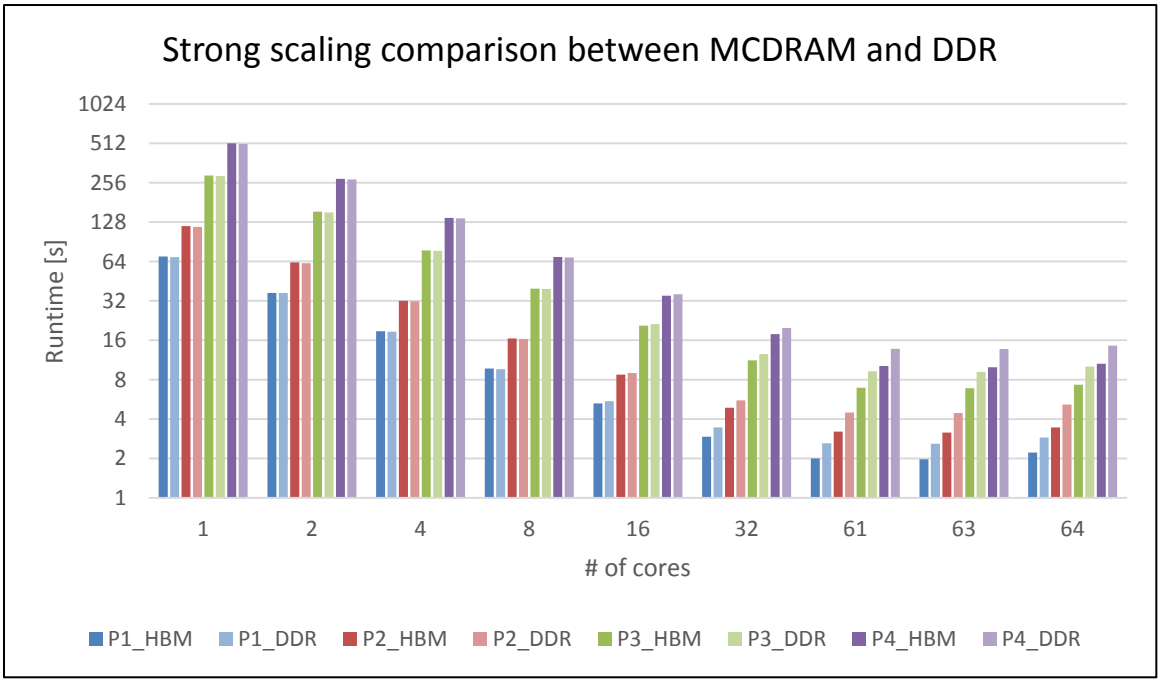


Figure 27: Strong scaling comparison between MCDRAM and DDR (Inria)

Figure 27 shows the walltime of the time-loop for different number of cores (strong scaling), using the DDR and using the MCDRAM in flat mode with all-to-all policy, for Lagrange basis functions of different orders (P1 to P4). It is recalled that the interpolation order changes the number of degrees of freedom per field (4 in P1 up to 35 in P4), and therefore the amount of required processing. The experiment details are listed in Table 17.

For a small number of cores, little memory movement is required and the DDR bandwidth is not saturated, therefore using the MCDRAM makes no difference. However, when the full node is exploited, using the MCDRAM yields a speedup in the range of 1.3x to 1.4x. The counterpart is a limited available space (16GB) compared with DDR (96 GB), such that larger cases (i.e. WOMAN in P3 or P4) are not tractable by a single KNL. However, the computational workload associated with such cases does justify the use of more than one node. Therefore, for the larger workloads considered in DEEP-ER, it makes more sense to aim at splitting data by using more than one Booster Node (KNL) rather than prioritising the

integration of the MEMKIND library to the application. It is recalled that the MEMKIND library is a heap manager that provides the possibility to allocate specific arrays to specific parts of the memory. Alternatively, MCDRAM in cache mode could also be experimented in such cases.

Number of cells	1853832
Other experiment details	Walltime of 20 iterations of the time-loop / No checkpointing
Used system	DEEP-ER SDV, KNL 7250 (knl04 node)
Compiler version	ifort version 16.2
MPI runtime versions	ParaStation MPI version intel-mt-5.1.4
Compilation flags	-align dcommons -qopenmp -qno-opt-prefetch -qopt-report=0 -O3 -r8 -axMIC-AVX512 -fpp
MPI processes per node	1
KNL setting	MCDRAM mode: Flat; Clustering mode: all-to-all.

Table 17: Experiment setup regarding MCDRAM tests (Inria)

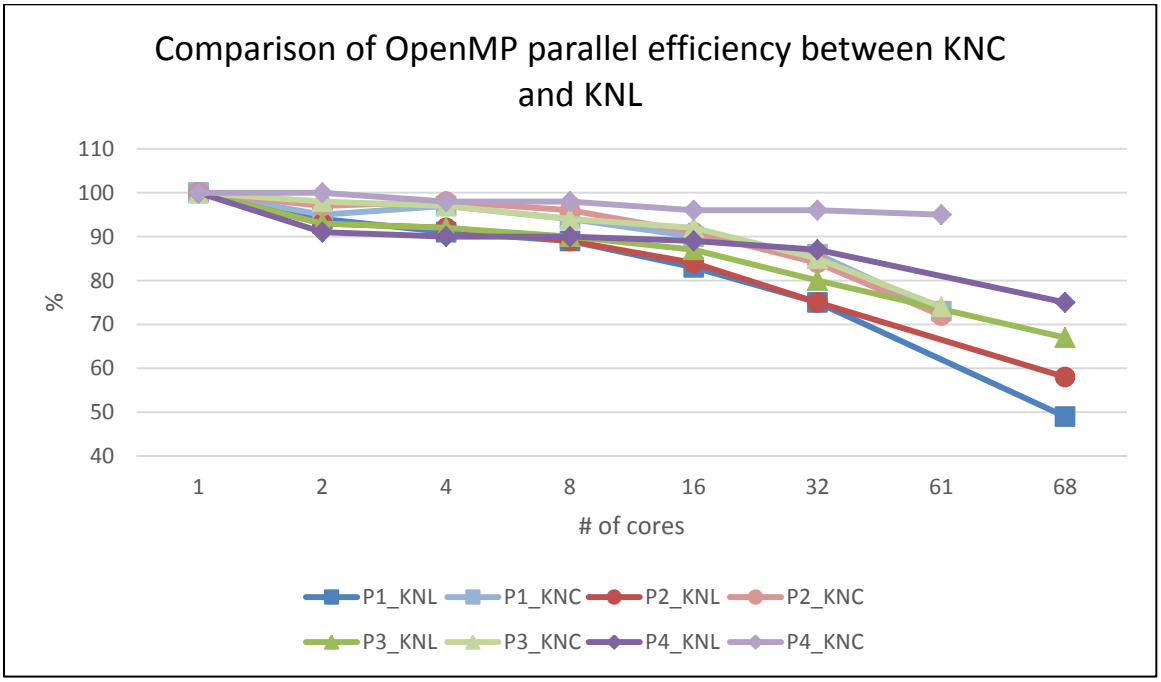


Figure 28: OpenMP parallel efficiency comparison between KNC and KNL (Inria)

Figure 28 shows the parallel efficiency achieved by the flat OpenMP parallelisation at the level of one single KNL, compared to one KNC. The considered reference value is the walltime achieved on one core using 4 threads. Table 18 shows the experiment details. Efficiency values are notably inferior on the KNL. One of the reasons for this has to do with an architectural difference: the KNL is conceived around tiles of two cores that share a L2 cache, which is not the case of the KNC where the L2 level is exclusive to cores. Therefore, the scaling from one to two cores might be spoiled by spurious L2 effects on the KNL.

Nevertheless, the runtime is better on the KNL than the KNC (see Figure 29) - for more details on performance comparison between architectures please refer to Section 6.5.

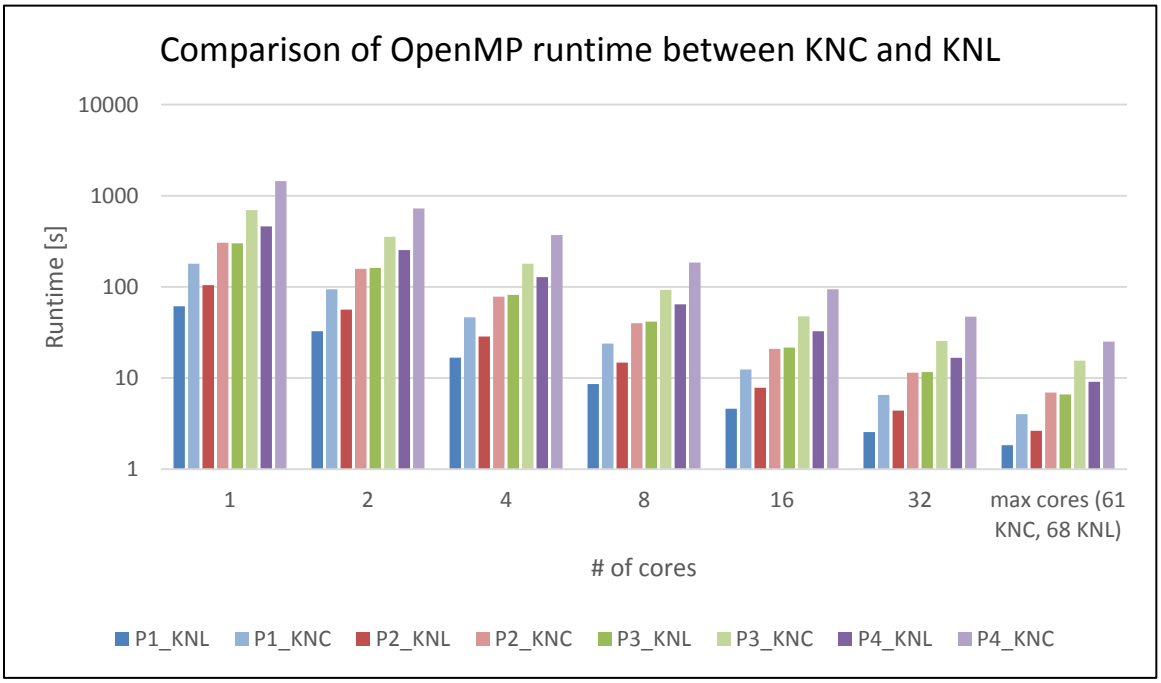


Figure 29: OpenMP runtime comparison between KNC and KNL (Inria)

Number of cells	1853832
Other experiment details	Walltime of 20 iterations of the time-loop / No checkpointing
Used system	DEEP-ER SDV, KNL 7250 (knl04 node) - 7120A KNC node on miclogin system
Compiler version	ifort version 16.2 on the SDV - intel version 15.1 on miclogin
MPI runtime versions	ParaStation MPI version intel-mt-5.1.4
Compilation flags	-align dcommons -qopenmp -qno-opt-prefetch -qopt-report=0 -O3 -r8 -axMIC-AVX512 -fpp
MPI processes per node	1
Threads per core	4
KNL setting	MCDRAM mode: Flat; Clustering mode: all-to-all.

Table 18: Experiment setup regarding the comparison between KNC and KNL (Inria)

6.2.1.2 OmpSs integration

Deliverable D6.2 [3] concluded with the discussion on the possible integration of OmpSs for two purposes: First, the offload of I/O operations from Booster to Cluster; and second, the taskification of main loops to expose task-based shared memory parallelism. The offload of I/O will be discussed in Section 6.4.

Taskification with OmpSs has been investigated during the third year of the project, but had to be left out. Indeed, a first and intuitive idea for taskifying the main routines of GERShWIN consists in assigning to elementary tasks the processing of elementary kernels F, M and K

(see D6.2) at the level of single cells. This situation is extremely convenient: in this case it is easy to define dependencies between tasks based on the mesh connectivity[1]. Unfortunately, this strategy is too "fine-grained" to be applied to GERShWIN: the processing-time required for applying elementary kernels is not much more than the time required for creating the task.

A workaround consists in assigning to elementary tasks the processing of elementary kernels F, M and K at the level of many cells. However, in that case it is not trivial to define dependencies between tasks unless we rely on a second level of partitioning of the mesh, at the level of each MPI process. This scenario would unfortunately require large and complex code development that was not possible in the time-frame of the DEEP-ER Project.

OmpSs can be used for synchronous parallelisation of loops by creating tasks from chunks of iterations. Figure 30 shows the resulting parallel efficiency at the level of one node of the DEEP-ER SDV, using the Nanos++ runtime using either OmpSs or OpenMP. The experiment setup is shown in Table 19. The performance of the two modes is quite similar with a little advantage for the OpenMP mode.

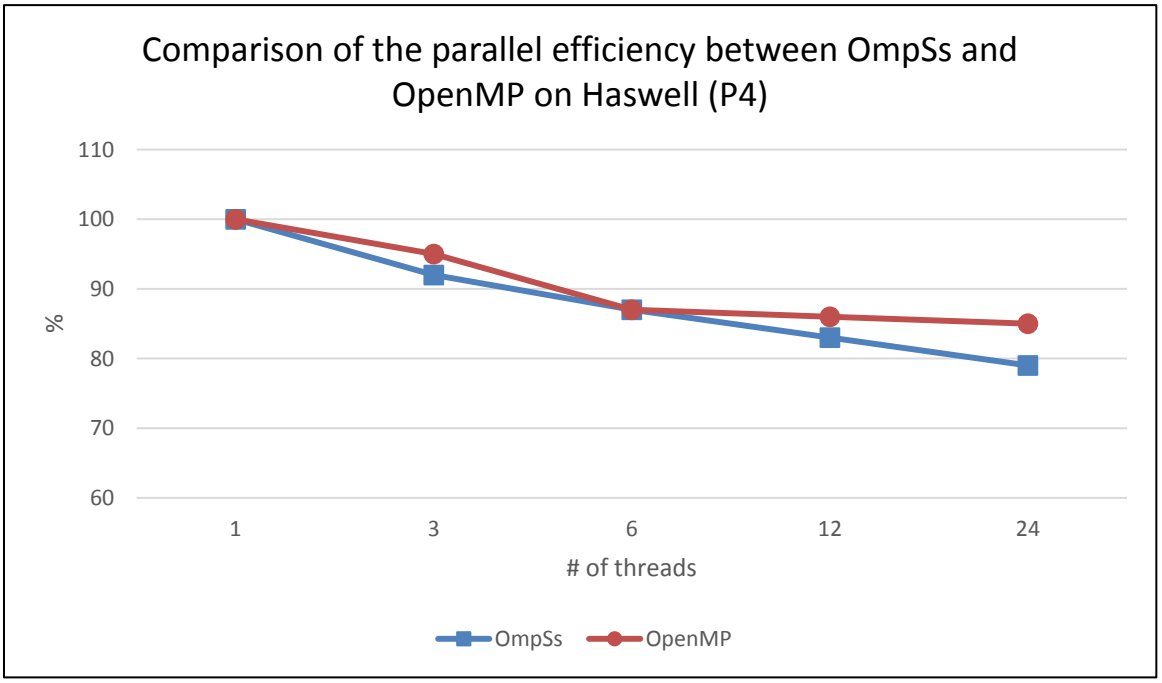


Figure 30: Parallel efficiency comparison between OmpSs and OpenMP on Haswell (Inria)

A similar set of experiments has been conducted on a KNL node. The achieved parallel efficiency is reported on Figure 31. Using Nanos++ on the KNL, best performance can be achieved by avoiding multithreading. Therefore only one thread per core was used, and the default binding policy. On the KNL, OmpSs and OpenMP modes achieve very similar performance both in terms of runtime and efficiency, with a slight advantage for OpenMP. Interestingly, it is not the case for P1 basis functions (the most cache-bound case) where using the OmpSs mode yields a 1.16x speedup over the OpenMP mode for the full node.

[1] It is recalled that updating fields for one cell only depends on the values of unknowns at its 4 direct neighbors, and itself.

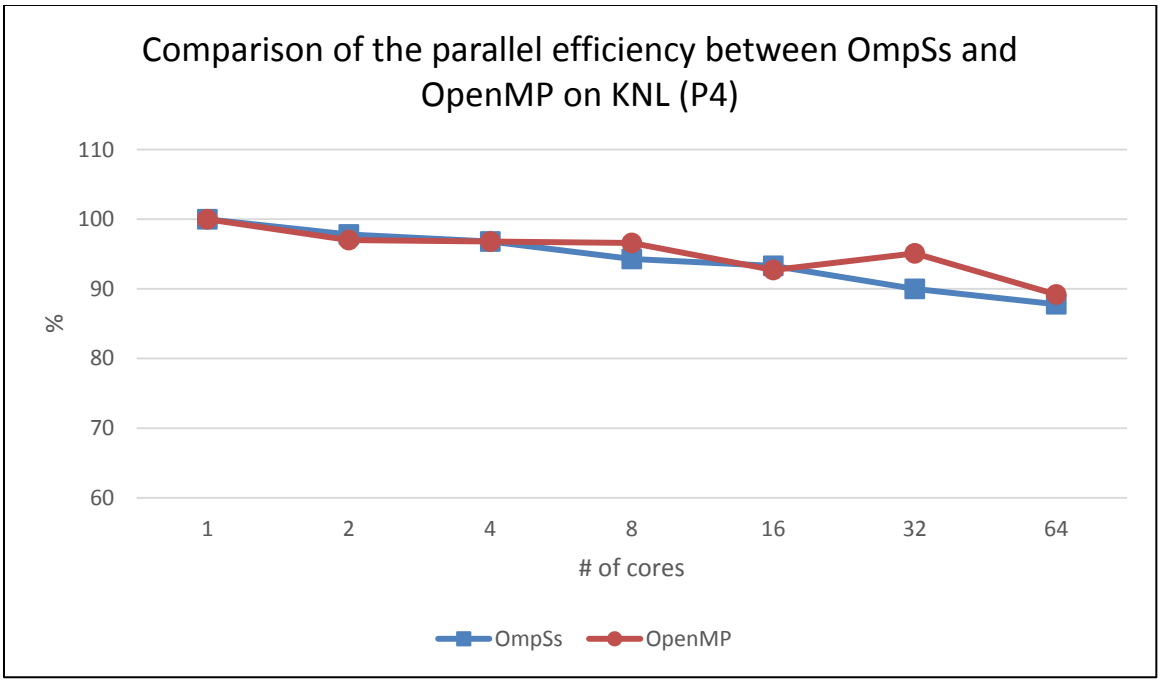


Figure 31: Parallel efficiency comparison between OmpSs and OpenMP on KNL (Inria)

Number of cells	1853832
Other experiment details	Walltime of 20 iterations of the time-loop / No checkpointing / Lagrange order P4
Used system	DEEP-ER SDV (Xeon Haswell node), KNL 7250
Compiler version	ifort version 16.2
MPI runtime versions	ParaStation MPI version intel-mt-5.1.4
OmpSs version	OmpSs MPI offload offload/16.08 - nanox git HEAD 860d546
Compilation flags SDV	-align dcommons -qopenmp -qno-opt-prefetch -qopt-report=0 -O3 -r8 -axCORE-AVX2 -fpp
Compilation flags KNL	-align dcommons -qopenmp -qno-opt-prefetch -qopt-report=0 -O3 -r8 -axMIC-AVX512 -fpp
MPI processes per node	1
Threads per process	Scaled
HBM setting (KNL)	Flat, all to all

Table 19: Experiment setup regarding the OmpSs and OpenMP comparison (Inria)

6.2.1.3 OpenMP optimisation

Finally, it was possible to identify optimisation opportunities at the level of the OpenMP parallelisation of the main loops - which is described in Deliverable D6.1, Section 4.3.2. These optimisations consisted in affecting to some "read-only" arrays the `SHARED` attribute instead of the `FIRSTPRIVATE` attribute. By doing so, threads do not perform a local-copy of the concerned arrays, which alleviates requirements in cache resources. These optimisations

yield results that are presented in Figure 32, in which the largest improvement is met by the P1 solver with a 1.19x speedup over the previous version.

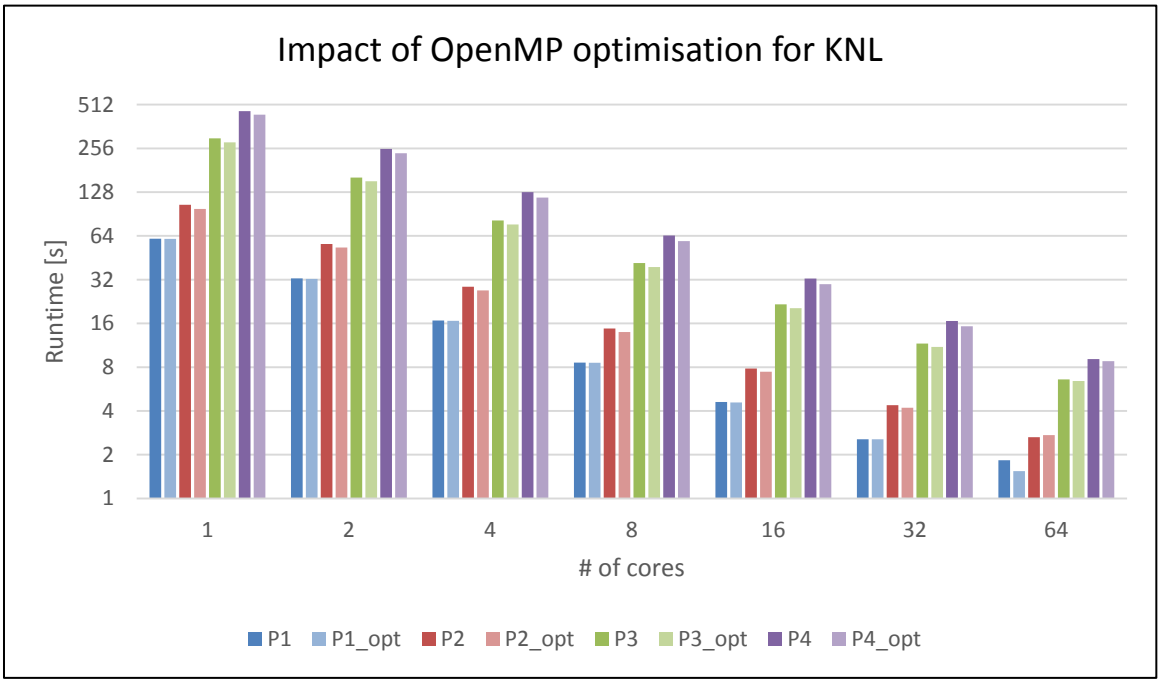


Figure 32: Impact of OpenMP optimisation for KNL (Inria)

6.2.2 Progress overview

Workflow elements	M1-M6	M6-M12	M13-M18	M19-M24	M25-M30	M31-M36	M37-M42
Analysis	1	2	2	2	2	2	2
Writing D6.1	0	1	2	2	2	2	2
Port to KNC	0	2	2	2	2	2	2
Threading	1	1	2	2	2	2	2
Vectorisation	1	1	2	2	2	2	2
General code optimisation	1	1	1	1	1	1	2
Implement OmpSs parallelisation and/or offload	0	0	1	2	2	2	2
Cluster-Booster division	0	0	1	1	1	1	1
Port to SDV	0	0	0	0	2	2	2
Use NVM	0	0	0	0	1	2	2
SIONlib integration	0	0	2	2	2	2	2
E10 integration	-	-	-	-	-	-	-
General I/O optimisation	1	1	1	1	1	1	2
Implementing a mockup	-	-	-	-	-	-	-
Writing D6.2	0	0	0	0	2	2	2
Checkpointing on NAM	0	0	0	0	0	0	0
Optimise for KNL	0	0	0	0	0	1	2
Implement SCR	0	0	0	2	2	2	2
Implement OmpSs task based resiliency	0	0	0	0	1	1	2
JUBE integration	0	0	0	2	2	2	2
Final benchmarking	0	0	0	0	0	0	2
Writing D6.3	0	0	0	0	0	0	2

Table 20: Progress overview (Inria)

- E10 has not been implemented as it targets collective I/O which is not in the framework of GERShWIN.
- Benchmarking offload of I/O with OmpSs is under progress.
- Waiting for libNAM to be integrated in SIONlib.

6.3 Resiliency

For driving application-level checkpointing, the SCR library has been integrated to GERShWIN during year 2 of DEEP-ER (see D6.2). During year 3, the persistent checkpoint restart feature of OmpSs has been implemented. It is recalled that this feature is in fact built on top of SCR, to which it provides a pragma interface. For this, tasks were assigned to single time-loop iterations, specifying the unknown and Fourier transform data arrays that should be saved for one checkpoint (`temps, ifour, efield, hfield, pol, ufour, ufourl`).

6.3.1 Evaluation of the persistent CP/RS overhead

Figure 33 shows a comparison of 3 scenarios. The first is a reference computation, with no checkpoint or restart. The second and third scenarios perform checkpoints and meet a failure with a consequent restart using SCR and OmpSs respectively. The experiment setup is listed in Table 21.

The CP/RS scenarios show less than 1% of difference in terms of overhead with respect to the reference. This implies in particular that the OmpSs persistent CP/RS does not generate any significant overhead on top of SCR.

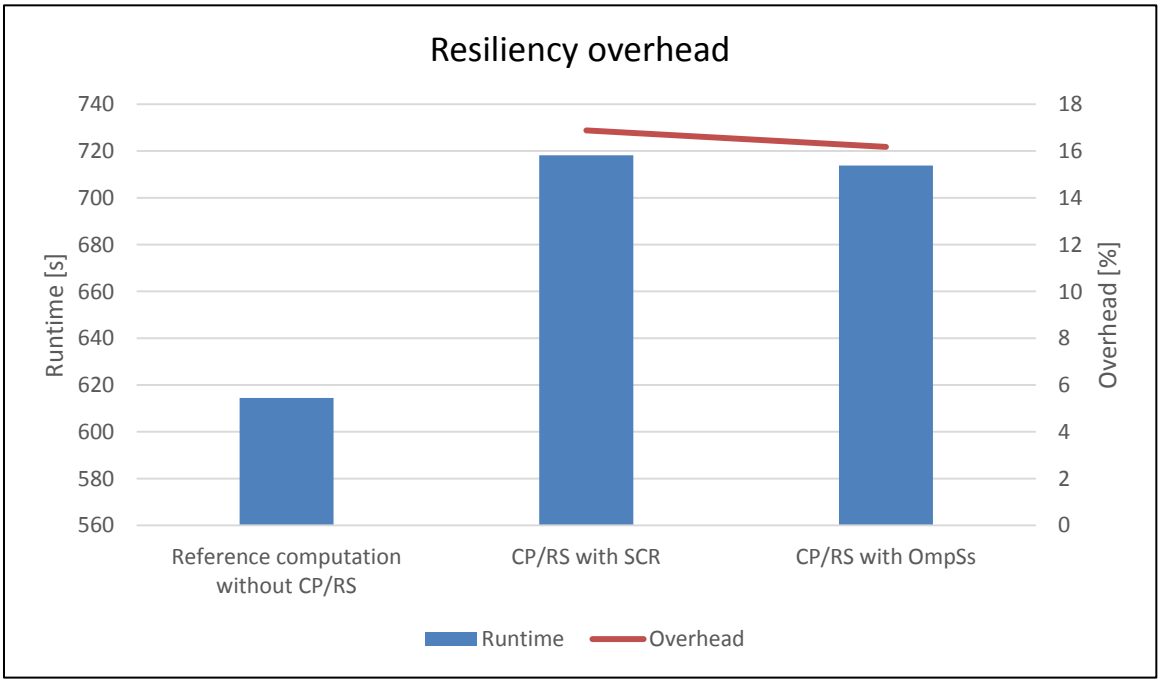


Figure 33: Resiliency overhead (Inria)

Number of cells	1853832
Other experiment details	P1, 37000 time iterations, CP every 3700 iterations, one failure & restart
Used system	Mare Nostrum - 2x Intel E5-2670 SandyBridge-EP nodes - GPFS file system
Number of nodes	32
MPI processes per node	2
Threads per process	8

Table 21: Experiment setup regarding resiliency overhead (Inria)

6.3.2 Checkpointing performance on the DEEP-ER SDV

Figure 34 represents the runtime required for SCR to perform 10 checkpoints using the /sdv-work filesystem and the on-node NVMe device provided on Haswell and KNL nodes. P1 and P3 interpolations are considered, leading to writing 1.1GB and 7.71GB per checkpoint respectively. The experiment setup is shown in Table 22. First, it can be observed that for this little amount of nodes (4), there is little difference of output runtime between the NVMe device and the /sdv-work BeeGFS. One should remark that this observation does not coincide with benchmarking results on one node achieved using /sdv-work and a NVMe device presented in D6.2 (with a speedup larger than 3x for the P3 case). The reason lies within updates on the filesystem and EXTOLL in the third year of the project improving the performance of /sdv-work. Detailed results concerning NVMe device performance with GERShWIN can be found in D3.3. Second, output time is significantly larger (about 4x) when performed from KNL nodes than from Haswell nodes. This consists in a good motivation to experiment with the Cluster-Booster division for offloading I/O operations from Booster Nodes to Cluster Nodes.

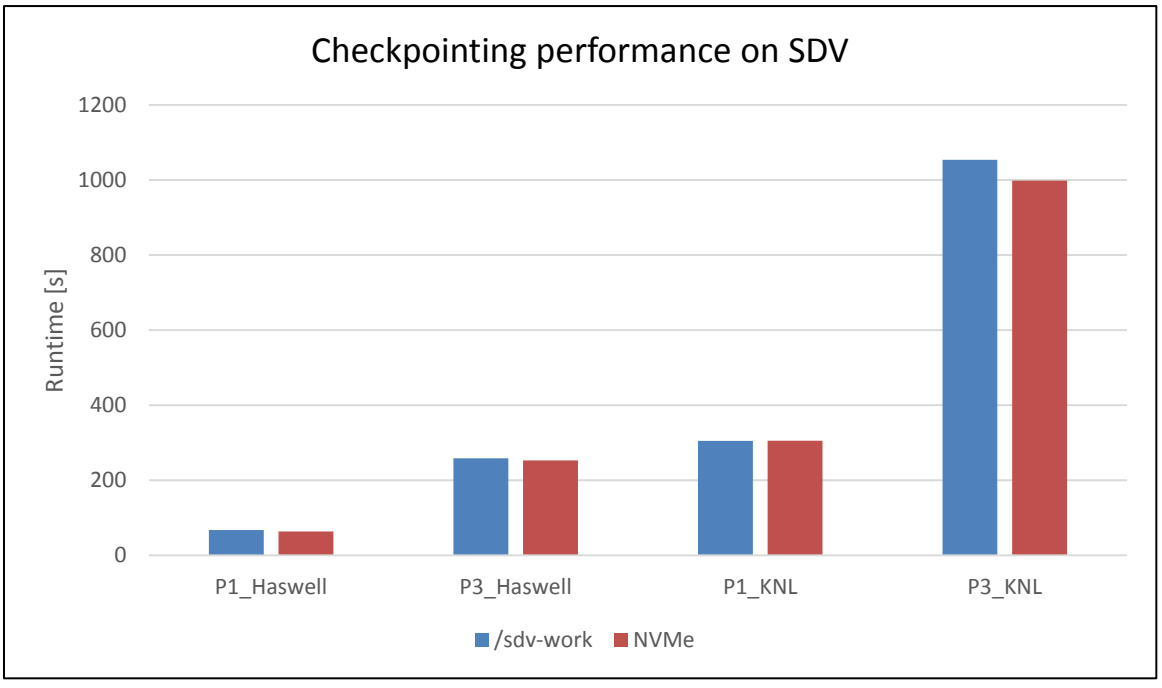


Figure 34: Checkpointing performance (SCR) on SDV (Inria)

Number of cells	1853832
Other experiment details	1000 time iterations / 10 checkpoints
SCR version	1.1.8
Used system	SDV
Compiler version	ifort version 17.1
MPI runtime version	parastation/intel-mt-5.1.7-1
Compilation flags	-align dcommons -qopenmp -qno-opt-prefetch -qopt-report=0 -O3 -r8 -fpp -axMIC-AVX512 (KNL) or -axCORE-AVX2 (Haswell)
Number of nodes	4
MPI processes per node	Haswell: 24 KNL: 16 (P1) / 8 (P3)
Threads per process	Haswell: 1 KNL: 16 (P1) / 32 (P3)

Table 22: Experiment setup regarding checkpointing performance on the SDV (Inria)

6.4 Input/Output

6.4.1 SIONlib

The integration of SIONlib has been performed in year 2 of DEEP-ER and has been reported in Section 5.3.1 of D6.2, including results achieved on the Haswell part of the SDV. In the following, results are presented achieved using one KNL node of the SDV, leveraging NVMe.

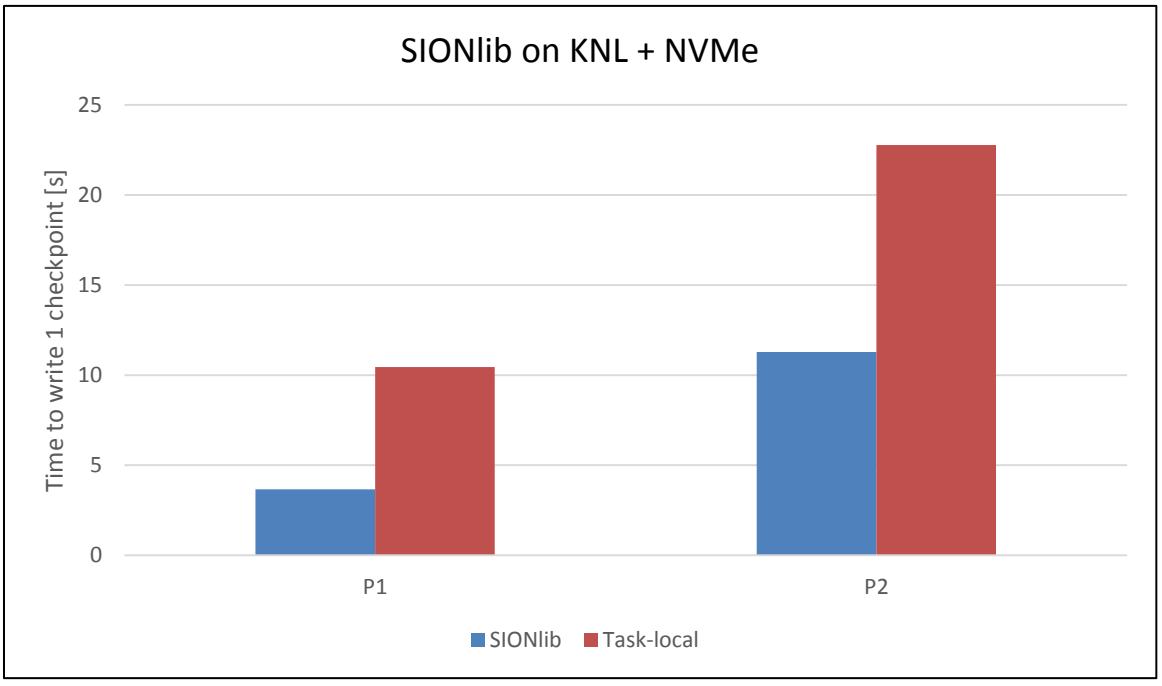


Figure 35: SIONlib on KNL + NVMe (Inria)

Figure 35 compares the time required to write one checkpoint to the node-local NVMe device into a SIONlib file, and using task-local files. The written data amounts to 1.1GB and 2.5GB in the P1 and P2 scenario respectively. It is shown that SIONlib yields a speedup of 2.86 and

2.01 respectively over task-local I/O. This is especially satisfying as the number of processes for these experiments - 64 - is moderate. Higher speedups are expected for larger number of MPI processes. The experiment setup is shown in Table 23.

Number of cells	1853832
Other experiment details	P1/P2 - 20 Time iterations, one checkpoint at iteration 11
Used system	DEEP-ER SDV KNL
Compiler version	ifort version 16.2
MPI runtime version	ParaStation MPI version intel-mt-5.1.4
Number of nodes	1
MPI processes per node	64
Threads per process	4

Table 23: Experiment setup regarding SIONlib on KNL + NVMe (Inria)

6.4.2 NVMe

GERShWIN gained a promising I/O performance win when using the NVMe devices. This is already explained in D3.3 [6].

6.4.3 I/O offload with OmpSs

Following the results of Section 6.3.2, the application is expected to take advantage of offloading I/O from Booster to Cluster. Of course, the main target is application-level checkpointing: first, the application would benefit from the I/O performance of the Cluster (see Section 6.3.2) and second, this opens the possibility of overlapping write operations with the computation of the following time-steps. For this, a one Booster to one Cluster process mapping was built, with one Cluster core per Cluster process. This is done by the following MPI initialisation sequence:

```
CALL NANOS_MPI_INITF()
  CALL MPI_INIT_THREAD(MPI_THREAD_MULTIPLE, provided, mpierr)
  CALL MPI_COMM_RANK(icomm0, myid , mpierr)
  CALL MPI_COMM_SIZE(icomm0, nproc, mpierr)
    CALL DEEP_BOOSTER_ALLOC(icomm0, nproc, 1, icommB) ! Allocation of cluster resources.
```

As a first step, the offload of mesh-reading operations was implemented with OmpSs. This is easily achieved by offloading the opening and closing of the concerned files. Then, reading loops are offloaded by taskifying them and setting the concerned geometrical data with the `inout` attribute. These reading tasks are intertwined with verification steps, which is not a concern as soon as adequate sequencing of reading tasks is assured, for instance with the TASKWAIT construct.

Although the I/O-offload model could be successfully tested on small simplified pieces of code provided by OmpSs developers, the above-mentioned strategy could unfortunately not be tested and benchmarked at the level of the full application.

For reasons that remain unexplained by the end of the project, the application crashes when entering the first task lying in the scope of any subroutine following the initialisation subroutine. For instance, the following task will run from the initialising subroutine, but fail afterwards:

```
!$OMP TASK ONTO(icommB,myid)
          WRITE(6,*) "Hi from booster",myid
!$OMP END TASK
!$OMP TASKWAIT
```

By the end of DEEP-ER, this issue remains under investigation.

6.5 Comparison between architectures

Figure 36 shows the walltime of the time-loop for one single node of the different architectures available within the project, that is to say KNL (SDV Booster Node), KNC (DEEP Booster Node), Sandy Bridge (DEEP Cluster Node) and Haswell (SDV Cluster Node). These numbers result from selecting for each case the fastest combination of MPI processes / OpenMP threads.

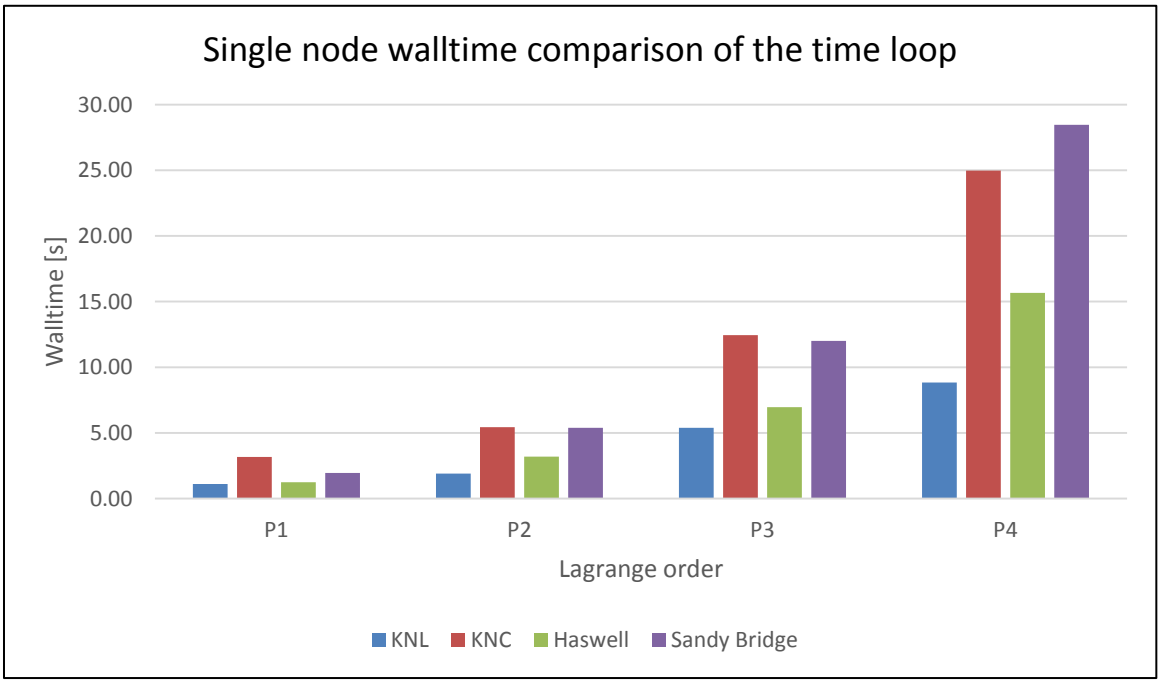


Figure 36: Single node walltime comparison of the time loop (Inria)

It is remarkable that using a full KNL node leads to walltimes that are 2.31x up to 2.86x smaller than using a full KNC node, which is a very positive result. It follows, in particular, that for all interpolation orders KNL nodes give a performance that is superior to Haswell nodes. The speedup resulting from using a SDV Booster Node with respect to a SDV Cluster Node is ranging from 1.12x (for P1) up to 1.77x (for P4). This is a satisfying result as it is recalled from D6.2 [3] that despite all the fine tuning on vectorisation of linear algebra

kernels, KNC-based results measured on the DEEP Booster Nodes did not show any acceleration with respect to Xeon nodes on the DEEP Cluster, except for the P4 case (with a quite moderate speedup of 1.13x).

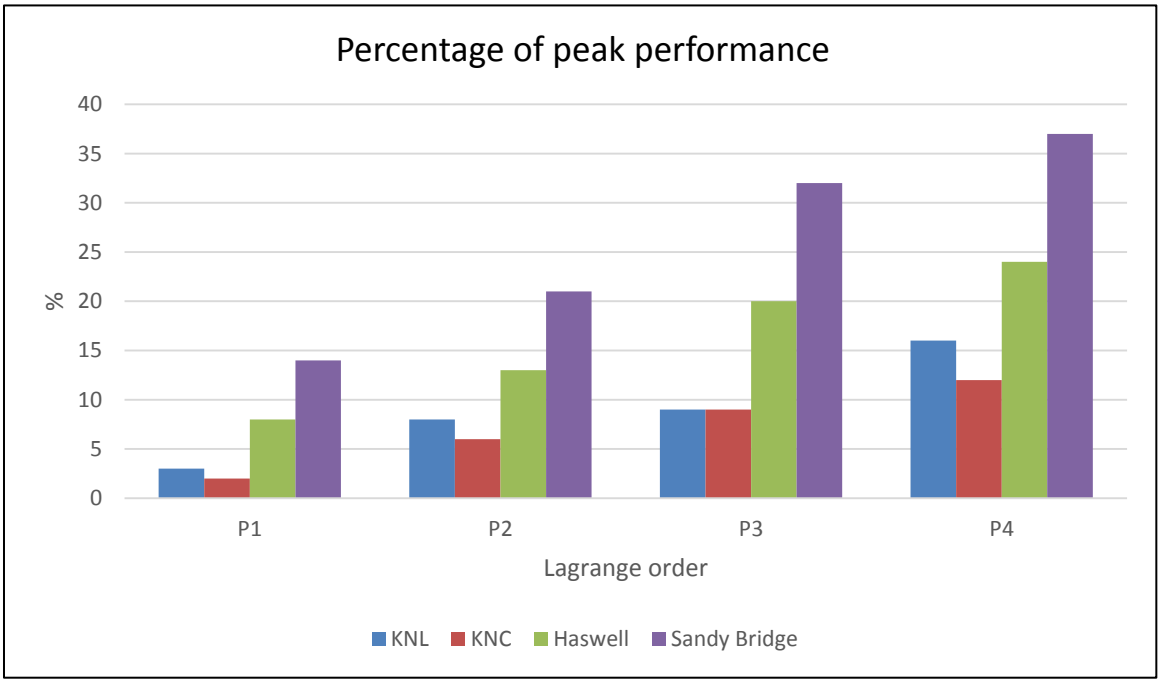


Figure 37: Percentage of peak performance (Inria)

Figure 37 shows the same results under the perspective of the percentage of peak performance. It shows the relative difficulty of optimising for accelerators and many-core CPUs compared to standard processing units for this application. Interestingly, a higher percentage of peak performance on KNL than on KNC can be observed, especially in the P4 case - which means the performance improvements compared to KNC are not only resulting from the extra cores and the higher clock-rate, but also from a better match between the algorithm and the hardware.

Number of cells	1853832
Other experiment details	Walltime of 20 iterations of the time-loop / No checkpointing
Used system	DEEP System (SNB) - DEEP-ER SDV (HSW / KNL) - 7120A KNC node on miclogin system
Compiler version	ifort version 16.2
MPI runtime version	ParaStation MPI version intel-mt-5.1.4
Compilation flags	-align dcommons -qopenmp -qno-opt-prefetch -qopt-report=0 -O3 -r8 -axMIC-AVX512 -fpp
MPI processes per Node	Xeon: 1 process per core. Xeon Phi: Best process-threads combination

Table 24: Benchmarking setup regarding architecture comparison (Inria)

6.6 Conclusion

The DEEP-ER Project has considerably impacted the overall performance of the Inria application.

First, it is important to remark that along the way towards adapting GERShWIN to the DEEP-ER architecture using DEEP-ER partner's tools - which we will sum up in the following - a number of more generic optimisations have been identified and implemented. Some of the knowledge acquired during the first year while profiling the application as well as subsequent improvements, in particular on communications and data locality enhancement, have diffused into other Inria in-house codes for the simulation of electromagnetic wave propagation problems.

On the computing-performance side, significant gains have been achieved in the process of porting and implementing fine-grain optimisations on the Knights Corner and Knights Landing architectures. Although such gains have also impacted performance on traditional cluster nodes, the fastest executions are expected to be achieved by exploiting the Booster side of the DEEP-ER architecture, as demonstrated in Section 6.5.

To achieve these results, it has been necessary to make progress on the programming models side and implement thread-based parallelism for a flexible exploitation of the Intel MIC architecture (it is recalled that GERShWIN was a pure MPI application by the start of the project). Although efficient task-based shared memory parallelism has been investigated, it could not be implemented as explained in Section 6.2.2 but consists in an interesting subject for further developments at Inria beyond DEEP-ER.

On the I/O and resiliency side, SIONlib is the tool of choice in the DEEP-ER software stack to improve GERShWIN's parallel output performance (which is mostly process-local), directly impacting application-level checkpointing performance. Also, the process-locality of outputs is such that it is natural to exploit on node NVMe devices for which results are reported in D3.3. On top of using on node NVMe devices, the integration of SCR and the persistent checkpoint-restart feature of OmpSs provides a convenient solution for managing checkpoints and restarts in a scalable way.

With all these optimisations and development efforts within the DEEP-ER Project, GERShWIN is not only well-prepared to exploit a Cluster-Booster-type architecture, but also made a significant leap towards efficiently harnessing Exascale supercomputers.

7 Task 6.5: Rapid crustal deformation & earthquake source equation (Task leader: BADW-LRZ)

The application under investigation (SeisSol) solves the partial differential elasto-dynamic equations for general seismological problems including geological faulting. The solver uses the Ader-Discontinuous-Galerkin (ADER-DG) method on unstructured tetrahedral computational meshes [7] [8].

Since the first open source release version optimised generated compute are using hybrid parallelism based on MPI and OpenMP. The optimised generated kernels are assembled within a pre-processing step [9]. While the first release version could not be considered production ready (as has been reported within Deliverable D6.1 [5]), it can be said that since recently the code base is in a much more mature state and scientists can now use the code for actual production runs. Since summer 2015 the code has been put into Open Source with a BSD-like license, with a repository hosted at GitHub [10]. For task 6.5 switching to the development branch of the code on GitHub positively impacts collaboration efforts. By using recent clones of development branches of the code base, effort is being saved and the sustainability of results is assured beyond project end. Improvements of the code continue to be a joint effort of registered SeisSol developers. Many shortcomings or design flaws of the baseline version of the code have been tackled outside of the DEEP-ER Project. Close collaboration with the main code-developers is on-going since DEEP-ER Q9 for I/O related improvements, which fostered (in joint effort) significant reimplementation for all I/O related code-regions compared to baseline version. Other code improvements include conditional compilation, hybrid parallelisation, explicitly vectorised code (SIMD instructions in generated kernels), most of which has been implemented externally but have been successfully adopted, ported and evaluated for chosen application use cases on DEEP-ER hardware and its specific software-stack. Fix bugs and shortcomings reported to either other DEEP-ER Work Packages or the application-developer-team increased robustness of the application.

7.1 Application overview

As a solver for the elasto-dynamic equations including nonlinear frictional sliding and plastic deformation SeisSol has excellent properties for simulating rapid crustal deformation and earthquake source dynamics. In the latter field many seismological solvers used in the community lack accuracy and/or flexibility of scenario description. The SeisSol solver in fact allows for both: accurate results through arbitrary high order numerical schemes (ADER-DG) and flexibility through unstructured tetrahedral meshes generated by external meshing software. Large computational meshes are partitioned by additional external partitioning software (e.g. Metis [11]). These partitions are then directed to the MPI tasks during execution.

A sophisticated clustered local time-stepping scheme has been implemented recently [12]. The introduction required changes of data-structures and a restructured time-loop workflow, which is not presented here in more detail. Practically the classic global time stepping remains now a simple realisation of the local time-stepping cluster with local time steps that do not vary. Central part of the code is now a program unit called time-manager, which orchestrates communications, clusters of elements and their integration. Optionally MPI communications may get a dedicated communication thread on architectures where beneficial (e.g. Haswell). There are two separate loops over the elements to compute local

integration and compute neighbouring integration. Within these required physical terms get integrated (time-, volume-, boundary-kernels as well as the constitutive equations). Figure 38 shows a simplified workflow of the application. Indicated by the colour (green for Cluster, blue for Booster) the solver part in the workflow can run on the Cluster or the Booster.

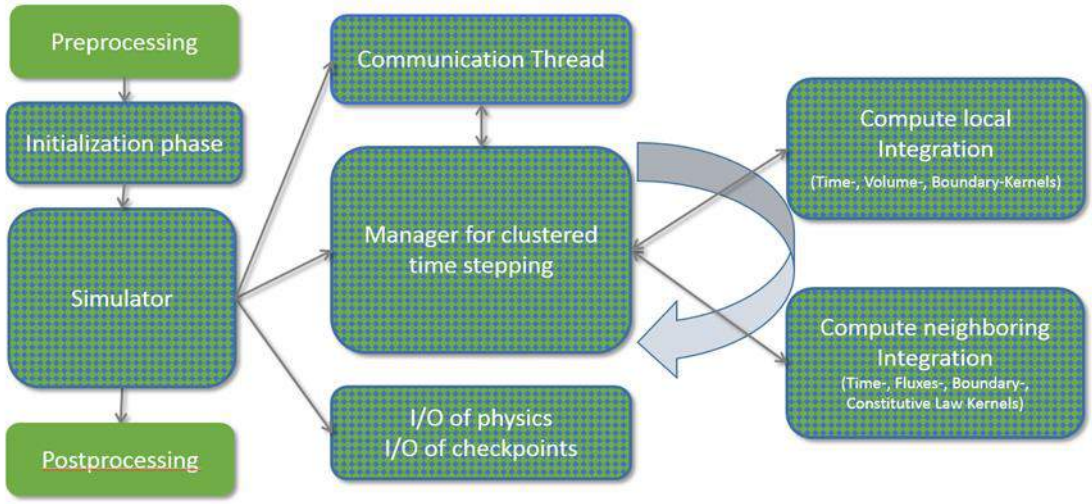


Figure 38: Workflow of SeisSol (BADW-LRZ)

7.2 Optimisations

Architecture specific optimisation is enabled through the combination of conditional compilation and (pre-) generated highly optimised compute kernels. Depending on the used subset of available physical terms (use case dependent) and the order of the applied numeric scheme (ADER-DG) there exist a fixed number of small matrix-matrix multiplications with a set of different sparsity patterns. Within the baseline of the code the implementation used sparse matrix notation using vector indexing to refer to non-zero matrix elements. These classical compute kernels perform reasonably on serial floating point units (~10% serial floating point performance). On the DEEP-ER target architectures however efficient use of SIMD vector instructions (AVX, AVX2, AVX512) and fast caches is essential for achieving maximum performance.

Eminent improvements in different parts of the code, with respect to the baseline-version, could be achieved by the application developers during the project time. These include model initialisation and efficient parallel read of the computational mesh, conditional compilation, usage of (pre-) generated highly optimised compute kernels, introduction of OpenMP thread parallelism, improved communication, improved I/O, and a new implementation of checkpointing and restart. The exchange of data was re-organised as an asynchronous communication scheme using only a small subset of the MPI-standard: `MPI_Isend`, `MPI_Irecv` and `MPI_test`.

7.2.1 Work done during the last project year

The last year of the project allowed for further evaluation of previously applied optimisation steps as well as for tackling few additional steps. The additions are (among others), extending capabilities of the modular checkpointing implementation as well as incorporating special DEEP-ER technology, e.g. E10 (with MPIWRAP), NVMe (through SIONlib or MPIWRAP), and broadening of hardware-specific optimised kernel support (e.g. KNL) for

additional physical aspects of the code (e.g. bulk plasticity), which could recently be tested on Intel KNL-systems provided though the DEEP-ER-Project. After fixing/working around some trouble leading to NAN-floating-point results and internal compiler error on KNL systems, very good performance of the application may be reported for the benchmarking cases used on the DEEP-ER-SDV KNL nodes.

7.2.2 Progress overview

Workflow elements	M1-M6	M6-M12	M13-M18	M19-M24	M25-M30	M31-M36	M37-M42
Analysis	1	2	2	2	2	2	2
Writing D6.1	0	1	2	2	2	2	2
Port to KNC	2	2	2	2	2	2	2
Threading	1	1	1	2	2	2	2
Vectorisation	1	1	1	2	2	2	2
General code optimisation	1	1	1	1	1	1	2
Implement OmpSs parallelisation and/or offload	-	-	-	-	-	-	-
Cluster-Booster division	-	-	-	-	-	-	-
Port to SDV	0	0	0	0	2	2	2
Use NVM	0	0	0	0	1	1	1
SIONlib integration	0	1	1	1	1	1	2
E10 integration	0	0	0	0	0	0	1
General I/O optimisation	1	1	1	1	1	1	2
Implementing a mockup	0	0	0	0	1	2	2
Writing D6.2	0	0	0	0	2	2	2
Checkpointing on NAM	0	0	0	0	0	0	0
Optimise for KNL	0	0	1	1	1	1	2
Implement SCR	0	0	0	0	0	0	1
Implement OmpSs task based resiliency	-	-	-	-	-	-	-
JUBE integration	0	0	0	0	0	0	0
Final benchmarking	0	0	0	0	0	0	2
Writing D6.3	0	0	0	0	0	0	2

Table 25: Progress overview (BADW-LRZ)

- OmpSs: potential use case discussed early during a F2F meeting with partner BSC, but found to be no alternative to the OpenMP implementation -> this means no task-based resiliency either.
- Cluster-Booster division: artificial division for application -> requires extra load-balancing effort. Code runs efficiently on either Cluster or Booster.
- E10 has been implemented, its build and batch processing workflow has been tested for functionality, its performance evaluation is pending due late delivery of the E10 software stack, with last-minute performance evaluation trails conducted, but not yet without run-time failure when using MPI-IO: will hopefully have results until the final review
- libNAM not envisioned directly, but with SIONlib-integration could be tested (delay in software stack).
- SCR has been prototyped, testing is still pending, considered E10 of higher importance, and will hopefully have results for the review meeting.

- JUBE integration dropped in favour of supporting other DEEP-ER technology mentioned above.

7.3 Resiliency

The resiliency strategy is based on application-based writing of checkpoint files, which allows for automatic restart of the simulation from the latest valid checkpoint files in case such files are found inside the simulation directory. In case no valid checkpoints are found, the computation would alternatively start from the beginning. The frequency for writing checkpoint files may be specified within the regular application-parameter input file. However, SCR provides the possibility to let the system trigger checkpointing and allows for optionally overriding the user-specified checkpointing-frequency. This frees the user from making considerations on meaningful frequency. A focus within the DEEP-ER Project has been on supporting a range of different I/O-backends that allow for a system specific “best” decision on the I/O-backend.

The used test case is characterised in Table 26. Figure 39 left panel shows an estimate on the resiliency overhead of the application for this test case when checkpointing every 1000 iterations, while Figure 39 right panel shows the restart overhead with respect to a checkpointing interval (here 1000 iterations), both as a function of the number of SDV nodes (strong-scaling). The resiliency overhead for checkpoint writing (left panel) for the SIONlib backend is best, it is always below 2%. As a linearly related measure the number of iterations missed per checkpoint is also given (right y-axis-scale). The restart times relative to checkpointing interval (right panel) is more costly and becomes a little bit more independent on the I/O-backend, but also is best for using SIONlib: checkpoint restart in the given example is y<15% for the assumed checkpointing period (1000 iterations).

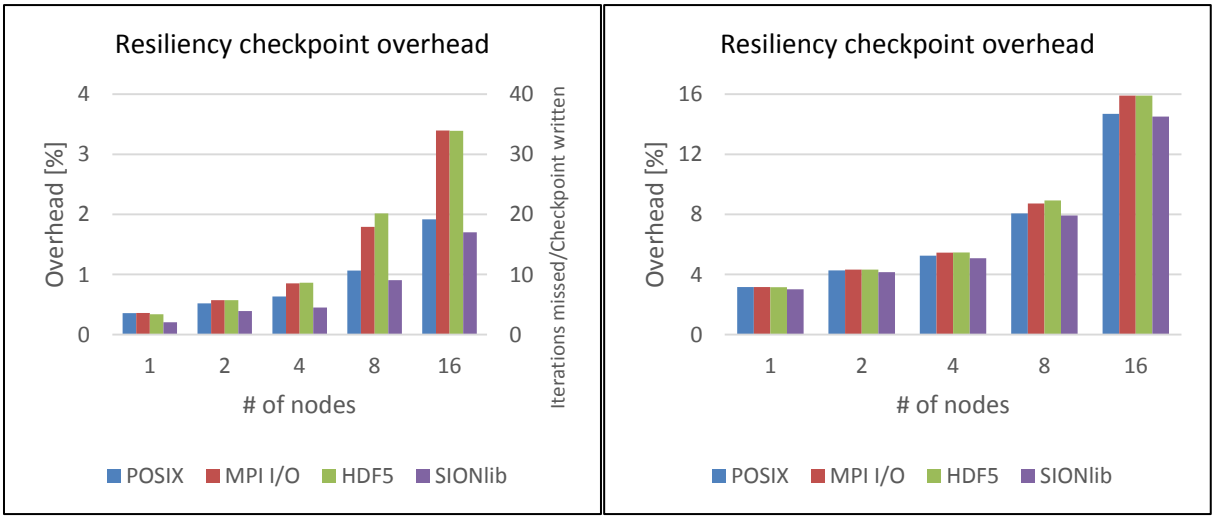


Figure 39: Resiliency checkpoint and restart overhead (BADW-LRZ)

Experiment details	LOH4, small (250000 elements)
Used system	DEEP-ER-SDV (Haswell)
Compiler version	16.3
MPI runtime versions	ParaStation-MPI (DEEP-ER)
Compilation flags	-O2 –AVX2 –ip

MPI processes per node	1
Threads per process	24

Table 26: Experiment setup regarding resiliency (BADW-LRZ)

7.4 Input/Output

As already reported in Deliverable D6.2 [3], considerable work has been done on the implementation and improvement of the checkpointing-related I/O of the application. Its modular design allows for easily switching actual I/O backends for individual simulation runs. Existing backends currently are based on POSIX-I/O, MPI I/O, HDF5, and SIONlib. First results reported in D6.2 suggested excellent performance of SIONlib-based backend on the DEEP/DEEP-ER systems (as well as other large-scale installations, e.g. SuperMUC or HazelHen). However, the I/O-flush or I/O-sync facility of the SIONlib API does not force (and ensure) data being always flushed to the file system before returning to user code. This however has been assured by the other backends in the comparison, and therefore invalidates previously reported results using SIONlib. With recent version of SIONlib it is therefore necessary to explicitly call a file-close-procedure to ensure a complete data-flush to the file system (`sion_parclose_mpi`). This in turn requires the file-open-procedure (`sion_paropen_mpi`) being called each time a checkpoint has to be written. Using this scheme, SIONlib loses its previously reported excellent performance on the DEEP-ER SDV for the application code of WP6.5. This feature has been discussed with SIONlib developers but implementation of such a file system flush functionality in future versions remains uncertain until the project end.

As reported earlier in deliverable D6.2 as well as at the F2F meeting held on April 2016 in Sophia-Antipolis: on the SDV the I/O-characteristics differ from those of tested large scale installations (e.g. SuperMUC).

For the SIONlib backend the influence of the number of SION files was tested, the choice of SION backend and of collective I/O option. Figure 40 shows part of the results for this effort. Note here that recent updates of the I/O software stack brought significant performance improvements, but Figure 40 refers to data that was collected on an earlier version of the I/O software components and therefore shows out-dated absolute bandwidth numbers. The used benchmark (LOH4) is the same as described in Table 28 of Section 7.5.1 but with checkpointing enabled and running on 16 SDV-HSW nodes. It can be seen that the choice of backend for SIONlib has the most relevant effect on performance (~50% performance decrease going from ANSI-C to POSIX) after this the number of SION files has the second most relevant effect (e.g. about 36% performance impact increase going from writing 1 to 4 files in non-collective mode). While there exist parameter cases where using the collective mode on the SDV brings also some benefit, its impact being less important, with benefit always less than 20% (in all cases) and typically below 10%.

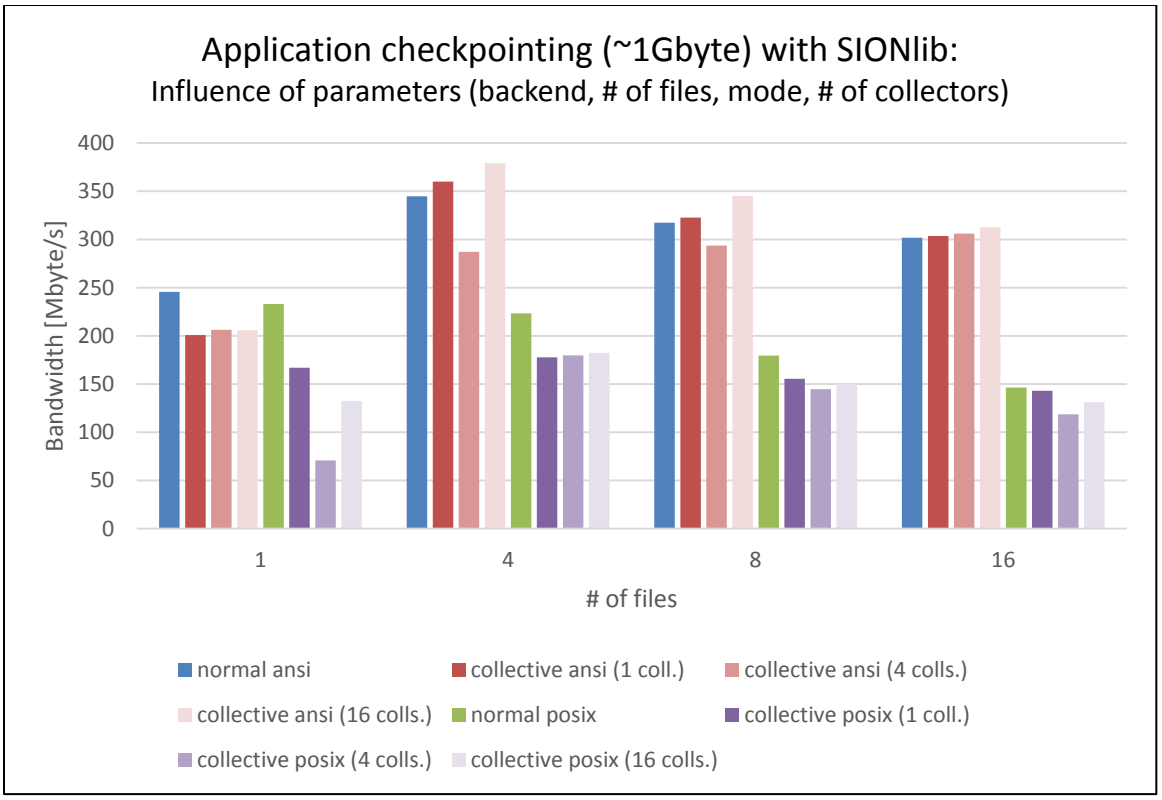


Figure 40: Tuning of SIONlib, exploring SIONlib-I/O-options (BADWLRZ)

Despite intermediately reported poor bandwidth results on SDV using SIONlib (Q12), since recent updates of the I/O software stack, including upgrades of BeeGFS and SIONlib, the WP6.5 application achieves much better and now satisfying I/O performance results on the SDV, especially with respect to SIONlib checkpointing backend. Recent results obtained on the SDV for all mentioned I/O-backends are shown in Figure 41, with corresponding parameters shown in Table 27. The plot shows I/O bandwidth in a strong scaling setup on 1 to 15 SDV-HSW nodes for the different I/O backends available (colour coded) and for different problem sizes (line style coded). Here the SIONlib backup uses default parameters (backend: ANSI-C, number of SION files 1 (which is automatic default on the SDV), non-collective). All backends score a bandwidth of almost 800 MiB/s for all problem sizes on a single node, except for SIONlib, which scores around 1200 to 1300 MiB/s on a single node and for all problem sizes. The POSIX backend scales with increasing bandwidth when using more nodes, reaching around 1200 MiB/s for all problem sizes when running on 15 nodes. The SIONlib backend also shows increasing performance with increasing number of nodes, reaching a bandwidth of around 1500 to 1700 MiB/s on 15 nodes. However, the maximum bandwidth achieved is found when utilising either 4 or 8, depending on the problem size. For the MPI I/O and the MPI-based HDF5 backends, the maximum bandwidth achieved is 900 MiB/s on 2 or 4 nodes with decreasing bandwidth for higher node counts. E.g. for the problem size of 1 M elements or 4032 MiB/checkpoint the bandwidth drops to around 600 to 650 MiB/s for both: the HDF5 as well as the MPI I/O backends, that is below 40% of SIONlib performance.

Scaling	SDV: 1 to 16 cluster Nodes (= 24 to 384 Haswell cores)
Number of cells	[125k, 250, 500, 1000] k elements ~ [504, 1008,2016,4032] MiB/cp
Other parameters	LOH4-like strong-scaling setup
Compiler version	16.2
MPI runtime version	ParaStation MPI
Compilation flags	-O2 –AVX2 –ip
MPI processes per Node	1
Threads per process	24

Table 27: Benchmarking setup regarding I/O optimisation (BADW-LRZ)

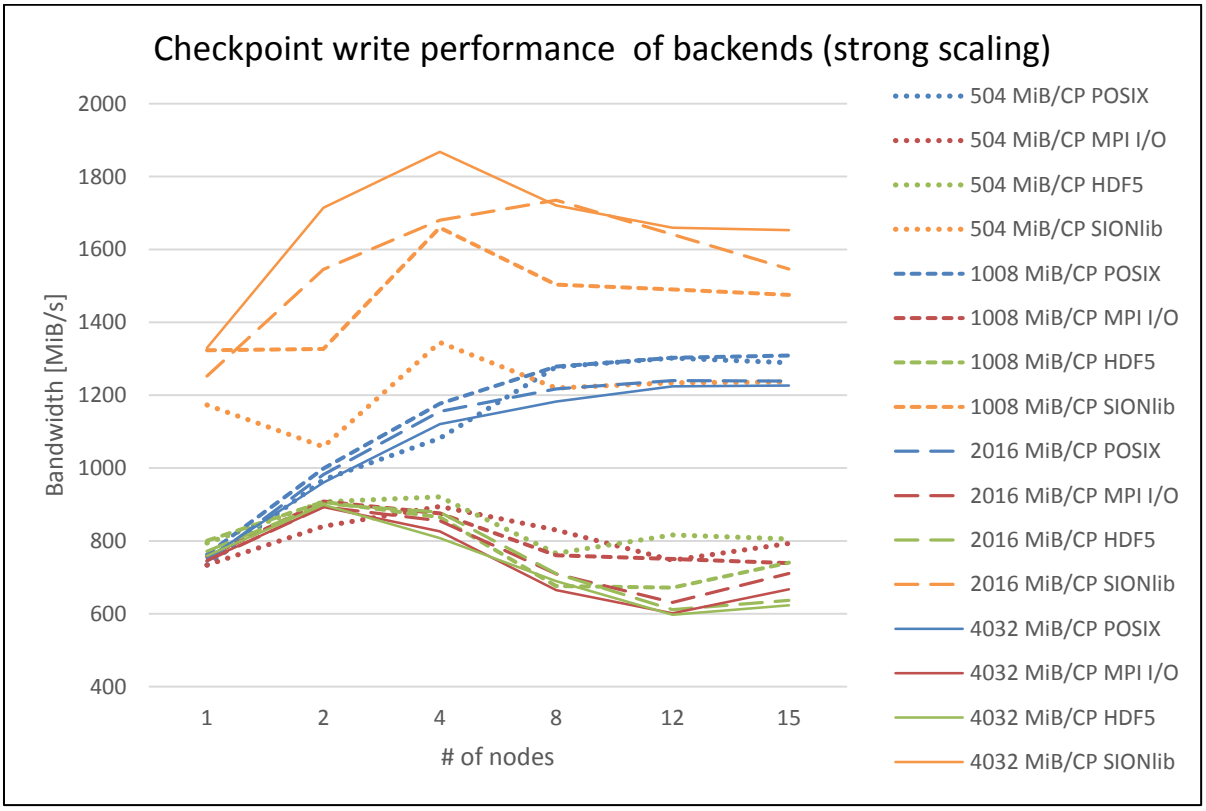


Figure 41: I/O backends in comparison on the DEEP-ER SDV (BADW-LRZ)

The range of available options for SIONlib makes it possible (and potentially necessary) to tune for better I/O performance. A checkpointing mockup has been created to more efficiently explore the parameter space available. The mockup can be run as a pre-processing step inside the job batch file to find most promising mode for SIONlib before execution of the actual application. One realisation of such a parameter space sampling is illustrated in Figure 40. Some of the parameter combinations do not necessarily modify the performance of SIONlib too much. However, we found as a minimum tests should comprise the number of files and the choice of SIONlib backend. While ANSI-C is the default backend for SIONlib and is the most well performing backend on the SDV (up to 2x faster than POSIX), on other systems this may not be the case: e.g. on SuperMUC our tests indicate that switching to POSIX backend is recommended. It can be observed that the latter is about 3x faster than using the default (ANSI-C) option.

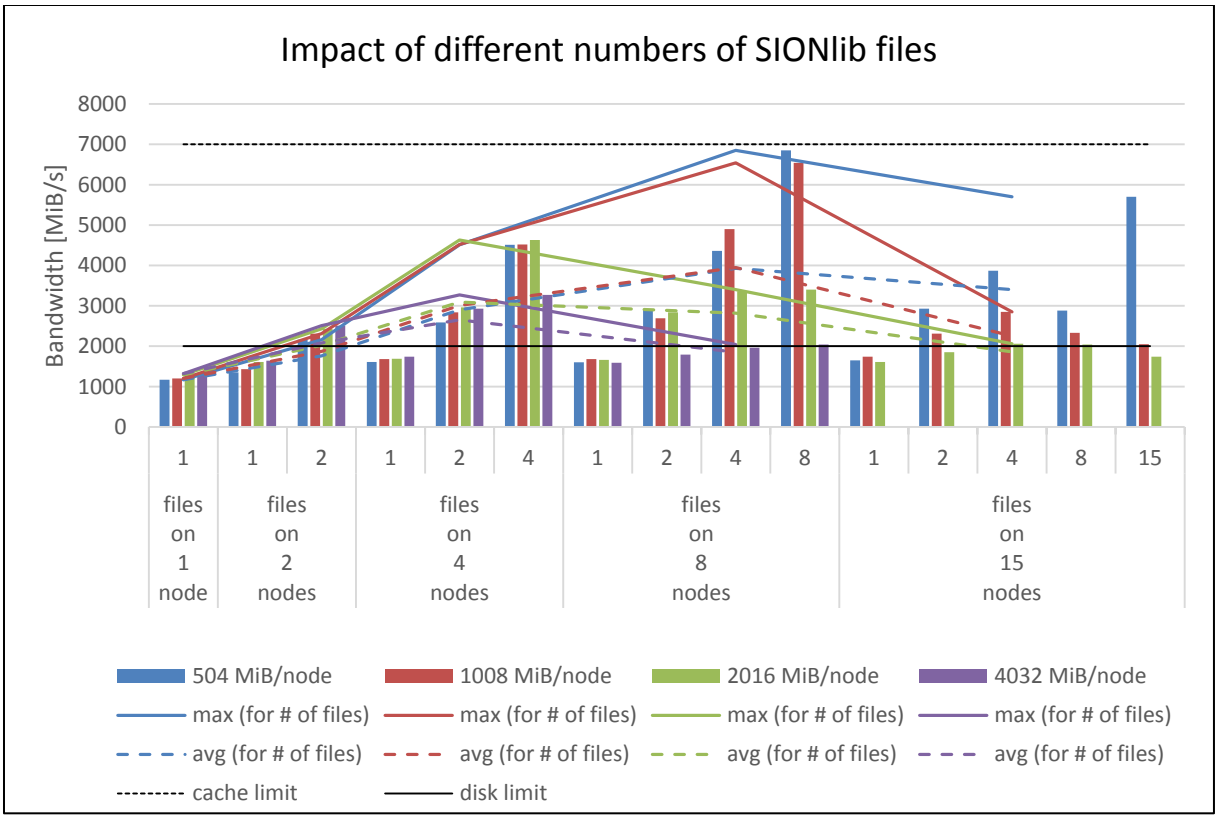


Figure 42: Tuning SIONlib I/O backend on the DEEP-ER SDV (BADW-LRZ)

Figure 42 shows weak scaling I/O benchmark results of write performance in MiB/s for 4 different data volumes per node (colour coded). The main focus here has been on writing into different number of SIONlib files (keeping the SIONlib backend as ANSI-C). It was tested by fixing the number of used nodes as the upper limit for number of files to be written, e.g. when using 2 nodes [1,2] was tested for number of files or when using 8 nodes it was tested for number of files being [1,2,4,8]. It can be seen that the maximum bandwidth stays below the accumulated disk bandwidth of the BeeGFS partition /sdv-work, which is about 2000 MiB/s. But when using 2 nodes and 2 files the measured bandwidth is already beyond that limit, which gives indication that the I/O subsystem enables usage of cache buffers for data storage. Maximum buffering can be observed when utilising 8 nodes and writing into 8 files. In this case the measured write performance is slightly below 7000 MiB/s, which also denotes about the limit of cache bandwidth, for the tested volumes of 504 MiB/node and 1008 MiB/node in this weak scaling test. General observation that were made here is that when increasing the number of nodes, increasing the number of files will improve performance. However, for the case of 15 nodes used (at the time of testing 16 nodes have not been available) the best performing number of files is 4, except for the smallest data volume for which it is 15 files. Figure 42 therefore gives indication that there exist a parameter range for which cache buffering of data may make I/O-overhead for the application (e.g. for checkpointing) very small.

In previous deliverables and project meetings it was reported that the POSIX-based implementation performs best on tested large-scale systems (e.g. SuperMUC, see Figure 43). However, it is also worth mentioning that on SuperMUC the POSIX-based backend did fail completely from time to time for very large parallel runs (high number of nodes = high number of MPI tasks). One therefore may conclude that when striving for large scale simulation scenarios that require a large number of MPI tasks (=number of POSIX files) to

choose POSIX task-based I/O pattern may potentially be risky (e.g. on SuperMUC with GPFS) if not taking extra effort (e.g. grouping of output in several subfolders on the file-system). However, on the small-scale DEEP-ER SDV installation this kind of task-based I/O pattern can't be pushed to large scale.

Figure 43 also shows the POSIX-based checkpointing achieving write performance beyond the file system limit. Measured bandwidth for the POSIX backend has also been observed on large-scale strong scaling I/O-benchmark on SuperMUC (filesystem: GPFS), see Figure 43, to go beyond denoted file system limit.

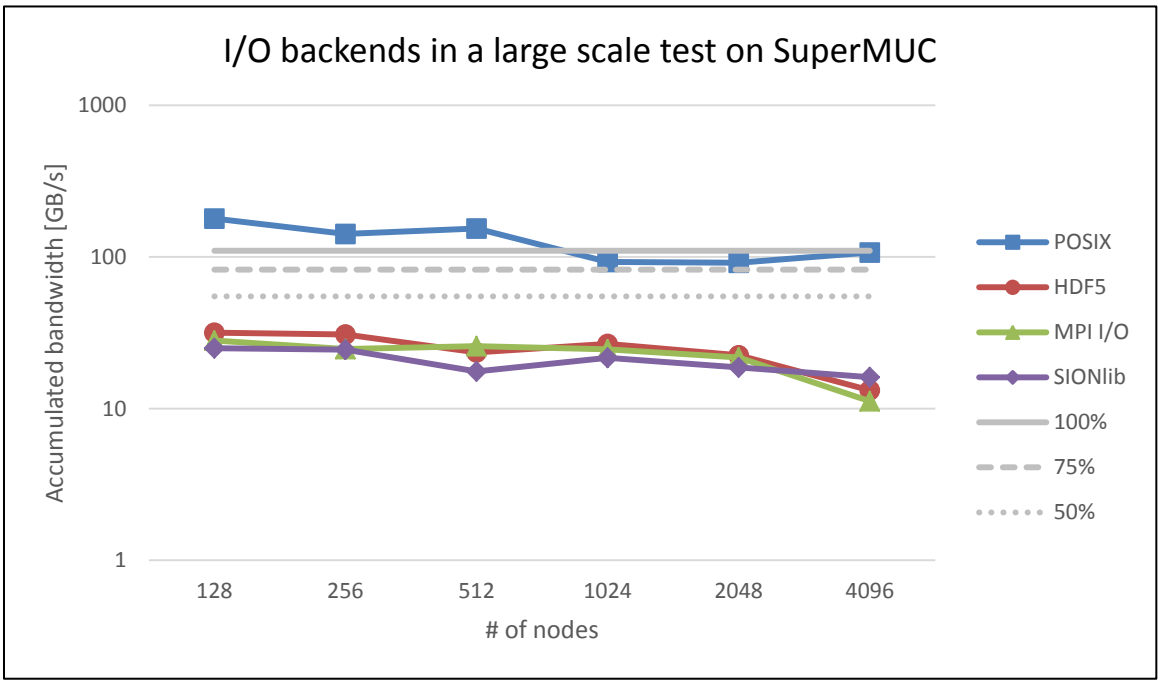


Figure 43: I/O backends in large scale strong scaling I/O tests on SuperMUC (BADW-LRZ)

This behaviour may be explained by file system internal buffering (e.g. on SuperMUC the buffer is of the order of several hundreds of Gigabytes), this means data is flushed/synchronised onto the parallel file system but not necessarily being written onto a disk yet. However, in case of compute node failure or some other trouble causing program execution to stop just after writing a checkpoint, with data still just inside the file system cache, the checkpoint-files would be expected to be valid and consistent in the file system nonetheless, if failure is not triggered by a failing client that holds the data. A restart from these checkpoints should therefore be possible, but this assumption could not be tested rigorously. In any case the checkpoint files are not consistent, a restart from previous checkpoints should be possible instead.

Efforts have been successfully applied to the build-, link- and batch-processing of the application for use of the E10 related I/O facilities. These allow for further tuning of the MPI I/O as well as the HDF5 backends. This is done by linking the MPIWRAP library [13], which may trigger asynchronous I/O as well as transparently use the NVMe devices as fast storage buffers. The required generation of the MPIWRAP configuration file is incorporated into the application workflow and batch file. Initial results are yet pending since bug-fixed version of the underlying software stack just became available at the time of writing, but results can be made available for the final review meeting of the DEEP-ER Project.

A downside of providing several I/O backends is the potential need for installing the additional software dependencies, which takes extra effort and has been proven to be a potential source of additional trouble and delay. Since the I/O facility is part of MPI standard and POSIX API is typically available on HPC systems, the backends that do not require additional libraries are MPI I/O and POSIX and may be valid choice in situations in which these extra dependencies cannot quickly or easily be resolved.

Recent application developments add asynchronous options for all backends. This is done by explicitly forcing data writing to be asynchronous from the program execution (overlapping computation and checkpoint-writing). A number of MPI tasks has to be reserved to do the actual I/O operations.

Figure 44 shows the I/O single node performance using POSIX and SIONlib I/O backends. The measured performance is slightly less than achieved for single node performance on the BeeGFS partition /sdv-work. However – this performance is expected to scale with the number of used NVMe devices and would give in total a larger I/O performance.

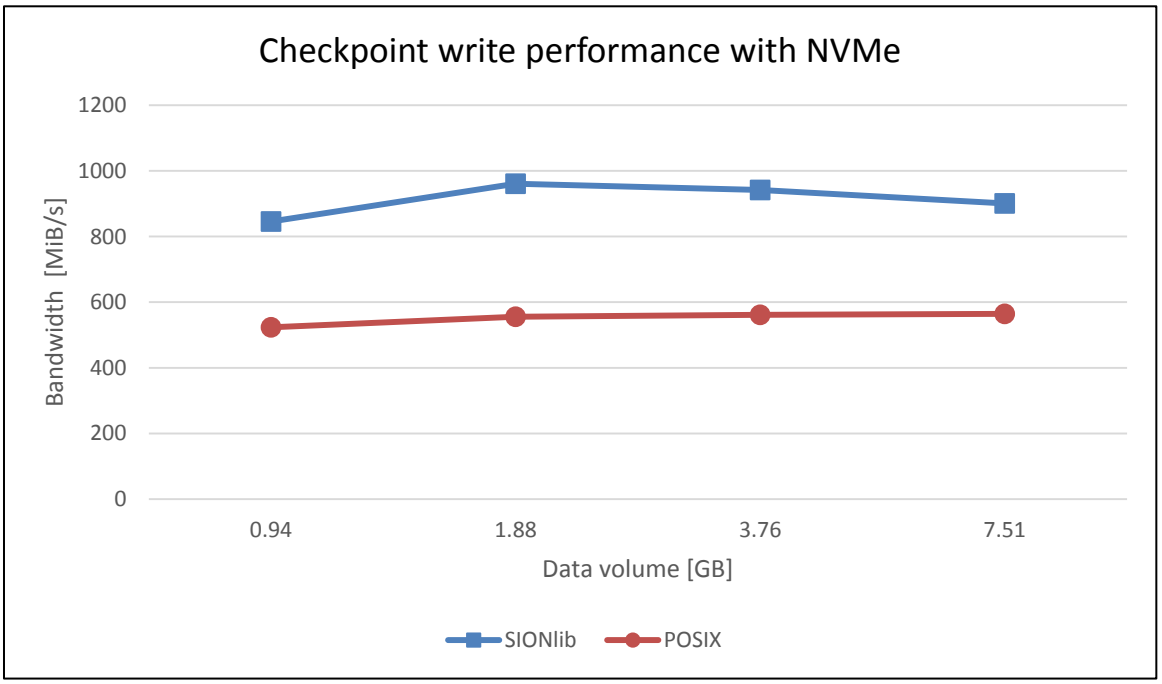


Figure 44: I/O on the NVMe devices using BeeOND (BADW-LRZ)

7.5 Comparison between architectures

7.5.1 Scaling and per node performance measures

As mentioned previously, for computationally efficient simulations the code relies entirely on the highly optimised compute kernels for small matrix-matrix multiplications as provided by the Intel supported libxsmm library [14]. Recent code versions on GitHub now provide the possibility of kernel-generation on the fly within the SCons build procedure. With this the generation step becomes more generic and the framework may be used in connection with potential other code generation backends that may be suitable for other architectures.

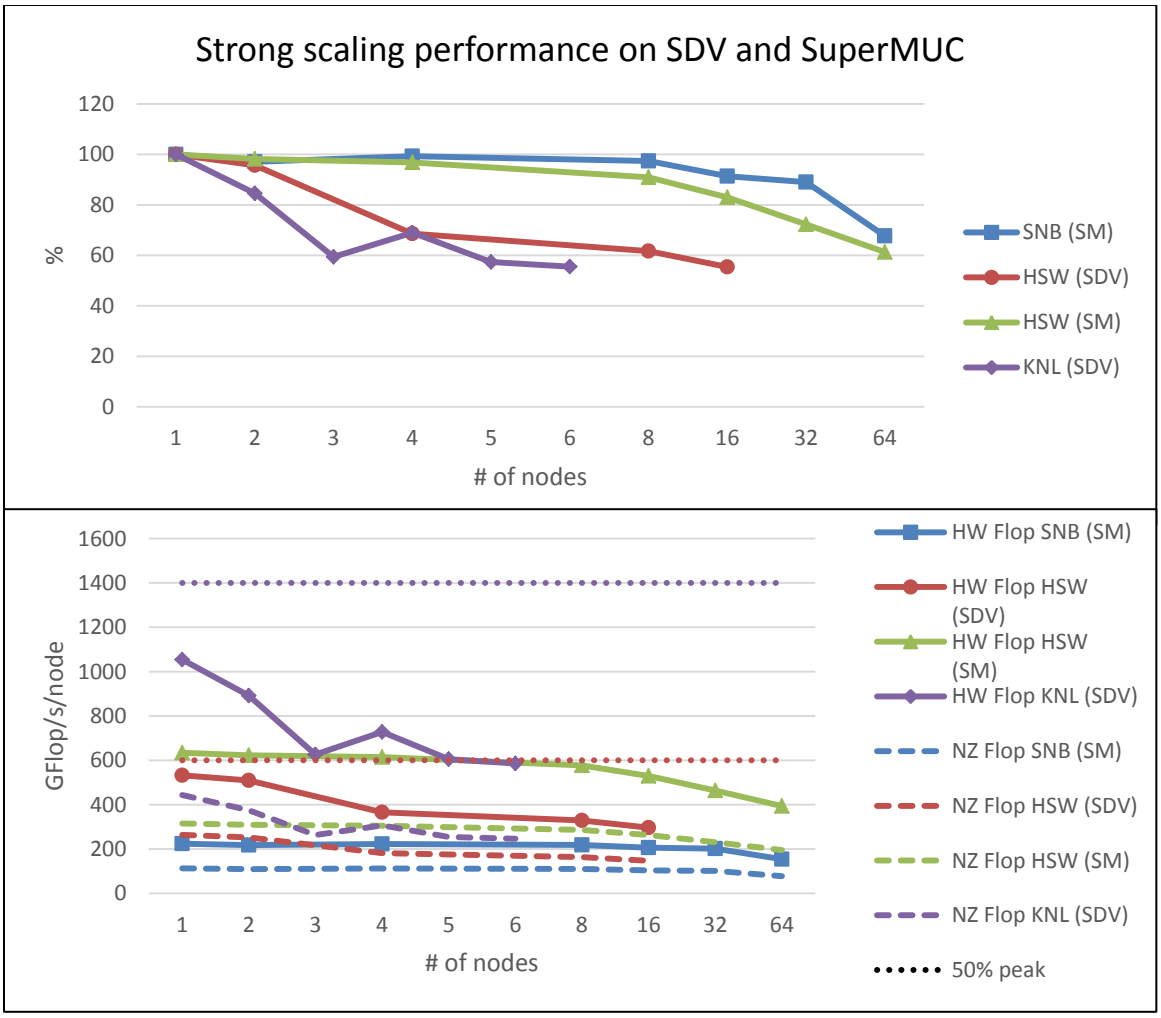


Figure 45: Performance results for the considered Intel CPUs: SNB/HSW/KNL (BADW-LRZ)

Experiment details	LOH4, small
Used system	SuperMUC (SandyBridge), DEEP-ER-SDV (Haswell + KNL)
Compiler version	16.1
MPI runtime versions	Intel MPI (SuperMUC), ParaStation MPI (DEEP-ER SDV)
Compilation flags	-O2, -ipo, -ip AVX/AVX2/AVX512
MPI processes per node	1 (SNB) and 1 (HSW) 1 to 4 (KNL)
Threads per process	16 (SNB) and 24 (HSW) 16 to 64 (KNL)

Table 28: Benchmarking setup regarding LOH4 run (BADW-LRZ)

Given the presently relatively small DEEP-ER SDV System available (with 16 Haswell nodes and 8 KNL nodes only) the LOH4 benchmark case was used. The benchmark has been described in previous deliverables and is summarized in Table 28. In Figure 45 scaling results are shown for the achieved parallel efficiency (top-panel) and floating-point performance (bottom-panel) as a function of the number of nodes. Shown are results on SDV-Haswell nodes (HSW), SDV-KNL nodes (KNL) in comparison with corresponding results on the older SuperMUC Phase 1 Sandy Bridge (SNB SM) and SuperMUC Phase 2 Haswell (HSW SM) nodes. The top panel shows the scalability of code use case in terms of

parallel efficiency, while the bottom panel shows the corresponding floating-point operations achieved per node (in GFlop/s/node). Bottom panel show that a very good node-level performance is achieved on all tested Intel architectures, while top-panel shows that the application scales better on SuperMUC nodes than on the DEEP-ER SDV.

On the per node level comparison it can be seen the best performance on KNLs followed by Haswell nodes then Sandy Bridge. The different architectures provide different core counts and therefore one may alternatively put up such a ranking with respect to the number of cores. Since the core counts per node are 64, 24, 28, and 16 for the systems DEEP-ER KNL / DEEP-ER HSW / SuperMuc HSW / SuperMUC SNB, Figure 45 bottom panel shows that HSW nodes provide the best performance per core. However, having 64 KNL cores per node the achieved performance gain of KNL over HSW is in the range of 1.4x to 1.7x for the inspected use case configuration.

The different architectures provide different core counts and therefore one may alternatively put up such a ranking with respect to the number of nodes. When comparing results shown in D6.2 for the KNC (here based on slightly out-dated code version), a performance gain of KNL over KNC of about 2.5x was achieved. Because of the small core count per node and smaller vector length of the SIMD instructions, Sandy Bridge nodes provide the smallest performance for the SeisSol code. Recalling the performance of the original baseline (as reported in D6.1) a total performance gain in the order of 25x was gained when comparing the baseline code on Intel Sandy Bridge based systems and recent code version (as published on GitHub) on KNL systems. On the SDV Haswell and KNL nodes the code runs efficiently using a single MPI task per node and 24 or 64 OpenMP threads (corresponding to the number of cores on the nodes). However, the performance is mostly unchanged when placing several MPI tasks and a reduced number of OpenMP threads per node instead for the tested setups. Note that for large scale systems (many thousands of nodes) and with respect to I/O, keeping the number of MPI tasks low is favourable.

Strong scaling is always limited since the problem size is limited. This is also reflected in the result presented in Figure 45. However, what can quickly be seen (especially in the top-panel plot, parallel efficiency) is that scaling behaviour on SNB-SM and HSW-SM nodes is superior over the nodes on the DEEP-ER SDV for both KNL as well as HSW. Increasing node-level hardware performance may result in decreasing scalability and parallel efficiency since the fraction of time spend in communication relatively increases. Here the HSW-SM nodes have a slightly larger theoretical performance than the SDV-HSW nodes because of the higher core count (28 over 24) and therefore this cannot explain observed deficit of scaling on the DEEP-ER SDV. At the time of writing, efforts are still ongoing to find the origin of trouble that, from current perspective, appear to be related to the used MPI (ParaStation MPI) and/or network (EXTOLL) or its current system configuration.

7.6 Conclusion

The DEEP-ER WP6 provided very good opportunities for porting, evaluating, and improving a state-of-the-art earthquake rupture dynamics source code that is in active development to latest hardware technology and an innovative software stack. This project also helped improving I/O capabilities in general as well as exploring efficient checkpointing capabilities in detail. New technologies (SIONlib, E10/MPIWRAP, BeeOND with NVMe) could be tested and either became (or will become) valuable I/O options in the application. Although the DEEP-ER related systems available for evaluation were rather small throughout the project-

time, the code development could benefit directly as well as indirectly by given feedback to other code developers (outside the DEEP-ER Project). At the time of writing, the code can already be considered ready for use on next-generation supercomputer installation soon to come, and possibly being an application-software candidate for the first Exascale systems in in the future.

8 Task 6.6: Radio Astronomy (Task leader: ASTRON)

In the previous two Deliverables the focus was on two computational parts of creating sky images from radio telescope data: correlation and imaging. This report addresses the parts that have been left out of the discussion before such as reading/writing data from/to persistent storage, performing a data transpose via the network, optimisations for Intel Xeon Phi Knights Landing (KNL), and incorporating DEEP-ER resiliency features.

8.1 Application overview

The application consists of two parts: correlation and imaging; both steps have been discussed in detail in Deliverable 6.1 [5] and 6.2 [3], respectively. Roughly speaking, the correlation step combines the signals from the individual stations (groups of antennas) by multiplying and accumulating the signals over time. The data products of the correlation are called visibilities. The imaging step converts visibilities into sky images. This workflow is shown in Figure 46.

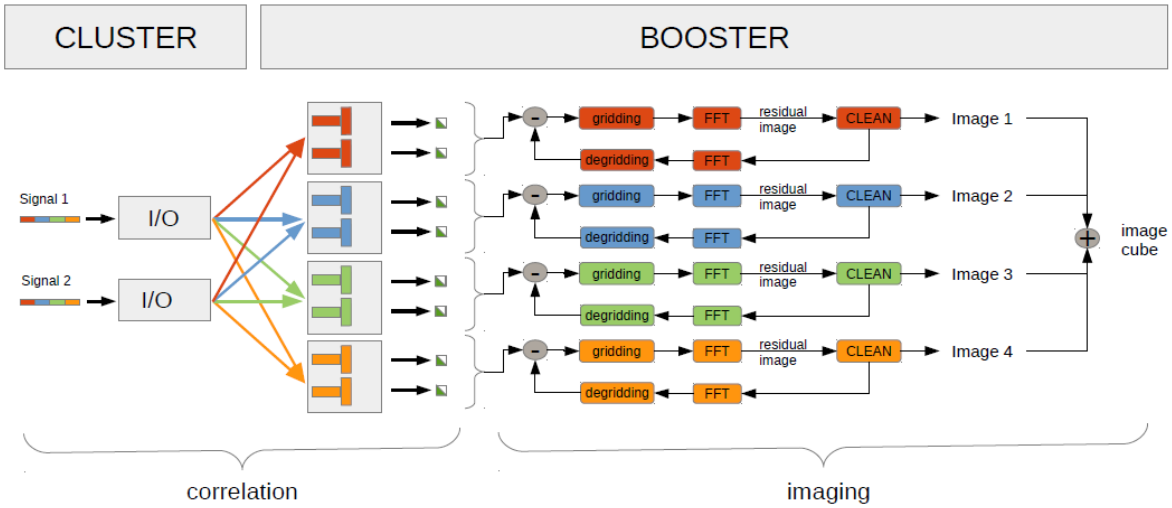


Figure 46: Workflow of the data processing pipeline (ASTRON)

8.1.1 Correlation

The telescope data is streamed and processed in the following order: For a set of I/O processes, UDP packets of data are received over wide-area network and placed into buffers. Each buffer is distributed in a Round-robin schema to a set of compute processes (the data distribution is often referred to as “the corner-turn”). The compute processes filter and correlate the data (the entire computation is referred to as “the correlation”). Finally, the resulting visibilities are written to persistent storage. The entire process is done in real-time, provided each operation can handle the incoming data streams. For details, see Deliverable 6.1 and [15].

8.1.2 Imaging

The creation of a sky image is an iterative process (see Figure 46): At each iteration the visibilities are gridded to obtain (up to a simple correction) the discrete Fourier transform of a sky image. From the image (obtained via an inverse Fourier transform), the bright sources are extracted to successively build a model of the sky. The extracted sources are used to predict (FFT and degridding) corresponding measurements. The predicted visibilities are

subtracted from the real measurements. The iterative sky model creation stops when the instrument noise level is reached. For details, see Deliverable 6.2 [3] and [16].

8.1.3 Mapping onto the DEEP-ER Prototype

For the correlator, the I/O processes are executed on the Cluster Nodes and the compute processes are executed on the Booster Nodes (see Figure 46). The resulting visibilities are written to local NVMe devices, which serve as a temporary buffer. The imager runs entirely on the Booster Nodes and converts the locally buffered data into images that are written to the global file system.

8.2 Progress overview

Workflow elements	M1-M6	M6-M12	M13-M18	M19-M24	M25-M30	M31-M36	M37-M42
Analysis	1	2	2	2	2	2	2
Writing D6.1	0	1	2	2	2	2	2
Port to KNC	0	1	2	2	2	2	2
Threading	2	2	2	2	2	2	2
Vectorisation	2	2	2	2	2	2	2
General code optimisation	1	1	1	1	1	1	2
Implement OmpSs parallelisation and/or offload	0	0	1	1	2	2	2
Cluster-Booster division	0	0	1	1	2	2	2
Port to SDV	0	0	0	0	2	2	2
Use NVMe	0	0	0	0	0	0	2
SIONlib integration	-	-	-	-	-	-	-
E10 integration	-	-	-	-	-	-	-
General I/O optimisation	1	1	1	1	1	1	2
Implementing a mockup	0	0	0	0	2	2	2
Writing D6.2	0	0	0	0	2	2	2
Checkpointing on NAM	-	-	-	-	-	-	-
Optimise for KNL	0	0	0	0	0	1	2
Implement SCR	0	0	0	0	0	0	2
Implement OmpSs task based resiliency	0	0	0	0	0	0	2
JUBE integration	0	0	0	0	0	0	1
Final benchmarking	0	0	0	0	0	0	2
Writing D6.3	0	0	0	0	0	0	2

Table 29: Progress overview (ASTRON)

Mockup implementations (restricted implementations with minimal external dependencies) of both correlation and imaging were created. Both have been optimised for the Xeon (SandyBridge, Haswell) and Xeon Phi (KNC, KNL) architectures of the DEEP/DEEP-ER Projects. The imager makes use of a newly developed library of a novel algorithm called image-domain gridding (IDG) [16]. The resulting library is currently being incorporated in the widely-used WSClean imager [17].

Based on absolute performance, the correlator uses both Cluster Nodes and Booster Nodes (the latter for the scalable computation) and the imager works on the Booster Nodes exclusively.

As both correlation and imaging are essentially streaming processes working on distinct data sets, the use of scalable parallel I/O in form of E10 and SIONlib was not found beneficial in this application after a detailed discussion with the experts of WP4. However, the local use of persistent storage has been optimised (using the HDF5 library) and the use of the NVMe devices as fast, scalable buffers for data and checkpoints has been evaluated. The checkpointing uses the Scalable Checkpoint/Restart (SCR) library extension provided by WP5. Furthermore, the OmpSs tasked-based resiliency and ParaStation MPI resiliency was evaluated.

8.3 The correlator

8.3.1 Optimisations

The scaling rules of the correlator, as the number of stations, the number of subbands, the sample rates, and the sample type vary, are not as obvious as it may seem at first glance. For instance, the physical separation of the stations (baseline length) is a major factor determining the short term integration time of the correlation and therefore the balance of compute to I/O. Other factors such as the station diameter have similarly non-obvious effects. As a consequence, all stages of the correlator need to be optimised separately [18]: handling the input streams, the data transpose via the network (“corner-turn”), the actual correlation, and writing the results to persistent storage.

8.3.1.1 Work done during the last project year

In Deliverable 6.1 the optimisation of the computation for Xeon and Xeon Phi (KNC) was discussed. For large number of stations, the five stage computation is dominated by the last stage of correlating the signals. This stage is a dense matrix-matrix operation and therefore performs better on the more powerful Xeon Phi than the regular Xeon processors. In the following, the remaining parts of the correlator are discussed.

The correlator mockup does not include the input section of receiving UDP packets and placing them into buffers, but instead sends artificial data to the compute processes. In contrast to the imaging part, the behaviour of all stages in the correlator is data-independent. Thus, the following points were evaluated: the performance of the corner-turn, the writing of visibilities to persistent storage, and fault-tolerance using ParaStation MPI and/or OmpSs task based resiliency.

8.3.2 Resiliency

Telescopes are very expensive instruments and any failure of the correlator means a loss of valuable observation time. Consequently, if failures occur, the most important goal is to lose as little incoming data as possible. If partial data loss (for a short period of time or a few subbands or channels) is unavoidable, it is to be kept to a minimum. A coarse-grain strategy is to monitor the entire application and restart it if it failed for any reason (thus, losing data for only a limited period of time). In this section, more fine-grain strategies are investigated, which enable restarting or ignoring of only parts of the application.

In order to handle fatal errors in compute processes at a finer grain, a ParaStation MPI fault-tolerance feature is used: `MPI_Comm_spawn` can be used in such a way that failing spawned child processes do not cause the spawning parent processes to fail as well (enabled by

setting `MPI_Info_set(info, "parricide", "disabled")` and changing the default error handler `MPI_ERRORS_ARE_FATAL`). Instead of spawning all compute processes with one `MPI_Comm_spawn` call (as originally done), `MPI_Comm_spawn` is used for each compute process to create intercommunicators containing all I/O processes and one compute process each (see Figure 47). Each I/O process periodically checks the health status of all compute processes using `MPI_Irecv` and `MPI_Testany` to test the connection. Then all I/O processes call `MPI_Allreduce` in order to have a consisted view on the health status of the compute processes. All spawned processes that are erroneous are collectively respawned. While a failing I/O process still leads to a complete failure of the correlator, failing compute processes are in this way replaced without the entire application stopping to run.

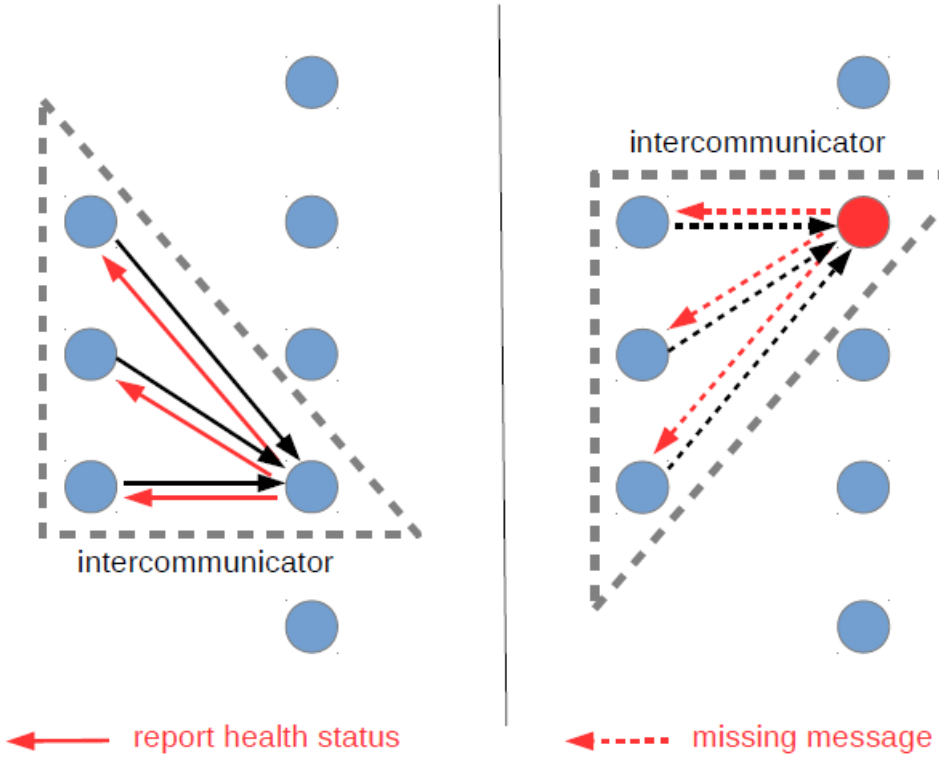


Figure 47: MPI fault tolerance (ASTRON)

The approach was verified by randomly calling `MPI_Abort` within the compute processes. A remaining problem with the approach above is the placement of the respawned processes. If for instance a node fails temporarily, it might be possible to replace the processes on the same node. This is not possible to control this yet. However, if a node goes “offline”, a new process is spawned to replace it in a spare node (allocated when the application was started).

The above strategy can equally be implemented using OmpSs task based resiliency for offloaded tasks, which handles the respawning of tasks transparently using a similar mechanism. This approach was not tested, because `MPI_Comm_spawn` is used for offloading, not OmpSs. However, OmpSs task based parallelism can be used within the computation (see Figure 51). Using the OmpSs recover mode does not make sense in our situation, but OmpSs tasks might still be helpful to deal with transient soft errors. An option to simply drop tasks together with their dependencies, if an error occurs, was proposed.

8.3.3 Input/Output

The I/O parts of the mockup include the network I/O of the corner-turn and the I/O of writing visibilities to persistent storage. Both stages are discussed in turn. In all experiments in this document, the settings specified in Table 30 are used.

Reordering the data in a streaming, real-time way across the entire system is a challenge for the next generation of telescopes. The corner-turn is a pure network transfer, which can be overlapped with the computational stage. The I/O processes, which normally receive the station data, spawn the compute processes using `MPI_Comm_spawn`. While the correlator is running, the data is buffered in the I/O processes and distributed from the group of I/O processes on the Cluster Nodes to the group of compute processes on the Booster Nodes in large chunks.

Compiler version	Intel 17.0.1 20161005
MPI runtime versions	ParaStation MPI version 5.1.4
Compilation flags	-std=c++11 -O3 -qopenmp -xHost -mkl
Threads per core	2 (SDV), 4 (KNL)
Affinity	KMP AFFINITY=compact
HDF5	Version 1.8.18
VML/SVML accuracy	VML LA / medium:sinf,cosf

Table 30: General settings for all experiments (ASTRON)

The data distribution was implemented with MPI using various methods: `MPI_Send/MPI_Recv`, `MPI_Scatter`, `MPI_Gather`, `MPI_Alltoall`, and their non-blocking equivalents, as well as the one-sided `MPI_Put` and `MPI_Get`. The performance is shown in Figure 48 for the DEEP Cluster with InfiniBand network and the DEEP-ER SDV using EXTOLL, respectively. The graphs show the performance when the message size (the chunk of data) is varied. In both cases 8 I/O processes and 8 compute processes are used (one multithreaded process per node). The performance is measured as the total bisection bandwidth (number of bytes moved from all I/O processes to all compute processes per seconds) per I/O process (i.e., the maximal input stream bandwidth that could be supported in real-time), BW_r. In both cases, it can be seen that station input streams of more than 2 GiB/s could be served. Comparing this number with the requirements of LOFAR (approx. 380 MiB/s) and next generation telescopes (approx. 1.0-1.4 GiB/s), shows that there is still headroom for a performance drop when scaling to a large system, without losing real-time behaviour. However, this is only true provided that there exist no interfering workloads that stress the network at the same time.

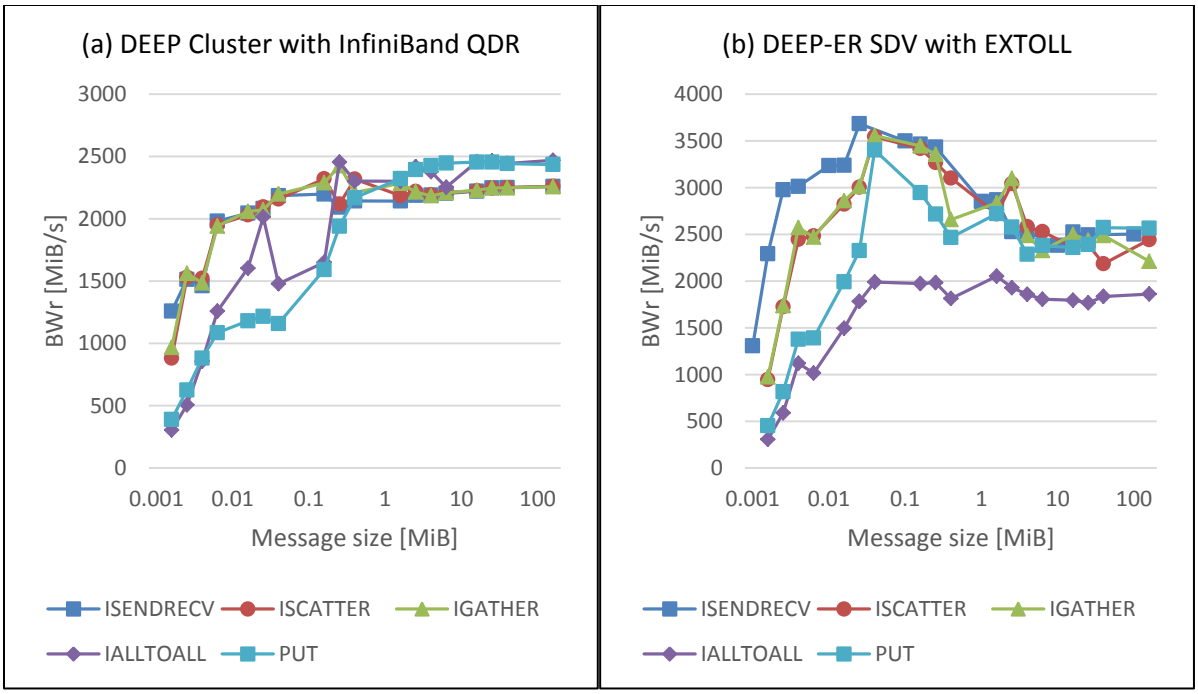


Figure 48: Performance of the data distribution (ASTRON)

In Figure 49, the DEEP Cluster is used to scale the problem to a larger system with up to 32 I/O processes and 32 compute processes (the message size is 4 MiB) with one process per node. As expected performance drops, but remains reasonably high. While LOFAR data with only about 80 stations is not extremely demanding for today's hardware, the SKA AA-Low has already 512 stations in the first construction phase. The corner-turn remains challenging for a system of such size or larger.

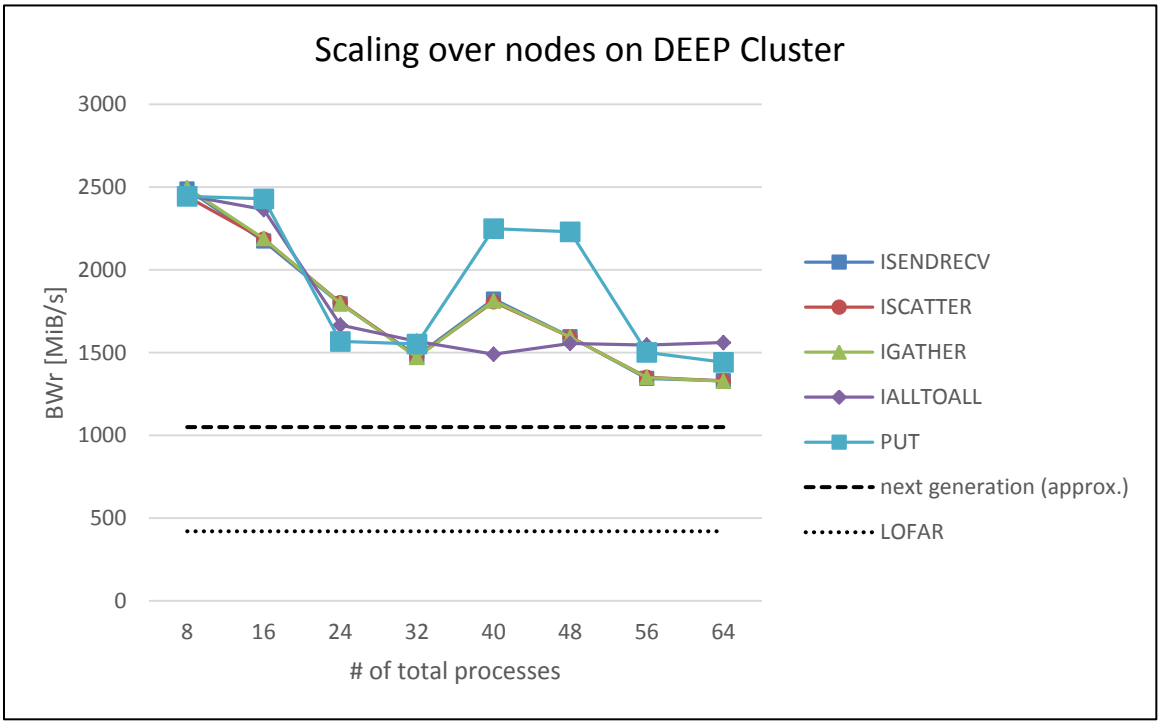


Figure 49: Scaling of the corner-turn on DEEP Cluster (ASTRON)

Another important component for achieving real-time behaviour is writing the data products efficiently to persistent storage. Figure 50 shows the write bandwidth to global (BeeGFS) and local (NVMe) storage. The highest achieved bandwidth of three short runs is reported.

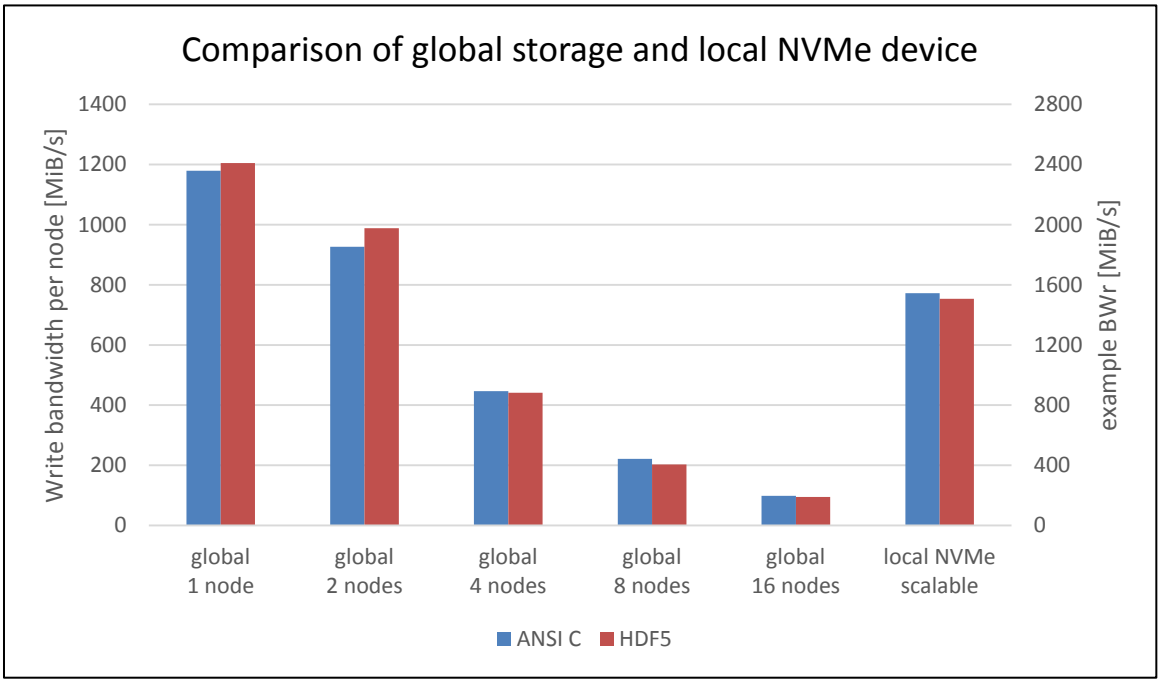


Figure 50: Bandwidth comparison of global storage and NVMe devices (ASTRON)

The bandwidth per node to global storage depends on the number of nodes writing simultaneously (or the behaviour of other users). To put the achieved bandwidth into context, a second scale was added in Figure 50 on the right. In this scale, it is assumed for simplicity that the number of stations is equal to the number of subbands and the correlator compresses the input volume by a factor 2 (not unreasonable assumptions).

Writing data to global storage is not a scalable solution. Not only gets the performance worse as the number of compute processes increases, but it is difficult to guarantee a minimum performance needed for real-time processing. A scalable solution makes use of local storage. LOFAR uses hard disks, but this is impractical for future telescopes, due to the relatively low bandwidth of disks. (Thus, many disks are needed per node to guarantee the required bandwidth.) Very suitable candidates to store intermediate visibility data are the NVMe devices used in the DEEP-ER SDV. They have more than an order of magnitude more bandwidth than hard disks and are more energy efficient. Also, NVMe devices have much lower latency, making it feasible to read and write data from the same device concurrently.

The results in Figure 50 suggest that NVMe devices are very promising devices to store intermediate visibilities in a complex processing pipeline containing not only correlation and imaging, but also flagging [19] and calibration [20]. The possibility of using NVMe devices will be further investigated in the future.

8.3.4 Comparison between architectures

For a large number of stations, the performance is dominated by dense matrix-matrix correlation. Consequently, the correlation is best performed on the Booster Nodes (KNC and KNL nodes on the DEEP and DEEP-ER System, respectively). The absolute performance is

shown in Figure 51. For this test the same configuration as for the measurements in Deliverable 6.1 is used with 512 stations. In addition to the regular (non-task-based) kernels, the performance for ‘taskified’ kernels using OmpSs is shown. The performance of OmpSs without recover is the same as of OpenMP tasks. The performance and energy consumption of the correlation kernel is discussed in detail in Deliverable 6.1 and [15].

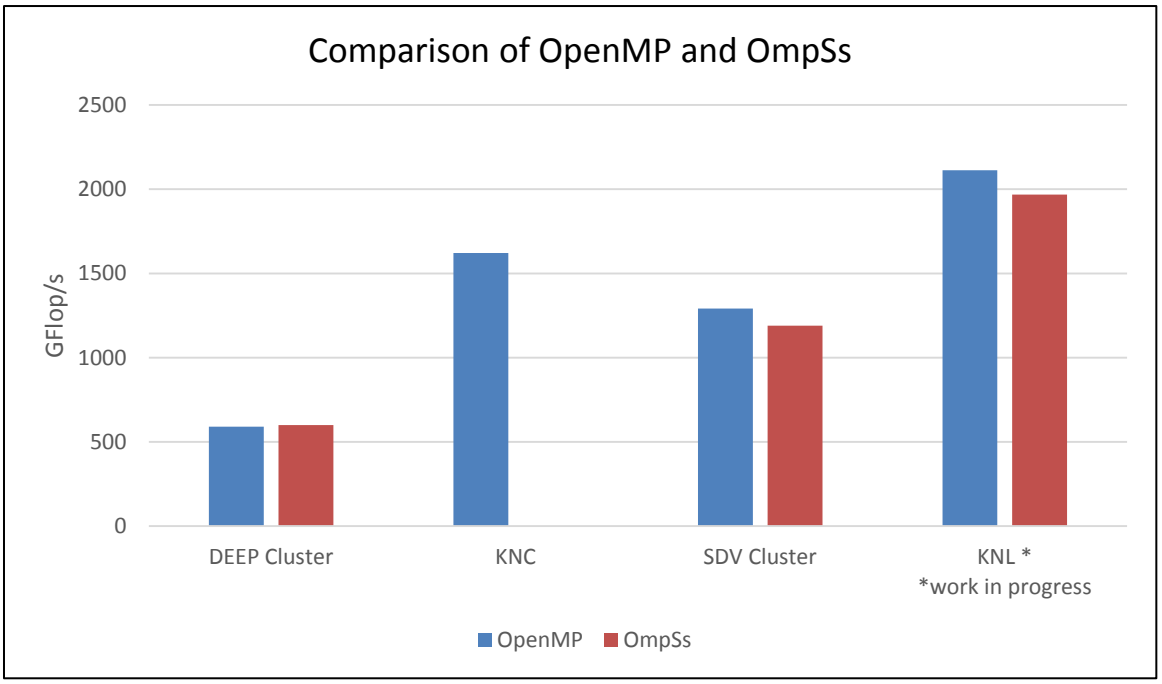


Figure 51: Comparison of OmpSs and OpenMP parallelisation in the correlator (ASTRON)

8.4 The imager

8.4.1 Optimisations

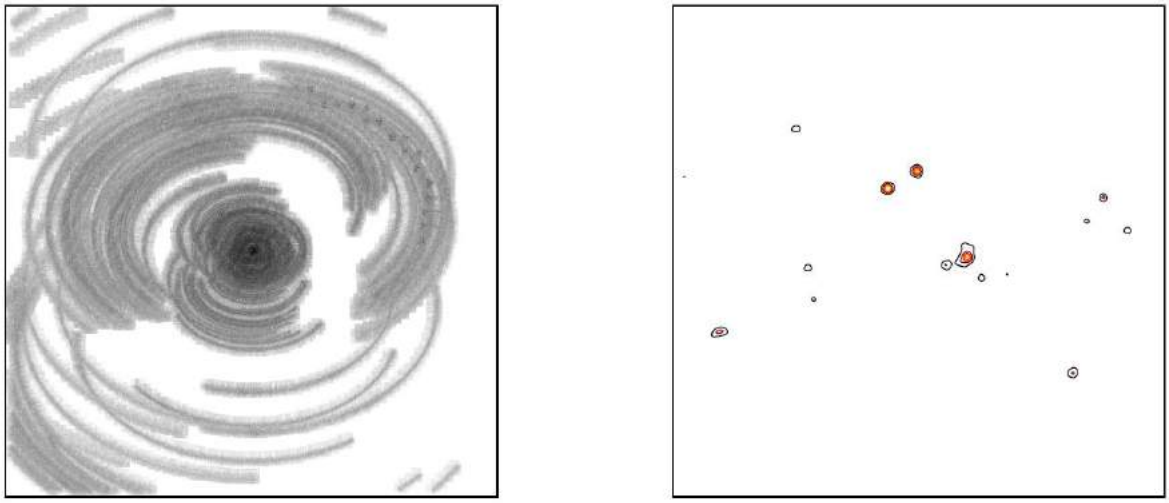
The requirements of imaging in future telescopes such as the SKA approaches Exascale and is dominated by gridding and degridding [21]. Consequently, these fundamental building blocks must be highly optimised for performance. These optimisations were the main topic of Deliverable 6.2. Since then the work on the IDG library has been continued. In the following, the new developments and the integration of persistent storage I/O and fault-tolerance into the mockup imager are discussed.

8.4.1.1 Work done during the last project year

In the last year of the project, the IDG library was further developed, including optimisations for Xeon Phi Knights Landing (KNL). Reading and writing data products from and to persistent storage was included and optimised. Furthermore, user-level checkpointing was integrated such that failed executions can be easily restarted.

8.4.2 Resiliency

The input for the experiments in the following subsections of the imager is the data set RX42 for which the settings are specified in Table 31 and which is shown in Figure 52.



(a) Gridded data (b) Sky image

Figure 52: Visualisation of test data set RX42 (ASTRON)

Antenna layout	LOFAR telescope (HBA dual inner)
Number of stations	55 (thus, 1485 baselines)
Number of channels	20
Number of correlations	4 (2 polarisations x 2 polarisations)
Number of time steps	3123 (in 1 time slot)
Observation time	31273.4 seconds
Size of the grid	4096 x 4096 pixels

Table 31: A single subband of the data set RX42 (ASTRON)

An entire imaging pipeline including correlation, flagging, calibration and imaging buffers data on storage devices between stages. This can be seen as a form of coarse-grain checkpointing. In this section, a more fine-grain approach within the imager was evaluated to enable restarts for long lasting imaging runs.

The Scalable Checkpoint/Restart (SCR) library is used to cache the updated visibilities and sky model on local NVMe devices. Since the other input data products (antenna coordinates, time stamps, visibility weights, flags, etc.) do not change during execution, only a fraction of the input data size needs to be written at every checkpoint. When recovering from failure the current updated visibilities and sky model are read from the checkpoint data, the remaining input is read from the original input data set. In all cases, the time to recover from failure is the same as reading the input data set once.

In Figure 53 results of experimenting with different checkpointing strategies are shown by modifying the checkpointing and flushing frequencies. The SCR settings are further specified in Table 32. The imager is executed for 17 clean cycles (in the last iteration, no checkpoint is written) and checkpoints are written every cycle, every second cycle, etc. To estimate the

maximal overhead of flushing the checkpoints to global storage, an execution was added that synchronously flushes every checkpoint. (This would not been done in practice as asynchronous flushing is available.) Additionally, the timings for synchronously flushing to global storage depend on the load of the storage system and should not be seen as strict upper bounds. Writing to the local NVMe devices however provides guaranteed high performance for checkpointing.

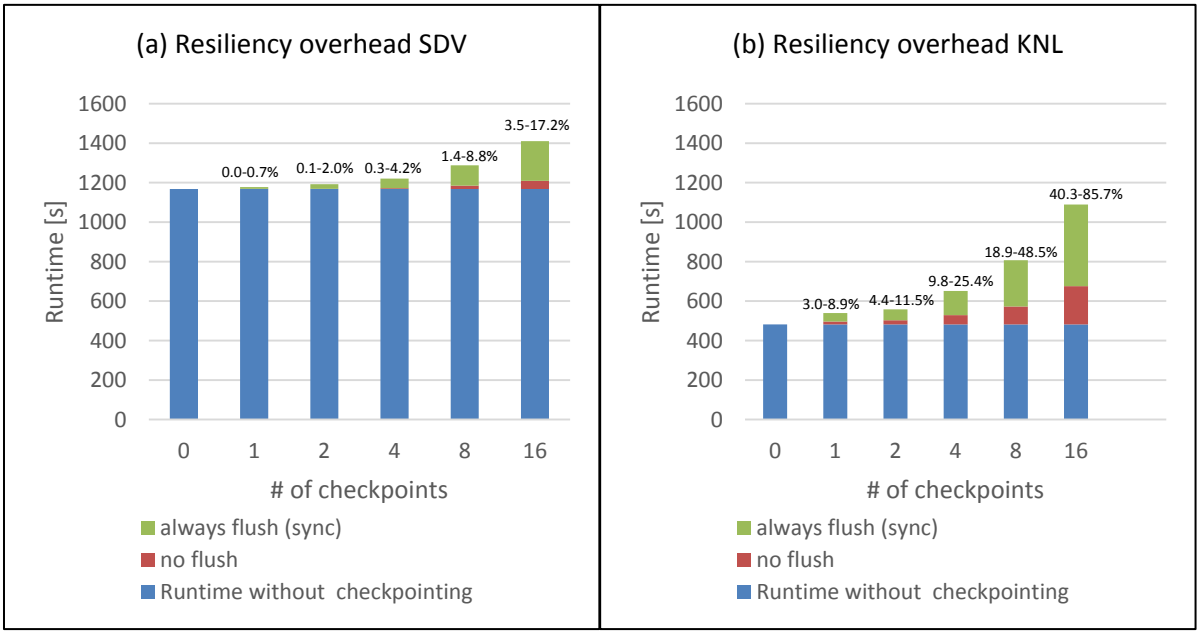


Figure 53: Overhead of using the SCR library (ASTRON)

Naturally, as the checkpointing frequency increases the overhead increases. In the extreme case of writing a checkpointing every cycle, the total overhead is about 3.5% on the SDV, but about 40% on the KNL. However, a realistic setting of checkpointing frequency would have to be set according to the failure model of the architecture developed in WP5.

Library	SCR version 1.1.8
SCR_PREFIX	path to the global file system to flush checkpoints
SCR_CACHE_BASE	path to NVMe device for local checkpoints
SCR_CACHE_SIZE	1
SCR_MODE	default
SCR_COPY_TYPE	single
SCR_CHECKPOINT_INTERVAL	p to write a checkpoint every p-th iteration
SCR_FLUSH	q to flush every q-th checkpoint to global storage
SCR_FLUSH_ASYNCH	0 – synchronous flush, 1 – asynchronous flush

Table 32: General settings used for SCR tests (ASTRON)

8.4.3 Input/Output

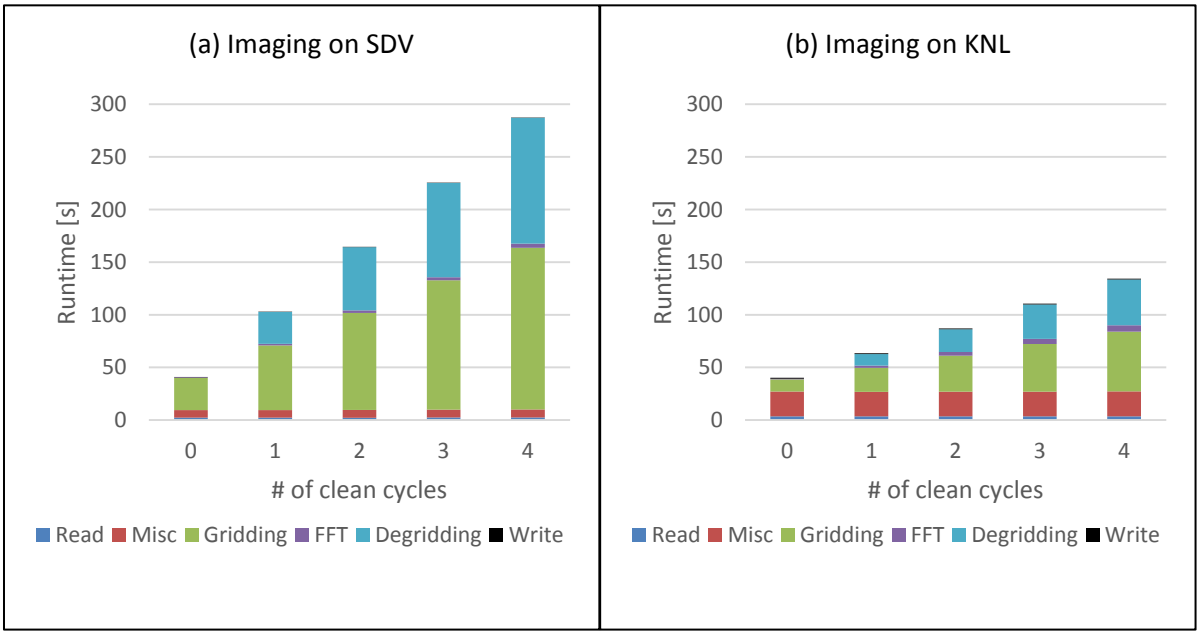


Figure 54: Imaging for various numbers of clean cycles (ASTRON)

Figure 54 shows the imaging of a single subband for various numbers of clean cycles (that is imaging iterations) on respectively a Cluster Node (SDV Cluster) and a Booster Node (SDV KNL). Different nodes work on different subbands in parallel.

The data reading and writing is only a small part of the execution time - especially if more than one clean cycle is necessary (which is usually the case) and the entire data set fits into memory (which is not always the case). The input data is read from the local NVMe devices and the output image is written to global storage. The HDF5 format is used for both input and output, in contrast to the more common MeasurementSet [22] and FITS [23] formats usually used in radio astronomy. The main reason for using HDF5 is that it is a generic library and a lightweight dependency, as it is usually available on supercomputing systems or can easily been installed (especially, in contrast to the more domain-specific MeasurementSets). The mockup does not overlap I/O and computation, although this is possible and is done for IDG within the WSClean imager. Given the time spent in I/O, the attention was focused to the other parts of the imager - namely, performance and checkpointing.

The imager runs normally entirely on the Booster Nodes, which is faster than using the Cluster Nodes. In IDG the kernels are currently runtime compiled. The compilation time is the major part in the “misc” category. Figure 54 (a) and (b) show a manifestation of Amdahl's law: as the runtime compilation does not parallelise efficiently, it becomes a significant part of the computation for small data sets with few clean cycles. However, in most cases, the execution time is dominated by gridding and degridding. This behaviour justifies concentrating much of the performance optimisation on these basic computational building blocks.

8.4.4 Comparison between architectures

In Deliverable 6.2, performance optimisations of the gridder and degridder codes were discussed for Xeon Haswell and Xeon Phi Knights Corner. Since then a better understanding was gained on how to analyse the observed performance and this insight was used for

further optimisation of existing code. Additionally, optimised code for Xeon Phi Knights Landing was produced. The analysis is based on a modified roofline analysis that provides practical upper bounds for our application kernels (see Figure 55).

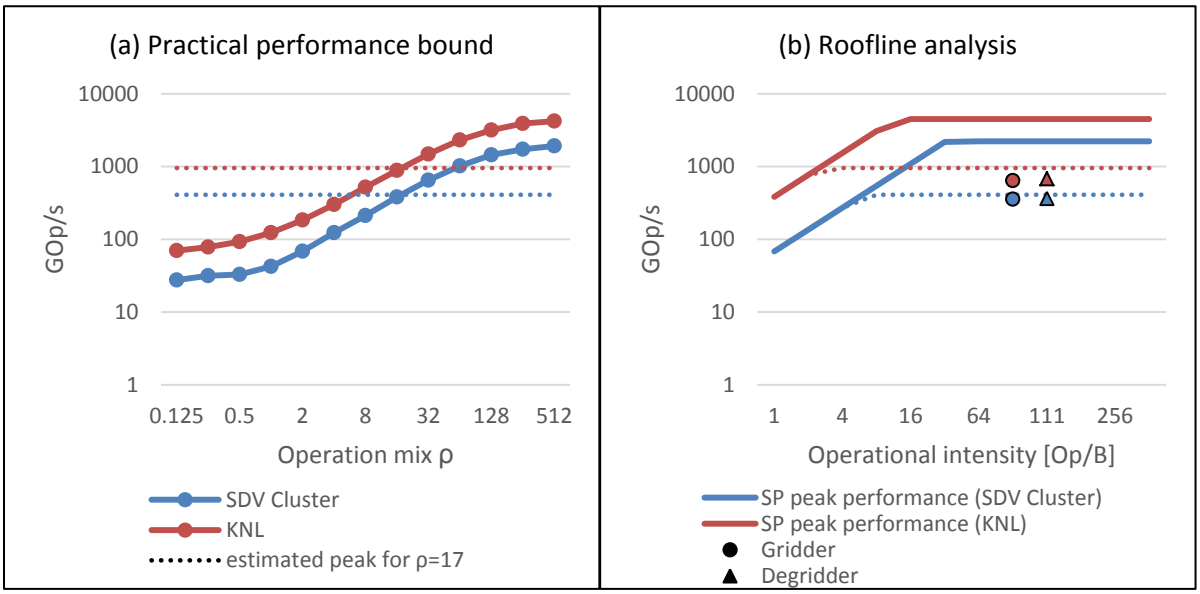


Figure 55: Roofline analysis with practical upper bounds (ASTRON)

The following is defined as an operation: +, -, *, /,sin(), cos(); that is, additionally to the usual floating point operations, the evaluation of a sine or cosine function is considered as a single operation. With this definition, the peak performance is still achieved by code exclusively using fused-multiply add and subtract (FMA and FMS) instructions (each being two operations). However, a good practical upper bound on performance can be found by taking the operation mix of our application into account: IDG gridding and degridding requires for every 17 FMAs about one evaluation of sincos() - that is, the evaluation sine and cosine of the same argument. Defining the operation mix ρ as the number FMAs divided by number sincos() evaluations, it thus is $\rho \approx 17$.

Figure 55 (a) shows the maximal achievable performance for various values of ρ on the SDV Cluster and KNL, respectively. Both use Intel's SVML with medium accuracy. The performance for $\rho = 17$ is used to provide practical, application-specific upper bounds in the roofline analysis in Figure 55 (b).

As 17 out 18 (94%) of the operations are in form of FMAs, Figure 55 (b) can be interpreted as an approximation of a traditional roofline graph; i.e., the gridder and degridder kernels run at about 17% and 15% of the theoretical peak performance on the SDV Cluster and KNL, respectively. While the AVX2 code on the SDV Cluster achieves almost 90% of the practical bound, the AVX512 code on KNL achieves only about 70%. Thus, it might be possible to further optimise the KNL code or the performance might be bound by other resources.

Figure 55 (b) also suggests the best ways on how to significantly improve performance on both architectures: by faster sine/cosine evaluations or by avoiding/reducing the need of sine/cosine evaluations algorithmically.

8.5 Conclusion

The project contributed to the development of highly-optimised correlation and imaging code for Intel Xeon and Xeon Phi platforms. In particular, the imaging code is based on a new gridding and degridding library (IDG), which is optimised for various architectures and is being incorporated in the widely-used WSClean imager. The results of IDG on KNL and within WSClean are currently evaluated and will be part of a future publication.

Also features of the DEEP-ER Architecture were explored for future usage in production. These include the NVMe devices as fast, local buffers for data storage and checkpointing, the EXTOLL network for fast data transposes, and resiliency features of ParaStation MPI, OmpSs, and SCR. All of these aspects should be further investigated for large scale problems.

8.5.1 Acknowledgment

We would like to thank Bram Veenboer at ASTRON for his collaboration. Within DEEP-ER, we profited from various collaborations within the project: among them, the various workshops on various topics (KNL, OmpSs, and SCR), the constructive discussions with other application developers and other Work Packages. Furthermore, we obtained support from the project partners at JUELICH, BSC and ParTec. On the other hand, we hope that we could contribute to Work Packages involved in the planning and design of the hardware with the special requirements of radio astronomical imaging. Last but not least, we would like to thank all the people involved in the DEEP-ER Project for discussions, organisation and support.

9 Task 6.7: Enhancing oil exploration: Full Waveform Inversion (Task leader: BSC)

During the last year of the project three major points have been addressed:

- Porting of the Full Waveform Inversion (FWI) application to the KNL architecture.
- Implementation of a resiliency strategy.
- Analysis and implementation of different input/output approaches.

Previous work on vectorisation [3] resulted in a smooth transition to the new Xeon Phi processor (KNL), requiring only minimal changes to the code. OmpSs task based resiliency perfectly matched the requirements of the application and was implemented with negligible overhead. Regarding input/output, it was decided not to implement SIONlib or E10 support after analysing the application in depth (more details can be found in Section 9.2.2). Local I/O using the NVMe units results in a great performance boost.

9.1 Application overview

The acoustic Full Waveform Inversion (FWI) method aims to generate high resolution subsoil velocity models from acquired seismic data through an iterative process. The time-dependent seismic wave equation is solved forward and backward in time in order to estimate the velocity model. From the differences between acquired data and the computed velocity model, a gradient volume is calculated and used to update the velocity model on the next iteration.

The inverse problem is nonlinear and ill-conditioned. This makes it difficult solving the problem at high frequencies. Instead, the initial stimulus is decomposed into a spectrum of frequencies. Then, low frequencies are solved first on a coarse grid providing a good guess for higher frequencies.

Conceptually, FWI can be divided into three main steps. A pre-processing step estimates the computational resources needed to solve the problem according to the number of shots, wavelet frequency and domain dimensions. Then, the wave propagator solves the time domain formulation of the wave equation forward and backward in time. Finally, a post-processing step gathers the information from the computation of all different frequencies into a single final velocity model. The workflow is shown in Figure 56.

All three stages of the FWI require intensive I/O operations. While pre and post-processing steps perform sequential read and write operations on large shared velocity model files, the wave propagator mostly performs local I/O. From the computational point of view, the pre and post-processing stages do not represent a major issue on the performance of the FWI. They are isolated from the solving phase and executed on the Cluster Nodes.

On the other hand the Booster Nodes take care of the heavy computation on the FWI, the wave propagator kernel. The wave propagator is used twice to propagate the wave forward and backwards in time. These two steps are commonly referred to as “Forward step” and “Backward step”, as represented in Figure 41. An 8^{th} order stencil is needed in order to ensure the stability of the numerical method. Given the length of the stencil, a good balance between high bandwidth memory access and computing power is needed in order to get good performance from the code. However, the wave propagator requires writing relatively large files at almost each time step. This results in inefficient computation of the propagator on most accelerators. Thus, the wave propagator is the key to performance of the FWI,

involving local I/O on the coprocessors, non-contiguous memory accesses and moderate arithmetic intensity.

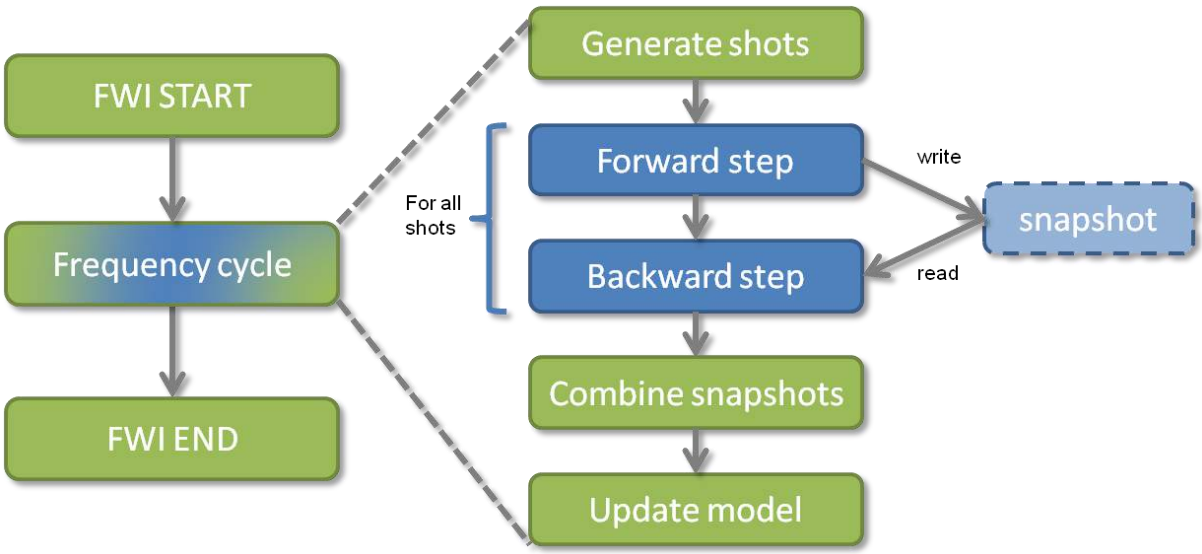


Figure 56: Workflow of the FWI mockup (BSC)

9.2 Optimisations

Our major concern for the last year of the project was I/O time. Most codes on the oil & gas industry are seriously constrained by the time spent on compressing and storing data. DEEP-ER provides us with different options to alleviate this problem, from both software and hardware points of view.

The optimisation of the stencil code, used to compute the centred, 8th order finite differences scheme, was previously vectorised using high-level compiler directives. At this point, only minor modifications were required to ensure that the compiler produces high quality vectorised code for the KNL processor.

9.2.1 Work done during the last project year

During the last year of the project, the efforts were concentrated on three key issues; porting the mockup to the last generation of Intel's Xeon Phi processors, implementing a resiliency mechanism and optimising the I/O segment of the FWI algorithm.

Porting the code to the Knights Landing architecture comprised three tasks. First, the performance of the mockup was evaluated on the KNL. The mockup was already optimised for KNC and Haswell architectures, as described in D6.2. At this point, it was ensured that all the key loops were vectorised correctly by inspecting the compiler report and the tools provided by Intel toolkit. Once the compiler was able to generate high quality code for the computational intensive loops, the focus was on providing the compiler with some advanced flags intended to tune how the compiler evaluates certain mathematical expressions. For example, in our case, we took advantage of the new optimised primitives that KNL provides to compute the reciprocals of natural numbers and of square roots, since this two expressions are used extensively during the interpolation phase of the wave propagator. Finally, the different options that the KNL processors offers were evaluated regarding the use of the HBM memory and the execution parameters were tuned by means of `numactl` command.

Regarding the I/O optimisations, the different options that the DEEP-ER Project provided were evaluated. Finally it was decided that the combination of BeeGFS and the local NVMe devices was the option that suited most the particulars of the FWI algorithm. For a more in depth discussion see Section 9.4 of this document.

Resiliency has been the third action point this year. An analytical analysis of the FWI algorithm was conducted, taking into account the particulars of the programming paradigm being used to determine what of the available mechanisms would be the most efficient for us. Finally, it was decided to implement OmpSs task based resiliency, since it provides all the tools needed while it is transparently integrated on the runtime. Section 9.3 of this document describes the resiliency mechanism in detail.

9.2.2 Progress overview

Workflow elements	M1-M6	M6-M12	M13-M18	M19-M24	M25-M30	M31-M36	M37-M42
Analysis	1	2	2	2	2	2	2
Writing D6.1	0	1	2	2	2	2	2
Port to KNC	2	2	2	2	2	2	2
Threading	1	1	1	2	2	2	2
Vectorisation	1	1	1	2	2	2	2
General code optimisation	1	1	1	1	1	1	2
Implement OmpSs parallelisation and/or offload	1	1	1	2	2	2	2
Cluster-Booster division	1	1	1	2	2	2	2
Port to SDV	0	0	0	0	2	2	2
Use NVM	0	0	0	0	1	2	2
SIONlib integration	-	-	-	-	-	-	-
E10 integration	-	-	-	-	-	-	-
General I/O optimisation	1	1	1	1	1	1	2
Implementing a mockup	1	2	2	2	2	2	2
Writing D6.2	0	0	0	0	2	2	2
Checkpointing on NAM	-	-	-	-	-	-	-
Optimise for KNL	0	0	0	0	0	1	2
Implement SCR	-	-	-	-	-	-	-
Implement OmpSs task based resiliency	0	0	0	0	1	2	2
JUBE integration	0	0	0	0	1	2	2
Final benchmarking	0	0	0	0	0	0	2
Writing D6.3	0	0	0	0	0	0	2

Table 33: Progress overview (BSC)

The FWI application has covered the main topics of the project. Instead of using a real application, a mockup code was developed, which shows all relevant characteristics of a full scale proprietary industrial code developed at the Barcelona Supercomputing Center. We refer to this mockup code as the FWI application in the remainder of this Deliverable.

The core of the FWI application, a wave propagator engine, was designed to enable efficient code vectorisation. The idea was to create OmpSs tasks for the outer iterations of the stencil code while the inner-most loops where vectorised making full usage of wide SIMD vector units. Later on, this code was optimised for Intel's KNC architecture and then ported to KNL, which provides 512-bit wide vector units.

At a higher level, the OmpSs offload is used to manage and schedule the execution of task-based parallelism. This enables the FWI to expose a great amount of parallelism with little programming effort, even when using heterogeneous systems like the Cluster-Booster machine.

The resiliency was implemented using OmpSs task-based resiliency. Since the code was already expressed in terms of tasks, only minor modifications were needed to use this feature of the runtime. FWI choose to implement task-based resiliency because of the nature of the algorithm as explained in depth in Section 9.3.

Regarding I/O, the use of local NVMe units promised a breakthrough on the performance of the application. Full Waveform Inversion problem is known to be dominated by the time required to create and store temporal snapshots of the velocity field on disk. These snapshots are local to computing nodes and are not shared until the computation of a single frequency is finished. In contrast, global I/O takes place only at the beginning and the end of the modelling process.

Although SIONlib and Exascale10 libraries provide efficient solutions for thousands of MPI ranks to access a few files, the FWI application does not fit into this category. Only the master nodes read the velocity model at the beginning of the computation of each frequency, sharing the data with their worker nodes. As a result, only a few nodes access the same file at a given time. Thus, the combination of BeeGFS and NVMe suits best our specific needs.

At this point, FWI application has been ported completely to the new KNL environment on the DEEP-ER SDV and it is able to run on hybrid configurations on the DEEP Cluster-Booster System and the DEEP-ER SDV.

9.3 Resiliency

The FWI application approaches the resiliency problem using OmpSs task-based resiliency. From the algorithmic point of view, this approach to resilience matches perfectly the features of the application.

Full Waveform Inversion generates embarrassingly parallel tasks at the level of shots. Each shot can be computed independently and the synchronisation is imposed at the end of each iteration of the frequency loop. On industrial applications of the FWI method, the number of shots is really large (in the order of tens of thousands) and the time cost for each one of the shots is negligible compared to the cost of computing a frequency iteration. Thus, an ideal resiliency model for the FWI would provide a mechanism to restart the computation of multiple shots within a frequency. In order to do that, only a small set of parameters and the initial velocity model are required. The syntax of the offloaded tasks was modified to receive these parameters as input. In this way, the OmpSs task-based resiliency is able to restore a task efficiently.

Once the computation of all the shots is completed for a frequency, the velocity model is updated for the next iteration. The updated velocity field acts as a checkpoint for the next frequency iteration, so there is no need to store checkpoints at this point.

Since the FWI application actually uses OmpSs to expose parallelism at different levels, enabling the resiliency support required virtually no work for the programmer.

The results in Figure 57 show that the resiliency support does not add additional overhead to the simulation time.

Figure 57 summarises the results of the experiment detailed in D5.4 [24], Section 5.2.2. In such experiment, multiple shots were executed simultaneously, ranging from 2 to 16. Each shot involved 4 worker nodes. Here, the figure compares the time-to-solution of four different scenarios: i) an execution without resiliency support, ii) an execution with resiliency support activated while no error occurred, iii) resiliency support activated with an error injected on one of the slave nodes and iv) with an error injected at the workers level.

It is noticeable that the runtime is able to recover from an error occurring at every level of the offloading hierarchy with little overhead, even though an error at the worker level requires a much more elaborate process.

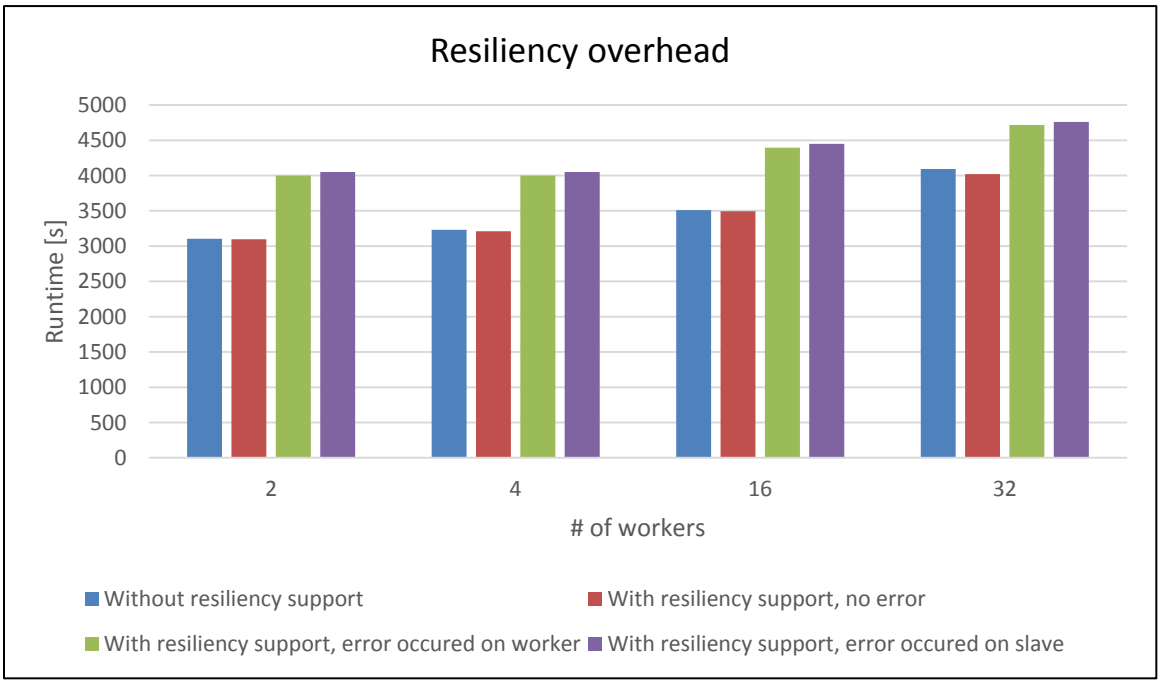


Figure 57: Resiliency overhead (BSC)

Experiment details	Multiple shots, ranging from 2 to 16. Each shot involved 4 worker nodes.
Used system	SDV
Compiler version	Intel 17.0 gcc 5.3.0 ompss 16.08
MPI runtime versions	Intel 5.1
Compilation flags	-std=c99 –ompss –no-copy-deps –Wn,-fma –Wn,-align –Wn,-finline-functions –Wn,-xHost
MPI processes per node	1
Threads per process	16
Describe the error (location, time, ..)	Errors were injected on each one of the levels of the application's offload hierarchy.

Table 34: Experiment setup regarding resiliency (BSC)

9.4 Input/Output

Reducing the time spent on input/output is critical for the FWI algorithm to scale to higher wavelet frequencies. The effect of increasing the wavelet frequency on the I/O pressure is two-fold. On one hand, finer computational meshes are required to capture the effects of the wavelet traveling through the space, so the number of points in the velocity field that is needed to store/read is larger. On the other hand, the Nyquist–Shannon sampling theorem enforces one to increase the frequency at which velocity field snapshots are stored.

Global input/output only occurs at the beginning and at the end of the computation. The application needs to load an initial velocity model from the global filesystem before starting the computation of each shot and, once these computation is finished, the information must be gathered into a single file. These two steps are done really quickly, even for large domains at high wavelet frequencies. The reason for this is that only a small number of computational nodes (typically less than 12 for industrial applications) are needed to process each shot, but thousands of shots may be required to generate reliable simulation results.

In contrast, local I/O takes a large fraction of the simulation time. Local I/O is performed by each one of the nodes independently and does not need to be shared or coordinated among them. Compression algorithms have been used on industrial oil & gas applications to shrink the size of the local snapshots. Although effective, this approach has two drawbacks. It steals CPU cycles from the simulation time and deteriorates the quality of the snapshots in some cases. Fast local storage units effectively reduce the cost of local I/O.

As a result of implementing the local I/O approach, a performance improvement of a factor of 11.7 has been obtained for the forward propagation case and a factor of 8.6 was measured for the backward propagation step. The forward modelling writes the data into storage devices while the backward modelling reads the snapshots from disk, so these two numbers are slightly different.

Figure 58 illustrates the performance benefit of using NVMe devices to store the velocity fields during the forward and backward wave propagation. Notice that the time spent on local I/O is invariant to the number of nodes used to compute a shot. Moreover, this strategy does not increase the pressure on the global file system of the machine since these devices are not interconnected.

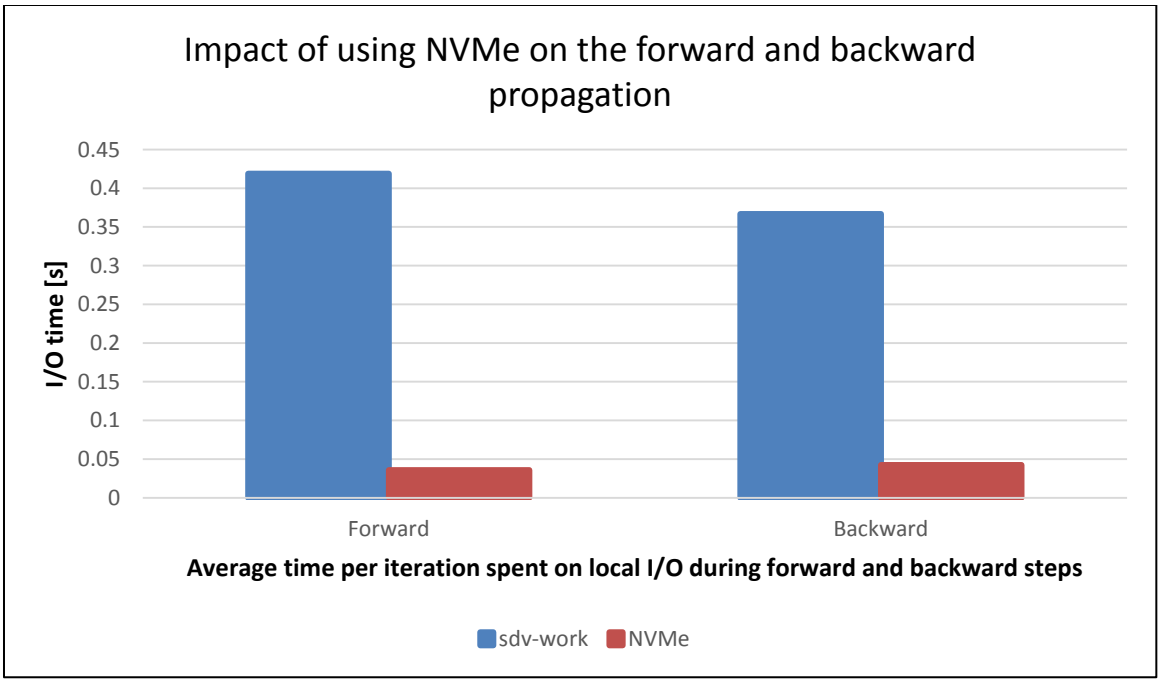


Figure 58: Impact of using the NVMe devices (BSC)

Number of cells	288x80x70
Other experiment details	Wavelet frequency, 5 Hz
Compiler version	Intel 17.0 gcc 5.3.0 ompss 16.08
MPI runtime version	Intel 5.1
Compilation flags	-std=c99 –ompss –no-copy-deps –Wn,-fma –Wn,-align –Wn,-finline-functions –Wn,-xMIC-AVX512
MPI processes per Node	1
Threads per process	134 (2 threads per core)

Table 35: Benchmarking setup regarding I/O optimisation (BSC)

9.5 Comparison between architectures

Last year of the project was mainly devoted to the optimisation of the code for Intel multi- and many-core architectures. Among other changes, a proper data layout was created and the compiler was provided with high-level directives. This approach was chosen over writing low-level code using architecture-specific intrinsics. Although low-level primitives can offer quite a performance benefit in some cases, they have a number of drawbacks. Primarily, they are machine-specific, hard to maintain and hard to debug. Instead, it relied on compiler assisted vectorisation. As a result, only minimal adjustments are needed to ensure effective loop vectorisation on different machines.

The same code has been tested on Xeon (Sandy Bridge and Haswell generations), KNC and KNL processors. Intel's compiler was able to vectorise the inner loops of the wave

propagator kernel for all these four architectures. Figure 59 and Figure 60 compare the runtime and parallel efficiency of the FWI application executed on KNC and KNL nodes

Being a co-processor, Intel's KNC does not have access to its own storage unit. Although it is possible to mount a remote directory on device's on-chip memory, this solution does not offer a good performance for I/O intensive applications like the FWI. This fact degrades the intra-node performance of the code as so the parallel efficiency.

In contrast, KNL implements interfaces for local I/O units, so disk access does not penalise more than on regular, general purpose nodes. Also, this enables the usage of NVMe storage devices for fast local I/O.

To summarise, the access to local storage units combined with the improvements at micro-architecture level (out-of-order execution, HBM, extended instructions set, etc) result in a high per-node performance and a great parallel efficiency of the code.

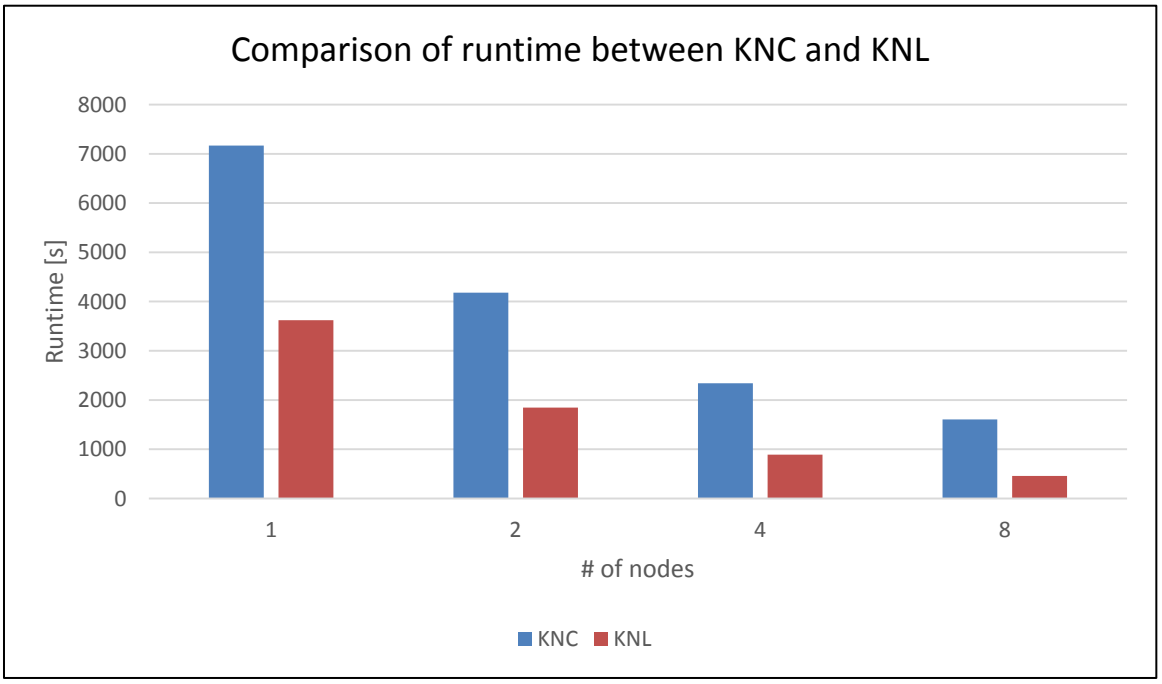


Figure 59: Runtime comparison between KNC and KNL (BSC)

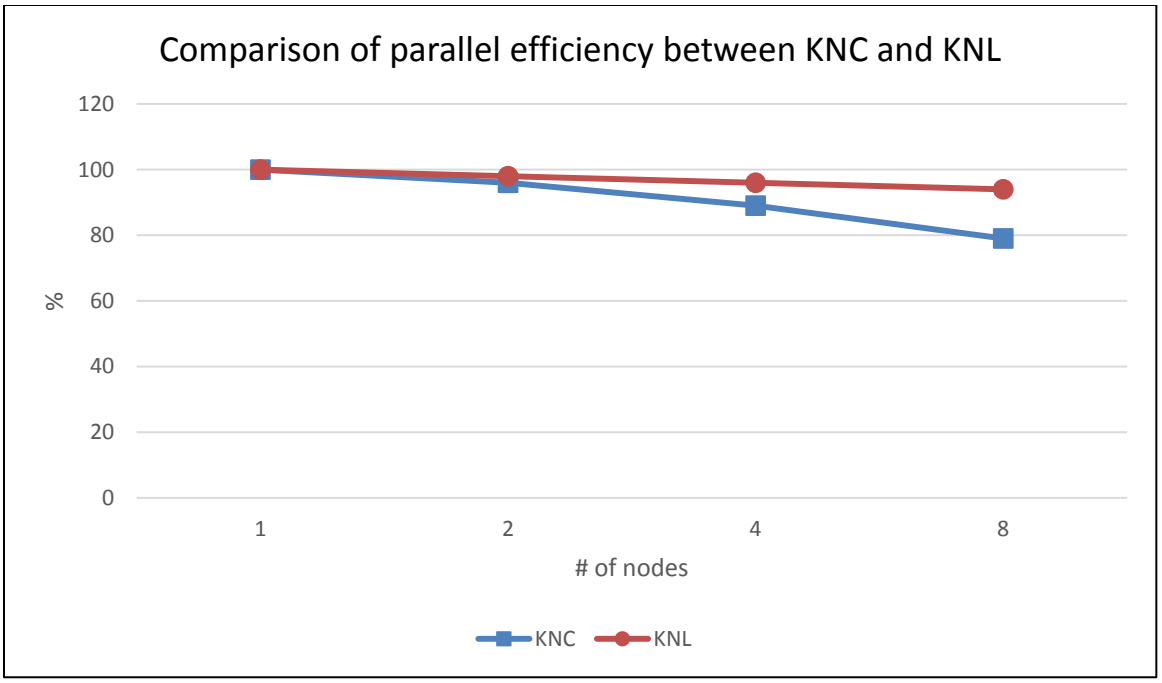


Figure 60: Parallel efficiency comparison between KNC and KNL (BSC)

Scaling	Strong
Number of cells	336x336x520
Other experiment details	Single shot, wavelet frequency 9 Hz
Compiler version	Intel 17.0 gcc 5.3.0 ompss 16.08
MPI runtime version	Intel 5.1
Compilation flags	-std=c99 –ompss –no-copy-deps –Wn,-fma –Wn,-align –Wn,-finline-functions –Wn,-xMIC-AVX512 for KNL and –Wn,-mmic for KNL
MPI processes per Node	1
Threads per process	2 threads per core

Table 36: Benchmarking setup regarding runtime comparison between KNC and KNL (BSC)

9.6 Conclusion

The use of the OmpSs offload results in a cleaner and easier to maintain application. The OmpSs offload functionality simplifies the process of creating, monitoring and scheduling a large number of shots that, otherwise involves a large number of lines of code.

Before DEEP-ER, the code had no resiliency features. Now, thanks to the tools provided by the project, the code is robust and we are able to recover the application from a number of failures with confidence.

The performance of the wave propagator has increased. The advice of the experts and the use of specialised analysis tools were critical to obtain a high performance version of this

kernel. Moreover, the code has demonstrated to be portable while maintaining a consistent performance across different Intel architectures.

The performance of the kernel does not come from the vectorisation alone. Input/output time is quite significant. By using fast local NVMe storage units we achieved a great performance boost. DEEP-ER gave us the opportunity to test these devices that are not yet so common on supercomputers. Further research would determine the impact of this kind of technology on the oil & gas industry and how it could shape a new generation of wave propagator kernels.

The optimisation of the code was approached in a holistic manner inside the project. The application was analysed with experts from different fields to take the most out of the hardware and the software resources available. From our perspective, this was a great opportunity to understand the performance issues that may occur when scaling our applications to next-generation supercomputers. Performance analysis tools and the advice of the experts were critical at this point.

DEEP-ER gave us – the applications developers - the chance of taking part into both the hardware and software design process. Hardware and software tools have been designed around the applications to solve critical HPC problems. This unique opportunity of being asked what is needed for your application to scale further pushes you to think and explore problems from new perspectives.

10 Task 6.8: Lattice QCD (Task leader: UREG)

As soon as KNL systems became available to the project, the application was ported to the new architecture. Some parts (like the domain decomposition solver) were previously optimised for KNC and contained low-level KNC code (intrinsics), which had to be ported to the new architecture in order to compile. Also this was used as an opportunity to remove vendor specific idioms from the code. Other parts (libHadronAnalysis) were written in a largely architecture-independent way from the start.

It was started to explore the possibility of integrating SIONlib in the application. In particular for storage based on BeeGFS, SIONlib can be an alternative to the currently used methods, namely regular POSIX file I/O and HDF5.

As a side remark, the solver has the option to use a specialised communication library (pMR [25]) instead of MPI. In certain cases, this avoids unnecessary buffering, as memory regions can be re-used for communication instead of allocating new memory each time. However, this library is currently only available for InfiniBand and Omni-Path, and not for EXTOLL.

10.1 Application overview

The Chroma QCD application can perform a wide range of simulation tasks for lattice QCD. However, these tasks can be grouped in two different kinds of tasks:

One is the generation of lattice gauge configurations. The application uses a Markov chain process to generate a set of gauge configurations (an ensemble) that represent the physics in a given environment. Input parameters like coupling constant, lattice size and type of action are provided via a small XML file. As this is a Markov process, gauge configurations in the same ensemble can only be produced one after another.

The other kind of task is analysis. For each configuration in a given ensemble of gauge configurations, physical observables are computed, and thereafter averaged over all configurations in that ensemble. Typically, the analysis step has a higher I/O to computation ratio. The computation of each observable can usually be done for each configuration separately, leaving room for trivial parallelisation.

Compute tasks (blue) are typically memory bandwidth bound, although if the local lattice size becomes too small, network latency will dominate. It is favourable that the network chip itself performs most communication tasks, freeing the CPU for numerical tasks.

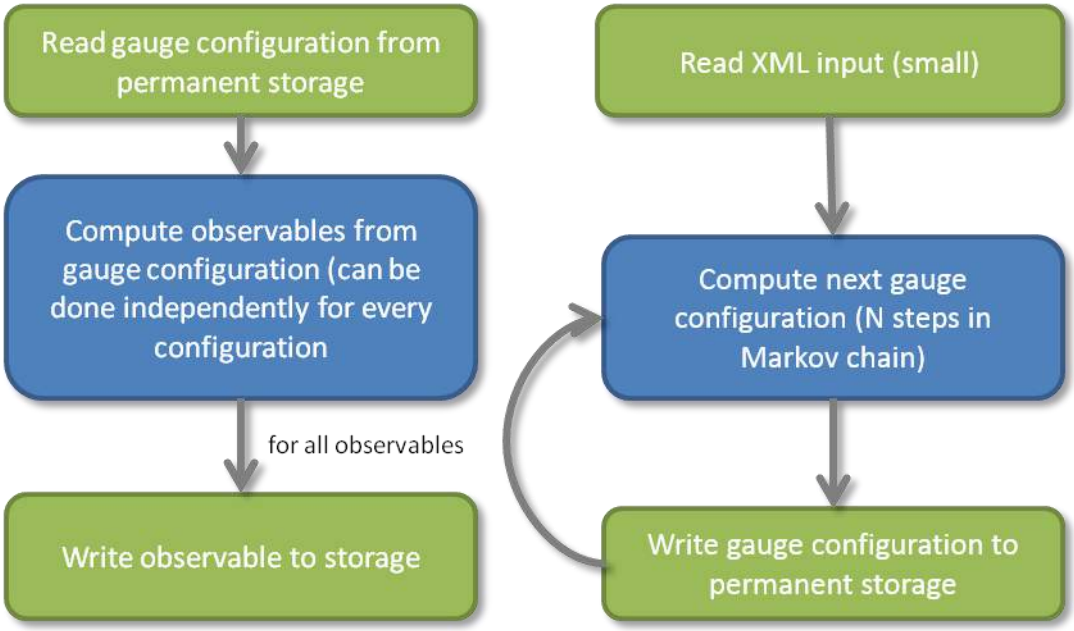


Figure 61: Workflow of the analysis mode (left) and generation mode (right) (UREG)

10.2 Optimisations

Optimisations have been done in the solver (approximate matrix inversion), which is a core part of both the generation mode and the analysis mode, and with respect to parallel I/O.

10.2.1 Work done during the last project year

10.2.1.1 Porting and benchmarking on KNL

Threads can be distributed “compact” or “scattered”, i.e., if the software threads 0,1,2,3 are placed on the same core, and 4,5,6,7 on the next core, it is called “compact”, whereas if thread 0 is placed on core 0, thread 1 on core 1, etc. and then thread n on core 0, thread n+1 on core 1 etc. it is called “scattered”. This actually makes a performance difference on KNL, as shown in Figure 62.

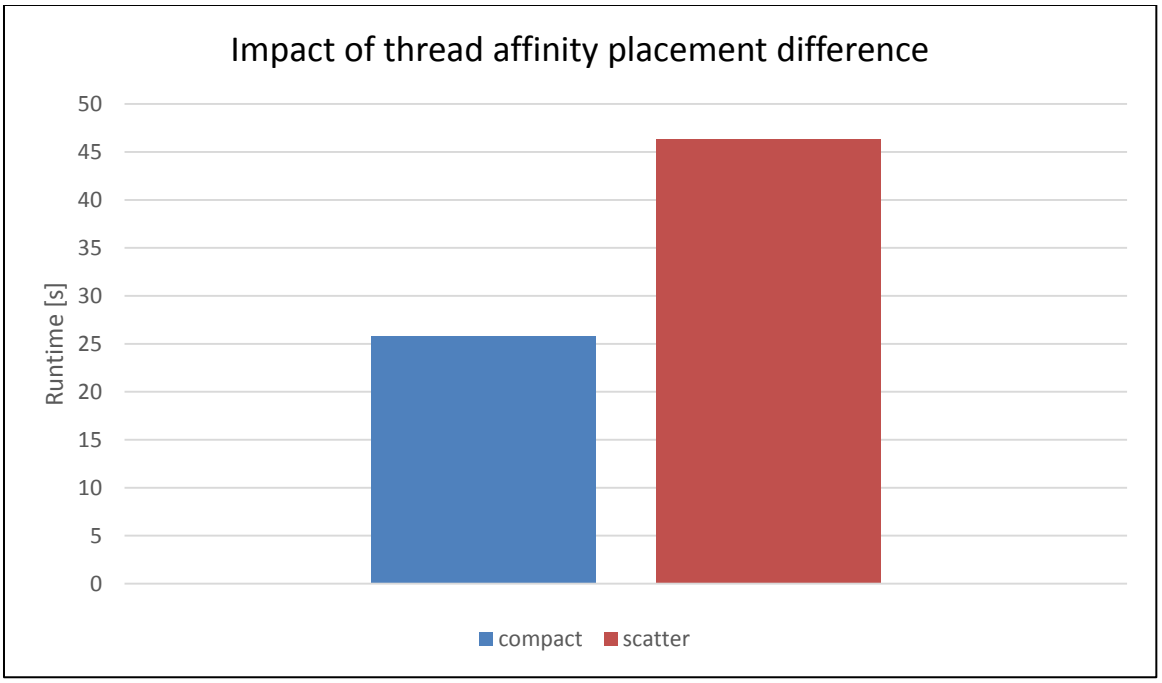


Figure 62: Impact of thread affinity placement difference (UREG)

Further figures only show benchmarks with the “compact” thread layout. They use the SNC4 processor configuration and the “flat” KNL memory model. Since Chroma benchmarks were taken on the QPACE3 machine, where an OmniPath network is used, SNC4 is advantageous. There can only be one OmniPath endpoint per process, and in order to saturate the off-chip bandwidth, at least 4 processes are needed. These do not share memory, therefore SNC4 was chosen.

Experiment details	Chroma (total time, full app) on one node
Used system	QPACE3
Compiler version	gcc6
MPI runtime versions	mvapich2-2.2-psm2
Compilation flags	-fopenmp -O3 -std=c++11 -march=knl -fargument-noalias-global -funroll-all-loops -fpeel-loops -finline-limit=50000
MPI processes per node	64
Threads per process	4

Table 37: Experiment setup regarding … (UREG)

10.2.1.2 E10 integration

Results are shown in Section 10.4. The MPIWRAP library allows us to make use of E10 without any code restructuring, just via relinking. HDF I/O benefits directly.

10.2.1.3 SIONlib integration

It was started explore SIONlib, however at the moment only some micro benchmarks were done to see if SIONlib offers sufficient flexibility to write QCD analysis data. As stated in the last Deliverable (D6.2) [3] (Section 9.3.1) a full integration of SIONlib into Chroma is a very substantial effort that might not pay off in the long term (adoption of file format for other collaborators). Full integrating of SIONlib will not be possible. Nonetheless, efforts to integrate SIONlib for specific routines will continue outside this project.

10.2.2 Progress overview

Workflow elements	M1-M6	M6-M12	M13-M18	M19-M24	M25-M30	M31-M36	M37-M42
Analysis	1	2	2	2	2	2	2
Writing D6.1	0	1	2	2	2	2	2
Port to KNC	2	2	2	2	2	2	2
Threading	1	1	1	1	2	2	2
Vectorisation	1	1	1	1	2	2	2
General code optimisation	1	1	1	1	1	1	2
Implement OmpSs parallelisation and/or offload	-	-	-	-	-	-	-
Cluster-Booster division	-	-	-	-	-	-	-
Port to SDV	0	0	0	0	1	2	2
Use NVM	0	0	0	0	1	2	2
SIONlib integration	0	0	0	0	0	0	1
E10 integration	0	0	0	0	1	2	2
General I/O optimisation	1	1	1	1	1	1	2
Implementing a mockup	-	-	-	-	-	-	-
Writing D6.2	0	0	0	0	2	2	2
Checkpointing on NAM	-	-	-	-	-	-	-
Optimise for KNL	0	0	0	0	0	1	1
Implement SCR	-	-	-	-	-	-	-
Implement OmpSs task based resiliency	-	-	-	-	-	-	-
JUBE integration	0	0	0	0	0	0	2
Final benchmarking	0	0	0	0	0	0	2
Writing D6.3	0	0	0	0	0	0	2

Table 38: Progress overview (UREG)

- **OmpSs parallelisation and/or offload**: Chroma runs natively on the Booster. Therefore, there is no Cluster-Booster division and no offloading. (See „Cluster-Booster division" bullet below for details). Furthermore, the most work intensive mode — the generation mode — has a sort of built-in resiliency. Therefore, OmpSs's resiliency features are not needed. Considering this, porting to OmpSs would introduce a major code change without an apparent use case for our application.
- **Cluster-Booster division**: Our application is memory bandwidth bound (except for very small local lattices, i.e., a very large number of nodes where it becomes network latency bound). Although some parts of the code would run faster on the Cluster, the application does not benefit from Cluster-Booster division because of the time lost in data transfer between Cluster and Booster. A Cluster-Booster division would require a major rewrite, and would possibly involve changes to the data layout between Cluster and Booster, adding to the latency. As the application is somewhat

monolithic, to justify the immense coding effort of a Cluster-Booster division, the gain would have to be huge. Such a huge benefit couldn't be observed, as only serial parts would benefit from the Cluster (because of higher clock speed). Therefore, such a split was not attempted.

- **Use NVM**: The NVMe devices are only used indirectly, via the MPIWRAP library to exploit the SSD cache features provided by Exascale10. The only use case for temporary files is the built-in resiliency feature of the generation mode. Since these temporary files do not differ in any way from the real configurations.
- **SIONlib**: replacing all I/O with SIONlib is not feasible or practical. Replacement of some I/O with SIONlib will continue outside this project
- **Implementing a mockup**: This is not necessary, as real data can be used. There are no privacy or IP issues connected to the real code and data.
- **Optimise for KNL**: Porting to KNL is done, and relevant parts have been rewritten to use either architecture independent code or KNL intrinsics. If further optimisation are necessary or not can only be assessed once we run on a large number of KNLs. Therefore this can be considered 90% finished.
- **Implement SCR**: No special resiliency feature is needed. See Section 10.3.
- **OmpSs task based resiliency**: See bullet point "Implement SCR".

10.3 Resiliency

Resiliency or checkpointing is not needed, because the generation mode already has a built-in resiliency, and analysis mode jobs are typically short lived and do not need checkpointing either.

10.4 Input/Output

This section shows several I/O benchmarks on QPACE3 and the SDV. First a good scaling behaviour is shown when using the parallel file system BeeGFS on QPACE3. Then synthetic benchmarks show a better write performance of SIONlib wrt. HDF5 output even on systems without special features (like QPACE3). The next subsections points out that the effects of MPIWRAP used to support E10 on the DEEP-ER SDV were quite promising. This led to the decision to also provide this library on QPACE3. Also some full application benchmarks are shown here.

10.4.1 HDF5 write mockup

For the scaling tests with BeeGFS a QDP++ application (part of the Chroma software stack) was used. Figure 63 shows a weak scaling run on 1-8 nodes keeping the total number of MPI tasks constant (32) as well as the problem size. It can be seen, that it makes quite a difference how to distribute the 32 MPI tasks. The best distribution might depend on the application but for the QDP++ application it is best to use 4 nodes with 8 MPI tasks each.

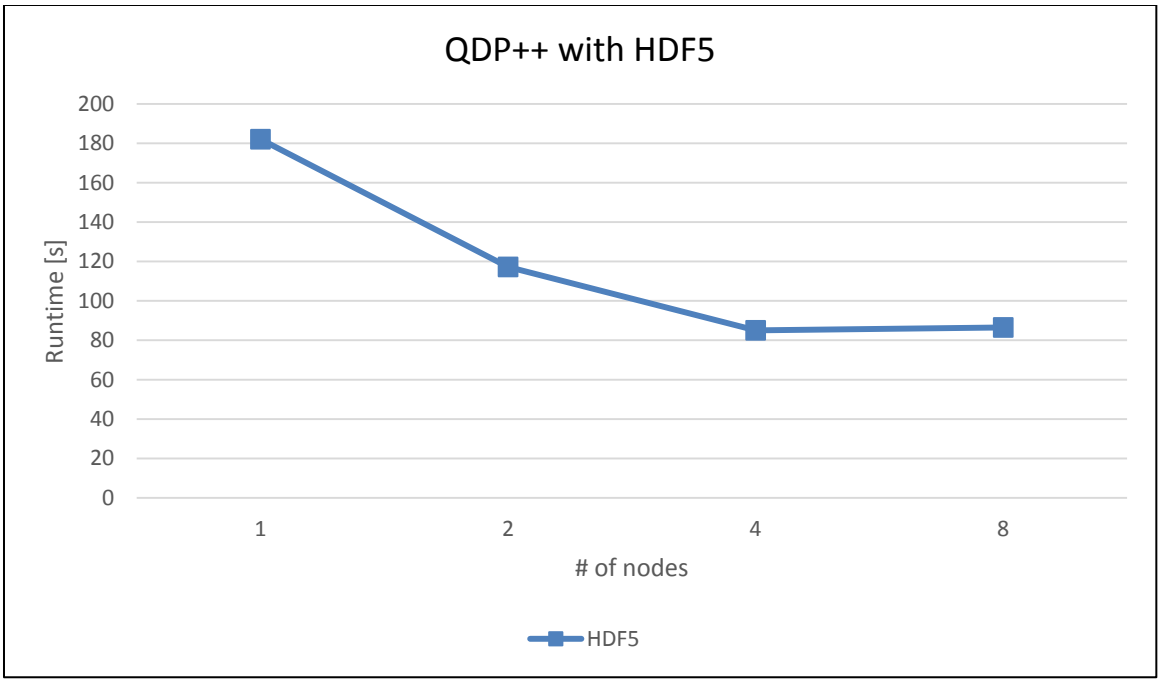


Figure 63: Scaling of QDP++ on BeeGFS (UREG)

10.4.2 SIONlib integration

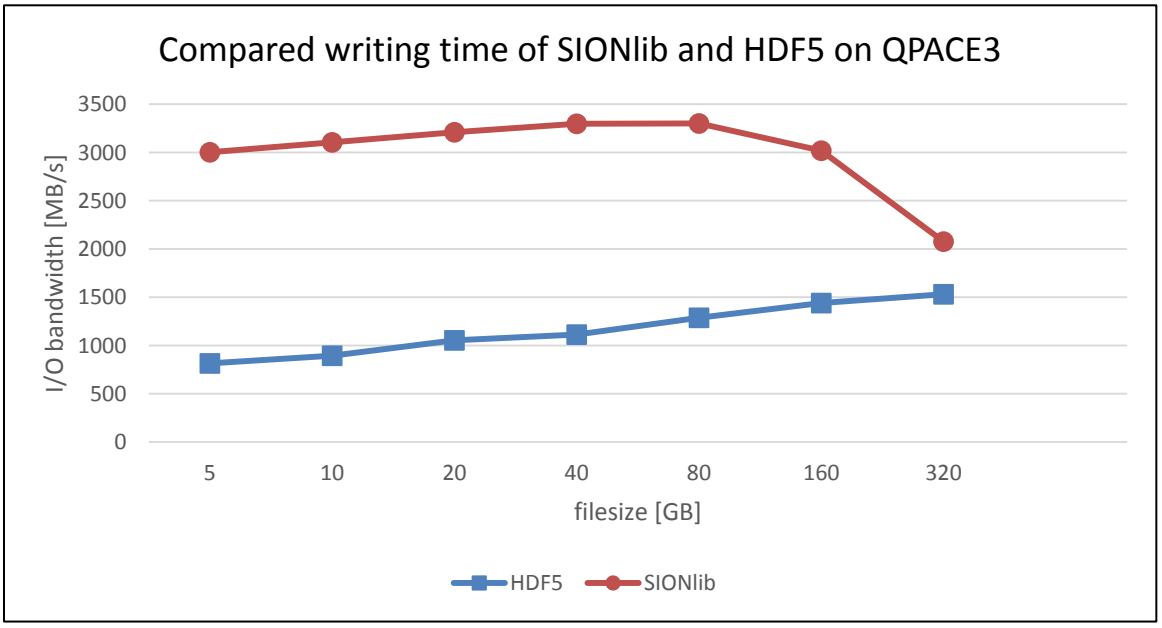


Figure 64: Synthetic benchmark on QPACE3 comparing SIONlib and HDF5 writing time (UREG)

Figure 64 shows a synthetic benchmark of SIONlib vs. HDF5 on BeeGFS of QPACE3, running with 1 rank, BeeGFS chunk size of 8M, buffer size = 500M, targets = 8.

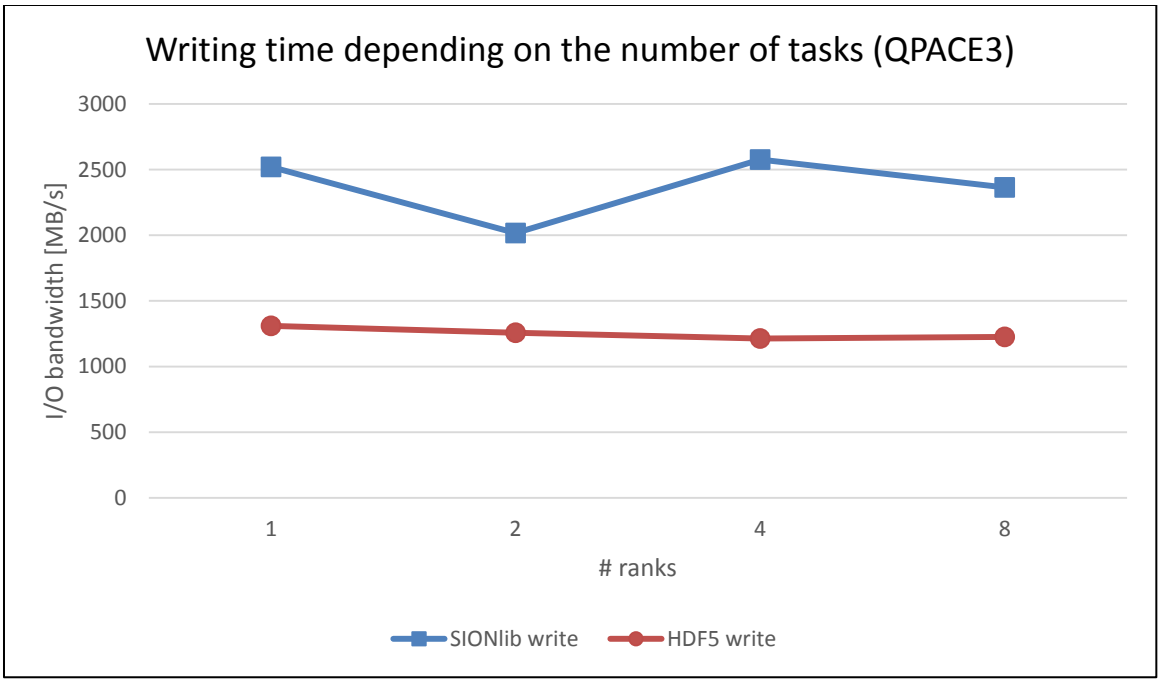


Figure 65: Synthetic benchmark on QPACE3 comparing SIONlib and HDF5 writing time depending on the number of tasks (UREG)

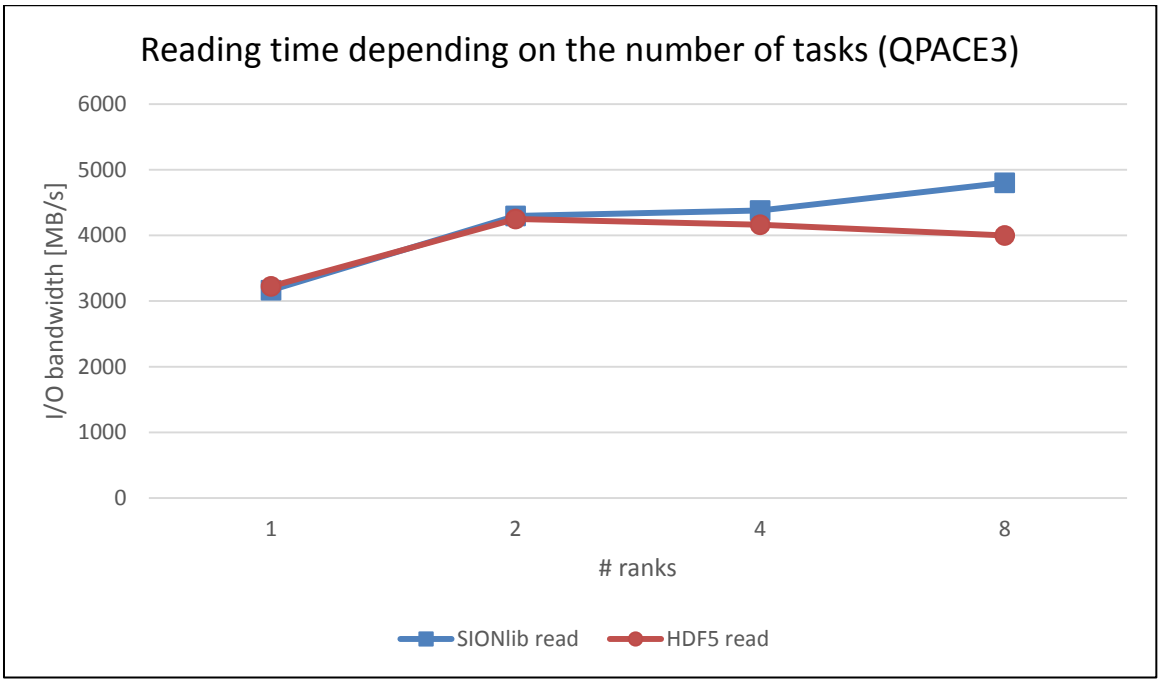


Figure 66: Synthetic benchmark on QPACE3 comparing SIONlib and HDF5 reading time (UREG)

Experiment details	Synthetic benchmark
Other experiment details	Libraries: HDF5 and SIONlib Filesystem: BeeGFS Chunk size: 8 MB Buffer size: 500 MB Targets: 8

Used system	QPACE3 (with BeeGFS)
Compiler version	gcc 6.2.1
MPI runtime version	mvapich2-2.2-psm2
Compilation flags	No extra flags
MPI processes per Node	1

Table 39: Benchmarking setup regarding SIONlib/HDF5 comparison (UREG)

10.4.3 E10 integration

Figure 67 shows results measured on the DEEP-ER SDV by using MPIWRAP for E10 support. The left part of the figure shows the writing time for Chroma for a small file size of 10 MB when running with and without E10. When using E10 the writing time could be decreased by 25%. The right part of the figure shows that the performance win is even higher when writing bigger files like it is shown there for 7 GB. Here E10 leads to a performance win of 57%.

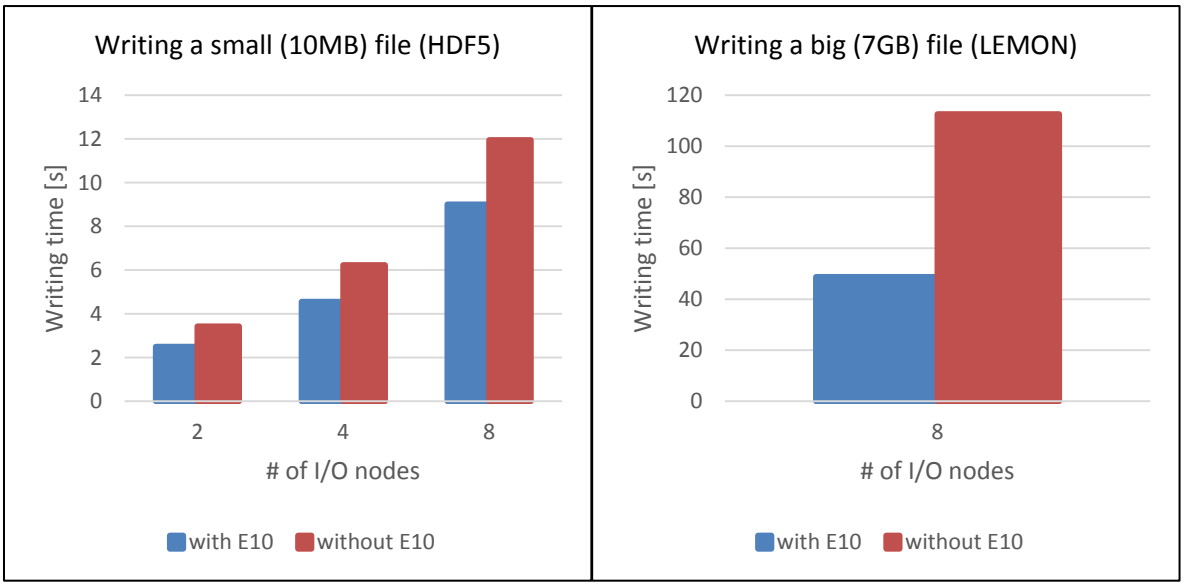


Figure 67: Full application benchmark with and without MPIWRAP on the SDV (UREG)

Experiment details	Chroma full application benchmark
Compiler version	gcc
MPI runtime version	ParaStation MPI
Compilation flags	No extra flags

Table 40: Benchmarking setup regarding E10 integration (UREG)

10.4.4 Full application I/O benchmark

This subsections shows I/O measurements for the full Chroma application on QPACE3. The same data is written with different methods: QIO serial (POSIX mode, serial write), QIO parallel (QIO routines based on POSIX) and LEMON (always parallel, uses MPI I/O

internally). Figure 68 shows a weak scaling test keeping the number of total MPI tasks constant (32) as well as the physical problem size and written data.

QIO serial naturally gets worse if the number of nodes is increased, as all data needs to be aggregated on a single node prior to writing data to disk. Lemon and QIO parallel can actually reduce the I/O time if more nodes are used, whereas QIO parallel actually scales better than LEMON.

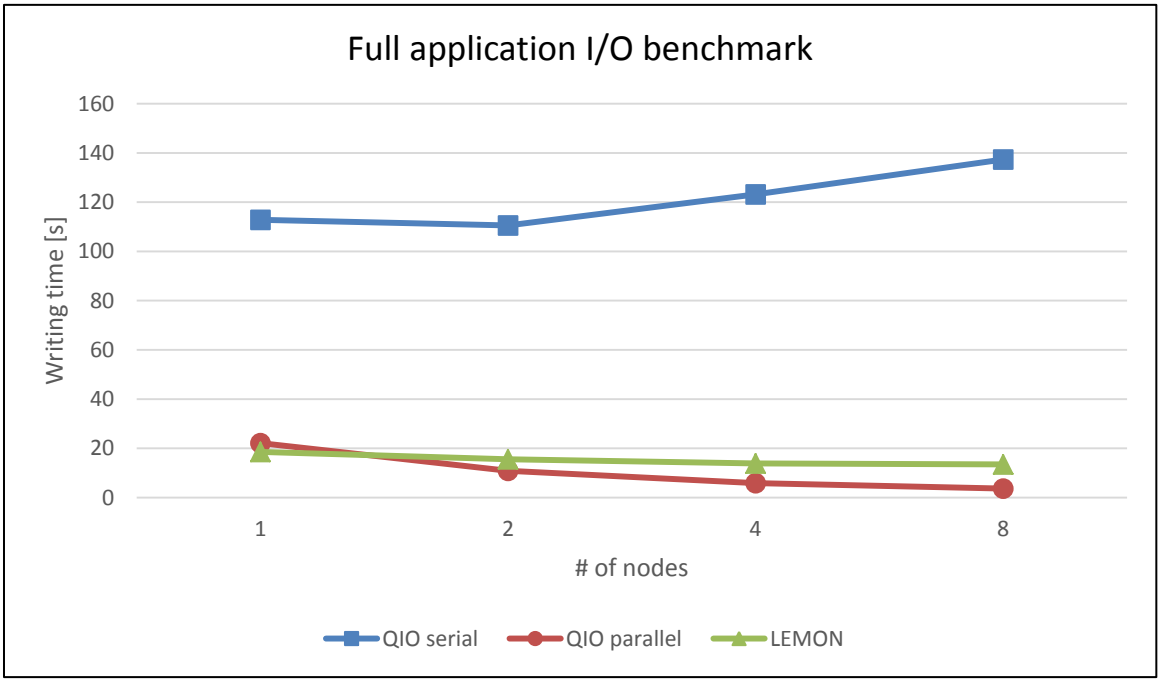


Figure 68: Full application I/O benchmark (UREG)

10.5 Comparison between architectures

10.5.1 Intra node scaling

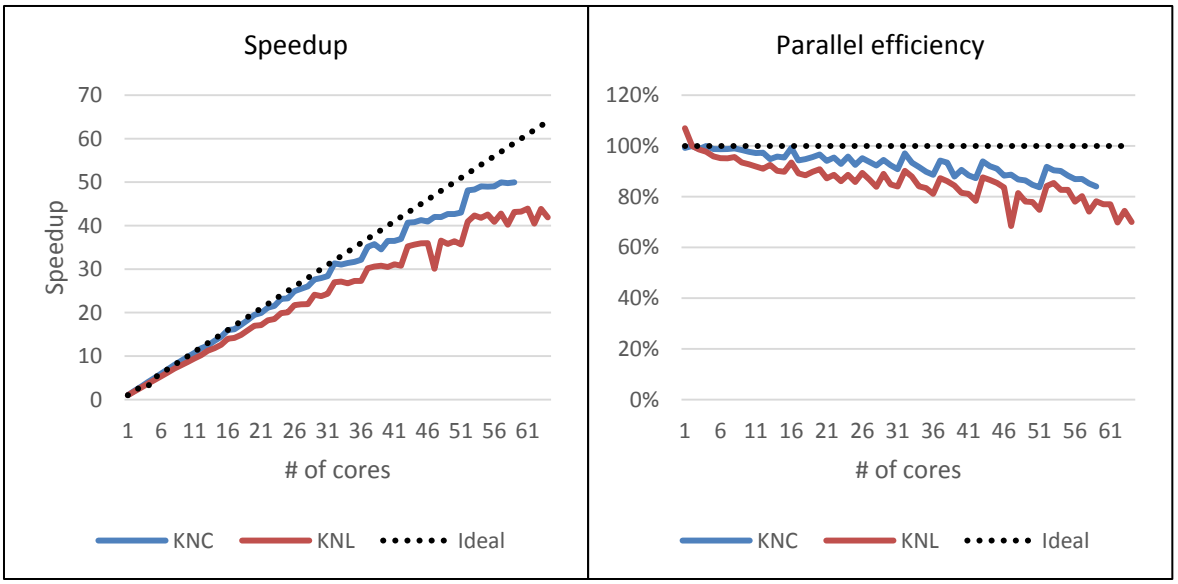


Figure 69: Intra node scaling speedup and parallel efficiency comparison between KNC and KNL (UREG)

Intra node scaling shows a rather good scaling behaviour (Figure 69). However, although the KNL is better in absolute numbers (Figure 70) the scaling behaviour is not quite as good as for the KNC. It is believed flat mode is the best for the application, because of its very regular memory access pattern. However, this is not a generic result and depends highly on the application in question.

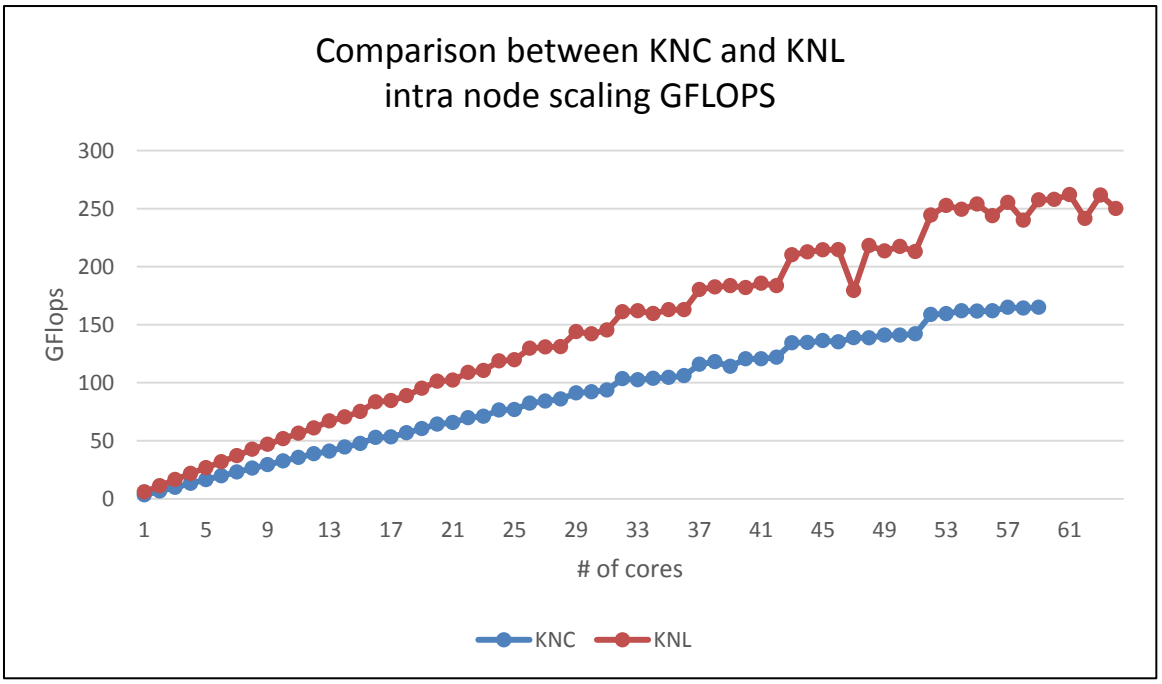


Figure 70: Intra node scaling GFlop/s comparison between KNC and KNL (UREG)

10.5.2 Inter node scaling

The intra node scaling evaluates the solver. The inter node scaling now shows effects of communication. Now the scaling behaviour does not follow the ideal line any more (see Figure 71). It can be seen that parallel efficiency drops to less than 50% for 32 nodes. This is an effect of the increased communication. (This is a plot for strong scaling, i.e., with a constant problem size.). Communication network was 100 Gbit OmniPath. As expected, KNL is better (faster) than KNC in terms of absolute performance also during the inter node tests (see Figure 72). Note that we only have two data points for the KNCs, we could not vary the number of nodes more for realistic lattice sizes, because of memory limitations.

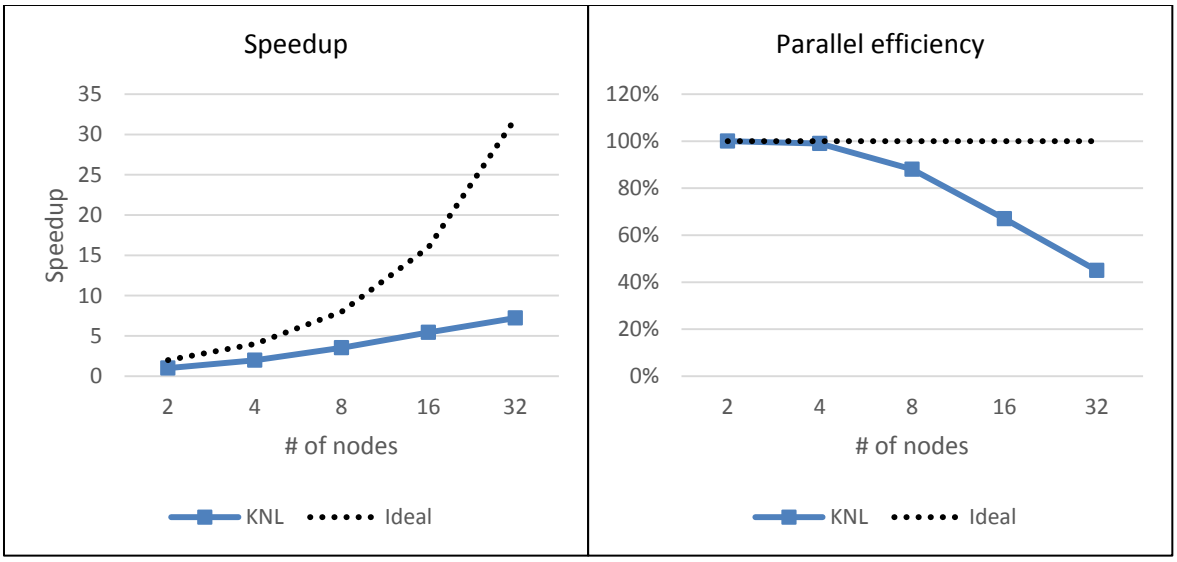


Figure 71: Inter node scaling speedup and parallel efficiency for KNL (UREG)

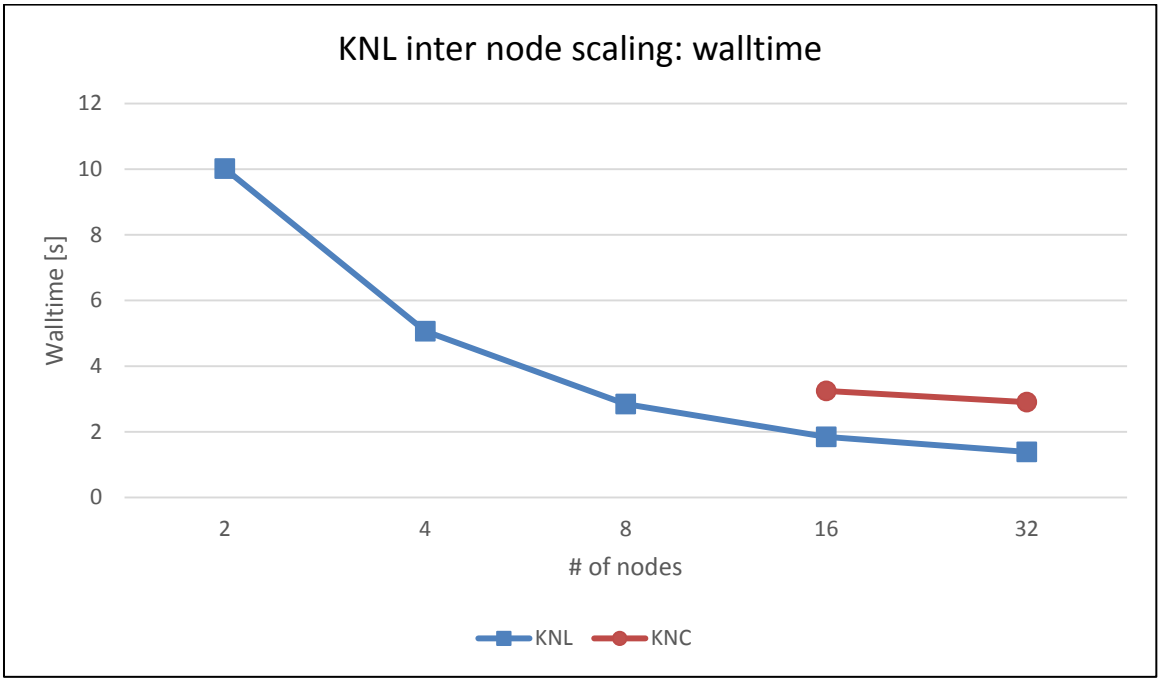


Figure 72: Inter node scaling walltime comparison between KNC and KNL (UREG)

Scaling	Strong scaling
Number of cells	Intra node: $16^3 \times 32$, inter node: $32^3 \times 96$
Other experiment details	Inter node scaling figures show fGMRES-DR solver (flexible generalized minimal residual method with deflated restarts) and DD (domain decomposition) preconditioner. Intra node scaling figures show DD preconditioner only.
Compiler version	gcc 6.2.1
MPI runtime version	mvapich2-2.2-psm2
Compilation flags	CFLAGS="-fopenmp -O3 -std=c99 -march=knl

	-fargument-noalias-global -funroll-all-loops -fpeel-loops" CXXFLAGS="-fopenmp -O3 -std=c++11 -march=knl -fargument-noalias-global -funroll-all-loops -fpeel-loops -finline-limit=50000"
MPI processes per Node	4
Threads per process	256 (for 1 core) or 64 for more cores

Table 41: Benchmarking setup for intra and inter node scaling (UREG)

10.5.3 Full application benchmark

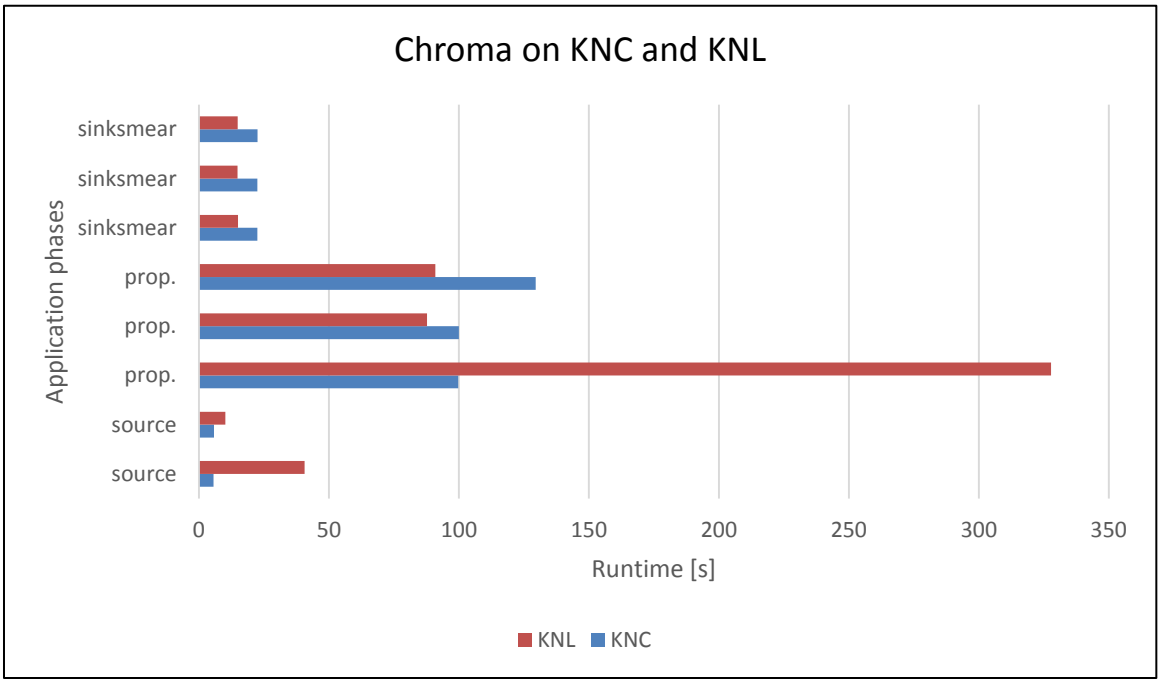


Figure 73: Comparing Chroma on KNC and KNL (UREG)

The full application benchmark shown in Figure 73 uses a $48^3 \times 128$ lattice for KNL, and a $48^3 \times 96$ lattice for KNC (numbers for KNC have been rescaled for comparison). The used benchmarking setup is shown in Table 42. This figure shows a mixed picture between architectures. Keep in mind that the code for KNC had a lot of architecture specific code in it, whereas the KNL code was rewritten with a more general approach in mind. The different groups show the different phases during a full application run: The generation of a point source, generation of a smeared sink, and then the computation of the propagator. The physical meaning is that a quark travels (propagates) from the source to the sink. Technically, the computation of this propagation corresponds to the (approximate) inversion of a sparse matrix.

Scaling	Time consumed by the different phases of a full application run.
Number of cells	483×128 (KNL), 483×96 (KNC)

Other experiment details	gauge configuration N303r000
Compiler version	gcc 6.2.1
MPI runtime version	mvapich2-2.2-psm2
Compilation flags	CFLAGS="-fopenmp -O3 -std=c99 -march=knl -fargument-noalias-global -funroll-all-loops -fpeel-loops" CXXFLAGS="-fopenmp -O3 -std=c++11 -march=knl -fargument-noalias-global -funroll-all-loops -fpeel-loops -finline-limit=50000"
MPI processes per Node	4
Threads per process	64

Table 42: Experiment setup for the full application benchmark (UREG)

10.6 Conclusion

This project made it possible to develop a well working version of Chroma for the KNC and KNL architectures. As a side effect, the implementation also improved for standard Xeon architecture as the new models also implement the AVX instruction sets. Restructuring the code to be more portable and less architecture dependent made the code also more accessible for optimising compilers. KNL, being currently the most efficient architecture for our specific code (in terms of energy efficiency), has also reduced our cost-to-solution. (KNL system is #5 in the Green500 list from November 2016, all systems before are graphics cards or not generally available CPUS.)

Some discussions with other team members showed that some special features of DEEP‑ER are not really applicable to our specialised application (notably NAM) or the direct use of the NVMe devices (Chroma uses them indirectly via E10). However, it was expected that it wouldn't be possible to make use of all the features of a machine that is suited for such a wide range of applications.

Although our own machines QPACE2 and QPACE3 use different networks and different network topologies (InfiniBand with hypertorus topology, Omni-Path with tree topology) than the DEEP-ER Booster, the co-design discussions gave us a deeper insight of what is feasible and what is desirable with respect to networks.

The exchange with other Work Packages, notably WP4, lead to design decisions for our own new machine – QPACE3. For this machine, BeeGFS is used as the main filesystem, for /scratch as well as for /home. Ease of use of E10 via the MPIWRAP library is a viable improvement for I/O, especially in conjunction with BeeGFS. It is expected that the good experiences with the BeeGFS filesystem and improved I/O throughput will carry over not only to lattice QCD, but also to other emergent projects that will be engaged in in the future.

11 Global conclusions

The DEEP-ER Project has demonstrated the potential of the DEEP-ER Architecture and its three strength points:

(1) *Flexibility of the DEEP/DEEP-ER concept:* The new architecture makes different usage models possible. Every application in DEEP-ER used different system configurations that included either Cluster or Booster, or a combination of both.

(2) *I/O optimisations*: This is one of the two fundamental improvements of the DEEP-ER Project over its predecessor DEEP. Efficient I/O at scale is vital to allow applications run in reasonable time and to make sure they can progress towards the solution of the problem. As previously discussed, DEEP-ER has three different I/O components (or libraries) that optimise different aspects of I/O. The SIONlib library focuses on task local I/O. The BeeGFS file system provides scalable I/O based on local NVMe devices and new cache APIs, finally E10 provides an improved version of collective I/O exploiting the available caching infrastructure in the DEEP-ER System.

(3) Resiliency support: This is the second fundamental improvement of the DEEP-ER Project over its predecessor DEEP. The motivation for having reliability support in the system is that at extreme scale failures will be frequent. In order to be able to use the system effectively it is necessary to make sure that applications can make progress in the computation. In DEEP-ER different resiliency features were provided like SCR, optimising user-level checkpointing, or OmpSs task based resiliency, supporting transparent task checkpointing.

Seven real-world HPC applications evaluated the DEEP-ER hardware and software. These applications were chosen to reflect a wide user range. They reach from processing radio astronomy data over enhancing oil exploration to researching human exposure to electromagnetic fields.

Admittedly most of the results shown in this document are limited to a small setup, since only the SDV System was available at the time of writing this report. However, efforts were made to do some larger scaling tests. The recently installed QPACE3 system was chosen, because its environment made it possible to use some of the DEEP-ER features. QPACE3 consists of 352 KNL nodes and BeeGFS is used as file system. With the help of the system administrators we were able to create a DEEP-ER like set up. The local NVMe devices of DEEP-ER were emulated through RAM discs. The same ParaStation MPI environment as the one present on the SDV was installed on QPACE3. In the limited amount of time in which access to the platform was available, it was possible to port one of the DEEP-ER applications to QPACE3, obtaining with it benchmark results of I/O scaling up to 64 nodes. The results of those measurements are shown in Section 4.5.3. Nevertheless, the small scale setup used was enough to show very promising results:

- The space weather simulation from KU Leuven was brought to a new generation Particle-in-cell code called xPic during the DEEP-ER Project. The new code outperforms the old iPic3D code in terms of computation and I/O: The new particle solver was optimised for KNL. The I/O performance of xPic was improved by using the SIONlib library and the local NVME devices. Also the new code is now resilient and very easy to use thanks to the integration of SCR.
- TurboRVB and 2degas are the two Quantum Monte Carlo codes used by CINECA. 2deags was optimised for KNL and achieved a good speedup compared to KNC. This could also be increased by using the MCDRAM. TurboRVB is now resilient and achieved an impressive restart performance improvement by using the SIONlib library and the local NVMe devices.

- The GERShWIN application from Inria benefitted greatly from the DEEP-ER Project. First there have been significant improvements in performance on KNC and KNL through extensive porting and optimisation efforts. The next achievement is the highly increased I/O performance. The SIONlib library, in combination with the usage of NVMe devices, was perfectly suited for the parallel task local I/O of GERShWIN. Moreover, GERShWIN now is resilient and the integration of SCR and OmpSs persistent checkpoint/restart feature makes it possible to manage the checkpoint/restart mechanism in a scalable way.
- SeisSol was able to port, evaluate, and improve the state-of-the-art earthquake rupture dynamics source code during the DEEP-ER Project. The I/O capabilities of SeisSol were greatly improved with SIONlib. Since the results were very good a SIONlib backend was added to the already existing I/O backends (i.e. HDF5 and POSIX). Another approach of I/O optimisation was to use E10 and BeeOND to make use of the local NVMe devices. The code can already be considered an application-software candidate for the first Exascale systems in the future.
- The DEEP-ER Project had a great impact on the development of a highly optimised correlation and imaging code for the data processing pipeline from ASTRON. The new developed image domain gridding library was integrated in the widely used WSClean imager. The I/O was optimised by using the NVMe devices as fast, local buffers for data storage and checkpointing. Thanks to the wide range of resiliency features, both parts, the correlator and the imager, are now resilient.
- The FWI application from BSC achieved several great improvements. The use of the OmpSs offload was one significant optimisation. It simplifies the process of creating, monitoring and scheduling a large number of shots and also provides a resiliency feature. The performance of the FWI application was increased and the code demonstrated to be portable with consistent performance on different Intel architectures. Another great improvement was achieved by using the NVMe devices to increase the I/O performance. Since the I/O time is quite significant in the FWI application this improvement was very important on the way to Exascale.
- During the DEEP-ER Project the QCD code Chroma from UREG was optimised to run efficiently on KNC and KNL architectures. One big achievement during the project was to make the code more portable and less architecture dependent. The other great achievement was the I/O optimisation by using the MPIWRAP library for E10 support. The great experiences with the BeeGFS file system made in the DEEP-ER Project led to the decision of using BeeGFS for the new installed QPACE3 machine, administrated by UREG.

Concluding, this report on applications experience clearly shows that the DEEP-ER System is flexible enough to accommodate the requirements coming from different problem domains. Each application in the DEEP-ER Project has benefitted from the experience made by other applications, and from the support of technical Work Packages, such as WP3, WP4 and WP5, to use the system in the most efficient way.

Annex A

A.1 Best practices Guide

One of the biggest advantages of the DEEP-ER architecture is its flexibility. So it can be a target for a wide range of applications. This guide shows how to get the optimal performance out of the applications on this architecture. Each kind of application may have different ways to use the DEEP-ER architecture (like the different applications described in this document). Below all the opportunities will be explained, hints will be given about which way will be the best for the application to get all the benefits from the DEEP-ER architecture and examples on the improvements that can be achieved will be shown.

Step 1 Analysis:

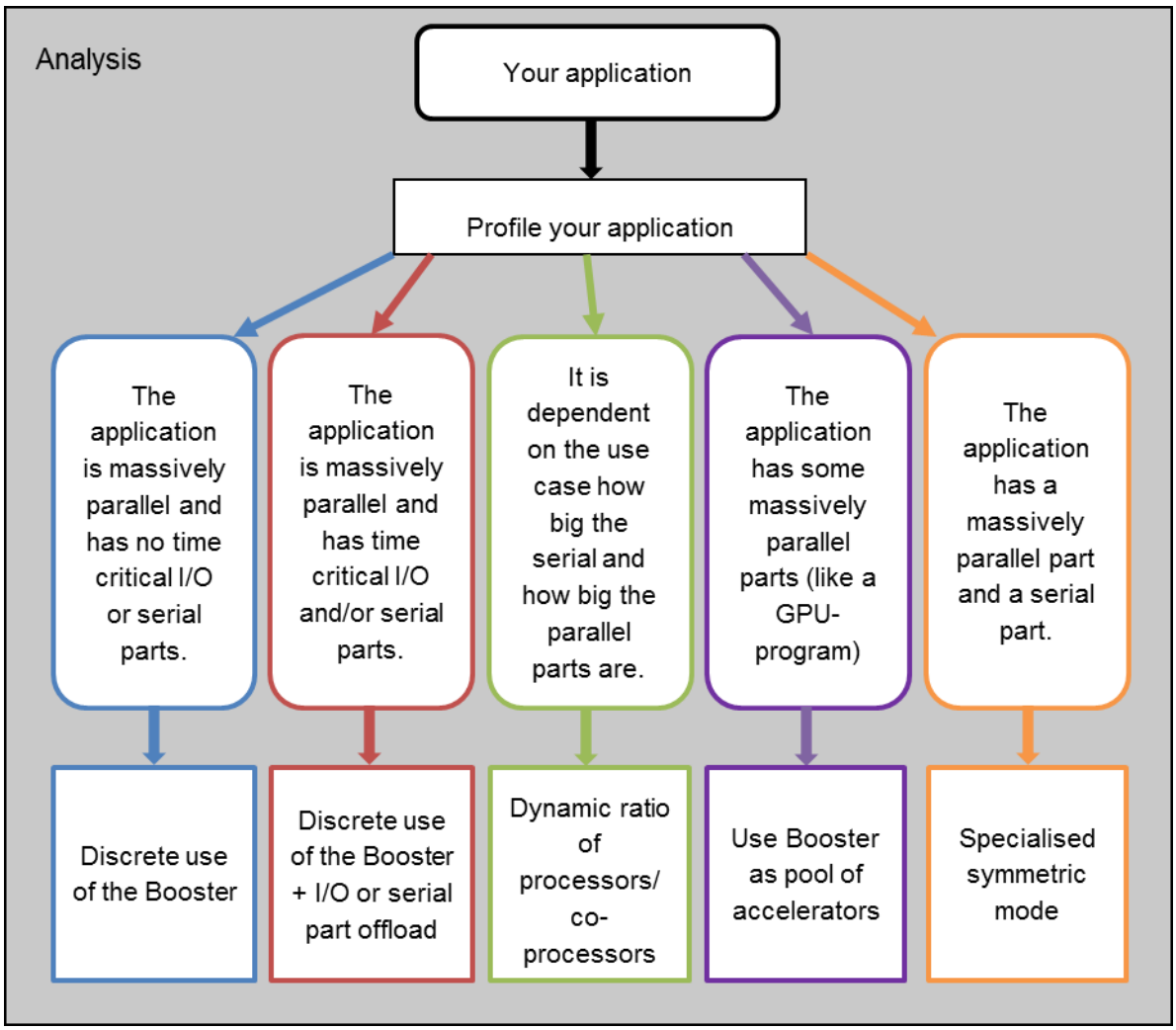


Figure 74: Analysis (Step 1)

The first step is to analyse the application. A detailed analysis of the code is essential to get to know which parts of the code can benefit from which parts of the architecture. Without this it's not possible to get all the benefits out of the DEEP-ER architecture. Here are some recommended profiling tools (all available on the DEEP-(ER) systems):

- Intel Vtune Amplifier [26]
- Intel Vector Advisor [27]
- Scalasca [28]
- Extrae/Paraver [29]

The tools will determine e.g. what the most time consuming parts are, whether the application is compute, memory or bandwidth bound or how well balanced the application is. Some of the tools like Intel Vector Advisor will give hints about the parallelisation and vectorisation potential.

The detailed analysis will show to which of the 5 categories shown in Figure 74 the application belongs to. Each application with massively parallel parts and without any time critical I/O or serial parts (e.g. Chroma from UREG) belongs to the first category (blue) and should use only the Booster. Applications with massively parallel parts and time critical I/O and/or serial parts (like GERShWIN from Inria) belong to the second category (red) and should run on the Booster and offload the time critical I/O and serial parts to the Cluster. For applications where it's dependent on the use case how big the serial and how big the parallel parts are (e.g. the FWI application from BSC) the third category (green) is the best where a dynamical ratio of Cluster and Booster Nodes will be used. If the application has some massively parallel parts that are controlled by a serial part (like the OmpSs version of TurboRVB from CINECA) it belongs to the fourth category (purple) and should use the Booster Nodes as a pool of accelerators. The last category (orange) an application can belong to is when the amount of serial and massively parallel parts is about the same and both parts should run in parallel (e.g. xPic from KU Leuven). In this case the application should run in a specialised symmetric mode on Cluster and Booster.

Step 2 Cluster-Booster division:

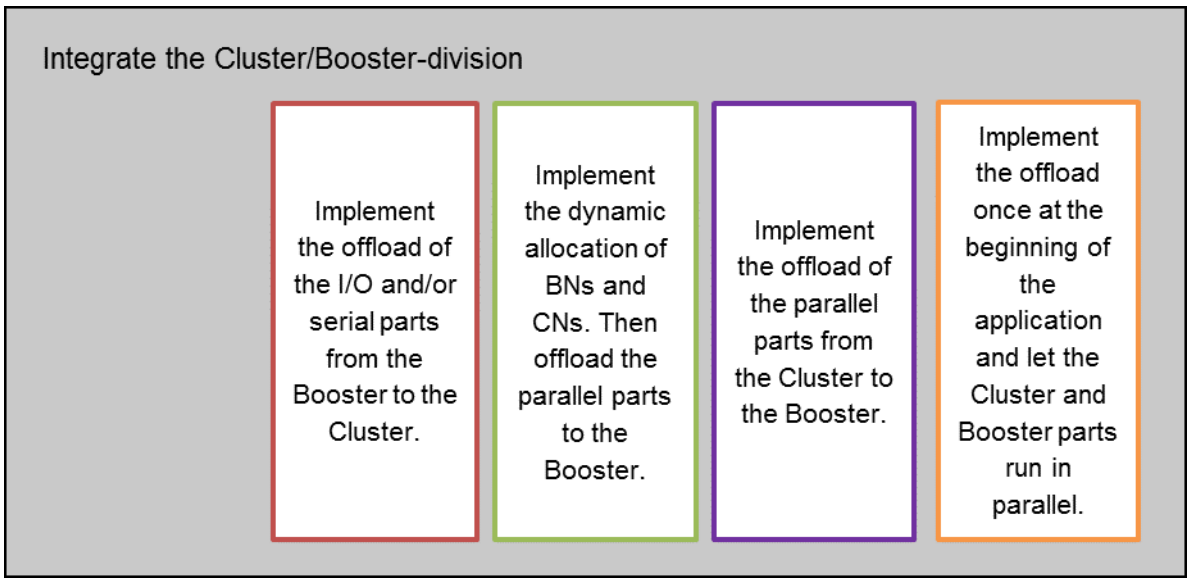


Figure 75: Cluster-Booster division (Step 2)

The next step for all categories except the blue one (discrete use of the Booster) is to implement the Cluster-Booster division like it is described in Figure 75.

There are 2 techniques for the Cluster-Booster division:

- Offload tasks with OmpSs [30]:
  Pragmas will be used to mark the code sections that should be offloaded. Communication between the host and the offloaded parts is done through the input and output variables within the pragma.
- Offload code parts with MPI_Comm_spawn [31]:
  With MPI_Comm_spawn a new MPI communicator is created. The command determines where the processes within this communicator should run (Cluster or Booster) by a host list. Another parameter for the command is an executable of the code part that should be offloaded. The new and the former MPI communicator can communicate with each other the whole time with the usual MPI send and receive commands.

Belonging to the fifth category (orange) you can both start on the Booster Nodes or on the Cluster Nodes and then offload the other part. You should implement the offload once at the beginning of the application and then both parts will run in parallel. In most cases the parts will have to communicate so the MPI_Comm_spawn offload is recommended here. For that you have to include the following command to do the offload:

```
MPI_Comm_spawn("offload.c", args, procs, inf, rank, com, intercom, err);
```

In xPic from KU Leuven this type of offload is used. A detailed describtion of the offload was given in the final application Deliverable of the D8.3 of the DEEP project [32] in Section 3.2.3.

In case of the second category (red) you will start the application on the Booster (host part) and the I/O operations and serial parts will be offloaded to the Cluster (offload part). This approach is used within GERShWIN from Inria and Section 5.2.3 in D6.2 [3] gives some ideas on how to do this.
In the third category (green) you can either start on the Booster Nodes or on the Cluster Nodes depending on the structure of your program (try to minimise the data transfer for the offload). Then offload as many processes as you need to the other part. You can also use a nested offload here like BSC within the FWI application. For more details please see Section 8.2.3 of D6.2.

In case of the fourth category (purple) you will start the application on the Cluster (host part) and offload the massively parallel parts to the Booster (offload parts). This is e.g. the case for applications with a master-slave fashion like TurboRVB from CINECA (see Section 4.2.4 in D6.2.

Since communication in those cases between both parts (host and offload) is only needed at the beginning and ending of the offload section, the OmpSs offload is recommended here. Then all parts to be offloaded would be surrounded by a pragma like this:

```
#pragma omp task device(mpi) onto(com,rank) in(var1, var2) out (var3)
  {
      Code segments to be offloaded…
  }
```

The communicator named in the `onto` clause has to be created with a `deep_booster_alloc` call.

To summarise it: To perform the offload it is recommended to use OmpSs, except when the host and the offload part should communicate not only at the beginning and the end (orange category). In any other case OmpSs should be used, because when offloading code parts with OmpSs the application developer is able to make use of the resiliency features of OmpSs (Step 5).

Step 3 KNL optimisation:

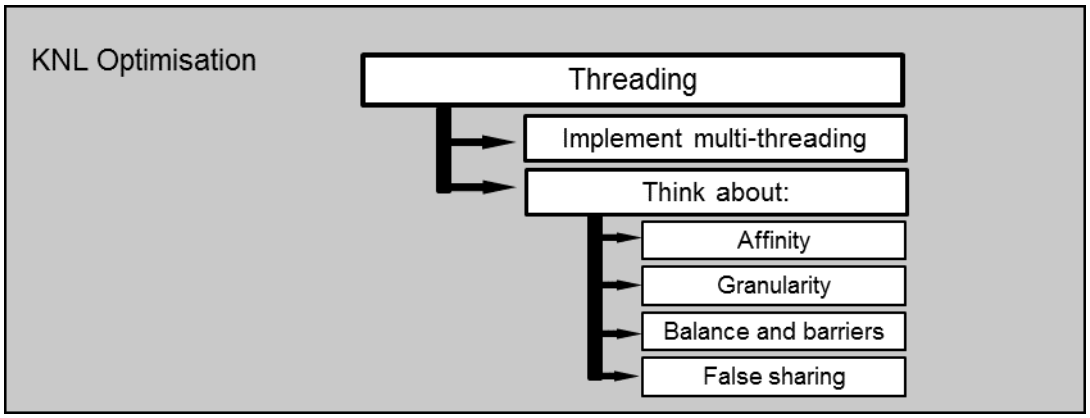


Figure 76: KNL optimisation – Threading (Step 3a)

The next step is optimising the code for KNL. This step is essential to make efficient use of the Booster Nodes. There are several things to consider. The first optimisation step is the threading (Figure 76). Codes that should run on KNLs should be able to use many cores and scale well when increasing the number of cores. So using multi-threading is an important point. To optimise the scaling behaviour when using multiple cores there are several things to take into account e.g. affinity or false sharing. Detailed information about the threading can be found in Intel's Guide for Developing Multithreaded Applications [33].

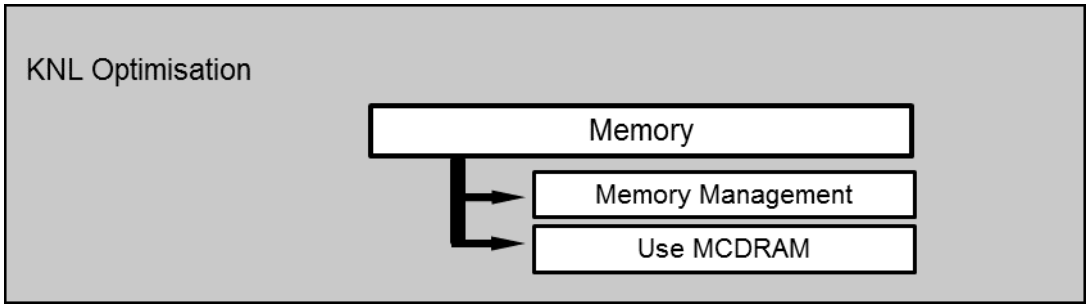


Figure 77: KNL optimisation – Memory (Step 3b)

The next point is to optimise the memory access (Figure 77). The above mentioned guide from Intel gives also some hints regarding the memory management. Another important point is the use of the MCDRAM. The KNLs provide some on-package memory (16 GB), which is called MCDRAM. Data that is used often should be stored there, because the MCDRAM is much faster than the DDR4. If the application uses not more than 16 GB, the whole application can run within this on-package memory. The MCDRAM can be used in 3 different modes: cache mode, flat mode, and hybrid mode (Figure 79). In cache mode the MCDRAM is treated as Last Level Cache and it's used automatically. In flat mode the MCDRAM is like a NUMA node and the usage is controlled by the application developer. For this mode the application can be bound to the MCDRAM during the execution command like this:

```
numactl -m 1 ./example_code
```

In this case everything will be stored in the MCDRAM. If there is not enough space on the MCDRAM the application will be terminated. With the following command the MCRDAM will be preferred as long as there is enough space. The rest of the data will then be stored in the DDR4:

```
Numactl –p 1 ./example_code
```

To control which data should be stored within the MCDRAM the memkind library has to be used. When running in flat mode there are 2 NUMA nodes. NUMA node 0 is the DDR4 and NUMA node 1 is the MCDRAM. With the following 2 lines all data will be allocated on the DDR4 (NUMA node 0) except for the memkind allocations on the MCDRAM (NUMA node 1).

```
export MEMKIND_HBW_NODES=1

numactl --membind=0 ./example_code
```

The hybrid mode is a combination of cache and flat mode. The different modes are selected at boot time. The presentation from Intel about MCDRAM provides more detailed information [34].

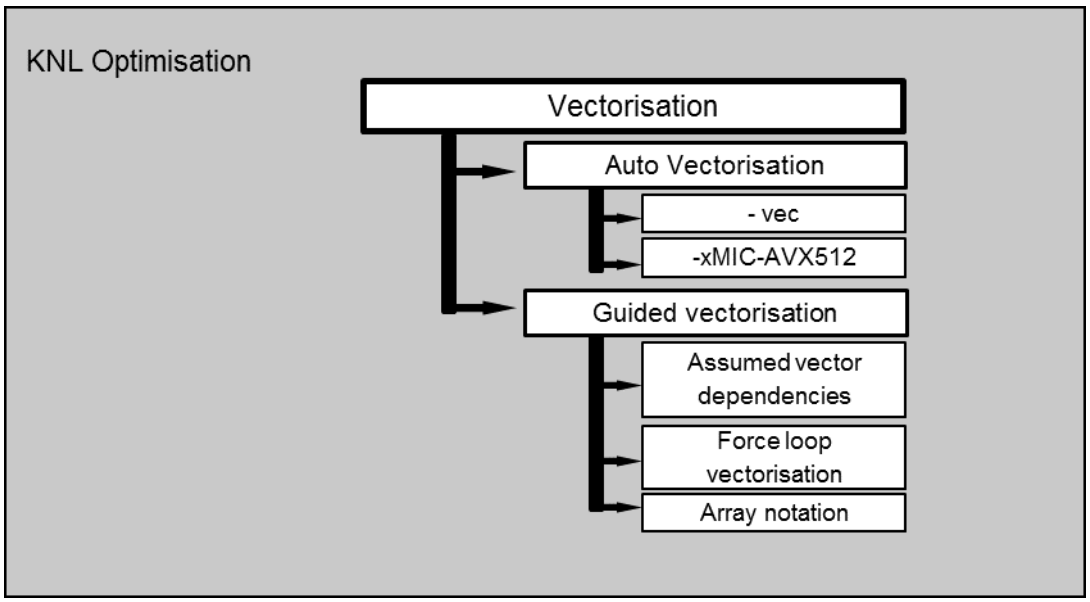


Figure 78: KNL optimisation – Vectorisation (Step 3c)

The third step of KNL optimisation is vectorisation (Figure 78). The easiest way is to use the auto vectorisation with the help of compilation options like `–vec` or `–xMIC-AVX512`. With this approach the compiler will decide what will be vectorised. It is the easiest way, because no code changes are needed. But it might be that the compiler does not vectorise everything the application developer would like to be vectorised. Therefore the guided vectorisation approach can be used. In some loops it could be necessary to inform the compiler that there are no vector dependencies with some hints, e.g. `#pragma ivdep` (ignore any assumed vector dependencies). Another opportunity is to force the compiler to ignore all dependencies and vectorise the loop, e.g. with `#pragma simd` or `#pragma vector`. This approach only needs minor code changes (adding the pragmas in front of the loops). A third way in the guided vectorisation is the use of the array notation. This technique requires some more code changes since the for-loops have to be

replaced by the array notation. When array notation is used the compiler will use the SIMD instruction set. The Best practice Guide from Intel about vectorisation shows some more detail and code examples [35].

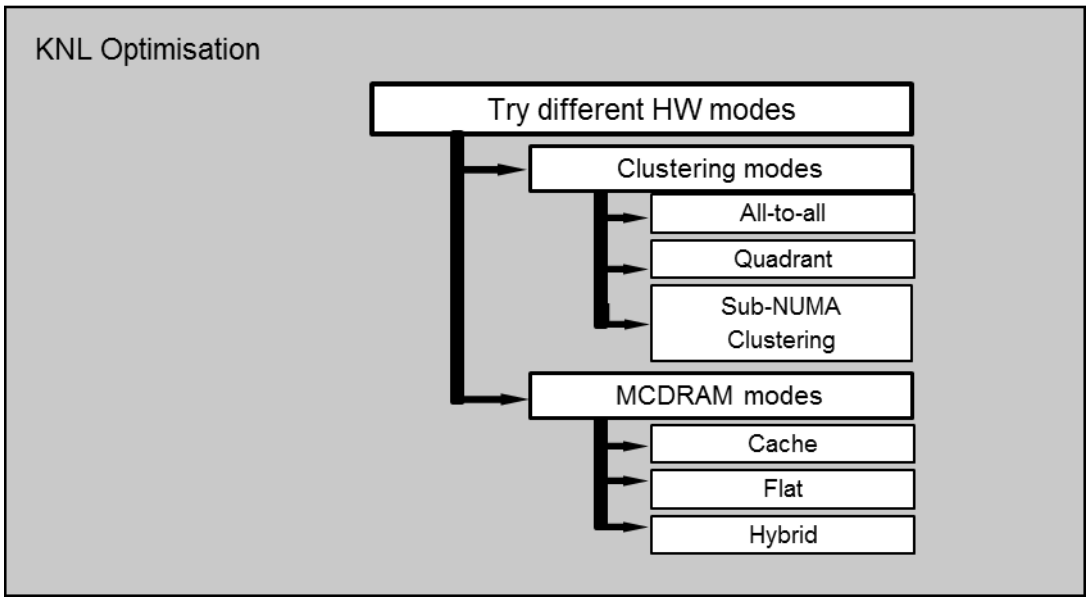


Figure 79: KNL optimisation – HW modes (Step 3d)

The next thing to investigate is the different hardware modes the KNL provides (Figure 79). Besides the above mentioned different MCDRAM modes the KNL mesh interconnect supports 3 different clustering modes: all-to-all, quadrant and sub-NUMA clustering. In the all-to-all mode the address is uniformly hashed across all distributed tag directories and there is no affinity between the tag directory and the memory. The quadrant mode divides the chip in four quadrants with affinity between the tag directories and memory in each quadrant. It has a lower latency and higher bandwidth than the all-to-all mode. For both modes no code changes are needed. When using sub-NUMA clustering the cores appear as 4 (or 2) NUMA nodes to the OS. This is analogous to a 4-socket Xeon. When using OpenMP with multiple MPI ranks per processor, descriptors like `scatter` or `compact` should be used. In OpenMP codes without MPI the NUMA bindings have to be handled manually. For affinity control MPI mechanisms like `I_MPI_PIN_MODE` (Intel MPI) can be used. The GERShWIN application from Inria is a good example to see what could be achieved by optimising the application for KNL. The vectorisation efforts led to a speedup of 1.3 (compared to the threaded version). Through identifying and resolving bad memory alignments the speedup was increased to 2.1. Finally by using the MCDRAM a speedup of 1.4 was achieved compared to using the DDR (both using the optimised version). To get a compact overview on how to optimise for KNL, the optimisation and performance tuning guide from Intel [36] is recommended. The COLFAX research blog provides a webinar about KNL optimisation (and the corresponding slides) [37] which is also interesting.

Step 4 I/O optimisation:

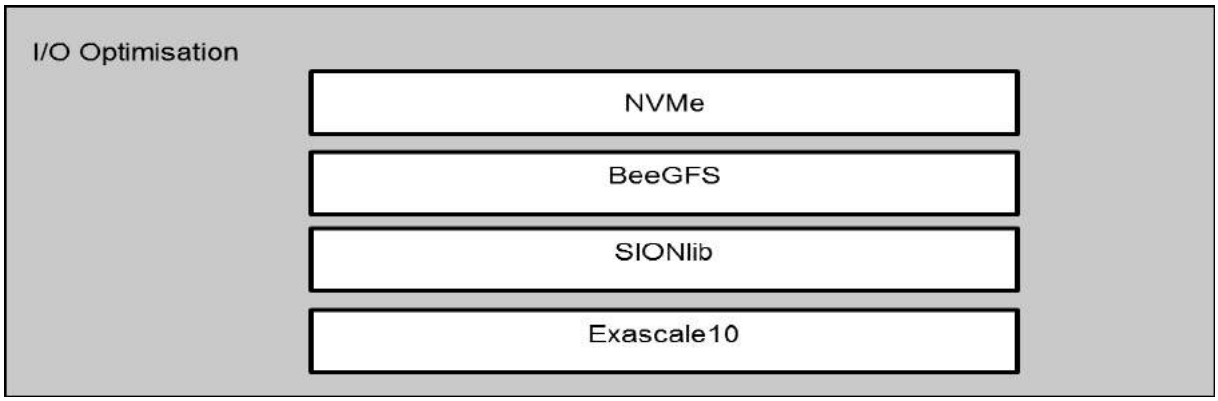


Figure 80: I/O optimisation (Step 4)

One important aspect of the DEEP-ER Project is the I/O optimisation. In lots of massively parallel applications the parallel I/O limits the performance. Since I/O didn't evolve as much as computation in the past and probably won't in the future, even applications which haven't performance limitations because of I/O now, might run into I/O bottlenecks when reaching for Exascale. To address this issue the DEEP-ER Project provides software as well as hardware features (shown in Figure 80).

The software stack for parallel I/O consists of three components: BeeGFS [38], SIONlib [39] and Exascale10 [40]. BeeGFS is the used parallel filesystem on the DEEP-ER storage system. It is split into two tiers. The upper tier is a partitioned non-coherent cache layer based on the NVMe devices (see below). This provides a linear scalability and very high throughput. The lower tier has high capacity for longer term data storage based on the HDD. The application developer can control the staging of data between the tiers with BeeOND [41].

Parallel task local I/O leads to a huge amount of files for large-scale parallel applications. The SIONlib library provides the opportunity to read/write data from/to thousands of processors into a small amount of files. Only small code adaptions are needed to use SIONlib. The open/close and read/write operations from C or FORTRAN have to be replaced with calls from the SIONlib API. The more MPI tasks are used for the I/O the more the application will benefit from SIONlib. SIONlib can potentially lower the writing time and also reducing the time for read operations like it is the case in TurboRVB. Here the amount of data that is written is not very large, so the writing time is not a critical point. But when using thousands of MPI task this would produce thousands of files without SIONlib. But if SIONlib is used the number of files is significantly reduced and so the TurboRVB application could speed up the reading time by a factor of 1000.

In parallel applications the workload is distributed over all processes and so is also the I/O. Exascale10 (E10) is a parallel I/O mechanism that overcomes current collective I/O limitations. E10 provides wrappers for `MPI_Init`, `MPI_Finalize`, `MPI_File_open` and `MPI_File_close`. The MPIWRAP wrapper library can be used to exploit the SSD cache features provided by E10. With MPIWRAP application developers can dynamically change the information passed to MPI-IO without the need of modifying the application. This approach was used in Chroma from UREG during their I/O optimisation. Chroma's

I/O time could be reduced to up to 50% (large files with 7 GB) and 25% (smaller files with 10 MB). More information about MPIWRAP can be found here [13].

On the hardware side the Non-Volatile-Memory (NVMe) devices will help to optimise the I/O performance. Each Booster Node will have its own NVMe devices. The NVMes are an SSD technology with 400 GB that is directly connected with PCIe which allows a higher bandwidth. When using the NVMe device the reading and writing time should be much faster compared to normal HDD. Only the paths to input and output files have to be changed. /nvme/tmp/filename can be used for data that is only temporally needed. When using /mnt/beeond/filename the file is stored on the local NVMe device, too. But the application developer has the chance to migrate it asynchronously to the global filesystem with the help of BeeOND. No other code changes are needed. In some initial tests with the data processing pipeline from ASTRON the achieved bandwidth was increased from 490 GByte/s (global BeeGFS) to 658 GByte/s (NVMe). More details about NVMe can be found in Deliverable D3.3 of the DEEP-ER Project [6].

Step 5 Resiliency:

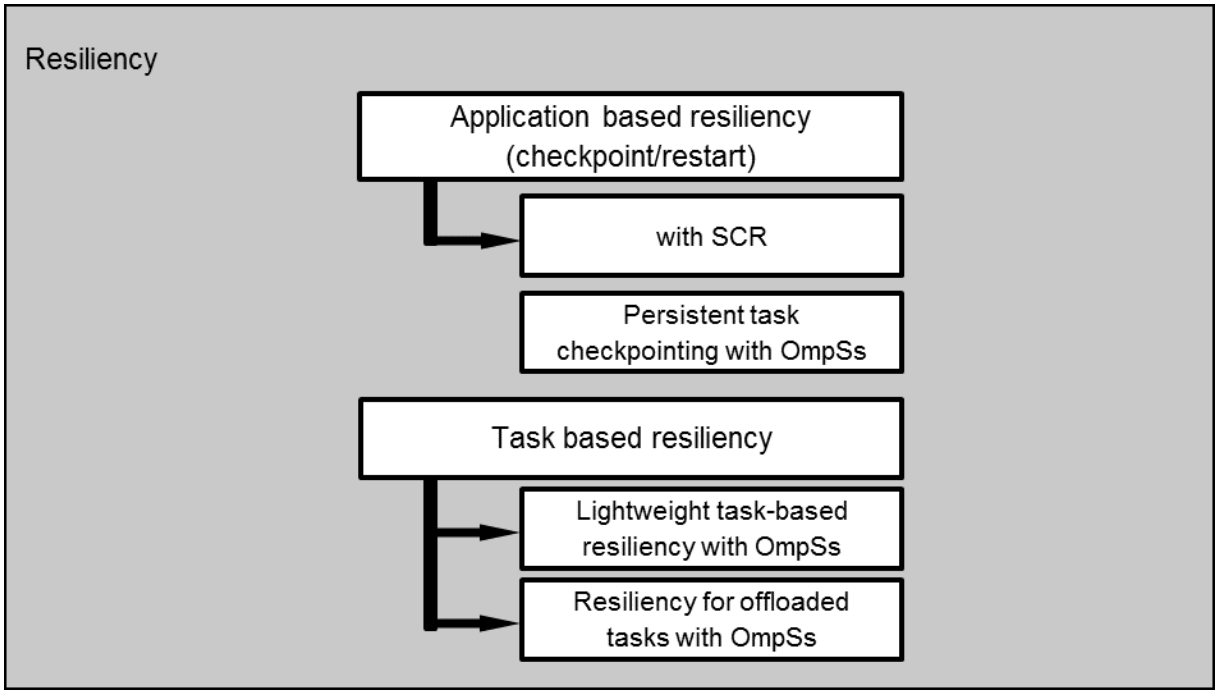


Figure 81: Resiliency support (Step 5)

The second important aspect of the DEEP-ER Project is resiliency. The resiliency methods developed in the project are flexible enough to accommodate the heterogeneous nature of systems like the DEEP-ER Prototype. The idea is to avoid the necessity of full application restart. Figure 81 shows an overview of the resiliency techniques within the DEEP-ER Project.

The first technique mentioned there is the application (or user) based checkpointing. The idea behind this is to write checkpoints from where the application can be restarted. This is necessary if a fatal hardware failure occurs. In this case the node has to be rebooted so the application won't be able to continue. But instead to start all over again after the reboot the application will start from the last written checkpoint. There are two possible

techniques for this approach: The scalable checkpoint-restart library SCR and the OmpSs persistent task-based checkpointing.

SCR supports multi-level checkpointing and redundancy (buddy-checkpointing). SCR also decides if writing a checkpoint is necessary. It can be used within MPI codes. The library and documentation can be found here [10].

SCR has to be started with `SCR_Initialize()`. This will initiate the checkpoint database and look for checkpoints to restart from. With `SCR_Need_Checkpoint(int *flag)` can be checked if a condition for writing a checkpoint is met. If this is the case a checkpoint can be created with `SCR_Start_Checkpoint()`, writing the checkpoint and `SCR_Complete_Checkpoint(int valid)`. At the end of the code (before `MPI_Finalize()`) `SCR_Finalize()` should be called. SCR was also integrated in the SIONlib library, so on the DEEP-ER architecture both libraries are combinable.

The other technique for application based resiliency mentioned in Figure 81 is the persistent task checkpointing (PTC) with OmpSs. Techniques, such as SCR, focus mainly on providing advanced I/O capabilities to minimize checkpoint/restart time. However, application developers are still in charge of manually serialize and de-serialize the application's state. Therefore new features have been implemented into the OmpSs programming model like the directive-based approach to perform application-level checkpoint/restart in a simplified way. The `checkpoint` clause allows the user to specify easily the state of the application that has to be saved and restored. The runtime system, which relies on SCR to perform scalable and efficient I/O operations, will do the serialization and deserialization activities. The checkpoint clause needs to be combined with the dependencies semantics (e.g. `in()` or `inout()`). The checkpoint task then will store these dependencies. Section 6.3.1 shows some results of GERShWIN when using PTC. The following OmpSs pragma shows an example where `cp_data` is checkpointed:

```
#pragma omp task in (cp_data) checkpoint ( )
```

In this example the checkpoint frequency is given by the underlying checkpoint/restart library (in our case SCR). The `checkpoint` clause also accepts an optional condition expression to let the user control the checkpoint frequency. In the following example a checkpoint of `cp_data` will be performed every five iterations:

```
#pragma omp task in (cp_data) checkpoint (iter%5==0)
```

The second topic of the resiliency features of the DEEP-ER Project is the novel task-based resiliency technique. The key idea is to "in-memory" checkpoint task inputs, so in case of a task failure the runtime can isolate the error and re-execute the affected task. The first task-based resiliency technique mentioned in Figure 81 is the lightweight task-based resiliency with OmpSs. The idea behind this technique is to checkpoint the task inputs when the task is ready to execute. When an error occurs within a checkpointed task, the runtime is able to restore and restart its execution. If no error occurs during the task execution, the runtime releases the dependencies and removes the checkpoint and related software data structures. To use the lightweight task-based resiliency the user has to add a `recover` clause to the task pragma:

```
#pragma omp task in(in_data) recover
```

In this example the task has been marked as recoverable, which means that the runtime will checkpoint the input data (`in_data`) before execution.

It is also possible to mark OmpSs offload tasks with the recover clause. This would be the second technique mentioned in the Figure 81, the resiliency for offloaded tasks with OmpSs:

```
deep_booster_alloc(MPI_COMM_WORLD, n, ppn, &comm);

#pragma omp task onto (comm, rank) recover
```

This code fragment allocates `ppn` offloaded processes in each of `n` hosts. Then a task is offloaded to each offloaded process. Without using resiliency a crash of one of the used offloaded processes would mean that the process manager terminates all processes (even the host processes). The idea of the resiliency for offloaded tasks now is that the process manager will limit the process cleanup upon failure to those processes sharing the same MPI_COMM_WORLD communicator of the failed process. So if one of the offloaded processes crashes, the failed processes will be replaced with a freshly spawned set of them. The FWI application from BSC uses this resiliency approach. Some results are shown in Section 9.3.

More information about all the resiliency features of OmpSs and the SCR library can be found in the deliverables of WP5 [42] [43].

A.2 Recommendation for improvements

During the DEEP-ER Project WP6 evaluated the DEEP-ER architecture on the SDV. Since the project developed several new features there are some areas that could be further optimised. This section lists some of the recommendations gathered by WP6.

- *Switching the KNL hardware modes:* The KNLs provide several clustering and MCDRAM modes. It is dependent on the application which mode gives the best performance. For this reason it would be very useful if the mode could be chosen during the job allocation. This would mean the batch system has to be able to switch the modes for the selected KNL(s). Unfortunately, both types of modes can only be selected at boot-time. Therefore, implementing such feature is non-trivial and requires fundamental changes to the batch-system.
- *More features for libNAM:* Several ideas concerning usage-models of the NAM and additional functionality that it could provide were discussed in DEEP-ER. During the project's lifetime, the XOR checkpointing functionality was developed. Other use-cases, such as collective operations, would be of interest for the applications and will be considered for future evolution of the NAM technology in the DEEP-EST context.
- *NAM capacity:* The NAM prototypes constructed in DEEP-ER are restricted to 2GB memory capacity due to the hardware characteristics of the current Hybrid Memory Cube technology. For some applications, a higher memory capacity of the NAM would be desirable. Other applications would focus more on the computing capabilities of such devices and the fact that it resides within the fabric and the performance benefits emanating from this. For this reason, the DEEP-EST project will further develop the NAM in two parallel directions: the "NAM", a high capacity (but medium speed) memory connected to the network; and the "Global Collective Engine" (GCE), a processing element combined with a reduced amount of volatile memory directly attached to the EXTOLL network.
- *SCR+SIONlib:* During the project the SCR+SIONlib coupled buddy-checkpointing has been implemented for C applications. The application developers recommended providing also FORTRAN support, to be able to use this feature in a wider range of applications. The SIONlib developers are aware of the requirement and plan to implement the FORTRAN support in the future.
- *Extending the DEEP-ER architecture:* The xPic application represents just one part of the full chain of the space weather simulation package of KU Leuven. The development team is currently planning to enlarge the simulation pipeline by an additional component to forecast energetic events on the Sun using Data Analytics. For this purpose, an additional computing module would be required. Such element is foreseen for the DEEP-EST Prototype, allowing KU Leuven to implement the full application chain utilising a single modular platform.

Annex B

B.1 Characteristics of the SDV

The SDV (Software Development Vehicle) installed at JUELICH was used by the application developers in WP6 for tests and performance measurements. Its main characteristics are summarised here.

Node type	Supermicro (19” Rackmount 1 U)
Number of nodes	16 Cluster nodes; 8 KNL nodes
Cluster:	
Processors per node	2x Intel Xeon E5-2680v3 (Haswell)
Cores per node	2x 12 cores (@2.5 GHz)
Threads per node	using hyper-threading: 48 threads/node
KNL:	
Processors per node	1x Intel Knights Landing 7210/7250
Cores per node	1x 64/68 cores (@1.3 GHz)
Threads per node	256/272
Total number of cores	384 Haswell and 516 KNL cores
Memory per node	128 GB (Cluster), 96 KNL plus 400GB NVMe device on each node
Interconnect network	EXTOLL
Filesystem	GPFS from JUST cluster exported through NFS to compute nodes BeeGFS as a parallel scratch using local storage (24 x 6 TB)
Operating system	CentOS 6.5
Batch system	Torque + Moab
MPI runtime	ParaStation MPI
Compilers	Intel Professional Fortran, C + C++

Table 43: System information: SDV

B.2 Characteristics of DEEP Cluster

The DEEP Cluster installed at JUELICH was used by the application developers in WP6 for tests and performance measurements. Its main characteristics are summarised here.

Node type	Eurotech Aurora nodes
Number of nodes	128
Processors per node **Cores per node** **Threads per node**	2x Intel Xeon E5-2680 (Sandy Bridge) 2x 8 cores (@2.7 GHz) using hyper-threading: 32 threads/node
Total number of cores	2048 CPU cores
Memory per node	32 GB
Total memory	4 TB
Peak performance	44 Teraflops
Interconnect network	InfiniBand 4x QDR 40 Gbps
Filesystem	GPFS from JUST cluster exported through NFS to compute nodes BeeGFS as a parallel scratch.
Operating system	CentOS 6.5
Batch system	Torque + Moab
MPI runtime	ParaStation Global MPI
Compilers	Intel Professional Fortran, C + C++

Table 44: System information: DEEP Cluster

Figure 82: DEEP Cluster at JUELICH

B.3 Characteristics of miclogin

The miclogin is a test machine installed at JUELICH. It is used by the application developers in WP 6 for tests and performance measurements on Xeon Phis. Its main characteristics are summarised here.

Number of nodes	2 (4x Xeon Phi 7120, 2x Xeon Phi 3120)
Xeon Phi **7120**	61 cores (@ 1.2 GHz) 16 GB GDDR5 4 threads per core 512-bits-wide vector unit 1.171 Teraflops of peak performance
Xeon Phi **3120**	57 cores (@ 1.1 GHz) 6 GB GDDR5 4 threads per core 512-bits-wide vector unit 1.003 Teraflops of peak performance

Table 45: System information: miclogin

Figure 83: KNC

B.4 Characteristics of Marconi

The Marconi Cluster installed at CINECA was used by the Tk6.3 in WP6 for tests and performance measurements. Its main characteristics are summarised here.

Partition Name	A1	A2
Nodes	1512	3600
Processors/node	2x18 core Broadwell at 2.3 GHz	1x68 core Intel KNL 7250 at 1.40 GHz
RAM/node	128 GB	98GB DDR4 + 16 GB MCDRAM
Peak performance (system)	2 PFlop/s	11 PFlop/s
Internal Network	Intel® Omni-Path Architecture	

Table 46: Characteristics of the Marconi

B.5 Characteristics of GALILEO cluster

The GALILEO cluster installed at CINECA was used by the application developers of Task 6.3 in WP6 for tests and performance measurements. The main characteristics of the Xeon Phi nodes are summarised here.

Node type	NeXtScale
Number of nodes	384
Processors per node **Cores per processor** **Threads per node**	2 x Intel Haswell 2 x 8 cores (@2.4 GHz) using hyper-threading: 32 threads/node
Xeon Phi **cards per node**	2x Intel Xeon Phi 7120p
Xeon Phi **cores per card**	61 cores (@1.2 GHz)
Memory per Xeon Phi **card**	16GB GDDR5
Total number of cores	8256 CPU cores + 46848 Xeon Phi cores = 55104 cores
Memory per node	128 GB+32 GB = 160 GB
Total memory	61.44 TB?
Peak performance	1 Pflop
Interconnect network	InfiniBand 4x QDR switches
Filesystem	GPFS
Operating system	CentOS 7.0
Batch system	PBSPro
MPI runtime	IntelMPI 5.1.1 OpenMPI 1.8.5
Compilers	Intel (16.0.0) , GNU (4.9.2), PGI (15.3)

Table 47: System information: GALILEO cluster

Figure 84: GALILEO cluster at CINECA

B.6 Characteristics of SuperMUC

The SuperMUC cluster installed at BADW-LRZ was used by the application developers of Task 6.5 in WP6 for tests and performance measurements. It is comprised of different kind of nodes. Here just the thin nodes are described. Their main characteristics are summarised here.

Node type	IBM System x iDataPlex
Number of nodes	9216
Processors per node **Cores per node** **Threads per node**	2x Intel Xeon E5-2680 (Sandy Bridge) 2x 8 cores (@2.7 GHz) using hyper-threading: 32 threads/node
Total number of cores	147456 CPU cores
Memory per node	32 GB
Total memory	288 TB
Peak performance	3185 Teraflops
Interconnect network	InfiniBand 4x FDR-10 40 Gbps
Filesystem	GPFS
Operating system	SuSE Linux Enterprise Server
Batch system	LoadLeveler
MPI runtime	IBM MPI, Intel MPI
Compilers	Intel Professional Fortran, C + C++

Table 48: System information: SuperMUC

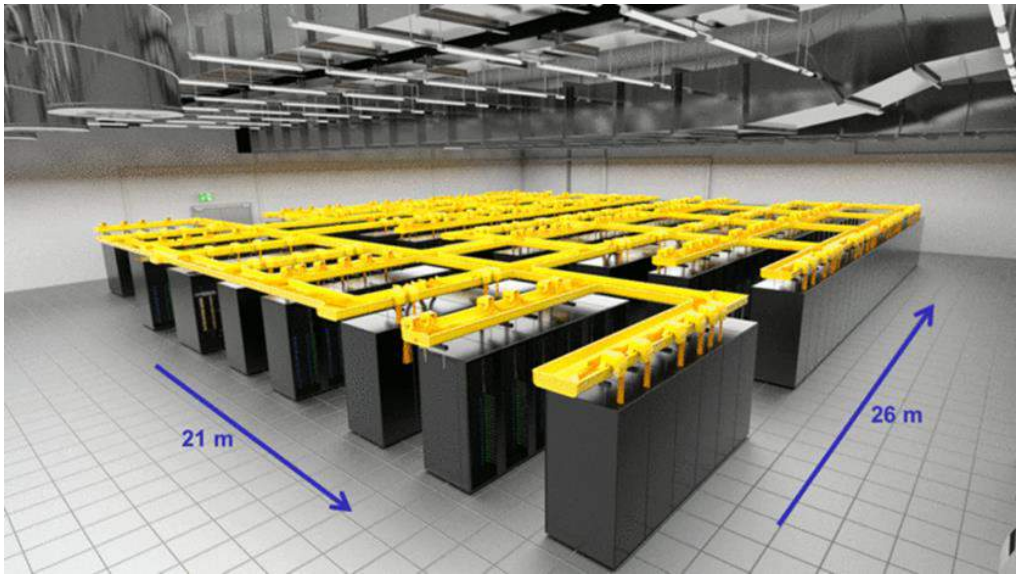


Figure 85: SuperMUC at BADW-LRZ

B.7 Characteristics of QPACE3

The QPACE3 cluster installed at JUELICH and administrated by UREG was used by the some of the application developers in WP6 for tests and performance measurements. Its main characteristics are summarised here.

Node type	Fujitsu PRIMERGY CX1640 M1
Number of nodes	352
Processors per node **Cores per node** **Threads per node**	1x Intel Xeon Phi 7210 (KNL) 1x 64 cores (@1.3 GHz) 256
Total number of cores	22528
Memory per node	48 GB
Total memory	16.896 TB RAM
Peak performance	937 Teraflops
Interconnect network	Intel Omni-Path 100 Gb/s
Filesystem	BeeGFS
Operating system	CentOS 7.3
Batch system	Slurm
MPI runtime	mpich, mvapich and openmpi
Compilers	gcc

Table 49: System information: QPACE3

List of Acronyms and Abbreviations

A

ADER-DG: Ader-Discontinuous-Galerkin (numerical scheme)
API: Application Programming Interface
Aurora: The name of Eurotech's cluster systems
AVX: Extensions to the x86 instruction set architecture for microprocessors from Intel and AMD
AVX2: Expands AVX to 256-bit support
AVX512: Expands AVX to 512-bit support

B

BADW-LRZ: Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften. Computing Centre, Garching, Germany
BeeGFS: The Fraunhofer Parallel Cluster File System (previously acronym FhGFS). A high-performance parallel file system to be adapted to the extended DEEP Architecture and optimised for the DEEP-ER Prototype.
BeeOND: BeeGFS-on-demand, parallel storage based on BeeGFS
BN: Booster Node (functional entity); refers to a self-booting KNL board (Node board architecture) including the NVM and NIC devices connected by PCI Express or a Brick (Brick architecture).
BNC: Booster Node Card is a physical instantiation of the BN
BoP: Board of Partners for the DEEP-ER Project
BSC: Barcelona Supercomputing Centre, Spain

C

CINECA: Consorzio Interuniversitario, Bologna, Italy
CN: Cluster Node (functional entity)
Coordinator: The contractual partner of the European Commission (EC) in the project
CP/RS: Checkpoint / Restart
CPU: Central Processing Unit

D

DEEP: Dynamical Exascale Entry Platform
DEEP-ER: DEEP Extended Reach: this project
DEEP-ER Booster: Booster part of the DEEP-ER Prototype, consisting of all Booster nodes and the NAM devices.
DEEP-ER Network: High performance network connecting the DEEP-ER BN, CN and NAM; to be selected off the shelf at the start of DEEP-ER
DEEP-ER Prototype: Demonstrator system for the extended DEEP Architecture, based on second generation Intel® Xeon Phi™ CPUs, connecting BN and CN via a

single, uniform network and introducing NVM and NAM resources for parallel I/O and multi-level checkpointing

DEEP Architecture: Functional architecture of DEEP (e.g. concept of an integrated Cluster-Booster Architecture), to be extended in the DEEP-ER Project

DEEP System: The prototype machine based on the DEEP Architecture developed and installed by the DEEP project

DGTD: Discontinuous Galerkin – Time Domain solver

DRAM: Dynamic Random Access Memory. Typically describes any form of high capacity volatile memory attached to a CPU

E

E10: Exascale 10. Parallel I/O software developed by a consortium of partners around the EOFS community. Partner Xyratex is responsible for the development needed for the DEEP-ER Project.

EC: European Commission

EU: European Union

Eurotech: Eurotech S.p.A., Amaro, Italy

Exascale: Computer systems or Applications, which are able to run with a performance above 10^{18} Floating point operations per second

EXTOLL: High speed interconnect technology for cluster computers developed by University of Heidelberg

F

FFT: Fast Fourier Transform

FhGFS: Fraunhofer Global File system, a high-performance parallel I/O system to be adapted to the extended DEEP Architecture and optimised for the DEEP-ER Prototype

FLOP: Floating point Operation

FP7: European Commission 7th Framework Programme.

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

FWI: Full Waveform Inversion

G

GERShWIN: New name for MAXW-DGTD the program code from Inria, means "discontinuous **G**al**ER**kin **S**olver for micro**W**ave **IN**teraction with biological tissues"

GFlop/s: Gigaflop, 10^9 Floating point operations per second

GitHub: Web-based Git repository hosting service

GMRES: Generalized Minimal RESidual method

GPFS: General Parallel File System (GPFS), a high-performance clustered file system developed by IBM

GPU: Graphics Processing Unit

GREEN500 list: Provides rankings of the 500 top most energy-efficient supercomputers in the world

H

H5hut: Library implementing several data models for particle-based simulations that encapsulates the complexity of parallel HDF5.

HDF5: Hierarchical Data Format: A set of file formats and libraries designed to store and organize large amounts of numerical data

Host Module: Self-booting computer board with an Intel KNL CPU, DDR4 memory and a PCI Express root complex. In the Brick Architecture, multiple host modules are connected to each other and NVMe and NIC devices by a PCI Express switch. In the Node Board architecture, a host module provides PCI Express slots to plug NVMe and NIC devices in.

HPC: High Performance Computing

HSW: Haswell, codename for a processor microarchitecture by Intel

HW: Hardware

I

IB: InfiniBand

IBM: International Business Machines Corporation

ICT: Information and Communication Technologies

IEEE: Institute of Electrical and Electronics Engineers

Intel: Intel Germany GmbH Feldkirchen,

IPC: Instructions Per Cycle

iPic3D: Programming code developed by the University of Leuven to simulate space weather

I/O: Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation

ITWM: Institut für Techno- und Wirtschaftsförderung. An Institute of the Fraunhofer Society

J

JUBE: Jülich Benchmarking Environment

JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany

K

KNC: Knights Corner, Code name of a processor based on the MIC architecture. Its commercial name is Intel® Xeon PhiTM.

KNL: Knights Landing, second generation of Intel® Xeon PhiTM

KU Leuven: Katholieke Universiteit Leuven, Belgium

L

libxsmm: Library for small matrix multiplications
LLNL: Lawrence Livermore National Laboratory
LOFAR: Low-Frequency Array, an instrument for performing radio astronomy built by ASTRON

M

MCDRAM: Multi-Channel DRAM, a high bandwidth on package memory on KNL
MIC: Intel Many Integrated Core architecture
MKL: Math Kernel Library
MPI: Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

N

NAM: Network Attached Memory, nodes connected by the DEEP-ER network to the DEEP-ER BN and CN providing shared memory buffers/caches, one of the extensions to the DEEP Architecture proposed by DEEP-ER
NUMA: Non-Uniform Memory Access
NVM: Non-Volatile Memory. Used to describe a physical technology or the use of such technology in a non-block-oriented way in a computer system
NVMe: Short form of NVM-Express
NVM-Express: An interface standard to attach NVM to a computer system. Based on PCI Express it also standardises high level HW interfaces like queues.

O

OmpSs: BSC's Superscalar (Ss) for OpenMP
OpenMP: Open Multi-Processing, Application programming interface that support multiplatform shared memory multiprocessing
OS: Operating System

P

ParaStation Consortium: Involved in research and development of solutions for high performance computing, especially for cluster computing
ParaStation MPI: Software for cluster management and control developed by ParTec
Paraver: Performance analysis tool developed by BSC
ParTec: ParTec Cluster Competence Center GmbH, Munich, Germany
PCI: Peripheral Component Interconnect, Computer bus for attaching hardware devices in a computer
PCIe: Short form of PCI Express

PCI Express: Peripheral Component Interconnect Express started as an option for a physical layer of PCI using high-performance serial communication. It is today's standard interface for communication with add-on cards and on-board devices, and makes inroads into coupling of host systems. PCI Express has taken over specifications of higher layers from the PCI baseline specification.

PETSc: A suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations

PFlop/s: Petaflop, 10^{15} Floating point operations per second

PFS: Parallel File System

PMT: Project Management Team of the DEEP-ER Project

POSIX: Portable Operating System Interface. A family of standards specified by the IEEE for maintaining compatibility between operating systems

PPF: Poly-Phase Filter

PR: Public Relations

Project Coordinator: Leading scientist coordinating and representing the DEEP-ER Project

PTC: Persistent Task-based Checkpoint

Q

QCD: Quantum Chromodynamics

QMC: Quantum Monte-Carlo

QPACE: QCD Parallel Computing Engine. Specialised supercomputer for QCD Parallel Computing

R

RAM: Random-Access Memory

S

Scalasca: Performance analysis tool developed by JUELICH and GRS

SCR: Scalable Checkpoint/Restart. A library from LLNL

SDV: Software Development Vehicle: a HW system to develop software in the time frame where the DEEP-ER Prototype is not yet available.

SIMD: Single Instruction Multiple Data

SIONlib: Parallel I/O library developed by Forschungszentrum Jülich

SISSA: International School of Advanced Studies, Trieste, Italy

SKA: Square Kilometre Array

SNB: Sandy Bridge, codename for a microarchitecture by Intel

SSD: Solid State Disk

SSE: SIMD instruction set extension to the x86 architecture

SVML: Intel's Short Vector Math Library

SW: Software

T

TBR: Task-based resiliency
Tier-0, Tier-1, …: Different classes of supercomputers ordered by their performance
TK: Task, Followed by a number, term to designate a task inside a Work Package of the DEEP-ER Project
TurboRVB: Quantum Monte Carlo Software for electronic structure calculations developed by SISSA

U

UREG: University of Regensburg, Germany

V

VMC: Variational Monte-Carlo
VML: Intel's Vector Math Library
VTune: Commercial application for software performance analysis

W

WAN: Wide Area Network
WP: Work Package

X

XML: Extensible Markup Language
x86: Family of instruction set architectures based on the Intel 8086 CPU

Bibliography

[1] K. Thust, “Results of I/O performance measurements,” in *Deliverable 4.5, DEEP-ER Project*, 2017.

[2] V. Beltrán, C. Clauss and A. Galonska, “Resiliency performance measurements,” in *Deliverable 5.4, DEEP-ER Project*, 2017.

[3] A. Wolf and A. Zitz, “Applications ported to the SDV,” in *Deliverable 6.2, DEEP-ER Project*, 2016.

[4] J. Schmidt, “Network Attached Memory (NAM) (updated),” in *Deliverable 3.4, DEEP-ER Project*, 2016 (updated in 2017).

[5] D. Alvarez, “Application code analysis,” in *Deliverable 6.1, DEEP-ER Project*, 2014.

[6] M. Cintra, “NVM Assessment,” in *Deliverable 3.3, DEEP-ER Project*, 2016.

[7] M. Dumbser and M. Käser, “An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes II: The Three-Dimensional Isotropic Case,” *Geophysical Journal International,* vol. 167(1), pp. 319-336, 2006.

[8] C. Pelties, J. De la Puente, J. Ampuero, G. Brietzke and M. Käser, “Three-Dimensional Dynamic Rupture Simulation with a High-order Discontinuous Galerkin Method on Unstructured Tetrahedral Meshes,” *J. Geophys. Res. - Solid Earth,* vol. 117 B2, 2012.

[9] A. Breuer, A. Heinecke, M. Bader and C. Pelties, “Accelerating SeisSol by Generating Vectorized Code for Sparse Matrix Operators,” in *Parallel Computing - Accelerating Computational Science and Engineering (CSE)*, vol. 25, I. Press, Ed., 2014, pp. 347--356.

[10] [Online]. Available: https://github.com/LLNL/scr.

[11] Karypis Lab, “Family of Graph and Hypergraph Partitioning Software”.

[12] A. Breuer, High Performance Earthquake Simulations, München: Technische Universität München, 2015.

[13] G. Congiu, “MPIWRAP,” 2015.

[14] A. Heinecke, H. Pabst and G. Henry, “LIBXSMM: A High Performance Library for Small Matrix,” in *Supercomputing Conference Poster Presentation*, 2015.

[15] J. Romein, “A Comparison of Accelerator Architectures for Radio-Astronomical Signal-Processing Algorithms,” in *International Conference for Parallel Processing (IPCC)*, 2016.

[16] B. Veenboer, M. Petschow and J. Romein, “Image Domain Gridding on Graphics Processors,” in *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.

[17] A. R. Offringa et al, “WSClean: an implementation of a fast, generic wide-field imager for

radio astronomy," in *Mon. Not. R. Astron. Soc, vol. 444*, 2014.

[18] J. W. Romein et al., "The LOFAR correlator: implementation and performance analysis,," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[19] A. R. Offringa et al., "Post-correlation radio frequency interference classification methods," in *Monthly Notices of the Royal Astronomical Society, Vol. 405 (1)*, 2010.

[20] U. Rau at al., "Advances in Calibration and Imaging Techniques in Radio Interferometry," in *Proceedings of the IEEE, Vol. 97, Issue 8*, 2009.

[21] R. Jongerius et al., "An End-to-End Computing Model for the Square Kilometre Array," in *Computer Vol. 47 (9)*, 2014.

[22] G. van Diepen, "Casacore Table Data System and its use in the MeasurementSet," in *Astronomy and Computing Vol. 12*, 2015, pp. pp. 174-180.

[23] R. J. Hanisch et al., "Definition of the Flexible Image Transport System (FITS)," in *Astronomy & Astrophysics 376*, 2001, pp. pp. 359-380.

[24] V. Beltran, C. Clauss and A. Galonska, "Resiliency performance measurements," in *Deliverable 5.4, DEEP-ER Project*, 2017.

[25] "https://github.com/pjgeorg/pMR," [Online].

[26] Intel Developer Zone, "Getting Started with Intel® VTune™ Amplifier," [Online]. Available: https://software.intel.com/en-us/node/544004.

[27] Intel Developer Zone, "Getting Started with Intel® Advisor," [Online]. Available: https://software.intel.com/en-us/get-started-with-advisor.

[28] Scalasca, "Scalasca 2.x series," [Online]. Available: http://www.scalasca.org/software/scalasca-2.x/documentation.html.

[29] BSC, "Performance Analysis Tools: Details and Intelligence," [Online]. Available: https://tools.bsc.es/. [Accessed 31 January 2017].

[30] Barcelona Supercomputing Center, "OmpSs User Guide - Offloading architecture," [Online]. Available: https://pm.bsc.es/ompss-docs/user-guide/run-programs-archs-offload.html.

[31] [Online]. Available: http://www.mpich.org/static/docs/v3.2/www3/MPI_Comm_spawn.html.

[32] D. Alvarez, "Final pilot applications report," in *Deliverable 8.3, DEEP Project*, 2015.

[33] Intel Developer Zone, "Intel Guide for Developing Multithreaded Applications," [Online]. Available: https://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications.

[34] Intel, "Intel® Xeon Phi™x200 processor: High-Bandwidthmemory," 2016.

[35] Intel, "Best Practices for Vectorization: Getting ready for Intel® Advanced Vector

Extensions 512 (Intel® AVX-512),” 2016.

[36] Intel Developer Zone, “Optimization and Performance Tuning for Intel® Xeon Phi™ Coprocessors - Part 1: Optimization Essentials,” [Online]. Available: https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization.

[37] Colfax International, “Performance Optimization for Intel® Xeon Phi™ x200 Product Family: Video,” 29 09 2016. [Online]. Available: http://colfaxresearch.com/how-knl/.

[38] Fraunhofer-ITWM, “BeeGFS The parallel cluster file system,” [Online]. Available: http://www.beegfs.com.

[39] “SIONlib 1.7.0 Scalable I/O library for parallel access to task-local files,” [Online]. Available: https://apps.fz-juelich.de/jsc/sionlib/docu/index.html.

[40] P. B. Meghan McClelland, “EIOW - Exa-scale I/O workgroup (exascale10),” 2013.

[41] Fraunhofer - ITWM, “BeeOND: BeeGFS On Demand,” [Online]. Available: http://www.beegfs.com/wiki/BeeOND.

[42] V. Beltran and J. Morillo, “Checkpointing architecture design,” in *Deliverable 5.1, DEEP-ER Project*, 2014.

[43] V. Beltran and J. Morillo, “Resiliency software,” in *Deliverable 5.3, DEEP-ER Project*, 2016.

[44] J. Schmidt, “Report on projections and improvements for the DEEP/DEEP-ER concept,” in *Deliverable 7.2, DEEP-ER Project*, 2017.