SEVENTH FRAMEWORK PROGRAMME

FP7-ICT-2011-7

DEEP

Dynamical Exascale Entry Platform

Grant Agreement Number: 287530

D8.3
Final pilot applications report

Approved
Project and Deliverable Information Sheet

<table>
<thead>
<tr>
<th>DEEP Project</th>
<th>Project Ref. №:</th>
<th>287530</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Title:</td>
<td>Dynamical Exascale Entry Platform</td>
<td></td>
</tr>
<tr>
<td>Project Web Site:</td>
<td><a href="http://www.deep-project.eu">http://www.deep-project.eu</a></td>
<td></td>
</tr>
<tr>
<td>Deliverable ID:</td>
<td>D8.3</td>
<td></td>
</tr>
<tr>
<td>Deliverable Nature:</td>
<td>Report</td>
<td></td>
</tr>
<tr>
<td>Deliverable Level:</td>
<td>PU*</td>
<td></td>
</tr>
<tr>
<td>Contractual Date of Delivery:</td>
<td>31 / August / 2015</td>
<td></td>
</tr>
<tr>
<td>Actual Date of Delivery:</td>
<td>31 / August / 2015</td>
<td></td>
</tr>
<tr>
<td>EC Project Officer:</td>
<td>Luis Carlos Busquets Pérez</td>
<td></td>
</tr>
</tbody>
</table>

* - The dissemination level 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

<table>
<thead>
<tr>
<th>Document</th>
<th>Title:</th>
<th>Final pilot applications report</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID:</td>
<td>D8.3</td>
</tr>
<tr>
<td></td>
<td>Version:</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>Status:</td>
<td>Approved</td>
</tr>
<tr>
<td></td>
<td>Available at:</td>
<td><a href="http://www.deep-project.eu">http://www.deep-project.eu</a></td>
</tr>
<tr>
<td></td>
<td>Software Tool:</td>
<td>Microsoft Word</td>
</tr>
<tr>
<td></td>
<td>File(s):</td>
<td>DEEP_D8.3_Final_pilot_applications report_v2.0-ECapproved</td>
</tr>
<tr>
<td>Authorship</td>
<td>Written by:</td>
<td>D. Alvarez, JUELICH</td>
</tr>
<tr>
<td></td>
<td>Contributors:</td>
<td>H. Merx, CYI</td>
</tr>
<tr>
<td></td>
<td>A. Wolf, JSC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>P. Kumbhar, EPFL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A. Emerson, CINECA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>J. Amaya, KULeuven</td>
<td></td>
</tr>
<tr>
<td></td>
<td>G. Staffelbach, CERFACS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>M. Tchiboukdjian, CGGVS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reviewed by:</td>
<td>E. Suarez, JUELICH</td>
</tr>
<tr>
<td></td>
<td>Approved by:</td>
<td>BoP/PMT</td>
</tr>
</tbody>
</table>

Document Status Sheet

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>31/August/2015</td>
<td>Final</td>
<td>EC submission</td>
</tr>
<tr>
<td>2.0</td>
<td>02/October/2015</td>
<td>Approved</td>
<td>EC approved</td>
</tr>
</tbody>
</table>
Document Keywords

| Keywords: | DEEP, HPC, Exascale, Pilot Applications |

Copyright notices

© 2011 -2015 DEEP Consortium Partners. All rights reserved. This document is a project document of the DEEP project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the DEEP partners, except as mandated by the European Commission contract 287530 for reviewing and dissemination purposes. All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.
Table of Contents

Project and Deliverable Information Sheet ................................................................. i
Document Control Sheet............................................................................................. i
Document Status Sheet ............................................................................................... i
Document Keywords ................................................................................................. ii
Table of Contents ........................................................................................................ iii
List of Figures ................................................................................................................ v
List of Tables ................................................................................................................ ix
Executive Summary ....................................................................................................... 1

1. Introduction .................................................................................................................. 2
2. Task 8.1: Detailed brain simulation (Task leader: EPFL) ........................................ 3
   2.1. Overview of application structure ..................................................................... 3
   2.2. Code modifications during the project ............................................................. 8
   2.3. Benchmarking ................................................................................................... 14
   2.4. Intranode metrics ............................................................................................ 20
   2.5. Conclusions ...................................................................................................... 21
3. Task 8.2: Space weather simulation (Task leader: KULeuven) .................................. 22
   3.1. Overview of application structure ..................................................................... 22
   3.2. Code modifications during the project ............................................................. 27
   3.3. Benchmarking ................................................................................................... 33
   3.4. Analysis of time evolution of different phases .................................................. 38
   3.5. Analysis of metrics for WP9 ............................................................................ 41
   3.6. Conclusions ...................................................................................................... 46
4. Task 8.3: Climate simulation (Task leader: CYI) ....................................................... 48
   4.1. Overview of application structure ..................................................................... 48
   4.2. Code modifications during the project ............................................................. 54
   4.3. Benchmarking ................................................................................................... 60
   4.4. Analysis of theoretical improved performance ................................................ 62
   4.5. Analysis of metrics for WP9 ............................................................................ 64
   4.6. Conclusions ...................................................................................................... 68
5. Task 8.4: Computational fluid engineering (Task leader: CERFACS) ...................... 70
   5.1. Overview of application structure ..................................................................... 70
   5.2. Code modifications during the project ............................................................. 74
   5.3. Benchmarking ................................................................................................... 83
   5.4. Analysis of metrics for WP9 ............................................................................ 92
   5.5. Conclusions ...................................................................................................... 92
6. Task 8.5: High temperature superconductivity (Task leader: CINECA) .................... 93
   6.1. Overview of application structure ..................................................................... 93
   6.2. Code modifications during the project ............................................................. 96
   6.3. Benchmarking ................................................................................................... 98
   6.4. Analysis of offload performance ..................................................................... 102
   6.5. Conclusions ...................................................................................................... 103
7. Task 8.6: Seismic imaging (Task leader: CGGVS) ..................................................... 104
7.1. Overview of application structure .............................................................................. 104
7.2. Code modifications during the project ....................................................................... 106
7.3. Benchmarking ........................................................................................................... 110
7.4. Analysis of metrics for WP9 ..................................................................................... 112
7.1. Conclusions ............................................................................................................... 114
8. Global conclusions ......................................................................................................... 116
9. Annex: Best practices and recommendation for improvements ............................... 118
  9.1. Leveraging DEEP ..................................................................................................... 118
  9.2. Recommendation for improvements ....................................................................... 120
Bibliography ...................................................................................................................... 123
List of Acronyms and Abbreviations ................................................................................ 124
List of Figures

Figure 1: Phases of CoreBluron. Green phases run on the Cluster, blue phases run on the Booster (EPFL) ................................................................. 4
Figure 2: Vampir trace view showing various phases of the CoreBluron simulator (EPFL) ............... 5
Figure 3: Structure of compute bound kernels (EPFL) ................................................................. 6
Figure 4: Structure of memory bound kernels (EPFL) ......................................................................... 6
Figure 5: Impact on run time of each phase of CoreBluron, when running on MareNostrum 3, using 2 MICs per host (EPFL) ................................................................. 7
Figure 6: Percentage of run time of each phase of CoreBluron, when running on MareNostrum 3, using 2 MICs per host (EPFL) ......................................................................... 8
Figure 7: Performance of the original MPI code vs. the new OpenMP implementation in a Xeon Phi 7120 (EPFL) ........................................................................................................ 9
Figure 8: Parallel efficiency of the original MPI code vs. the new OpenMP implementation in a Xeon Phi 7120 (EPFL) ........................................................................................................ 9
Figure 9: A] shows how mechanisms of neuron cells are stored. B] shows the original AoS layout of individual mechanism’s properties, which disables the vectorisation. C] shows the new SoA layout where mechanism properties are grouped together in order to enable compiler auto-vectorisation (EPFL) ........................................................................................................ 10
Figure 10: Vectorisation performance in a Xeon Phi 7120, in the main compute phase (EPFL) ............ 11
Figure 11: Vectorisation improvement in a Xeon Phi 7120, in the main compute phase (EPFL) .......... 11
Figure 12: Representation of complex, non byte-copyable data structure of CoreBluron (EPFL) .......... 12
Figure 13: Implementation of the reverse I/O offload in CoreBluron (EPFL) ........................................ 13
Figure 14: Example of a kernel written in neuron DSL (EPFL) ........................................................... 13
Figure 15: Performance of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL) .... 15
Figure 16: Parallel efficiency of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL) ........................................................................................................ 15
Figure 17: Performance of CoreBluron (computing + I/O) while running natively and using offload mode (EPFL) ........................................................................................................ 16
Figure 18: Performance of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL) .... 16
Figure 19: Parallel efficiency of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL) ........................................................................................................ 17
Figure 20: Performance of the CoreBluron compute phase on BlueGene/Q platforms (EPFL) ........... 18
Figure 21: Parallel efficiency of the CoreBluron on BlueGene/Q platforms (EPFL) ................................ 18
Figure 22: Performance of the CoreBluron compute phase in cluster platforms. The Xeon Phi experiments use I/O offload (EPFL) ................................................................. 19
Figure 23: Parallel efficiency of the CoreBluron compute phase in cluster platforms. The Xeon Phi experiments use I/O offload (EPFL) ................................................................. 19
Figure 24: Phases of iPc3D. Green phases run on the Cluster, blue phases run on the Booster (KULeuven) ........................................................................................................ 23
Figure 25 Speedup of the original version of the iPc3D code. Simulations performed in the Curie supercomputer using Intel Xeon E5 2680 processors (KULeuven) .................................................................... 24
Figure 26: Impact on runtime of each phase of iPc3D without offload, when running on the DEEP cluster (KU Leuven) ........................................................................................................ 26
Figure 27: Percentage of runtime of each phase of iPc3D without offload, when running on the DEEP cluster (KULeuven) ........................................................................................................ 27
Figure 28: Tests performed on the Xeon Phi architecture. Old: MPI only version. New: MPI+OpenMP version (KULeuven) ........................................................................................................ 28
Figure 29: Method of integration of OmpSs in the main code (KULeuven) ......................................... 31
Figure 30 Offloading using the MPI_Comm_Spawn method in the main section of the code (KULeuven) ........................................................................................................ 32
Figure 31: Parallel efficiency of iPc3D on Stampede and MareNostrum 3 (KULeuven) ................... 34
Figure 32: Performance of iPc3D in Stampede, using multiple modes (KULeuven) ......................... 35
Figure 33: Parallel efficiency of iPc3D in Stampede, using multiple modes (KULeuven) ................. 36
Figure 34: Performance of current version of iP PiC3D vs. old version of iP PiC3D, in DEEP (KULeuven) .......................................................... 37
Figure 35: Parallel efficiency of current version of iP PiC3D vs. old version of iP PiC3D, in DEEP (KULeuven) .......................................................... 37
Figure 36: Performance ratio between the old and the new versions of iP PiC3D (KULeuven) .......................................................... 38
Figure 37: Decomposition of the weak scaling results obtained in Stampede for the three offload methods. In columns, the three phases of the code. In rows, the three cases run. Blue bars: computing time, red bars: communication times (KULeuven) .......................................................... 40
Figure 38: Variation of number of MPI calls with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (KULeuven) .......................................................... 42
Figure 39: Variation of time spent on MPI with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (KULeuven) .......................................................... 43
Figure 40: Variation of useful time with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (KULeuven) .......................................................... 44
Figure 41: Variation of number of cycles with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (KULeuven) .......................................................... 45
Figure 42: Variation of number of instructions with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (KULeuven) .......................................................... 46
Figure 43: Phases of EMAC. Green phases run on the Cluster, blue phases run on the Booster (CYI) .......................................................... 49
Figure 44: The ECHAM main application loop. MESSy replaces and enhances the grid point calculations marked in yellow (CYI) .......................................................... 49
Figure 45: Paraver trace of major processor usage of one time step (CYI) .......................................................... 50
Figure 46: MECCA execution time analysed with Scalasca (CYI) .......................................................... 51
Figure 47: Wall time for one simulated day versus the number of nodes on JUDGE, at the beginning of the project (CYI) .......................................................... 52
Figure 48: Maximal MECCA kernel execution wall-time in microseconds. The adaptive time-step integrator shows a non-uniform run time caused by stratospheric photochemistry and natural and anthropogenic emissions (CYI) .......................................................... 53
Figure 49: Impact on run time of each phase of EMAC, when running on MareNostrum 3 (CYI) .......................................................... 54
Figure 50: Percentage of run time of each phase of EMAC, when running on MareNostrum 3 (CYI) .......................................................... 54
Figure 51: Performance of OpenMP threading in the DEEP Cluster (CYI) .......................................................... 56
Figure 52: Performance of OpenMP threading in the DEEP Cluster (CYI) .......................................................... 56
Figure 53: Performance of the vectorisation of MESSy in a Xeon E5-2680 (CYI) .......................................................... 57
Figure 54: Performance of the two-stage offload in MareNostrum 3 (CYI) .......................................................... 62
Figure 55: Parallel efficiency of the two-stage offload in MareNostrum 3 (CYI) .......................................................... 62
Figure 56: Time per simulated day in DEEP using a pure MPI approach, and a theoretical performance with offloading to Xeon Phi, based on the metrics collected in MareNostrum 3 (CYI) .......................................................... 64
Figure 57: Variation of number of MPI calls with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI) .......................................................... 65
Figure 58: Variation of time spent on MPI with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI) .......................................................... 66
Figure 59: Variation of useful time with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI) .......................................................... 66
Figure 60: Variation of number of cycles with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI) .......................................................... 67
Figure 61: Variation number of instructions with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI) .......................................................... 68
Figure 62: Phases of AVBP. Green phases run on the Cluster, blue phases run on the Booster (CERFACS) .......................................................... 71
Figure 63: Impact on run time of each phase of AVBP 6.2, when running on the DEEP Cluster (CERFACS) .......................................................... 73
Figure 64: Percentage of run time of each phase of AVBP 6.2, when running on the DEEP Cluster (CERFACS) .......................................................... 73
Figure 65: Impact on run time of each phase of AVBP 7.0, when running on the DEEP Cluster (CERFACS) .......................................................... 74
Figure 66: Percentage of run time of each phase of AVBP 7.0, when running on the DEEP Cluster (CERFACS) .................................................................................................................. 74
Figure 67: MPI domain decomposition structure (CERFACS) .................................................................................................................. 75
Figure 68: Sub-partition decomposition structure for tasks (CERFACS) .................................................................................................. 76
Figure 69: Task execution diagram with OmpSs (CERFACS) .................................................................................................................. 76
Figure 70: Performance of OmpSs threading for the computational phase in a Xeon Phi 5110 (CERFACS) .................................................................................................................. 77
Figure 71: Parallel efficiency of OmpSs threading for the computational phase in a Xeon Phi 5110 (CERFACS) .................................................................................................................. 78
Figure 72: Sample code of the gradient routine. Specialized version for 3 dimensional elements with 4 vertices (CERFACS) .................................................................................................................. 79
Figure 73: Sample code of the gather routine. Specialized version with forced inline directives (CERFACS) .................................................................................................................. 79
Figure 74: Sample code of the gather operation. Aligned SIMD execution kernel (CERFACS) .................................................................................................................. 79
Figure 75: Vectorisation performance in a Xeon Phi 7120 (CERFACS) .................................................................................................................. 80
Figure 76: Vectorisation improvement in a Xeon Phi 7120 (CERFACS) .................................................................................................................. 80
Figure 77: Performance of the standard initialisation and the recycling initialisation on the DEEP cluster (CERFACS) .................................................................................................................. 82
Figure 78: Performance improvement of the initialisation phase on thanks to the recycling initialisation, measured on the DEEP cluster (CERFACS) .................................................................................................................. 82
Figure 79: Performance of AVBP in Xeon Phi: SIMPLE test case. Number of Cores = 4 MPI x N threads (CERFACS) .................................................................................................................. 84
Figure 80: Performance of AVBP in Xeon Phi: NASAC3X test case. Number of cores = 4 MPI x N threads (CERFACS) .................................................................................................................. 85
Figure 81: Performance of AVBP in Xeon Phi: MERCATO test case. Number of cores = 4 MPI x N threads (CERFACS) .................................................................................................................. 85
Figure 82: Performance of AVBP in DEEP and BlueGene/Q platforms (CERFACS) .................................................................................................................. 87
Figure 83: Parallel efficiency of AVBP in DEEP and BlueGene/Q platforms (CERFACS) .................................................................................................................. 87
Figure 84: Performance of AVBP for BlueGene/Q on the refined Simple test case (CERFACS) .................................................................................................................. 88
Figure 85: Parallel efficiency of AVBP for BlueGene/Q on the refined Simple test case (CERFACS) .................................................................................................................. 88
Figure 86: Performance of current version of AVBP (without offloading) vs. old version of AVBP, in DEEP for the Simple test case (CERFACS) .................................................................................................................. 89
Figure 87: Parallel efficiency of current version of AVBP (without offloading) vs. old version of AVBP, in DEEP for the Simple test case (CERFACS) .................................................................................................................. 90
Figure 88: Performance of current version of AVBP (without offloading) vs. old version of AVBP, in DEEP for the NASA test case (CERFACS) .................................................................................................................. 90
Figure 89: Parallel efficiency of current version of AVBP (without offloading) vs. old version of AVBP, in DEEP for the NASA test case (CERFACS) .................................................................................................................. 91
Figure 90: Performance of current version of AVBP vs. old version of AVBP (without offloading), in DEEP for the MERCATO test case (CERFACS) .................................................................................................................. 91
Figure 91: Parallel efficiency of current version of AVBP vs. old version of AVBP (without offloading), in DEEP for the MERCATO test case (CERFACS) .................................................................................................................. 91
Figure 92: Main Phases of TurboRVB. Green phases run on the Cluster, blue phases run on the Booster (CINECA) .................................................................................................................. 94
Figure 93: Strong scaling behaviour on the Fermi (BG/Q) system (CINECA) .................................................................................................................. 95
Figure 94: Impact on run time of each phase of TurboRVB, when running on BlueGene/Q with 16384 walkers (CINECA) .................................................................................................................. 95
Figure 95: Percentage of run time of each phase of TurboRVB, when running on BlueGene/Q with 16384 walkers (CINECA) .................................................................................................................. 96
Figure 96: Performance of TurboRVB in Xeon Phis in MareNostrum 3 (CINECA) .................................................................................................................. 99
Figure 97: Parallel efficiency of TurboRVB in Xeon Phis in MareNostrum 3 (CINECA) .................................................................................................................. 100
Figure 98: Weak scaling as a function of number of Xeon Phis, using the offload code (CINECA) .................................................................................................................. 100
Figure 99: Strong scaling as a function of number of Xeon Phis, using the offload code (CINECA) .................................................................................................................. 101
Figure 100: Performance of TurboRVB in Xeon processors in MareNostrum 3 (CINECA) .................................................................................................................. 102
Figure 101: Parallel efficiency of TurboRVB in Xeon processors in MareNostrum 3 (CINECA) .................................................................................................................. 102
Figure 102: Branching time in Xeon processors, using the offload and the regular versions of TurboRVB (CINECA) ................................................................. 103
Figure 103: Phases of RTM (CGGVS) ................................................................. 105
Figure 104: Percentage of run time of each phase of RTM, when running on the CGG GPU cluster. Imaging and Finalization phases are not included since these phases cannot be run without I/Os. (CGGVS) ................................................................................. 106
Figure 105: Performance of the main wave propagation kernels in a Xeon Phi 7120 (CGGVS) .... 107
Figure 106: Parallel efficiency of the main wave propagation kernels in a Xeon Phi 7120 (CGGVS) ........................................................................................................ 108
Figure 107: Vectorisation performance in a Xeon Phi 7120. Kernels 1 to 5 have no optimized form and kernels 6 to 8 have no original form (CGGVS) ................................................................. 109
Figure 108: Vectorisation improvement in a Xeon Phi 7120. Kernels 1 to 5 have no optimized form and kernels 6 to 8 have no original form (CGGVS) ................................................................................. 109
Figure 109: Parallel efficiency of RTM in Xeon Phi platforms (CGGVS) ........................................ 111
Figure 110: Parallel efficiency of RTM in MareNostrum3 and GPU platforms. Both parallel efficiencies are basically equal (CGGVS) ................................................................................. 112
Figure 111: Ratio of RTM performance between Xeon Phi and GPU platforms (CGGVS) ........ 112
Figure 112: Variation of the number of MPI calls with the number of nodes, per phase. Average across ranks. (CGGVS) ................................................................................. 113
Figure 113: Variation of the fraction of time spent in MPI calls with the number of nodes, per phase (CGGVS) ................................................................................. 114
List of Tables

Table 1: Experiments setup for the scalability considerations subsection (EPFL) ........................................... 7
Table 2: Summary of main changes done during the project (EPFL) ................................................................. 13
Table 3: Experiments setup for the benchmarking subsection (EPFL) ............................................................ 14
Table 4: System setup details for Xeon Phi platforms (EPFL) ........................................................................ 14
Table 5: System setup details for BlueGene/Q (EPFL) ...................................................................................... 17
Table 6: System setup details for MareNostrum 3 without Xeon Phi (EPFL) ................................................. 18
Table 7: CPI and vectorisation intensity on a set of DSL generated kernels on Xeon Phi 7120 (EPFL) ........... 20
Table 8: Input parameters used for the benchmarking of the code iPic3D (KULeuven) ................................. 25
Table 9: Experiments setup for the scalability considerations subsection (KULeuven) ................................... 25
Table 10: Runs performed in one node, with and without intrinsic calls (KULeuven) ........................................ 30
Table 11: Summary of main changes done during the project (KULeuven) .................................................... 32
Table 12: Experiments setup for the benchmarking subsection (KULeuven) .................................................... 33
Table 13: System setup details for Stampede (KULeuven) .............................................................................. 33
Table 14: System setup details for MareNostrum3 (KULeuven) ...................................................................... 33
Table 15: System setup details for DEEP Cluster (KULeuven) ....................................................................... 36
Table 16: Experiments setup for the analysis subsection (KULeuven) ............................................................. 41
Table 17: System setup details for the analysis done in Stampede (KULeuven) .............................................. 41
Table 18: Experiments setup for the scalability considerations subsection (CYI) .......................................... 51
Table 19: System setup details for the analysis done in the JUDGE system (CYI) ............................................ 52
Table 20: Summary of main changes done during the project (CYI) .............................................................. 59
Table 21: System setup details for Xeon Phi platforms (CYI) ....................................................................... 61
Table 22: Experiments setup for the analysis subsection (CYI) .................................................................... 62
Table 23: System setup details for the analysis done in the DEEP system (CYI) .............................................. 63
Table 24: Experiments setup for the analysis of metrics for WP9 subsection (CYI) ........................................ 64
Table 25: System setup details for the analysis done in MareNostrum 3 (CYI) .............................................. 64
Table 26: Experiments setup for the scalability considerations subsection (CERFACS) .............................. 72
Table 27: Summary of main changes done during the project (CERFACS) ................................................... 82
Table 28: Experiments setup for the benchmarking subsection (CERFACS) ................................................ 83
Table 29: System setup details for Xeon Phi platforms (CERFACS) .............................................................. 83
Table 30: System setup details for BlueGene/Q and DEEP (CERFACS) ......................................................... 86
Table 31: System setup details for DEEP Cluster (CERFACS) ...................................................................... 89
Table 32: Experiments setup for the scalability considerations subsection (CINECA) .................................. 94
Table 33: Summary of main changes done during the project (CINECA) ..................................................... 98
Table 34: Experiments setup for the benchmarking subsection (CINECA) .................................................... 98
Table 35: System setup details for Xeon Phi platforms (CINECA) .............................................................. 99
Table 36: Experiments setup for the regular processor comparison (CINECA) ............................................. 101
Table 37: Experiments setup for the scalability considerations subsection (CGGVS) .............................. 106
Table 38: Summary of main changes done during the project (CGGVS) .................................................... 110
Table 39: System setup details for Xeon Phi platforms (CGGVS) .............................................................. 110
Table 40: System setup details for the GPU cluster (CGGVS) ................................................................. 111
Executive Summary

The main target of the last deliverable of Work Package 8 is to summarise the results obtained by the application partners when running their applications in the DEEP platform. In order to do that, all the partners made the necessary changes to their application to successfully run simultaneously in both parts of the system, that is, Cluster, and Booster. The way applications can use the system vary wildly. However, what all of them have in common is that, in theory, they are ready to benefit from the system.

Despite the readiness of the applications, the results are lacking. The reasons are various, and the difficulties encountered are both human and technical. Some partners were struck by lack of personnel, as the technical staff left some institutions without time to find appropriate replacements. In these cases the support team moved in to try to achieve the best possible outcome. Technical difficulties proved to be more difficult to solve. The lack of a working Cluster/Booster platform could be circumvented simulating it in existing clusters with Xeon Phi accelerators. However, there are a number of software interactions and bugs that were not foreseen:

- Xeon Phi with DAPL support for InfiniBand is crash prone in many platforms.
- MPI_Comm_spawn is not as mature as other parts of the MPI runtime, with related bugs present in MPICH that cascade down to other MPI implementations based on MPICH.
- Mercurium for Fortran was developed during the project and bugs were frequent, slowing down the development of the applications.
- New compiler/runtime versions added new bugs that required fixing, in some cases taking a very significant amount of time.

These and other reasons limited the capacity of WP8 to produce satisfactory results. Nevertheless, the impact of the changes done during the project is remarkable for many applications. They enjoy now a much better scalability and performance than before and these improvements take them one step closer to Exascale.

First results obtained with the iPiC3D application from KULeuven on the DEEP System -- distributed over Cluster and Booster nodes -- have been obtained and are described in the progress report (Deliverable D1.10). The rest of the results collected at this point are presented in this document. Many partners have acknowledged that the work will continue after the end of the project. They expressed their desire to further explore the concept, once the system is running stable enough for assessing empirically the benefits of the architecture.
1. Introduction

The purpose of this document is to explain the changes done to the applications by the WP8 partners, as well as the results obtained. In order to do that, the description of all the applications follow the same structure, reflected here:

- Overview of application structure
  - Phases
  - Dominant factors
  - Scalability considerations
- Code modifications during the duration of the project
  - Threading
  - Vectorisation
  - Cluster/Booster division
  - Other optimisations
  - Summary of changes
- Benchmarking
  - Xeon Phi platforms
  - Comparison with other architectures (BlueGene/Q or GPU clusters)
  - Comparison with cluster architectures without coprocessors (with and without offload to regular CPUs)
- Analysis of extended metrics
- Conclusions

Since during the benchmarking phase the DEEP Booster and the ASIC evaluator were not available to the application users, the Xeon Phi experiments were done mainly in MareNostrum 3, which has 42 nodes with 2 Xeon Phi 5110 each. Other benchmarking platforms are JUQUEEN—a BlueGene/Q system—, and the DEEP Cluster.

The deliverable initially aimed at covering an extensive set of benchmark results, comparing and analysing all the possible combinations:

- Hardware: DEEP Booster vs. ASIC evaluator vs. off-the-shelf Xeon Phi cluster vs. BlueGene/Q vs. standard cluster. For this reason the X axis units in most figures is number of nodes, since this is an easier to unit to compare platforms and processors with very different number of cores and HW threads.
- Software: Initial application version vs. optimised version vs. DEEP-ready version.

However, hardware and software issues, frequently outside of the control of WP8, resulted in a smaller set of results than desired.

The present report describes the results collected until the official deliverable submission deadline, on 31st August 2015. Any results achieved beyond that date will be added to the slides presented on the final project review meeting.
2. Task 8.1: Detailed brain simulation (Task leader: EPFL)

The Blue Brain Project is the first comprehensive approach to simulation-based research in neuroscience. The detailed multi-compartment conductance-based simulations of cable equations and phenomenological ion channels (Hodgkin Huxley) require solving a linear system and thousands of differential equations per nerve cell. Taking full advantage of DEEP's hybrid infrastructure requires carefully analysing the application workflow in order to properly map the computing needs of different compute kernels to the most appropriate part of the system. To support such a paradigm shift and allow fast prototyping, as part of DEEP project we have been contributing in the development of core modelling engine factored out from Bluron simulator, called CoreBluron, and optimizing its data structures for MIC coprocessors.

Using performance analysis tools, following performance classification leading to critical design decisions has been obtained: the kernels supporting the resolution of the cable equations – using multi-compartment conductance-based modelling and phenomenological ion channels (Hodgkin Huxley) leading to the setup and resolution of large sparse linear algebraic system – were ported and optimized onto the Booster architecture. The parts of the application workflow responsible for model initialization and the cell compartment level report generation were offloaded onto the Cluster part of the system, where I/O is more efficient.

Getting good performance on the Booster is challenging considering hundreds of small kernels auto-generated from the DSL converter. During this work, SoA (Structure of Array) memory layout support has been added to CoreBluron and the DSL converter to enable compiler vectorization of application kernels. We used the DEEP software stack, which includes MPI's dynamic process management interface, to avoid rewriting of whole application for the Cluster/Booster division. This allows us to easily manage the necessary data exchange between the various software components distributed at run time over different parts of the system. With these developments, CoreBluron is able to take advantage of the DEEP architecture as well as other computing platforms like Stampede.

2.1. Overview of application structure

CoreBluron is derived from the NEURON simulation environment that has been developed to support parallel network simulations. Each processor integrates the equations for its subnet over an interval equal to the minimum (inter-processor) presynaptic spike generation to postsynaptic spike delivery connection delay. The overall application structure and workflow has been discussed in next section.

2.1.1. Phases

The simulator starts with reading circuit input data and then initialize model data structure. Different compute phases are performed depending upon the biological time of the simulation. Once the model is initialized from an input dataset, the main simulation timestep starts. Note that the highlighted spike/voltage report component is being developed and not fully integrated with the simulator yet.
The important steps from the simulator are briefly described below:

- **I/O phases**: the simulator reads in the input dataset generated by the circuit-building software. The I/O phases are indicated by the amount of data read/written. *I/O phase I* reads few MBs per node and *I/O phase II* reads ~ 40% of the node’s DRAM.

- **Synaptic current**: matrix elements are updated with the synaptic current contributions from the active synapses and the contributions from the ion channels and pumps.

- **Solver**: as a consequence of the use of an implicit discretization scheme in time, a linear system is solved to obtain the membrane potential values at the current time step.

- **Channel State**: the state equations for channels and synapses are integrated by solving the corresponding equations.

- **Spike Exchange**: whenever the delay of a specific spike event has passed, it is “delivered” to the corresponding synapse by updating its state, which inevitably introduces a discontinuity in its evolution.

Figure 2 shows the high level structure of different phases in the simulator. At the beginning, a binary circuit is read-in by *I/O phases* followed by the serial *setup phase* (i.e. done by single thread). Depending on the simulation time, the *compute* phase shows the time spent in the
main timestep loop. In the current production simulations of CoreBluron, the I/O phase is very small, especially for simulations with large biological time. But it is important to note that the compartment-report functionality currently being implemented will have large I/O cost. The offloading of the I/O phase helps to understand the future impact of the compartment report.

Figure 2: Vampir trace view showing various phases of the CoreBluron simulator (EPFL)

2.1.2. Dominant factors

As explained in the above section, the simulator has a large I/O phase and a main compute time step loop. Though initialization happens only once, depending on the simulation use case (e.g. simulation with few milliseconds of biological time) the initialization phase could be expensive requiring large gigabytes of I/O on the coprocessor. Also, the on-going effort to support for compartment voltage reports will have similar I/O characteristics as the I/O phases discussed above. From the main time step loop, the compute kernels are divided into memory bound kernels and compute bound kernels:

- **Compute bound:** The kernels from the “channel state update” phase are compute intensive. These kernels have lots of division and exponential operations. The operational intensity is higher due to the calculation of the exponential. There are memory accesses with indirect index but their cost is largely hidden by the throughput of division and exponential operations.

- **Memory bound:** The kernels from the “synaptic current” and “solver” phases are memory bound. This is due to fact that they have lots of indirect memory accesses and very low math operations.

The structure of the kernels from the compute bound part is shown in Figure 3. These kernels are dominated by high latency operations like division and exponentials.
for( i = 0; i < count; i++) {

    int idx = node_index[i];
    double v = vec[idx];
    p3[i] = data[ion_index[i]]; 

    double qt = 2.952882641412121;

double mAlpha = (0.182*(v+32.0)) / (1.0-(exp(-v-32.0)/6.0));
double mBeta = (0.124*(-v-32.0)) / (1.0-(exp(v+32.0)/6.0));
double mInf = mAlpha/(mAlpha+mBeta);
double mTau = (1.0/(mAlpha+mBeta))/qt;

    p1[i] = p1[i] + (1.0-exp(dt*(-1.0/mTau))) * 
    ((-mInf/mTau)/((-1.0/mTau)-p1[i]));

}

Figure 3: Structure of compute bound kernels (EPFL)

In contrast, the memory bound kernels have very few floating point operations (addition and multiplication operations) and are dominated by indirect non-unit stride memory accesses. The structure of these kernels is shown in Figure 4.

    for( i = 0; i < count; i++) {

    int idx = node_index[i];
    v = vec[idx];
    p3[i] = data[ion_index[i]]; 

    double gNaTs2 = p0[i]*p1[i]*p1[i]*p1[i]*p2[i];
double ina = gNaTs2*(v-p3[i]);

    data[ion_index1[i]] += gNaTs2;
data[ion_index2[i]] += ina;

    vec_rhs[idx] -= ina;
    vec_d[idx] += gNaTs2;

}

Figure 4: Structure of memory bound kernels (EPFL)

2.1.3. Scalability considerations

To analyse how we could use the DEEP architecture for the CoreBluron simulator, we did profile the compute and I/O phases of the simulator in two scenarios:

- Running simulator on Xeon only
- Running simulator as native application on MIC

As you can see in Figure 2 and Figure 3, the I/O cost is about ~23% while running on Xeon. But as we run the simulator natively on MIC, I/O cost increases to 62% while compute cost decreases to ~38%. Note that the production simulations are 10x larger in terms of biological time (hence the initialisation I/O cost is very small). But, as previously indicated, the
measurements displayed in Figure 5 are representative of the I/O cost that we expect once we will have the compartment report implementation in place with the ReportingLib library.

**Table 1: Experiments setup for the scalability considerations subsection (EPFL)**

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of neurons</strong></td>
<td>1900 to 31000 (1900 per node)</td>
</tr>
<tr>
<td><strong>Iterations</strong></td>
<td>8000 (200 milliseconds of biological time)</td>
</tr>
<tr>
<td><strong>Other details</strong></td>
<td>Due to input dataset limitation and current implementation, the single cell cannot be divided across multiple compute units. Hence we focus on weak scaling simulations, which we use in production.</td>
</tr>
</tbody>
</table>

**Impact on run time of each phase**

![Impact on run time of each phase](image)

**Figure 5: Impact on run time of each phase of CoreBluron, when running on MareNostrum 3, using 2 MICs per host (EPFL)**
2.2. Code modifications during the project

During the duration of the project various improvements have been implemented, such as multi-threading, memory layout transformation, compiler vectorisation, etc. These code changes are described in the sections below. Note that we have used the dataset and simulation configuration described in Table 1, except for section 2.2.1 where benchmarks were performed with the older CoreBluron version and small number of timesteps.

2.2.1. Threading

Though the NEURON core simulator shows very good scaling on large number of MPI ranks, pure MPI implementation increases the communication overhead specifically on MIC if we run two to three ranks per core. Thread-level parallelism is exposed for individual neuron cell so that each OpenMP thread can compute single cells independently. This helps to expose lots of on-node parallelism in the simulator. Only during the spike exchange step the threads are synchronised so that the last finishing thread can perform spike exchange.

Figure 7 and Figure 8 show the comparison between the original pure MPI version and the OpenMP implementation of CoreBluron. Up to 60 MPI ranks we did not see any difference compare to pure OpenMP implementation. When we increase the number of MPI processes up to 240, we see a ~12% improvement due to less process overhead in the OpenMP version.
2.2.2. Vectorisation and NMODL changes

Due to the large number of auto-generated kernels from the DSL, the focus was on optimising these kernels with minimal changes, by utilizing the capabilities of the compiler auto-vectorization. Other approaches for code vectorisation using the Cyme SIMD library are also being explored within the Blue Brain Project. We rely on the “ivdep” compiler directive to provide hints of independent, vectorisable iterations of the loop.
In the original implementation, the individual mechanisms were stored in a SoA format (see Figure 9A), while their individual properties were stored in an AoS layout (Figure 9B) which leads to stride memory access. Due to this, the compiler was only able to vectorise some of the kernels achieving low performance improvements. In order to improve the memory accesses and reduce bandwidth pressure, we changed the layout to SoA as shown in Figure 9C. Further modifications – like using “shadow vectors” – were needed for synapse mechanisms (“shadow vector” temporarily buffers the updates to avoid data dependency conflicts and enable compiler auto-vectorisation). All these changes enable the compiler auto-vectorisation of all the compute kernels generated from the DSL.

Figure 9: A] shows how mechanisms of neuron cells are stored. B] shows the original AoS layout of individual mechanism’s properties, which disables the vectorisation. C] shows the new SoA layout where mechanism properties are grouped together in order to enable compiler auto-vectorisation (EPFL)

In order to improve the performance further, changes to the domain specific language NMODL, were made. We added new keyword called CONDUCTANCE to DSL, which helps the NMODL-to-C translator generating optimised code. The CONDUCTANCE keyword allows users to specify that the i/v (current/voltage) relation is ohmic and hence the extra derivative computation can be avoided. The performance improvements due to this change and vectorisation are shown in Figure 10 and Figure 11. The “Original” column shows the performance of the old “AoS” layout, the “SoA” column shows the performance after changing the memory layout to SoA to enable vectorisation, and finally the “NMODL Opt” describes the performance with additional memory access optimisation. The performance of the original code is very similar to the performance obtained with --no-vec and --no-simd, which shows that it was not possible to vectorise at all before the changes introduced.
Overall we see a 5.3x speedup compared to the original code with AoS layout. Note that the individual kernels generated from DSL show more speedup than 5.3x, but some parts of the simulator (e.g. linear algebra) are not vectorised yet.

2.2.3. Cluster/Booster division

The initial approach in Tk8.1 was to port the NEURON simulator to the DEEP System, offloading mechanisms computation to Booster nodes and using the Cluster to perform the spike exchange, linear algebra and model building parts. However, simultaneously to DEEP a stripped-down, lightweight version of the simulator called CoreBluron is being developed within the Blue Brain Project, aiming at reducing its memory footprint and time to solution.
If DEEP would have continued using the old version of NEURON, all investigations would be outdated by the time the project ended. To avoid this situation, during last year we decided to perform the work of Tk8.1 with CoreBluron instead of NEURON.

The major challenge to port the code onto the DEEP architecture was its complex, non byte-copyable data structures. Figure 12 shows the representative data structure of CoreBluron. Note that this is a small part of the data-structure showing the multi-levels of pointer indirection. The simulator loads neuron cells into the NeuronGroup or NrnThread container. Each cell has different types and numbers of mechanisms or channels. The channels of different types are represented by MechList as a linked list and its data is stored in the MechGroup arrays. These are bitwise copyable variables which can be transferred using offload programming models. Depending on the type of mechanism, there are additional data structures like ion data that is added. For linear algebra calculations, additional vectors like right hand side (RHS) and diagonal elements of the matrix are allocated. This makes offloading of compute kernels difficult.

In order to use the OmpSs offload (as well as MPI offload), we implemented a functionality to allocate byte-copyable data structures in CoreBluron. Basically we packed all required data-elements for compute kernels into contiguous vectors that can be easily transferred back and forth between Cluster and Booster.

![Figure 12: Representation of complex, non byte-copyable data structure of CoreBluron (EPFL)](image)

During last year, detailed performance analysis of the simulator was done with the help of JUELICH and BSC. With the multi-threading implementation and vectorisation, most of the compute kernels from the timestep loop were suitable to run natively on MIC. But during the initial setup phase, each rank/process needs to read data equal to about 40% of DRAM. As I/O on the Booster is slow, it was decided to implement a reverse offload strategy where the model initialisation with large I/O could be offloaded from Booster to Cluster. With the refactoring of I/O kernels of the simulator, the offload implementation looks like shown in Figure 13. While running the simulator on the Booster, the I/O offloader module was added in order to offload the I/O kernels to the Cluster nodes. This offload happens at the level of neuron cells, where every offload-request is identified by cell GID (unique global identifier). The binary circuit reader functions were refactored to execute them on the Cluster (shown as “Circuit Reader” in the Figure 13). When multiple threads from the Booster offload to the Cluster, reader processes spawned on the Cluster handle the I/O requests in FIFO order. The number of reader processes that are spawned on the Cluster depends on the number of cores available on the Cluster node, that is, 16 in the current implementation.
2.2.4. Other optimisations

Apart from the above described simulator-related code changes, it was important to understand and improve the models written by neuroscientists in the neuron NMODL language. For example, a sample compute kernel written in DSL is shown in Figure 14:

```plaintext
PROCEDURE rates()
    LOCAL qt
    qt = 2.3^((34-21)/10)

    mAlpha = (0.182 * (v - 32))/(1-\(\exp(-(v - 32)/6)\))
    mBeta = (0.124 * (-v - 32))/(1-\(\exp(-(v - 32)/6)\))
    mInf = mAlpha/(mAlpha + mBeta)
    mTau = (1/(mAlpha + mBeta))/qt

```

Figure 14: Example of a kernel written in neuron DSL (EPFL)

In the above kernel, variables like mAlpha, mBeta, mInf etc. are RANGE variables by default, which means that they are dynamically allocated on heap and available for reporting/plotting during every timestep. But not all variables are necessary to report/plot while running large simulations on supercomputers (as opposed to desktop where scientists are interested in looking at detailed properties of individual cell every timestep). Hence such selected set of variables can be converted into a LOCAL type that decreases memory footprint as well as memory bandwidth pressure.

2.2.5. Summary of changes

Table 2 summarises the changes done to the application in the time frame of the project.

| Threading | OpenMP parallelisation added with static load balancing, calculated prior to the execution taking into account the computational requirements of each kernel and the type of cell/neuron. Use of mutex lock (omp_lock_t) while spike enqueueing. |
Vectorisation | SoA enabled by changes in the data structure and DSL translator. Compiler hint “pragma ivdep” used
---|---
Offload | Reverse offload of I/O phases implemented using MPI_Comm_spawn and OmpSs
Other | Extraction of CoreBluron out of Bluron. Significant reduction (7x) in memory footprint. Analysis and optimisation of models written in DSL to reduce memory footprint and memory bandwidth.

### 2.3. Benchmarking

Benchmarking of the CoreBluron simulator has been done on Intel Xeon Phi, Xeon and BlueGene/Q. For the performance analysis on MIC and Xeon platforms, we have used the MareNostrum 3 cluster at BSC. For BG/Q, we have used BBP IV BlueGene system at Swiss Supercomputing Centre (CSCS).

**Table 3: Experiments setup for the benchmarking subsection (EPFL)**

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of neurons</td>
<td>1900 to 31000 (1900 per node)</td>
</tr>
<tr>
<td>Iterations</td>
<td>8000 (200 milliseconds of biological time)</td>
</tr>
<tr>
<td>Other details</td>
<td>Due to input dataset limitation and the current implementation, the single cell cannot be divided across multiple compute units. Hence we focus on weak scaling simulations, which we use in production.</td>
</tr>
</tbody>
</table>

#### 2.3.1. Xeon Phi platforms

Table 4 provides the details on the software stack, including the compiler options and configurations used for benchmarking on Xeon Phi.

**Table 4: System setup details for Xeon Phi platforms (EPFL)**

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>intel/14.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>impi/5.0.1.035</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O3 -fopenmp</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16/2 in MareNostrum 3 (Xeon/Xeon Phi). Single MPI process per socket on Xeon, and single MPI process per MIC card</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1/120 in MareNostrum 3 (Xeon/Xeon Phi)</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>Used KMP_PLACE_THREADS to bind threads to core</td>
</tr>
<tr>
<td>Offloaded ranks per MPI process</td>
<td>8 (single MPI rank on MIC spawns 8 MPI reader processes on host)</td>
</tr>
</tbody>
</table>

Figure 15 and Figure 16 show the scaling behaviour of CoreBluron on Xeon Phi while running natively. Though the original code also shows a good scaling behaviour, the on-node performance of the code was poor. As discussed in the section 2.2.2, overall a 5.3x improvement has been achieved in total runtime of timestep loop. While running on 16 nodes (i.e. 32 MICs), the parallel efficiency has dropped to 0.82. Also, note that the parallel efficiency of the original code was better (not the time to solution) due to the fact that the spike exchange part (MPI communication) was not significant before the on-node optimisations. The input dataset was generated on BlueGene/Q by considering the computation cost of mechanisms executing there. Also the mechanisms are not vectorised on this platform with the XL C compiler. The fact that the input data set was generated in another platform introduces a small load imbalance across ranks when we execute CoreBluron on a MIC, which results in decrease in parallel efficiency. In order to improve the scaling, we are
working on generating dataset in HDF5 format where compute costs can be re-calculated on the platform where CoreBluron is being executed.

**Performance of CoreNeuron in Xeon Phi platforms**

![Graph showing performance of CoreNeuron in Xeon Phi platforms](image)

Figure 15: Performance of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL)

**Parallel efficiency of CoreNeuron in Xeon Phi platforms**

![Graph showing parallel efficiency of CoreNeuron in Xeon Phi platforms](image)

Figure 16: Parallel efficiency of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL)

In order to measure the benefits of the offload, we measured the I/O-cost while running the simulator natively and using the offload mode. For the offload runs on MareNostrum 3, every MPI rank on a Xeon Phi card spawns eight reader-processes on each socket of the host. Then, every OpenMP thread offload I/O kernel to the host. Figure 17 shows the I/O performance improvement using the offload mode compared to a native run on Xeon Phi. As expected, the I/O on Xeon Phi is slow and we see 3x I/O performance improvements using the offload mode.
Performance of CoreNeuron on Stampede (compute phase only)

Figure 18: Performance of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL)

On MareNostrum 3 only 42 nodes (i.e. 84 mics) are available and, due to its stability issues, we were only able to use 32 mics. We did scaling studies up to 128 mics on the Stampede system at TACC. We used the same configuration shown in Table 3 and Table 4. The scaling results on Stampede are shown below. The parallel efficiency of the code drops after 32 nodes. We believe that this is due to an increased load imbalance in the dataset generated on BlueGene/Q platform.
Figure 19: Parallel efficiency of the main compute phase of CoreBluron in Xeon Phi platforms (EPFL)

2.3.2. Comparison with BlueGene/Q

Table 5 describes the software stack, including the compiler options and configurations used for benchmarking on BlueGene/Q.

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>bg-xl/12.1.0.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>bg-q mpich</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O3 -qtune=qp -qarch=qp -q64 -qhot=simd -qsmp -qthreaded</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>1</td>
</tr>
<tr>
<td>Threads per process</td>
<td>64 in BlueGene/Q</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>BG_THREADLAYOUT=1 (default breadth first thread placement)</td>
</tr>
</tbody>
</table>

Figure 20 and Figure 21 show the weak scaling performance of CoreBluron on BlueGene/Q. As we generated the input dataset on BlueGene/Q, the load balance is pretty good and scaling shows a parallel efficiency of ~0.96 when using 128 nodes.
2.3.3. Comparison with cluster architectures without coprocessors (without offload to regular CPUs)

In this section we discuss the performance of CoreBluron when running only on the host on MareNostrum 3 (i.e. without using accelerators).

Table 6: System setup details for MareNostrum 3 without Xeon Phi (EPFL)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>intel/14.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>impi/5.0.1.035</td>
</tr>
</tbody>
</table>
Compilation flags | -O3 -fopenmp
---|---
MPI processes per node | 2 (one per socket)
Total MPI threads per node | 8
Affinity setup | `I_MPI_PIN_DOMAIN=socket`
Offloaded ranks per MPI process | NA

Figure 22 and Figure 23 show the scaling and parallel efficiency of CoreBluron while running on MareNostrum 3 in host only mode (i.e. without offload). The vectorisation and multithreading has improved the on node performance of the simulator by 4x. The original code is showing almost linear speedup but the parallel efficiency of the optimised code is dropped to 0.88 while running on 16 nodes. We have seen similar behaviour on Xeon Phi and this has been discussed in section 2.3.1.

**Performance of CoreNeuron in cluster platforms**
(compute and I/O phase)

![Graph showing performance of CoreNeuron](image)

**Figure 22:** Performance of the CoreBluron compute phase in cluster platforms. The Xeon Phi experiments use I/O offload (EPFL)
2.4. Intranode metrics

In order to better understand the performance of individual compute kernels, we measured the cycles per instruction (CPI) and vectorisation intensity as shown in Table 7.

Table 7: CPI and vectorisation intensity on a set of DSL generated kernels on Xeon Phi 7120 (EPFL)

<table>
<thead>
<tr>
<th>Compute kernel</th>
<th>Cycles per instruction</th>
<th>Vectorisation intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_Ih</td>
<td>4.52</td>
<td>9.26</td>
</tr>
<tr>
<td>S_Im</td>
<td>4.81</td>
<td>9.03</td>
</tr>
<tr>
<td>S_Na</td>
<td>4.71</td>
<td>9.27</td>
</tr>
<tr>
<td>S_SKv</td>
<td>4.96</td>
<td>9.10</td>
</tr>
<tr>
<td>S_PGABA</td>
<td>7.31</td>
<td>8.44</td>
</tr>
<tr>
<td>S_PAMPA</td>
<td>6.57</td>
<td>8.43</td>
</tr>
<tr>
<td>B_Ih</td>
<td>9.63</td>
<td>3.21</td>
</tr>
<tr>
<td>B_Im</td>
<td>13.34</td>
<td>3.49</td>
</tr>
<tr>
<td>B_Na</td>
<td>11.76</td>
<td>3.97</td>
</tr>
<tr>
<td>B_SKv</td>
<td>12.78</td>
<td>3.55</td>
</tr>
<tr>
<td>B_PGABA</td>
<td>4.27</td>
<td>4.68</td>
</tr>
</tbody>
</table>

The first four kernels (S_Ih, S_Im, S_Na, S_SKv) in the table are compute bound kernels with high latency operations like exponential and division. As there is no branching / mask instructions, vectorisation intensity is very good and vector units are fully utilised. The kernels like B_Ih, B_Im, B_Na etc. are memory-bound kernels and use indirect memory accesses. Because of that the vectorisation intensity is low compared to the first four kernels.
2.5. Conclusions

During the duration of the DEEP project we have made significant progress in terms of code development, as well as performance improvements. Changes in memory layout, multi-threading, and auto-vectorisation have improved the on-node performance (5.3x improvement on MIC) as well as scaling of the simulator. Due to lots of small compute kernels and complex data structure it’s not easy to make code division for offload. The reverse offload capability of DEEP software stacks helped us to port CoreBluron application in such scenario. Some issues have been identified, such as load imbalance when data is generated on BlueGene/Q and used for simulation on MIC. These issues will be addressed in the future development of CoreBluron. Despite the challenges encountered, this work has improved the performance of the simulator across all platforms (2.5x on BG-Q and 3.4x on Xeon). We believe that the DEEP architecture will be beneficial for the CoreBluron simulations as well as for future use cases like interactive computing where visualisation software stack can be efficiently run on cluster nodes while simulator executing on Booster nodes.
3. Task 8.2: Space weather simulation (Task leader: KULeuven)

Within DEEP the attention was focused on improving the particle section of the code. The procedure to move each one of the particles and project their values on the grid nodes has been updated and adapted to the DEEP architecture. In the particle solver hundreds of particles per cell are constantly moved. In large-scale applications many billions of particles can be part of the simulation. These particles are completely independent from each other, ensuring very high scalability. MPI communications in the particle solver are limited to the transmission of particles from one subdomain to its neighbour.

A summary of the modifications introduced to the code in order to perform simulations in heterogeneous architectures, and an analysis of the performances obtained after such modifications are presented in the following sections.

3.1. Overview of application structure

The Particle-in-Cell code iPic3D is divided in two main components: 1) a finite element “field solver” that calculates the evolution of the electromagnetic fields, and 2) a “particle solver” that traces the individual movement of electron and ion particles composing the plasma flow. Information is exchanged between these two components by interpolation of data between the nodes of the grid and each one of the particles. Each particle recovers the information about the electric and magnetic fields by projecting the values stored in the closest nodes. A weighted sum of the node quantities is performed for each particle to obtain their linear interpolation in three dimensions. The weight coefficient depends on the distance to each node and the particle electric charge.

While transferring information from the fields to the particles is relatively straightforward, the opposite procedure, called moment gathering (moment calculation, moment sum), requires a more complex strategy. The moment gathering represents the statistical projection of information from each particle to the closest nodes in the domain. For simplification reasons the code iPic3D does not currently include any kind of particle sorting. This means that particles are located in random locations of the memory and are not organised by cells. To gather the moments at a single node the code has to scan and test all the particles located in the current process.

3.1.1. Phases

Currently the code is divided in three computing phases and two Cluster-Booster (CB) communication phases. Figure 24 shows the phase decomposition of the code. Bold arrows indicate the communication phases between Cluster and Booster and thin arrows indicate the internal communications inside each of the computing phase. In [1] it was shown that this computing decomposition suits best the numerical methods built in the iPic3D code.

The moment gathering procedure, which interpolates data from the particles to the grid nodes, requires a specific numerical treatment, independent from the particle mover. In this report the “particle solver” branch of the code is composed of a “particle mover” phase and a “moment gathering” phase.

In the same figure, the green section represents the “field solver” branch. In previous reports this branch was also divided into “hatted moment” phase and “field advancement” phase. However, the computing times for the hatted phase are negligible compared with the field
advancement. Therefore, in this report it is considered that the “field solver” is one phase that contains both the hatted moment calculation and the field advancement computation.

In summary, the blue branch is deployed in the Booster and contains two phases, and the green branch is deployed in the Cluster architecture and contains one phase.

![Figure 24: Phases of iPiC3D. Green phases run on the Cluster, blue phases run on the Booster (KULeuven)](image)

### 3.1.2. Dominant factors

The field solver that runs in the Cluster uses the finite difference method (FDM) to solve the implicit time evolution of the electric field using a Cartesian grid. This is translated into the resolution of a linear problem of the type $Ax=b$ where the dimension of the vector $Ax$ is three times the number of cells in the mesh (the electric field has three components). The numerical solution of this linear problem is obtained using an iterative GMRES method. That means that the matrix must be accessed and updated at every iterative step, and it involves global communications for the calculation of the norm of the residual $r=|Ax-b|$. This approach makes the field solver bound by communications, making it best suited for running on the Cluster.

The particle mover on the other hand features a large number of independent numerical points. Communications are only necessary to move particles from one subdomain to the next. The calculation of each particle movement is independent from the rest of the code. Space Weather applications can reach low noise levels by using large numbers of particles. Therefore, the particle mover is mainly compute bound. However, the upper limit of the number of particles depends on the available memory in the system.

The third phase, the moment gathering, is bound by memory accesses. The interpolation procedure does not require any interaction between the particles themselves. However, access to data is random and can cause multiple cache misses dramatically slowing down the gathering of information in the nodes. Vectorisation of the procedure using a smart data structure is required to overcome these limitations.
3.1.3. **Scalability considerations**

The iPic3D code has been tested in different supercomputers around the world. Runs on up to 32,000 processors have shown that the code has an almost linear strong scaling. Figure 25 displays the results obtained during tests in the Curie supercomputer. This cluster, located at CEA, in France, is composed of 5040 B510 BullX nodes, each one with of 2 eight-core Intel Xeon E5 2680 processors, and 64GB of memory.

![iPic3D Scaling](image)

*Figure 25 Speedup of the original version of the iPic3D code. Simulations performed in the Curie supercomputer using Intel Xeon E5 2680 processors (KULeuven)*

This good performance was obtained using Space Weather simulations with a computational load of around 15000 cells per processor. Communications at the interior of the linear iterative field solver struggle to perform at the ideal levels, shown in this figure, when the computational charge per process is low, i.e. when the number of cells per process is low. This is mainly due to the required use of global broadcasting of messages that update the values of the residual.

The particle mover is in principle highly scalable. Particles are independent and a careful vectorisation of the treatment of each one of them should lead to an increase in the performance of the code. Communication between the different processes is limited to the transfer of particles that leave one subdomain to enter the next one. An improvement in the computation speed is expected by deploying the code in the Booster, but the impact on the communications must be carefully followed.

Moment gathering is a somewhat more complex procedure than particle moving. It requires access to particle data located in different, non-contiguous memory sectors. To obtain the moments at a single grid point, the code recovers and projects data from the closest particles. These particles are not organized in memory. It is important to work on the locality of the gathered information. Communications are limited to the sharing of particle information between neighbouring processors. This internal communication procedure must be tracked to check if it performs in the same way in the Booster as it does in a regular CPU cluster.

It is important to note that the scalability of the code depends on the computational load of each process. It has been observed in previous studies that the iPic3D code shows an almost ideal scalability up to 32000 processors, when the number of cells per processor is larger than 1000 and lower than 30000. With a lower computational charge, communications take
precedence over computation and scalability stagnates. On the other limit, using a large number of cells depends on the available memory, but can lead to very slow iteration times.

The simulation of a drifting Maxwellian plasma inside a periodic box is the simplest case that can be run with iPic3D. At the beginning of the simulation all the particles are located inside the domain at random locations. They move with a constant “group velocity” called the drifting velocity. In addition, each independent particle has also a “thermal velocity” which is derived from a Maxwellian velocity distribution. The total velocity of each particle is equal to the addition of the drifting and the thermal velocities. This kind of simulation allows for a well-balanced distribution of particles on the different processors and threads.

<table>
<thead>
<tr>
<th>Drifting velocity of the plasma</th>
<th>( u_0 )</th>
<th>0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thermal velocity of the ions</td>
<td>( u_{th,i} )</td>
<td>0.0063</td>
</tr>
<tr>
<td>Thermal velocity of the electrons</td>
<td>( u_{th,e} )</td>
<td>0.0045</td>
</tr>
<tr>
<td>Mass ratio</td>
<td>( m_i/m_e )</td>
<td>256</td>
</tr>
<tr>
<td>Number of particle sub iterations in the mover loop</td>
<td>( npariter )</td>
<td>3</td>
</tr>
<tr>
<td>Maximal residual for the field linear solver</td>
<td>( \epsilon )</td>
<td>1e-3</td>
</tr>
</tbody>
</table>

The details of the parameters used for this simulation are given in Table 8. Two geometries were selected to perform the weak scaling of the code and the strong scaling. The computational charge of each case is slightly different but they are representative of the kind of simulations performed in the Space Weather applications. The parameters shown in Table 9 were selected to fit the case in different number of nodes without saturating the system memory. It was important to select a strong scaling test that fitted in memory for a low number of nodes but was not too small for Amdahl’s law to limit the scalability of the code.

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Strong scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>256 x 256</td>
<td>256 x 256</td>
</tr>
<tr>
<td>Time step</td>
<td>0.25</td>
<td>0.2</td>
</tr>
<tr>
<td>Number of cycles</td>
<td>200</td>
<td>50</td>
</tr>
<tr>
<td>Box length</td>
<td>20 code units</td>
<td>6 code units</td>
</tr>
<tr>
<td>Cell size</td>
<td>0.078 code units</td>
<td>0.023 code units</td>
</tr>
<tr>
<td>Particles/cell</td>
<td>684</td>
<td>250</td>
</tr>
</tbody>
</table>

Using the Xeon nodes of the DEEP Cluster, we performed measurements of the runtimes of each of the phases of the code, using the strong scaling benchmark case. Figure 26 displays the total runtime for the field solver, the particle mover and the moment gathering using 1 to 64 nodes (16 to 1024 processes). Acceleration of the execution from 1 to 8 nodes is evident. It will be shown later that, above 8 nodes, communications become more time consuming than computation, creating a stagnation of the speedup of the code.
It can be already observed in Figure 27 that the code starts spending more and more time in the field solver above 8 nodes. Communications take over computation and the field solver uses as much time as moment gathering at 64 nodes. However, for a good computing load, i.e. for less than 8 nodes, the field solver only takes around 5% of the total runtime. The particle mover (50%) and the moment gathering (30%) are the two phases that consume more runtime. For this reason the developments presented in this report focused on the acceleration of the particle solver branch of the code. This branch is meant to be deployed in the Booster section of the DEEP architecture, while keeping the less-scalable field solver on the Cluster.
3.2. Code modifications during the project

An initial analysis of the code revealed that there were three main considerations for the deployment of the iPic3D code in the DEEP System.

1. Cluster-Booster separation: the best strategy was identified to be loading the field solver in the Cluster without major modifications and focusing on the development and porting of the particle solver (mover and moment gathering) to the Booster.

2. Particle communications: from the initial tests it was observed that particle communications might present challenges when ported to the Xeon Phi architecture. The current communications method, developed ten years ago by the initial developers of the code, require an improvement adapted to modern architectures.

3. Moment gathering: due to the non-locality of the memory access during the moment gathering procedure it is important to develop a memory strategy that will minimise cache misses in the code. We will present the strategy used in the following sections.

3.2.1. Threading

In the past, iPic3D used exclusively MPI communications, which are suited for CPU clusters with distributed memory. Tests performed for deliverable 8.2 showed that using MPI only communications on 60-core MIC processors was ineffective.

The first major change included in the code was to implement OpenMP parallelisation on both phases of the particle solver (mover and moment gathering). Figure 28 shows an important improvement on the total time of the particle solver, mainly due to the use of all available cores of the MIC co-processor and by avoiding unnecessary MPI communications between cores.

In this figure the total runtime of four tests cases is reported. The application was deployed in a single Intel Xeon Phi co-processor. In this architecture 60 computing cores are available, each one containing 4 independent processing pathways. It is expected to reach a maximum of
acceleration for $60 \times 4 = 240$ independent processes (or threads). Figure 28 shows that the runtimes decrease when increasing the number of processes running in the co-processor. For details on the setup used for this test, please refer to section 3.1 of deliverable D8.2. The figure shows that the best performances are obtained using an MPI+OpenMP approach.

Figure 28: Tests performed on the Xeon Phi architecture. Old: MPI only version. New: MPI+OpenMP version (KULeuven)

In order to map MPI processes in the Xeon Phi co-processors with Xeon processors in the node, it was decided to use the MPI+OpenMP version of the code, and find the combination that results in shorter total run time.

The procedure used is the following: when a simulation starts, $X$ processes are launched in each Xeon Phi card. Each one of these processes is in charge of $Y$ threads. This assignment distributes the 240 threads on all the available hardware locations. At the interior of one process the particle vectors are divided among the threads.

The particle mover is the main computing sequence in the particle solver. A large loop performs the calculation of the new position and velocity for each of the particles. This main computational loop has been divided among the available threads. Additional improvements in computing time were obtained by using Xeon Phi intrinsics. Extremely fine tuning using this procedure is possible, but with the drawback that the code becomes less portable and more difficult to maintain. OpenMP sections were also included in book keeping procedures and in the buffering of data for subsequent MPI communications.

The second important loop of the particle solver is the moment gathering loop. Each process of the particle solver contains a local copy of the grid subdomain. The particles are distributed among the threads of the process. Each thread projects its assigned particles on the temporal copy of the grid. Particle information is communicated between adjacent subdomains to correctly represent the moment data at the shared nodes. Once the data is gathered in the local temporal array, the information is sent to the field solver in the Cluster. The use of multiple threads to process the particle information speeds up the moment gathering procedure.
3.2.2. Vectorisation

As reported in deliverable 8.2, section 3.2, the two basic and most important modifications of the code were: a) use pragmas to point the compiler at the vectorisable loops and b) re-build the main loops of the code to exploit vectorisation. These modifications did not radically improve the speed of the code though. We noticed that the simple deployment of the code on a Xeon Phi coprocessor was not performing as expected due to the nature of the data structures and the memory accesses used. The speedup reported in deliverable 8.2 was mainly gained thanks to the introduction of the multi-threading procedures presented in the previous section.

In order to make full use of the Xeon Phi architecture it was decided to modify two of the core elements of the code: a) changing the data structure of the particle solver in order to improve data locality and reduce cache misses, b) re-writing the main particle procedures using MIC intrinsics to make full use of the compiler and the processor capacities.

Each particle in the code is composed of eight properties stored in variables of type double: three components of the position, three components of the velocity, the particle charge and a tag (that can be used as an id). Traditionally, it was accepted that vectorisation is best served when each one of these elements are stored in a different vector. iPic3D was initially designed to use one vector for each one of these eight particle elements. However, as discussed in previous reports, the particle mover and the bookkeeping require all the data from one particle to be available at the same time. In such situations it is more efficient to store the eight elements of the particle in contiguous memory locations. Therefore, both memory location strategies had to be studied.

Saving the data in the classical approach is known as a Struct of Arrays (SoA). It keeps separated vectors for each one of the eight elements of all particles. The second approach is called an Array of Structs (AoS) and is defined as the vector in which each element contains the information about the eight properties of the particles. Finally it was identified that there are zones of the code where the AoS performs better and other zones where the SoA was more appropriate.

The final solution was to implement a mixed approach: the main mover loop and the particle bookkeeping use the AoS approach, while the moment gathering is done using the SoA approach. To avoid a duplication of information in two different structures all particle data is saved in a contiguous memory location. Taking advantage of the fact that the elements of a struct in C++ are allocated in consecutive memory locations, the new data structure is built on an extension of the std::vector class of C++. This vector of particles is called the AoS.

The particle moving phase is performed using the AoS approach. The moment gathering phase on the other hand uses the SoA approach. To switch between these two phases a transposition of the particle vector is needed. A matrix transposition is in general regarded as a time consuming procedure, however some tests showed that the time overhead of the matrix transposition is very well balanced by the performance improvements of the SoA approach in the moment gathering phase.

In addition to the introduction of the new particle data structure, an important step towards a good performance of the code on Xeon Phi architectures requires the correct use of a) the VPU registers of the MIC cards by calling intrinsic functions and b) special Intel calls that guarantee memory alignment, code optimisation and vectorisation.

It is also important to note that the Xeon Phi intrinsics work using the 8 Vector Processing Unit (VPU) registers of 512 bytes each one. Each register can thus contain eight values of type double (64-bytes each), making it convenient for the data representation: each register
can simultaneously contain the eight elements of one particle, or one element of eight particles. For simple loops the Intel compiler is able to translate the C++ code into VPU instructions. However, in more complex loops the VPU instructions must be explicitly given.

One of the main code optimizations is the transposition of the particle vector from an AoS to a SoA. The matrix transformation is performed by transposing chunks of 8x8 doubles, i.e. eight particles with their eight properties are transposed at a time. Eight VPU registers containing eight 64-byte doubles are involved in the procedure. Using the Intel memory alignment calls it is certified that the 8x8 blocks are contiguous in memory and the compiler is directed to particular zones were the code can be vectorised.

The inclusion of the intrinsics and the use of aligned memory allocation allowed for an increase in the computing performance of the code, especially in the moment gathering and particle pushing phases (around 10%). Table 10 shows the performance of the code running in one node, with and without intrinsics.

<table>
<thead>
<tr>
<th></th>
<th>Total run time</th>
<th>Total particle time (compute + communication)</th>
<th>Particle (only compute time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without intrinsics</td>
<td>309.197</td>
<td>157.575</td>
<td>94.935</td>
</tr>
<tr>
<td>With intrinsics</td>
<td>283.118</td>
<td>132.607</td>
<td>60.869</td>
</tr>
</tbody>
</table>

3.2.3. Cluster/Booster division

The code was split following the phase division shown in Figure 24. In the final DEEP architecture the field branch will be launched in the Cluster (Xeon) while the particle branch will be launched in the Booster (Xeon Phi). For testing purposes we also performed runs where the particle solver is launched in additional nodes of the Xeon Cluster.

While the initialisation of the matrices in the field solver is straightforward, the particle initialisation is both compute and memory intensive. This and the general logical structure of the code led to the decision to select the particle solver as the entry point (host) of the code on the Booster. Once the particle solver is launched it triggers the offloading of the field solver to the Xeon Cluster, performing a so called reverse offload. For testing purposes runs were performed where the particle solver is launched in additional nodes of the Xeon Cluster.

Two methods were considered to perform the offloading of the field solver: a) using the OmpSs runtime developed by BSC and extended during the DEEP project, and b) using the MPI_Comm_Spawn call from MPI.

OmpSs extends the functionalities of OpenMP, but the former is not fully compatible with the later. OmpSs does not support some of the pragmas introduced in the iPic3D code. To use this offloading method we commented out all the OpenMP pragmas aside from #pragma omp task and #pragma omp for. The code was then compiled with the Mercurium compiler. The offload is performed by a call to the field solver at the interior of a #pragma omp task device (mpi) call. OmpSs takes this section of the code and offloads the calculation to the Xeon processor. Figure 29 shows the exact calls used in iPic3D.
MPI_Comm clustercomm;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

deep_booster_alloc(MPI_COMM_WORLD, 1, size, &clustercomm);

//create offload task
#pragma omp task device(mpi) onto(clustercomm,rank) \ 
in([...] copy_deps
{
    solver.run_Cluster();
}

solver.run_Booster(clustercomm);

Figure 29: Method of integration of OmpSs in the main code (KULeuven)

All the objects required by the field solver need to be sent through the Cluster-Booster communicator from the Xeon Phi to the Xeon. However, OmpSs does not yet support the offload of C++ objects. A solution to this problem would be to serialize in the starting process and de-serialize in the spawned processes all the necessary objects, but initial tests showed that this task would be too complex. Instead, only the argv and argc parameters are offloaded to the Cluster, and each Xeon processor recreates the full initialization procedure itself.

This procedure was tested multiple times but led to complications without clear benefits in this particular code division. Therefore, it was decided to continue the development of the code using the second offloading approach.

Offloading the field solver, the host performs an MPI_Comm_spawn call. This creates an additional global MPI inter-communicator for the child processes, and an additional parent-to-child communicator to connect the particle branch to the field branch. The software deployed in the Cluster starts execution with the new child communicator. The main() routine points the execution towards the two branches depending on the communicator used. The run_Cluster() branch is deployed in the Xeon Cluster and the run_Booster() branch is deployed in the Xeon Phi Booster. To detect which branch has to be executed a conditional if clause checks if the current processor communicator is the parent (host) or the child (offload).

Dividing the code in run_Cluster() and run_Booster() branches requires the implementation of MPI communication procedures between the Cluster and the Booster. These additional features were included in iPic3D. Figure 30 shows the new offloading procedure.
In the benchmarking section we will show the performance of the full system using three different processor allocations:

1. Offload from MIC to Xeon (m2x): the application is launched in the Xeon Phi coprocessors and the fields are offloaded to the Xeon processors,
2. Offload from Xeon to Xeon (x2x): the application is launched in the Xeon nodes and the field solver is offloaded to additional Xeon processors.
3. No offload: the code is not split in two branches. The code runs as a classical parallel application in the Xeon cluster using only one global communicator. This is the baseline for the benchmarking procedure.

3.2.4. Other optimisations

The code iPic3D uses a large number of dynamically allocated vectors and matrices. To guarantee that these data structures are aligned in memory, new memory allocation routines were included in the code. In addition, memory alignment and concordance with the VPU registers of the MIC coprocessor is ensured by using the appropriate intrinsic routines.

3.2.5. Summary of changes

Table 11 summarises the changes done to the application in the time frame of the project.

<table>
<thead>
<tr>
<th>Threading</th>
<th>OpenMP parallelisation of particle processing and moments accumulation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorisation</td>
<td>AoS used for representation of particles, but transposed on the fly to SoA to enable vectorisation. Hints to the compiler for automatic vectorisation. Extensive use of intrinsic calls to load vectors into the MIC VPU registers. In particular for the on the fly transposition between the AoS and the SoA.</td>
</tr>
<tr>
<td>Offload</td>
<td>Reverse offload of field solver. Implementations with MPI_Comm_spawn and OmpSs</td>
</tr>
</tbody>
</table>
Other

Inclusion of subroutines to guarantee data locality of dynamically allocated vectors and matrices.

3.3. Benchmarking

The focus of the benchmarking section is to give a summary of the runtimes obtained using the three offloading methods in platforms using only Xeon processors and platforms using heterogeneous Xeon+Xeon Phi processors.

Table 12: Experiments setup for the benchmarking subsection (KULeuven)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>256x256 per node</td>
</tr>
<tr>
<td>Number of particles per cell</td>
<td>5x5x5x2 (18x18x1x2 on MareNostrum 3)</td>
</tr>
<tr>
<td>Box length</td>
<td>6x6 per node (20x20 on MareNostrum 3)</td>
</tr>
</tbody>
</table>

3.3.1. Xeon Phi platforms

For the benchmarking of the code on Xeon Phi platforms we had access to two systems: MareNostrum3 from the Barcelona Supercomputer Centre, and Stampede from the Texas Advanced Computer Center.

The weak scaling test case was used in these two systems to test the parallelisation of the computational sections of the code with a constant computational load in each process and constant communications. The details for each system are presented in Table 13 and Table 14. It has to be noted that MareNostrum 3 has 2 Xeon Phis per node, whereas Stampede has 1.

Table 13: System setup details for Stampede (KULeuven)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 15.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Intel MPI 5.0.2</td>
</tr>
<tr>
<td>Extrace version</td>
<td>extrace_lite 0.2</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-openmp -fno-exceptions -fp-model fast=2 -O3 [-xHost, -mmic]</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16/16 (Xeon/Xeon Phi)</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1/15 (Xeon/Xeon Phi)</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>KMP_AFFINITY=compact</td>
</tr>
</tbody>
</table>

Table 14: System setup details for MareNostrum3 (KULeuven)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>intel/14.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>impi/4.1.3.049</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O3 -fopenmp [-xHost, -mmic]</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16/8x2 (Xeon/Xeon Phi)</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1/30 (Xeon/Xeon Phi)</td>
</tr>
</tbody>
</table>

The experiments in these two systems are slightly different, and therefore the run times cannot be compared directly. However, parallel efficiency is a comparable metric. Figure 31 shows the parallel efficiency of iPic3D on the two Xeon Phi platforms. Both systems show a linear scaling with a very low slope.
We show no results for more than eight nodes in MareNostrum 3. Runs performed multiple times were showing inconsistent runtimes. We suspect that faulty nodes or interconnections were causing the large runtime variability observed: including more nodes in the simulation raised the probability to use one of the faulty nodes. It was decided that the runs performed on Stampede would give more accurate and consistent results.

The parallel efficiency of the benchmark performed in Stampede is presented in Figure 31.

![Parallel efficiency of iPiC3D in Stampede](image)

**Figure 31: Parallel efficiency of iPiC3D on Stampede and MareNostrum 3 (KULeuven)**

### 3.3.2. Comparison with cluster architectures without coprocessors (with and without offload to regular CPUs)

To compare the Xeon Phi platforms with classic Xeon-only clusters we performed two additional kinds of runs: a) a traditional “no offload” case were the application is run on Xeon processors without Cluster-Booster division, and b) a “x2x” case where the division Cluster-Booster is performed and run on different Xeon nodes of the supercomputer. Figure 32 shows the total runtime in seconds for each one of the three execution cases. Notice that there is an outlier point in the “Stampede (w.o. offload)” case when running on 16 nodes. We can only attribute it to an unexpected underperformance of the cluster at the moment of the run. For our discussion we will exclude this outlier point.
This figure shows that the division Cluster-Booster does not affect the run time of the application when the offload is performed on the Xeon processors. However, when the code uses the Xeon Phi coprocessors the application has a runtime 4 to 5 times slower than the case without offload.

Figure 32 shows the parallel efficiency of the code in all three running cases. It is clear that the problem is not associated either with the Cluster-Booster division, nor with the communications. The only variable that plays a role in the slowing down of the application is the introduction of the Xeon Phi coprocessors.
3.3.3. Comparison with the beginning of the project

The DEEP project improved the performance of the iPic3D code for applications other than the simple benchmark case presented in this report. Figure 34 shows the gain in computing time of the original version of the code, before DEEP, against the current version of the code. Both cases were run using only the Xeon processors of the DEEP system. The run used to create this figure is the strong scaling case presented in Table 9. Figure 35 shows the parallel efficiency of the old and new versions. It can be observed that both versions have similar efficiencies, even though the old version presents superlinear speedup for 4 and 8 nodes, due to better cache usage when reducing the data set per process. Figure 36 shows computational gain, as the ratio of the old run times vs the new run times. It consistently shows an increase in the performance of the code.

Table 15: System setup details for DEEP Cluster (KULeuven)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 14.0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Parastation MPI 5.0.28</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-openmp -fno-exceptions -O3</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>Default</td>
</tr>
</tbody>
</table>

This improvement in performance is a major advance, and is currently helping the researchers of KU Leuven to perform simulations using less than half of the resources they needed before the beginning of the DEEP project.
Figure 34: Performance of current version of iP3C3D vs. old version of iP3C3D, in DEEP (KULeuven)

Figure 35: Parallel efficiency of current version of iP3C3D vs. old version of iP3C3D, in DEEP (KULeuven)
3.4. Analysis of time evolution of different phases

This section analyses the run times of the different phases of the experiments performed on Stampede. It does so decomposing the times in computation and communication times. Figure 37 shows a detailed decomposition of the three weak scaling runs, and explains the reasons why the case running with the Xeon Phi coprocessors gives a higher total run time. The figure shows nine charts: the three columns display the three phases of the code and each one of the three rows correspond to the cases without offload, with offload to the Xeon and with offload including Xeon Phi coprocessors. Each of the nine charts contains the runtime for computations (blue) and communications (red) for 1 to 128 nodes. The abscissa gives the runtime in seconds. The maximum value of this axis is 20 seconds in all the charts, except in the last two charts, where the maximum value is set to 60 seconds (the charts with the grey background).

In the first column the computing and communication times for the field solver can be observed. For all the cases and all the number of nodes, it can be observed that the computing time and the communications time is very similar (around 3 seconds). These three charts show the reason why the field solver was chosen to run on the Cluster side of the system: computing and communications take almost similar times. This is only true for the number of cells selected for the benchmarking: a higher number of cells per process would lead to higher computing time compared with communications.

It is also interesting to observe that the total runtime of the field solver remains almost constant independent of the number of nodes used.

In the second column, the runtimes of the particle mover phase can be observed. For the three cases and the different node numbers, the computing time is almost identical, around 10 seconds, including for the case with Xeon Phi coprocessor (3rd row). Communication times on the other hand increases with an increasing number of nodes in all three cases. However, this value is critically bad in the 3rd case that includes the Xeon Phi coprocessor. This figure clearly shows that communications between Xeon Phi cores are very deficient. This critical aspect of the code was acknowledged in previous deliverables and Alec Johnson was...
intensely working on the re-formulation of particle communications. Unfortunately, when he passed away his latest developments and ideas were partially lost.

The third column shows the behaviour of the moment gathering phase in the three cases. Contrary to the previous column, the distribution of time between computation and communications is more favourable. However, in the case running with Xeon Phi coprocessors (3rd row), the moment gathering phase is almost seven times slower compared with the cases without the coprocessor.

This is a very unfortunate result obtained only recently. The previous tests performed using the Cluster-Booster division in Xeon-only systems were very encouraging and the good performances obtained by the vectorisation enhancements were very welcomed. The current results point towards the difficulty of porting Xeon code to Xeon Phi coprocessors.
Figure 37: Decomposition of the weak scaling results obtained in Stampede for the three offload methods. In columns, the three phases of the code. In rows, the three cases run. Blue bars: computing time, red bars: communication times (KULeuven).
3.5. Analysis of metrics for WP9

We used the extrae_lite tool developed by BSC to trace some of the important metrics of the new version of the code iPic3D. We used the weak scaling benchmarks performed in the Stampede supercomputer to gather the information. Table 16 is a summary of the computational load of this test, and Table 17 gives a summary of the system setup used to run the code. The offload case was used to perform these measurements, and as such they include the performances of the Xeon Phi section of the code.

Measurements were performed for each one of the three phases of the code, and for the communications between the Cluster and the Booster. However, in the sections below, we only present the metrics for the three phases, as they are various orders of magnitude larger than those measured in the Cluster-Booster communications.

Table 16: Experiments setup for the analysis subsection (KULeuven)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of cells</td>
<td>256x256 per node</td>
</tr>
<tr>
<td>Number of particles per cell</td>
<td>5x5x5x2</td>
</tr>
<tr>
<td>Box length</td>
<td>6x6 per node</td>
</tr>
</tbody>
</table>

Table 17: System setup details for the analysis done in Stampede (KULeuven)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 15.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Intel MPI 5.0.2</td>
</tr>
<tr>
<td>Extrae version</td>
<td>extrae_lite 0.2</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-openmp -fno-exceptions -fp-model fast=2 -O3 [-xHost, -mmic]</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16/16 (Xeon/Xeon Phi)</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1/15 (Xeon/Xeon Phi)</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>KMP_AFFINITY=compact</td>
</tr>
<tr>
<td>Offloaded ranks per MPI process</td>
<td>1</td>
</tr>
</tbody>
</table>

3.5.1. Number of MPI calls

Figure 38 shows the how many MPI calls are performed in each phase of the code. In a perfectly scalable code the height of the bars should remain constant even with an increasing number of nodes. In the present case, changing the number of nodes of the simulation does not affect the number of MPI calls performed in the field solver in the Xeon processors nor in the moment gathering phase in the Xeon Phi coprocessor. However, the number of MPI calls increases with an increasing number of nodes for the particle mover.

Cluster to Booster and Booster to Cluster communications take only around 3 seconds of run time, and for simplicity are not shown in the present figure.
3.5.2. Time spent on MPI

The variation shown in Figure 39 corresponds to the time that MPI calls take to perform their tasks in each one of the phases. The first thing that attracts the attention is the constant high value of the “Moments Xeon” phase. This phase of the code is located in the field solver. Its main job is to receive the moments calculated in the particle solver and distribute them in the correct arrays of the field solver. The reason why this value is so high is because, for most of the time, the field solver has to wait idle for the moment data to be transferred from the particle solver. In the present conditions the iPic3D code has an unbalanced load with a higher computational cost in the Booster side. This unbalance is transformed into waiting time in the Cluster side of the code.

In the same figure, the time spent in MPI calls increases in the particle solver side of the code when the number of nodes increases. The rate of increase is much higher for the particle mover than it is for the moment gathering phase. While the former increases three fold, the later increases only two fold, when comparing the runs in one node to the runs on 128 nodes. The figure also reveals that the times in the field solver MPI calls are so low that they almost do not appear in the chart. This is once again due to the total run times observed in previous sections: scaling is not the biggest issue, but the total run time and the total time spent in MPI calls is too high for the sections of the code that run in the Xeon Phi coprocessor.
### 3.5.3. Useful time

Once again in Figure 40 the measurements show that all the phases have a constant „useful time“ defined as the time spent in each process outside MPI or OpenMP calls. The moments section of the code, in the coprocessor, takes the largest useful time, followed by the particle mover. These numbers point again to the bad performances of the algorithms in the Xeon Phi coprocessor.

One intriguing measured value is the useful time in the moment reception section of the field solver. The value „Moments Xeon“ decreases by half every time that the number of nodes is doubled. This routine only takes care of receiving the moments from the particle solver and distributing them in the local arrays in the Xeon processes. Unfortunately we did not have the time to analyse the origin of this measurement, but we suspect that it is related to the idle waiting time observed in the field solver phase.
Figure 40: Variation of useful time with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (KULeuven)

3.5.4. Number of cycles

Figure 41 shows once again the same behaviour as the one presented in the previous subsection: the code running on the Xeon Phi coprocessor uses a much larger number of cycles to perform its tasks, in particular the moment gathering consumes up to nine times the number of cycles than the iterative field solver. The “Moments Xeon” measurement that counts the cycles used during the reception of the moments in the field solver, behaves again in a peculiar way. This value decreases again by half when doubling the number of nodes. As in the previous subsection we are not certain if this measurement is an artifice of the metrics gathering or is actually an effect that arises because of the waiting time observed in the field solver.
Figure 41: Variation of number of cycles with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (KULeuven)

3.5.5. Number of instructions

This quantity follows the same behaviour observed in the two previous subsections. This time however it can be observed that the number of instructions called in the field solver (“Fields Xeon”) is larger than the number of instructions called in the particle pusher. The instructions called by the iterative linear field solver are more complex than the simple particle transportation instructions. However, as we observed in the previous subsections, the final number of cycles used in the particle solver running in the Xeon Phi processor is much larger.

It should be noted that the number of instructions reported here corresponds to the number of instructions executed by the master thread in OpenMP environments. In order to simplify the gathering of metrics, extrae_lite assumes a constant number of threads for the analysis. The real number of instructions of the particle solver is close to 15 times higher, as we used 15 threads. The parallelisation of the moments is not as well balanced as the parallelisation of the particles, and therefore the total number of instructions is significantly higher than the one depicted on the figures. However, the ratio is not as high as for the particle solver. These effects have no impact on the analysis of scalability of iPiC3D, as long as the number of threads is kept constant.
Conclusions

The iPic3D code is the perfect candidate for the DEEP architecture: it is composed of two main solvers: the first, the field solver, performs an iterative resolution of a linear system in multiple processors, the second, the particle solver, transports millions of individual and independent particles across a domain.

The division Cluster-Booster was evident from the beginning of the project and the tests performed in the benchmark section showed that, as expected, the field solver uses a compute time that is in the same order of magnitude than the communications time. The particle solver on the other hand required communications only to move particles from one subdomain to the next.

In addition to the Cluster-Booster division, the main improvements of the code consisted in the addition of multithreading calls in the particle solver using OpenMP, ensuring memory locality using adequate dynamic memory allocation, using a new AoS approach to keep particle properties close together in memory, and using VPU registers with intrinsic calls for the vector processing of the MIC coprocessor.

Weak scaling tests allowed isolating communications from computation. It was observed that the overall application scaling is good for all architectures, including the Xeon-only and the Xeon+Xeon Phi clusters.

However, we are disappointed with the performances of the code on the Xeon Phi coprocessors. The tests performed in Stampede and MareNostrum3 show that, when using the Xeon Phi coprocessors, the code can be four to five times slower that the Xeon-only case. A detailed analysis of the compute times show that, in the particle branch of the code, only the computations of the particle mover give times comparable to those observed in the Xeon-only runs. The moment gathering phase and the communications in the particle mover are very slow in the Xeon Phi.

We conclude that there is a need for a more basic approach to the coding of the particle solver for the Xeon Phi architecture. We consider that the algorithms are not straightforward.
transferable from the Xeon to the Xeon Phi architecture but that there is still room for improvement at the core of the solver. In addition, the communications subroutines of the particle branch were developed for CPU clusters and are not adapted for the MIC architecture. The buffering methods and the book keeping routines were not designed with performance, memory or vectorisation in mind. An important restructuring of the present version of the particle solver is required to improve the performances.

On the other side of the code, in the field solver, it is observed that performance is limited by the ratio between computing time and communications time. This ratio is almost equal to one. To increase this ratio, the number of cells per processor must be kept between 1000 and 30000. However, in the particle solver, a high number of cells mean a high number of particles. The available memory is lower in the Xeon Phi cards, so at the end there is a limitation on the number of cells that each process can allocate.

In addition, in the current version of the Cluster-Booster division, there is a one-to-one mapping between the processes in the field solver and the processes in the particle solver. This means that the number of cells in one process in the Cluster is equal to the number of cells in one process in the Booster. Now let’s also remember the field solver spends most of its time waiting for information from the particle solver. It would be interesting to find a way to perform a one-to-many processes mapping between the Cluster and the Booster. This would load balance the computational charge in each side and give a lower number of cells per process in the Booster. This would save time and memory but would require additional Booster-Cluster communications, but it has been shown that these communications take an almost negligible time compared with other phases of the code.

Up until now the use of Xeon Phi coprocessors has not shown to be very efficient, but we believe that there is still room for improvement. It requires a delicate refactoring of some of the core sections of the particle solver. However the particle-in-cell method is simple enough as to take a leap and build from the bottom-up a new particle solver specially adapted to the Xeon Phi architecture. Intra-MIC and inter-MIC communications need to be studied in more detail to find a simple solution that does not fill the system with MPI calls.

Finally we think that the developments included in the code have improved its performance on Xeon-only platforms, and although such is not the objective of the present project, it is important to note that these advances have a direct positive impact in our scientific research.
4. Task 8.3: Climate simulation (Task leader: CYI)

At the Cyprus Institute the global climate model ECHAM/MESSy Atmospheric Chemistry (EMAC) is used to study climate change and air quality scenarios with a focus on the Eastern Mediterranean and the Middle East. Climate and atmospheric chemistry simulations are compute-intensive applications and create very large data sets. The EMAC model runs on several platforms, but its large memory demands make it currently unsuitable for massively parallel computers.

Studying the chemistry and physics of the atmosphere a complex Earth-system simulation is employed coupling a global circulation model (GCM) with local physical and chemical models. Running on parallel supercomputers the global meteorological processes are strongly coupled with high communication demands while the local physical processes are inherently independent with high computation demands. The DEEP architecture is naturally suited to these tasks with global components running on the Cluster nodes exploiting the high-speed Xeon processors and local components running on the highly-parallel Xeon Phi coprocessors. By balancing communication versus computation the DEEP concept provides a new degree of freedom allowing us to distribute the different components at their optimal parallelisation.

4.1. Overview of application structure

The EMAC model comprises two parts, the meteorological base model ECHAM, using a nonlocal, spectral algorithm with low scalability, and the modular framework MESSy, linking local physical and chemical processes to the base model with high scalability. While the number of processors used for the base model is limited by the non-local spectral representation of global physical processes, local physical and chemical processes described by framework submodels run independently from their neighbours and scale well.

4.1.1. Phases

The implementation of EMAC in the context of the DEEP project comprises three phases, the base model ECHAM integrating the dynamical state of the atmosphere, the MESSy framework that interfaces to \( n - 1 \) submodels calculating physical and chemical processes, and the MECCA submodel that computes the chemical kinetics of the homogeneous gas-phase chemistry of the atmosphere.

The ECHAM base model runs in parallel in the distributed-memory paradigm using the Message Passing Interface (MPI) library for communication; the MESSy framework inherits the parallel decomposition defined by the base model. While ECHAM has been shown to be able to exploit the shared-memory paradigm using the Open Multi-Processing (OpenMP) library, no such effort had been undertaken for the MESSy model so far.

It is, however, currently not possible to delegate the whole MESSy subsystem to full multi-threaded execution as some physical processes are naturally modelled in a column-based approach, and are strongly dependent on the system states at their vertically adjacent grid points. The implementation of submodels simulating these processes consequently relies on the column structure inherited from the base model. Furthermore, even a coarser column-oriented multi-threaded approach is hindered by global-variable interdependencies between submodels.

Describing homogeneous gas phase chemical kinetics, the MESSy submodel MECCA executes independently of its physical neighbours and is not limited by vertical adjacency relations. As more than half of the total run-time is spent in MECCA for a typical model
scenario, it seems adequate to concentrate on the MECCA kernel with strong algorithmic locality and small communication volume per task. As sketched in Figure 44 the current implementation of MECCA is delegated to the Booster using a task-based approach while both ECHAM and the remaining MESSy submodels are executed on the Cluster in the distributed-memory paradigm (Figure 43).

![Diagram](image)

**Figure 43:** Phases of EMAC. Green phases run on the Cluster, blue phases run on the Booster (CYI)

### 4.1.2. Dominant factors

Implementing a spectral model of the dynamical state of the atmosphere the ECHAM phase comprises six transform and six transposition operations in each time step, as seen in Figure 44. While communication and computation largely overlap, the data (size scales with the square of the model resolution) is transposed in an all-to-all communication pattern, and this phase is dominated by the network bandwidth.

![Diagram](image)

**Figure 44:** The ECHAM main application loop. MESSy replaces and enhances the grid point calculations marked in yellow (CYI)

Figure 45 displays one time step traced with Extrae/Paraver starting with the end of the grid point calculations of the last time step – in which most processors are already idle (orange) due to load imbalance and waiting for process 14 (blue) to finish running –. This is followed by the transpositions and Fourier and Legendre transformations (magenta), which execute simultaneously as further analysis showed. After the transpositions a short interval with all processors running (blue) can be identified with the time step integration in spectral space, followed by the inverse transformations and transpositions and transport calculations in ECHAM.
While the pattern described so far repeats towards the end of the displayed interval, the major fraction of the time step is spent without communication, running (blue) or waiting (orange) in calculations in MESSy in grid space. The MESSy phase comprises some 30 submodels that are tightly coupled by exchanging the atmospheric observables using global variables. Model performance depends largely on a virtual longitude run-time parameter exploiting cache line adjacency of the grid point variables. Investigations during the first phase of the project determined the load imbalance visible in Figure 45 to be caused by chemical processes computed in the MECCA submodel.

The observed load imbalance is one of the main factors determining application scalability. It is caused by an adaptive time-step integrator solving a system of differential equations. As the stiffness of these equations representing homogeneous photochemical reactions varies by up to two orders of magnitude due to changes in the intensity of sunlight the adaptive integrator demands varying amounts of run time accordingly (described in more detail in Section 4.1.3).

In the MECCA phase the algorithmically complex adaptive time-step differential equation integrator operates on chemical concentrations of only a few kilobytes size per grid point. Yet, as seen in Figure 46, this phase consumes the major proportion of the total execution time, it is compute-bound and an obvious candidate for offloading to accelerators.
4.1.3. Scalability considerations

In the initial preparatory phase of the project the EMAC application has been ported to JUDGE and JUQUEEN, and a representative benchmark with a horizontal resolution of 128 grid points in longitudinal and 64 grid points in latitudinal direction with 90 vertical levels and a spin-up period of 8 simulated months has been compiled, frozen and packaged to be used for measurements. Benchmarks on both JUDGE and JUQUEEN have been run and examined for the application analysis in Deliverable D8.1. Table 18 details the experimental setup for the results shown in this section.

<table>
<thead>
<tr>
<th>Table 18: Experiments setup for the scalability considerations subsection (CYI)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scaling</strong></td>
</tr>
<tr>
<td><strong>Number of columns</strong></td>
</tr>
<tr>
<td><strong>Number of grid points</strong></td>
</tr>
<tr>
<td><strong>Number of chemical species</strong></td>
</tr>
<tr>
<td><strong>Spectral resolution</strong></td>
</tr>
</tbody>
</table>

At the start of the project EMAC was benchmarked with different numbers of processors on JUDGE in order to determine the run time behaviour of the total application. As shown in Figure 47 the application scales up to 384 processors, at higher numbers the performance decreases. Parallel execution speed is determined by the balance of three factors, computation, communication, and load imbalance. The benchmarking setup for the JUDGE cluster can be seen in Table 19. While the computational resources increase with additional processors and therefore increase the application performance, communication demands diminish the positive effect of the additional processors. Additionally, increasing the granularity of the total workload also increases the load imbalance.
Table 19: System setup details for the analysis done in the JUDGE system (CYI)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 13.1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Parastation/Intel MPI 5.0.27</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O3 –fp-model source –r8 –align all</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>24</td>
</tr>
</tbody>
</table>

Performance of EMAC on JUDGE

Figure 47: Wall time for one simulated day versus the number of nodes on JUDGE, at the beginning of the project (CYI)

While the number of processors used for the distributed-memory part of the code is limited by the scalability of the non-local representation of global physical processes in ECHAM, the local processes in MESSy running independently from their neighbours can scale very well. The MESSy subsystem has not been designed for multi-threaded execution, though, and contains non-local code due to characteristics of the physical processes and algorithmic design decisions. Some physical processes are naturally modelled in a column-based approach, because they are strongly dependent on the system states at vertically adjacent grid points, e.g. sunlight intensity at lower grid points depending on the absorption at higher grid points, and precipitation depending on the flux of moisture from vertically adjacent grid cells. Sub-models simulating these processes consequently rely on the column structure implemented in the current model.

In the existing distributed-memory parallel decomposition, the three-dimensional model grid is split horizontally using two run-time parameters, setting the number of processes in latitudinal and longitudinal direction. As work is distributed independently for each direction, a rectangular decomposition is maintained. Restricting the individual work packets to a rectangular shape reduces the amount of information to be exchanged with neighbouring grid cells, thus limiting the communication demands.

Examining the benchmark calculations the physical load-imbalance caused by photo-chemical processes in the lower stratosphere and natural and anthropogenic emissions appears in the run time spent for each grid point. In Figure 48 the maximal MECCA kernel execution wall-time for one grid point in each column differs by up to a factor of four. The load imbalance is caused by the adaptive time-step integrator solving the differential equations that describe the chemical equations computed in the MECCA sub-model. The strongly varying light intensity...
at sunrise and sunset and night-time emissions lead to stiff differential equations that require more intermediate time steps with derivative function evaluations and increase the computational load by up to one order of magnitude.

Figure 48: Maximal MECCA kernel execution wall-time in microseconds. The adaptive time-step integrator shows a non-uniform run time caused by stratospheric photochemistry and natural and anthropogenic emissions (CYI)

While at high parallelisations the load imbalance becomes a limiting factor, the factors determining scalability in absolute numbers in Figure 49 are communication and computation. For the ECHAM phase (blue) the balance between these tilts at around 8 nodes when the communication demands of the underlying spectral model involving several all-to-all communication patterns start to dominate. The MESSy phase computing local physical processes gains considerably in employing more processors, while the MECCA phase benefits even more due to an MPI barrier introduced for the measurement diminishing the load balance sustained at low parallelisation.
Figure 49: Impact on run time of each phase of EMAC, when running on MareNostrum 3 (CYI)

In Figure 50 the turning point at around 8 nodes is clearly apparent; 16 nodes is commonly used in production runs of the EMAC atmospheric model as a scientific application balancing efficiency and time-to-result.

Figure 50: Percentage of run time of each phase of EMAC, when running on MareNostrum 3 (CYI)

4.2. Code modifications during the project

Over the duration of the project a number of changes were developed. The EMAC model atmospheric chemistry code (MECCA) of the new version of MESSy was taskified with an offload mechanism implemented using OmpSs directives. Furthermore, a shared memory parallelisation based on the OmpSs tasks directives was created. The Fortran code implementing the MECCA homogeneous chemistry submodel was refactored to decouple the OmpSs tasks. The distributed-memory offloading code was also redesigned to exploit shared
memory within the Xeon Phi many-core processors using two levels of nested tasks, in collaboration with BSC.

The new task-based MECCA implementation was optimised and we reduced the memory and network footprint of the distributed-memory offloading code by three orders of magnitude. To reduce the memory footprint by three orders of magnitude for off-loading tasks, the number of computational grid elements issued to MESSy is further split into individual elements for each task, by rearranging the grid point arrays in each time step to implement data locality at the grid-point-level, resulting in a massive total memory footprint reduction from 2.7 Mbytes down to 6.3 kbytes for each task. This was a result of refactoring both the data and code structures in MECCA. For example, at the benchmark resolution of T42L90MA a total number of 737 280 tasks is generated in each time step resulting in a total data size of 2 645 360 640 bytes (2.5 Gbytes).

Finally, during the duration of the project, and specifically after December 2013 there was a major new release of the EMAC model that saw rapid and wide adoption by the user community. We rebased our code base from the legacy (version 1.92) to the new updated version (2.42) and repeated the initial profiling and validation stages on the JUDGE cluster and later on the DEEP cluster. During the later stages of the project the model was ported to and all development effort was shifted to the MareNostrum 3 supercomputer to allow testing with Intel Xeon Phi accelerators on a production-size machine.

4.2.1. Threading

The MECCA submodel was refactored for the latest EMAC model version v2.42 through the creation of computational kernels for intranode parallelisation with shared memory tasks. An OpenMP implementation based on the OmpSs task offload directives was created. The implementation of the shared-memory code uses OmpSs as the programming paradigm for offloading locally to Xeon CPUs. The new version of EMAC, running ECHAM with MPI processes and MECCA with shared memory OmpSs tasks clearly outperforms the old EMAC using MPI (without offloading), over the full range of number of nodes tested, and continues to scale beyond the region where the original implementation scaling performance plateaus as can be seen in Figure 51 and Figure 52, showing the performance and parallel efficiency using threading on the DEEP Cluster. The benchmarking setup can be seen in Table 22.
4.2.2. Vectorisation

The computational core of MECCA is connected by an interface layer to the MESSy framework, integrating different submodel code and data structures into the ECHAM base model. It provides the gridpoint data as sub-arrays of the global simulation data—which have been re-arranged from their native longitude and latitude coordinates into a virtual longitude and an outer index variable—counting the virtual longitude blocks and assuming the role of a virtual latitude. The virtual longitude exploits cache line adjacency on non-vector architectures and serves as run-time vectorisation parameter for all MESSy submodels.
For the MECCA submodule an integrator kernel has been created that can be offloaded onto worker threads running on the main processor or hardware accelerators. The chemical mechanism is compiled by the KPP implementing a domain-specific language for chemical kinetics. The integrator kernel operates on the variables of one grid-point describing the local state of the atmosphere and the integrator parameters determining the solution of the chemical equations. As the kernel variables are passed as one-dimensional sub-arrays of global, four-dimensional arrays extending along the virtual longitude, the vector variables are transposed to extend contiguously along the dimension of chemical species.

**Performance of the vectorisation of MESSY**

![Performance of the vectorisation of MESSY](image)

In order to estimate the run-time effect of the changes in the code, the application was benchmarked on the DEEP Cluster using the Xeon main processors without vectorisation as baseline measurement of 548.65 seconds per simulated day. Compiling with the auto-vectoriser enabled for the AVX instruction set extensions decreased the run time to 466.40 seconds, resulting in a first speed-up of 1.18. Examination of the optimisation report identified several unaligned array accesses, which were solved using compiler directives and introducing aligned leading dimensions at 64-byte boundaries for multi-dimensional arrays as needed for the instruction set of Intel Xeon Phi. These changes improved the total application performance to 349.50 seconds per simulated day for a second speed-up of 1.33 achieving a total speed-up of 1.57.

### 4.2.3. Cluster/Booster division

The EMAC model comprises a nonlocal meteorological part with low scalability (ECHAM) and physical processes (MESSy), including chemical processes with high scalability (MECCA). A priori, the model’s structure naturally suits the DEEP Architecture using the Cluster nodes for the nonlocal part and the Booster nodes for the local processes.

The implementations of individual MESSy submodels simulating the physical processes rely on the column structure corresponding to their vertical physical dependencies. A coarser column-oriented multi-threaded approach is hindered by global-variable interdependencies between submodels. For this reason a hybrid Cluster-Booster approach was selected with computationally demanding MESSy sub-models offloaded for acceleration on the MIC.
As discussed in section 4.1.2, a detailed analysis of the EMAC run-time behaviour using Scalasca and Extrae/Paraver has identified that the MECCA submodel consumes a major proportion of the execution time, does not participate in communication, and is independent of adjacency constraints. It is thus well suited to be delegated to the Booster employing the large dynamical pool of accelerator resources provided by the DEEP concept for load balancing of the heavily varying computation demands discussed in section 4.1.3.

As described in Section 4.2.1, refactoring the MECCA KPP integrator kernel, reduced the memory and network footprint of the distributed-memory offloading code by three orders of magnitude. Furthermore, and in close collaboration with BSC, the distributed-memory offloading code was redesigned to exploit shared memory within the Xeon Phi many-core processors by nesting an OmpSs shared memory region within Cluster-to-booster tasks encompassing variable, runtime-defined number of individual gridpoint calculations. Thus, the number of tasks to be sent to the Booster can be controlled and optimised for each architecture, and host-specific configuration allows for optimum task size based on bandwidth, reducing task communication overheads.

With this approach, the specifics of the DEEP system architecture, and in particular the hardware present in MIC coprocessors is exploited by massively parallelising the chemistry calculations at the cell level and offloading to the Booster exposing a significant amount of thread parallelism. At the same time the load imbalance observed in MECCA is automatically alleviated through OmpSs’ dynamic load balancing by selecting a sufficiently fine task size and decoupling the model-domain location of the grid point from the task execution on the physical CPU. The Cluster-Booster offload in the full application is not functionally operational due to instabilities when running under the OmpSs library environment and as such no benchmarking results are readily available. Tests conducted to identify the root cause and simulations to enable the projection of the attainable computational performance are outlined in Section 4.3.

4.2.4. Other optimisations

The EMAC code base used for the DEEP project was changed from the EMAC development cycle 1 (ECHAM v5.3.01/MESSy v1.92) to the EMAC development cycle 2 (ECHAM v5.3.02/MESSy v2.41 and MESSy v2.42). The code re-base reflects to a large part the shift in the EMAC user community, as the new release is now widely adopted and more importantly, development of the chemistry code continues only for the 2nd development cycle. It is expected that the legacy development cycle 1 will be deprecated in the near future. Finally, changing to the new version allows profiting from improvements done in the vectorisation of the chemistry code by the developers of the application.

The EMAC model source code was significantly restructured for development cycle 2, including an improved and extended infrastructure for the basemodel independent coupling of process-submodels, new diagnostic capabilities for the evaluation with observational data, and an improved atmospheric chemistry setup.

Switching the code base to using MECCA instead of MECCA1 allows for a more extensive use of vectorisation when computing chemistry. In MECCA (based on KPP 2.2), the KPP Post Processor (KP4) is used to modify the KPP generated Fortran95 code, including modifications of the loop structure, and optionally vector blocking with a user-specified vector length, to achieve an improved run-time performance and can potentially be exploited in the SIMD architecture of the MIC. The core of KP4 consists of a Fortran95 parser and is implemented in C++. The drawback is the added code complexity and the challenge of instrumenting machine-generated code.
Due to the large code base that links multi-institutional codes, containing in many cases legacy and unmaintained segments, and machine-generated code, porting the EMAC development cycle 2 application to a new architecture and/or compiling with a new compiler suite is not a trivial task but one that requires considerable effort. To achieve compilation of all code objects and linking of executable binaries, supporting libraries had to be ported and the EMAC code base itself had to be modified, including the removal of numerous instances of OpenMP parallel pragmas in the ECHAM code that are used in pure ECHAM but ignored by design in ECHAM/MESSy and cause conflicts with the OmpSs syntax.

In an attempt to improve the performance of the MECCA submodel, trial runs were performed with the default Rosenbrock solver replaced by a version developed at CYI, placing a higher hard-coded limit on the maximum number of the integration step increase factor, itself heavily based on the Rosenbrock-Mainz implementation, which was developed at the Max Planck Institute for Chemistry (MPIC) Mainz, and that was shown to achieve speedups in the chemistry calculations without impacting the correctness of the results. Unfortunately, there were problems running with the new solver, namely lookup errors in the required MESSy convection sub-model within a few model days of initial startup, most likely related to numerical instability from unphysical dynamical values produced. The problem is under investigation, and the current working hypothesis is that the issue arises from the introduction of aerosol phase chemistry and its coupling with the atmospheric dynamical state above the troposphere in MECCA.

The part of the application that is offloaded to the Booster relies on a preprocessor that transforms code written in a domain specific language into Fortran for the chemical interactions in the atmosphere. Changes were required to be made to the preprocessor to effectively parallelize the code, rather than changes in the application’s code, to allow for the chemical species and mechanism to be defined by the end-user.

Unfortunately the full application Cluster–Booster offloading mechanism, developed as part of the DEEP project, was not available for benchmarking on all platforms tested (DEEP cluster, MIC-login, MareNostrum 3) because of instability of the underlying software stack and sensitivity to compiler and library releases. As part of these efforts to port and benchmark, in close cooperation with by the BSC, Intel and FZJ support teams, a number of bugs were identified and resolved in both the application and middleware. Nonetheless a full stable execution of the application in offload mode could not be achieved. The task was also hindered by the unavailability of the DEEP system hardware (Booster, ASIC evaluator). Thus to project the expected attainable performance an ad hoc chemistry offloading emulator code was developed and tested on the MareNostrum 3 supercomputer. The results are discussed in Section 4.4.

### 4.2.5. Summary of changes

Table 20 summarises the changes done to the application in the time frame of the project.

<table>
<thead>
<tr>
<th>Threading</th>
<th>Taskification of the chemistry mechanism at the individual grid-cell level using OmpSs directives.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorisation</td>
<td>Refactored chemistry code to allow for memory adjacency of vector elements.</td>
</tr>
<tr>
<td>Offload</td>
<td>OmpSs taskification with remote offload with massive task parallelisation and optional two-stage offload.</td>
</tr>
<tr>
<td>Other</td>
<td>Changed model version to implement upstream developments in base model</td>
</tr>
</tbody>
</table>
4.3. Benchmarking

Different implementations of the code memory and workload divisions for the DEEP project were developed to test different aspects of the achievable performance run configuration on the proposed architecture:

- The EMAC model code v2.42 out-of-the-box, ran as the baseline performance indicator on all platforms tested, using the setup of the benchmark scenario, on the JUDGE cluster and subsequently on the DEEP cluster. This is the initial application performance evaluation (Section 4.1.3).
- A shared-memory code using OpenMP, to test threading changes on the DEEP cluster. Results were presented in Section 4.2.1.
- A distributed-memory offloading code using OpenMP and the Nanos runtime on Xeon Phi accelerators ported to the MICLOGIN test workstation using the KPP integrator module as computational task using the refractored code to eliminate strided memory access. This version did not provide benchmarking results because of instability of the underlying middleware.
- The offload version was also ported to the BSC MareNostrum supercomputer utilizing Intel Xeon Phi accelerators, along with transferring all necessary initial and boundary condition data for testing and benchmarking purposes and compiling all necessary shared libraries (Section 4.3.1).

Development efforts on the DEEP Cluster and MareNostrum 3 were focused on obtaining a benchmark of the behaviour representative of the final target machine configuration. The benchmark scenario used is based on the T42L90MA model configuration corresponding to a horizontal spectral resolution that is equivalent to a grid of 128 × 64 points in longitude and latitude (NLAT * NLON) and with 90 vertical hybrid levels (NLEV) extending to include the middle atmosphere to a pressure level of 1 Pascal. The benchmark scenario covers both the troposphere in lower altitudes and the stratosphere at higher altitudes where a detailed, localised benchmark analysis had previously detected a large photochemical load-imbalance.

The chemical mechanism implemented in the new benchmark scenario is derived from the version that was used to validate the new development cycle of the model for scientific result output. The detailed description of the chemical specification used to validate the model is referred to as the ECHAM5/MESSy2–02b emissions scenario, applicable between 1998 and 2005 or with a different namelist set-up between 2005 and 2010. It includes complete gas chemistry and an aerosol mechanism, along with a full suite of MESSy sub-models. The experimental setup is the same as described in Table 18.

4.3.1. Xeon Phi platforms

The offloading mechanism was tested on the MareNostrum 3 supercomputer using not the full model, due to instabilities of the OpenMP experimental setup, but with an ad hoc chemistry mechanism emulator. The emulator creates in memory and transfers between the Xeon CPU and MIC (emulating the cluster and booster) the exact same data structures as the full application to test the offload mechanism through OpenMP. Individual chemistry tasks do not vary in size in the emulator but are fixed and controlled for benchmarking purposes, thus also emulating load balancing.

A version with task workload identical to the real-world application, through loading in memory previously stored chemical species concentrations and reaction rates from full model
simulations, was also developed. However, this version was found to be unstable when running under the OmpSs library environment and as such no benchmarking results are readily available. This was also hindered by the fact that only certain Intel compiler versions were compatible, requiring a special runtime environment to be created by BSC to support the application. An investigation into the causes of the sensitivity to the compiler version points to issues with the handling of source precision, but has not yet yielded any conclusive results.

In order to test the offloading mechanism in MECCA was emulated as a test case with tests conducted for different computational kernel task length sizes from 1ms to 1s, emulating variable model chemistry complexity (the performance is shown in Figure 54 and the equivalent parallel efficiency in Figure 55). The benchmarking setup can be seen in Table 21.

Table 21: System setup details for Xeon Phi platforms (CYI)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 13.0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Intel MPI 5.0 Update 1</td>
</tr>
<tr>
<td>Mercurium and nanox versions</td>
<td>Mercurium 1.99.6 and Nanox 0.9a</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>--ompss -O2 --Wn,-fp-model,precise,-r8,-align,all</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16 (Xeon)</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1/1-240 (Xeon/Xeon Phi)</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>Default, as controlled by Nanox</td>
</tr>
<tr>
<td>Offloaded ranks per MPI process</td>
<td>1-60</td>
</tr>
</tbody>
</table>
Performance of the Xeon Phi offload on MareNostrum 3

Figure 54: Performance of the two-stage offload in MareNostrum 3 (CYI)

Parallel efficiency of the Xeon Phi offload on MareNostrum 3

Figure 55: Parallel efficiency of the two-stage offload in MareNostrum 3 (CYI)

4.4. Analysis of theoretical improved performance

The system setup for running the strong scaling experiments (simulation details in Table 22) with the EMAC model can be seen in Table 23.

Table 22: Experiments setup for the analysis subsection (CYI)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Strong scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of columns/boxes</td>
<td>8192</td>
</tr>
<tr>
<td>Number of grid points</td>
<td>737280</td>
</tr>
<tr>
<td>Number of chemical reactions</td>
<td>139 (318 reactions)</td>
</tr>
</tbody>
</table>
Table 23: System setup details for the analysis done in the DEEP system (CYI)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 13.1.2 in DEEP systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Parastation MPI 5.1.1 in DEEP systems</td>
</tr>
<tr>
<td>Mercurium and nanox versions</td>
<td>Mercurium 1.99.6 and nanox 0.9a in DEEP systems</td>
</tr>
<tr>
<td>Extrac version</td>
<td>3.0 in DEEP systems</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>“O3 –fp-model source –r8 –align all” in DEEP systems</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>1-16 in DEEP systems</td>
</tr>
<tr>
<td>Offloaded ranks per MPI process</td>
<td>16</td>
</tr>
</tbody>
</table>

4.4.1. Attainable Performance

In order to project the performance of the full DEEP System Xeon-based measurements on the DEEP Cluster were combined with Xeon Phi-based measurements on MareNostrum 3 and the Xeon-based metrics measurements on MareNostrum 3. As displayed in Figure 56 the DEEP Cluster reference data weighted by the relative factors for each phase derived from the metrics measurements exhibit a performance maximum at 8 nodes for the sum of ECHAM MPI, ECHAM and MESSy times representing a good estimate for the optimal parallelisation of the Cluster phase. This estimate of 375 s per simulated day for the low-scaling Cluster phases was used to extrapolate the Xeon Phi data retrieved from MareNostrum 3 where benchmarks using one node had been run with varying numbers of processing elements within one Xeon Phi processor.
While the number of Booster nodes required to attain similar performance to the original distributed-memory based implementation corresponds to regular accelerator architectures with individual boosters directly attached to cluster nodes, the projected DEEP performance scales beyond the optimal performance achieved so far. The EMAC atmospheric chemistry global climate model seems therefore well suited to exploit an architecture providing considerable more hardware acceleration than provided by regular systems.

### 4.5. Analysis of metrics for WP9

The extrae_lite tool, developed by BSC for the purposes of WP9, was used to trace some of the important metrics of the EMAC model. Table 24 is a summary of the computational load of this test, and Table 25 gives a summary of the system setup used to run the code on MareNostrum 3. Measurements were performed for each one of the three phases of the code.

**Table 24: Experiments setup for the analysis of metrics for WP9 subsection (CYI)**

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Strong scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number of columns</strong></td>
<td>8192 columns with 90 levels</td>
</tr>
<tr>
<td><strong>Number of grid points</strong></td>
<td>737280 grid points</td>
</tr>
<tr>
<td><strong>Number of chemical species</strong></td>
<td>139 species in 318 reactions</td>
</tr>
<tr>
<td><strong>Spectral resolution</strong></td>
<td>T42L90MA with 42 coefficients</td>
</tr>
</tbody>
</table>

**Table 25: System setup details for the analysis done in MareNostrum 3 (CYI)**

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 13.0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Intel MPI 5.0 Update 1</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O3 –fp-model source –r8 –align all</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16 (Xeon)</td>
</tr>
</tbody>
</table>
4.5.1. Number of MPI calls

The number of MPI calls displayed in Figure 57 varies for the ECHAM phase between one and two nodes because the parallel decomposition changes from a one-dimensional to a two-dimensional scheme. It represents the data transpositions described in Section 4.1.2 for the spectral transformations in the meteorological base model computations.

The MPI calls visible in the MESSy and ECHAM phases are MPI_Barrier calls introduced for the phase measurements. Neither phase participates in MPI communication.

![Variation of MPI calls with the number of nodes, per phase](image)

*Figure 57: Variation of number of MPI calls with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI)*

4.5.2. Time spent on MPI

As discussed in 4.5.1, the time spent in MPI calls in the MESSy and MECCA phases displayed in Figure 58 originates in the MPI_Barrier calls introduced for the phase measurements. However, it can be interpreted for the MECCA phase where it represents the accumulated load imbalance that is statistically balanced with other MECCA calls when run without barriers.

As is to be expected the time spent in the MPI calls for the all-to-all communication of the data transpositions in the ECHAM phase grows with increasing numbers of participating processes.
Figure 58: Variation of time spent on MPI with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI)

4.5.3. Useful time

The distribution of time spent in computation in Figure 59 exhibits the MECCA phase as the most compute-intensive with the MESSy phase following second. This distribution is reflected in the decision to offload the MECCA submodel to the Booster.

Figure 59: Variation of useful time with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI)

4.5.4. Number of cycles

Similar to the discussion in 4.5.3, the distribution of cycles in computation in Figure 60 exhibits the MECCA phase as the most compute-intensive with the MESSy phase following
second. This distribution is reflected in the decision to offload the MECCA submodel to the Booster.

Variation of number of cycles with the number of nodes, per phase

Figure 60: Variation of number of cycles with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI)

4.5.5. Number of instructions

Similar to the discussion in 4.5.4, the distribution of instructions in exhibits the MECCA phase as the most compute-intensive with the MESSy phase following second. This distribution is reflected in the decision to offload the MECCA submodel to the Booster.
Figure 61: Variation number of instructions with the number of nodes, per phase. The shown number is the average for all the iterations and nodes (CYI)

4.6. Conclusions

The global climate model ECHAM/MESSy Atmospheric Chemistry (EMAC) is used to study climate change and air quality scenarios. The EMAC model is constituted by a nonlocal meteorological part with low scalability, and local physical/chemical processes with high scalability. The model’s structure naturally suits the DEEP Architecture using the Cluster Nodes for the nonlocal part and the Booster Nodes for the local processes. Different implementations of the code memory and workload divisions for the DEEP project were developed and benchmarked to test different aspects of the achievable performance run configuration on the proposed architecture. The use of the OmpSs API largely frees the programmers of implementing the offloading logic and, given that EMAC is developed and used in a large community working on all aspects of the model, facilitates adoption of the DEEP concept in the MESSy community.

The chemistry mechanism was taskified at the individual grid-cell level using OmpSs directives. The chemistry code was refactored to allow for memory adjacency of vector elements. Enabling the vectoriser achieves a total speed-up of 1.57 by aligning all arrays at 64-byte boundaries. The OmpSs taskification with remote offload allows for massive task parallelisation and the implementation of optional two-stage offload to control Cluster-Booster task memory size and optimum bandwidth utilisation. Finally, during the duration of the project the model version under development was changed to implement upstream developments in base model and chemistry mechanism.

The computational load imbalance arising from a photochemical imbalance is alleviated at moderate parallelisation by assigning grid points with differing run times to the each process and distributing the load over all processes. Due to the physical distribution of sunlight this load balancing does not require an explicit algorithm at moderate parallelisation, instead, the implicit assignment of the model grid in rectangular blocks suffices for this purpose. At higher numbers of processors this implicit load-balancing decreases and the resulting load imbalance has to be reduced by active balancing. The dynamic scheduling provided by the
OmpSs run-time system balances the computational load without a possible, but expensive prediction for the current time step.

With these approaches, the specifics of the DEEP System architecture, and in particular the hardware present in MIC coprocessors is exploited by massively parallelising the chemistry calculations at the cell level and offloading to the Booster exposing a significant amount of thread parallelism. At the same time the load imbalance observed in MECCA will be automatically alleviated through dynamic load balancing by selecting a sufficiently fine task size and decoupling the model-domain location of the cell from the task execution on the physical CPU.

Benchmark projections based on available hardware running the DEEP software stack suggest that the EMAC model requires the large numbers of Xeon Phi accelerators available in the DEEP architecture to scale beyond the current optimal performance point and exploit Amdahl’s law with the highly scalable grid point calculations while capitalising on the high performance and fast communication for the spectral base model on regular Xeon processors.

The changes in the code developed within the context of the DEEP project are currently backported to the main development branch of the application and are expected to contribute to the eventual adoption of MIC accelerated architectures for production runs, in the presently available implementations towards a full Exascale enabling platform.
5. Task 8.4: Computational fluid engineering (Task leader: CERFACS)

The target application for Computational fluid engineering is the CERFACS and IFP Energie Nouvelles code AVBP. It is a legacy code of over 300,000 lines of Fortran and C started over 20 years ago. At the start of DEEP the code scaled up to 8192 MPI tasks on BlueGene/Q and the objective was to modernize and extend the software to Exascale architectures with Many Integrated Core chips in mind.

The initial version of the code was version 6.2. In this release the code used a master/slave approach with sequential HDF5 I/O designed for scalar processors. Therefore it was imperative to modernize the data structures and adapt the coding for DEEP.

The first step required the full parallelization of the code by removing the master/slave paradigm and switching to a SPMD approach. I/O operations where refactored using parallel HDF5.

Then in two parallel phases, optimizations were done and are on-going for vectorisation and to introduce hybrid parallelism using OpenMP and OmpSs. These developments were performed in parallel branches and will be merged outside of DEEP.

The last change done in AVBP was the introduction of OmpSs offloading. Currently, the writing of temporals – used for diagnosing how the simulation is progressing – is offloaded to the Cluster. The writing itself does not pose a big obstacle. However, previous to this I/O operation, multiple MPI_Reduce and MPI_Allreduce functions are called, which are also offloaded, and therefore the cost of these collectives in large scale simulations is avoided.

Please note that throughout this whole section run time is defined as time per iteration for a 500 iteration simulation.

5.1. Overview of application structure

The following section describes the structure of the SPMD version of AVBP (version 7.0 and above).

5.1.1. Phases

As most codes, AVBP has 3 main phases: Initialization, computation and I/O.

The initialization phase consists on the declaration and input of data to compute, and the decomposition of the said data for parallel computing. During the initialization phase all exchanges use global MPI collectives (MPI_Gatherall[v], MPI_Scatterall[v], MPI_Broadcast). Data is read sequentially and broadcasted to all processes, which then select the data that they will keep. The data are windowed reducing any possibility of memory issues. Decomposition is performed via the ParMETIS4 library and distribution of the data is ensured via global collectives.

The computation phase comprises of the iterative loop that simulates the flow behaviour. It includes the numerical scheme, i.e. the boundary condition application. This phase uses two data structures vertices based data and element based data. Most computations are performed at the vertex level with simple loops on the mesh nodes. However, the numerical scheme in AVBP are based on the distributed residuals approach. Therefore in order to advance the iteration the numerical scheme requires element data. This element data is obtained by gathering the vertices value of a given element at the centre of the element. Element averaged data is computed by the scheme as well as gradients and fluxes and then redistributed to the
vertices to continue the simulation. Vertices shared between partitions are synchronised via asynchronous non-blocking bufferer communications (MPI_Isend/MPI_Irecv/MPI_Waitall).

The I/O phase comprises the output of data during computation and dumping a full 3d snapshot of the simulation when the computation phase is finished, allowing for restart where we left off. The I/O phase has two components: Full 3d snapshot dumping and periodic binary output of global values for scientific monitoring. This last component allows for simulation monitoring (checking whether the simulation is behaving according to plan) and for data mining purposes gathering simulation statistics. This I/O is not required from a computational point of view but from a physics one, and the dump frequencies is of about once every 100 iterations (a full case simulation requires hundreds of thousands of iterations).

This data gathering uses MPI_Reduce and MPI_Allreduce operations to compute statistics and dump global values to a single binary file. Besides that, in the I/O phase full 3D snapshots are collectively dumped via the HDF5 parallel interface.

Figure 62: Phases of AVBP. Green phases run on the Cluster, blue phases run on the Booster (CERFACS)

5.1.2. Dominant factors

The initialization phase is limited by the memory size, as well as by the performance of MPI collective calls.

The compute phase is compute bound. The data is clustered in groups of cells that ensure minimum memory access and yield 97% or more access for L2 in the computations.

Of course this is the case when the data per partition is sufficient (about 1000 thousand cells per partition). With less control volumes, communication becomes dominant and network will limit performance. The temporal I/O is limited by MPI reductions, which required full process synchronisation. The 3D filed I/O dump is also blocking, requiring MPI_Allreduce and MPI_Allgather operations.
5.1.3. Scalability considerations

Before DEEP, the application used the master/slave paradigm. Initialization and I/O were fully sequential and only computation was parallel. Additionally all communication was centralized on a master rank. Therefore initialization and I/O did not scale at all.

The computation phase scaled but was limited by the communication pattern used in a master/slave mode, with a centralized rank handling all communications. In this condition the code required at least 10,000 elements per partition to efficiently scale in the computation phase. The following graphs detail the time for each phase. For user reading ease, the I/O time embedded in the computation phase has been removed from the computational phase timer and explicitly shown as I/O. The test case chosen for the experiments is the so-called SIMPLE test case. It is a standard benchmark for combustion modelling based on the PRECCINSTA test rig and consists on the simulation of a simple gas turbine. This case is representative of current state of the art simulations.

Table 26: Experiments setup for the scalability considerations subsection (CERFACS)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Strong scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tetrahedron</td>
<td>2,958,592</td>
</tr>
<tr>
<td>Other details</td>
<td>SIMPLE TEST CASE (PRECCINSTA test rig)</td>
</tr>
</tbody>
</table>

The graphs bellow show the distribution between each phase for AVBP 6.2 (at the beginning of DEEP) and V7.0 (current AVBP distribution that includes the work from DEEP). Figure 63 to Figure 66 show the impact of each phase and their percentage of runtime for a 100 iteration simulation of the DEEP Cluster at JSC.

The drawbacks of the master/slave paradigm are evident in the graphs: the initialization phase takes a significant amount of time. The computation phase scales up to 128 tasks and then becomes communication dominated due to the centralized communications in the master.

Switching to SPMD allows the initialization phase to scale up to the 768 tasks. The scalability of the computation phase has also improved, and depends on the size of the input dataset. The I/O phase is not visible for AVBP 6.2 since the times scales are much larger for the other two phases. However the I/O phase is an order of magnitude faster with the PHDF5 approach that with the sequential I/O. The increased share of run time for the I/O justify the importance offloading this work so as to remove the synchronisation of all process to perform it.
Figure 63: Impact on run time of each phase of AVBP 6.2, when running on the DEEP Cluster (CERFACS)

Figure 64: Percentage of run time of each phase of AVBP 6.2, when running on the DEEP Cluster (CERFACS)
As mentioned in the introduction, the AVBP code is a legacy code with poor vectorisation and only MPI parallelisation. During DEEP an extensive overhaul of the application was carried out. This code modernization is a continuous work will be continued after the end of DEEP to adapt the application to the new architectures and programming models appearing in the road towards Exascale. During the DEEP project major tasks were performed. First, the switch to SPMD already mentioned and parallel I/O. Then, to reduce the I/O requirements of the application, in-code post processing capabilities were integrated. Additionally, threading
has been progressively introduced and changes to the algorithms have been operated so as to take advantage of the vector units in MIC.

Most optimizations have been introduced in the kernels used for the specific benchmark cases used throughout the project. Propagation to other simulation features is ongoing and will be carried out outside DEEP.

5.2.1. Threading

The initialization phase would benefit only marginally from threading, as it is negligible for long simulations. However, the impact on the computation phase is obvious. Given the time frame and resources in DEEP we introduced threading in the numerical scheme section of the computational phase. Propagation to the rest of the code is ongoing and will be completed after DEEP.

To introduce threading it was necessary to adapt the initialization phase to create the structures required in the computation phase. The SMPD approach of AVBP relies on domain decomposition. Figure 67 details the domain decomposition structure use for MPI communications. During the initialization phase the full mesh is segmented into domains by ParMETIS using the k-way algorithm resulting in balanced partitions. These partitions share duplicated nodes that are synchronized whenever an MPI communication is requested in the code.

A second level decomposition was introduced for each MPI domain to account for task execution. Figure 68 shows the structure for each sub partition. Each MPI domain is broken down in continuous sub-partitions each one become a cell group to execute the cell operations directly by the tasks as described in the numerical scheme paragraph.
Each group of cells shared nodes with its neighbours. However to avoid node duplication, these are assigned to a given sub-partition to whom each of the neighbours has a ghost dependency. The resulting execution scheme is described in Figure 69. Node data is embarrassingly parallel. However the gather/scatter operations entirely rely on the tasks execution scheme. When we arrive to the gather operation, ghost node value retrieval tasks are generated to synchronize sub-partitions, and then the independent sub-partitions are executed by tasks at the cell level. After that an inverse synchronization task is generated to again synchronize shared node values with the new cell value. If the sub partition contains an MPI communication frontier, the MPI synchronization is performed. If not, the node computations continue.

**Figure 68: Sub-partition decomposition structure for tasks (CERFACS)**

**Figure 69: Task execution diagram with OmpSs (CERFACS)**

Figure 70 and Figure 71 show wall-times and parallel efficiency of OmpSs compared to the MPI only execution. Wall-times enclose OmpSs task creation, task execution, and taskwait
barrier. Parallel efficiency is in the ratio $T_s / n.T_p$ where $T_s$ is the serial useful time, $n$ the number of cores, and $T_p$ the parallel wall-time.

Tests were performed on the so-called SIMPLE test case from the previous section, on 1 Xeon Phi 5510 P card (mic-native execution) of the MareNostrum 3 cluster. The MPI-only kernel is executed with a varying number of MPI ranks, whereas the OmpSs kernel is executed with 2 MPI ranks (fixed) and a varying number of OmpSs threads.

The code performs satisfactory and with similar runtime performance both in MPI and hybrid execution. The switch to SPMD has allowed the code to run on clusters equipped with Xeon Phi coprocessors, which was not possible before DEEP. Both the MPI and hybrid execution showed an excellent scalability up to the 60 tasks limit imposed by the physical core count. Using the virtual core count via hyper threading, however, yields a large drop in performance both in MPI and hybrid execution. The MPI+OmpSs execution yields a much worse results than the performance with MPI alone. The OmpSs execution behaves favourably compared to the MPI execution where it is at least as performant or better than full MPI without SMT.

The performance gap when using SMT is not seen on other platforms so far, and suggests a problem with the tasks placement. Profiling the application also shows that the ghost node synchronization tasks are executed in sequential order when using 120 threads. In this case the tasks become too small to overlap with computation or each other’s work.

**Figure 70: Performance of OmpSs threading for the computational phase in a Xeon Phi 5110 (CERFACS)**
5.2.2. Vectorisation

Vectorisation was introduced progressively on the code and has concerned mainly phase 2 (the computational phase). Analysis of the Vtune profiling and Intel Vector Advisor allowed for targeted optimization of hotspot kernels. The initial analysis of the code showed that autovectoriser was not vectorising efficiently. In fact, the -no-vec code executed 10% faster than using SSE4.2 instructions. On Xeon Phi, the performance of the original code is only slightly impacted by deactivating the SIMD option and the autovectoriser.

The initial work focused on the gradient computation that accounted for 20% of the inclusive computational time at the beginning of the project and replacing whenever possible redundant divisions inside the iterative loops (Figure 72). Progressive specialization of the DO loops was carried out by introducing hard coded values for loop indexes and using the forced vectorisation SIMD directive. To ensure application maintainability, code specialization was coupled with the FORCEINLINE directive reducing the amount of duplicated code to generate. The code is depicted in Figure 73.

Indeed, analysis of the vectorisation reports from the compilers suggested that in most cases loops could be vectorised but the compiler heuristics were too conservative and chose not to perform it. Adding the specialized hard coded values allowed the compiler to assess better the cost of vector operations, and the SIMD directive forced the compiler to vectorise in all cases the operations in question.
ELSE IF ( ncid==3 ) THEN

DO n=1,nlcell
   DO k=1,neq

      gradzc(k,1,n) = -1.0_pr*third + [ zc(k,1,n) + snc(1,1,n) + &
                                     zc(k,2,n) + snc(1,2,n) + &
                                     zc(k,3,n) + snc(1,3,n) + &
                                     zc(k,4,n) + snc(1,4,n) ]

      gradzc(k,2,n) = -1.0_pr*third + [ zc(k,1,n) + snc(2,1,n) + &
                                     zc(k,2,n) + snc(2,2,n) + &
                                     zc(k,3,n) + snc(2,3,n) + &
                                     zc(k,4,n) + snc(2,4,n) ]

      gradzc(k,3,n) = -1.0_pr*third + [ zc(k,1,n) + snc(3,1,n) + &
                                     zc(k,2,n) + snc(3,2,n) + &
                                     zc(k,3,n) + snc(3,3,n) + &
                                     zc(k,4,n) + snc(3,4,n) ]
   END DO
END DO

Figure 72: Sample code of the gradient routine. Specialized version for 3 dimensional elements with 4 vertices (CERFACS)

CASE (0)
   SELECT CASE (npnproduct)
      CASE (1)
         !DIRS FORCEINLINE
         CALL gather_o_cpy_flat_1D(nval, nval, 8, ielob, z, zobj)
      CASE (3)
         !DIRS FORCEINLINE
         CALL gather_o_cpy_flat_2D(nval, nval, 3, 8, ielob, z, zobj)
      CASE (5)
         !DIRS FORCEINLINE
         CALL gather_o_cpy_flat_2D(nval, nval, 5, 8, ielob, z, zobj)
      CASE (15)
         !DIRS FORCEINLINE
         CALL gather_o_cpy_flat_2D(nval, nval, 15, 8, ielob, z, zobj)
      CASE DEFAULT
         !DIRS FORCEINLINE
         CALL gather_o_cpy_flat_2D(nval, nval, npnproduct, 8, ielob, z, zobj)
   END SELECT

Figure 73: Sample code of the gather routine. Specialized version with forced inline directives (CERFACS)

Lastly, gather-scatter operations were rewritten with aligned SIMD operation using hard coded specialized loops (Figure 74). This yielded the current performance reported in Figure 75.

DO n=1,nlcell
   DO no = 1, nvert
      !DIRS SIMD
      DO k=1,npnproduct
         zobj(no * npnproduct + k, n) = z(ielob(no,n) * npnproduct + k)
      END DO
   END DO
END DO

Figure 74: Sample code of the gather operation. Aligned SIMD execution kernel (CERFACS)

Figure 75 and Figure 76 show the impact of the modifications to the code to use the vector units of the processor in a single MIC using 60 tasks (MPI only). The computational phase executes 2.4 times faster than in the original code on Xeon Phi.
Profiling using Vtune showed that we currently use a maximum of 47% of the vector length in Xeon Phi in our operations. This is due to the structure of the code where large amounts of small loops are executed. To remove this limitation and use the full vector length work began over 9 months ago to swap the array indexes and use small amounts of large loops. This work is ongoing and requires rewriting 90% of the application. It will be continued and completed after DEEP.
5.2.3. Cluster/Booster division

The AVBP application scales well and requires little memory. There are no specific kernel that uses large amounts of computing time and the code benefits from the extreme parallelism of today’s machines. Therefore, we decided to fully execute the code natively in the Booster.

However, that does not mean that the code cannot benefit from a Cluster/Booster division. The I/O performance of first generation Xeon Phi coprocessors, as well as the current iteration of the DEEP architecture itself, push towards offloading the I/O from the Booster to the Cluster. Therefore, it was decided to reverse offload the I/O. This can be overlapped with the next iteration of the compute loop. I/O in AVBP is performed in two different phases, as described before, Offloading of the temporal data has been successfully implemented with the help form JSC. Even though the amount of data written in this step is limited, it requires costly MPI collectives, which become problematic for large node counts. Data gathering, collective operations and data dumping to the disk have been, removing them out of the critical path of the application’s workflow. The other aspect of the I/O phase is the full dataset dumping. The 3D snapshot writing is a complex process which has not been offloaded yet. However the work has been started with the implementation of a hierarchical I/O scheme where processes are divided into groups. Each group has a single I/O root process that gathers the data from the others and is the only one that performs the MPI-IO call via HDF5. The following step consist on switching from a computing process to an external cluster process for the I/O root process selection and will be studied after DEEP.

Despite its implementation for the temporal data, the offload has not been benchmarked extensively. This was not possible due to interactions between OmpSs, the MPI runtime, high speed communications on Xeon Phi, which resulted in instabilities in both MareNostrum 3 and Stampede.

5.2.4. Other optimisations

Working with Xeon Phi and BlueGene/Q platforms showed that using ParMETIS on large core counts becomes problematic as it relies on heavy MPI_Alltoall operations and has a large memory footprint. Therefore, initialization-recycling techniques were implemented on the initialization phase. In this case, when re-starting a simulation using a previous result with the same amount of resources, data that have not changed – such as domain decomposition maps and wall distance maps – are read and not re-generated. This allowed for offline and portable generation of these maps. Using 768 MPI tasks, the initialization phase is 4.5 times faster using the recycling method than the standard one (it is also 12.1 times faster than the master initialization method from version 6.2).
5.2.5. Summary of changes

Table 27 summarises the changes done to the application in the time frame of the project.

**Table 27: Summary of main changes done during the project (CERFACS)**

<table>
<thead>
<tr>
<th>Threading</th>
<th>OmpSs taskification, with graph sub partitioning mimicking the MPI domain decomposition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorisation</td>
<td>Removal of redundant divisions, heavy specialization of the loops and vector directives for the compiler (SIMD, FORCEINLINE). Ongoing work to switch array structures indexes for large vector</td>
</tr>
</tbody>
</table>
vectorisation.

<table>
<thead>
<tr>
<th>Offload</th>
<th>Reverse offload of I/O phases. Implementation with OmpSs on the temporal I/O. Started offload of 3D dump I/O but not operational yet.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other</td>
<td>Switch from master/slave to SPMD paradigm. Implementation of parallel collective IO using HDF5. Use of modern collective MPI for global communications instead of master centralized system. Buffering of arrays for reduction operations.</td>
</tr>
</tbody>
</table>

5.3. Benchmarking

The AVBP code was benchmarked in the DEEP Cluster, the Mira BlueGene/Q system at ARNL and the MareNostrum 3 cluster. The DEEP Cluster is a standard cluster with 256 Intel Xeon CPU E5-2680. Mira is equipped with 48 racks and 786,432 cores and the MareNostrum 3 cluster is comprised of 42 heterogeneous nodes each with 2 Intel SandyBridge-EP E5-2670 and 2 Xeon Phi 5110 P.

As explained earlier, the OmpSS developments do not include the vectorisation improvements as they are in a separate code branch and this benchmarking results do not reflect the performance of the code when all the changes will be merged.

Table 28: Experiments setup for the benchmarking subsection (CERFACS)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Strong scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of elements</td>
<td>2,958,592 tetrahedra, 347,856 Prisms and 8,830,863 tetrahedra, 14,047,346 tests</td>
</tr>
<tr>
<td>Other details</td>
<td>SIMPLE, NASAC3X, MERCATO</td>
</tr>
</tbody>
</table>

5.3.1. Xeon Phi platforms

We tested 3 cases in the MareNostrum 3 system. We have chosen to perform tests with 4 MPI per MIC as this setup showed the best ratio between MPI processes and OmpSs threads on the Xeon Phis of MareNostrum 3. When increasing the number of used cores in the following graphs only the number of tasks is changed. Please note that the value shown here corresponds only to the computational phase as the other phases are not threaded.

As discussed before, the code is executed in fully native mode directly on the node MIC. In these tests the offloading is disabled because, at the time of writing, it was not possible to use offloading and the parallel threading in MareNostrum 3 due to bugs. The latest version of the Mercurium compiler supports correctly the offload, unlike the previous one. However, the tasks are not correctly parallelized. We are currently investigating the reason. Because of memory limitations the NASA (turbine blade) and the MERCATO test case (two phase flow combustion) could not be ran on less than 2 Xeon Phi.

Table 29: System setup details for Xeon Phi platforms (CERFACS)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 14.0.2 in MareNostrum 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Intel MPI 4.1.1.036 in MareNostrum 3</td>
</tr>
<tr>
<td>Mercurium and nanox versions</td>
<td>OmpSs 14.06 (mcxx 1.99.2, nanox 0.7) in MareNostrum 3</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O3 -mmic -unroll0 -model precise -diag-disable 10212 -fno-alias</td>
</tr>
<tr>
<td>MPI processes per MIC card</td>
<td>4 in MareNostrum 3 (Xeon Phi)</td>
</tr>
</tbody>
</table>
The figures below show the performance of the code for these test cases. The SIMPLE test case on 1 MIC achieved a stable parallel efficiency between 77% and 87%, though it lightly decreases as the number of tasks increases. This is understandable as the quantity of data per thread varies with the number of threads; with diminishing data, tasks reduce in size and computation time becomes limited by the task creation time.

On 2 MIC cards the parallel efficiency shows a drop at 2 threads per MPI rank on both SIMPLE and NASAC3X test cases. This singularity is reproducible but we have not yet been able to understand the reason. The two test cases appear to have similar profiles for parallel efficiency on 2 MICs.

Performance of AVBP in Xeon Phi
- Simple test case -

![Performance of AVBP in Xeon Phi](image)

*Figure 79: Performance of AVBP in Xeon Phi: SIMPLE test case. Number of Cores = 4 MPI x N threads (CERFACS)*
The MERCATO test case could not be run on less than 8 MIC. This problem has been communicated to the BSC support group. The efficiency of the parallelism is fairly good but again we see a drop with 16 tasks per MIC.

The performance of AVBP for these test cases is satisfactory. Further improvements are of course required to assess and bypass the performance drop at 16 tasks per MIC. Instabilities in the Xeon Phis in MareNostrum 3 does not allow us to perform a more in-depth analysis of the performance issue as 3 out of 5 runs fail due to MPI connection errors. Nevertheless, the new version of AVBP is able to benefit from the multi-core and shared memory aspects of the system and the task based algorithm works as expected. Scalability will be improved once the
whole computational phase is threaded and the limitations for the NASA and MERCATO test cases will be reduced when threading the initialization phase, which is currently the memory bottleneck. We have performed the same tests with the SIMPLE benchmark using the Intel OpenMP 4.0 implementation with near identical performance.

5.3.2. Comparison with BlueGene/Q

The code was tested in the DEEP Cluster and the MIRA BlueGene/Q system from Argonne. The test case chosen for this test is the SIMPLE test case (PRECCINSTA test rig). In BlueGene/Q, OmpSs was not used and we rely solely on the automatic threads generated by MPI with the SMT mode. The configurations tested were 1 task per core, 2 tasks per core and 4 tasks per core. The affinity is the default. In the DEEP cluster the standard MPI execution is used for comparison.

<table>
<thead>
<tr>
<th>Table 30: System setup details for BlueGene/Q and DEEP (CERFACS)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Backend compiler versions</strong></td>
</tr>
<tr>
<td><strong>MPI runtime versions</strong></td>
</tr>
<tr>
<td><strong>Mercurium and nanox versions</strong></td>
</tr>
</tbody>
</table>
| **Compilation flags** | BlueGene/Q: -O3 -qsuffix=cpp=f90 -qstrict -q64 -qarch=qp -qtune=qp  
DEEP: -fp-model precise -vec-report0 -unroll -xsSE4.2 -O3 -fno-alias -ip -assume byterecl -convert big_endian |
| **MPI processes per node** | 16 in BlueGene/Q, 16 in DEEP Cluster |
| **Threads per process** | 1, 2 and 4 in BlueGene/Q, 1 for the DEEP Cluster |
| **Affinity setup** | Default |
| **Offloaded ranks per MPI process** | No offloading |

For the small SIMPLE test case the AVBP code scales very well on both platforms. The AVBP code makes efficient use of BlueGene/Q’s SMT with the same run time for 1 thread per core and 2 nodes and 4 threads per core on one node. The SMT setting has little not impact strong scaling. Performance is of course not the same for 32 nodes with 1 process per core and 16 nodes with 2 process per node but strong scaling performance with reference to the smallest number of MPI processes used for a given rank is almost identical no matter the SMT mode.

The parallel efficiency drops linearly after 16 nodes for both the DEEP and the BlueGene/Q systems. This suggests that a latency vs computational work limit is reached at this level. To test this, a heavily refined version of the SIMPLE test case has been generated with over 432,027,836 tetrahedras. Figure 84 displays the strong scaling using 1 thread per core and 4 threads per core, for up to 8192 nodes (262,144 task). This shows that with enough work the code remains scalable up to Petaflops limits. The current limitation for higher core counts is translated to the mesh generation, as currently we cannot create meshes larger than 1 Billion elements on complex geometries.
Performance of AVBP in DEEP and BlueGene/Q platforms
-Simple test case-

![Graph 1: Performance of AVBP in DEEP and BlueGene/Q platforms](image1)

Figure 82: Performance of AVBP in DEEP and BlueGene/Q platforms (CERFACS)

Parallel efficiency of AVBP in DEEP and BlueGene/Q platforms

![Graph 2: Parallel efficiency of AVBP in DEEP and BlueGene/Q platforms](image2)

Figure 83: Parallel efficiency of AVBP in DEEP and BlueGene/Q platforms (CERFACS)
5.3.3. \textit{Comparison with cluster architectures without coprocessors (with and without offload to regular CPUs)}

JUELICH has supported CERFACS by implementing offloading on the I/O temporal phase. This allows the offloading of all-reduce and I/O operations, which do not scale well and hinder performance. Unfortunately, at the time of writing it was not possible to use the offloading model on Mercurium 14.06 and the latest version is not able to handle correctly the threading dependencies between the tasks on MareNostrum 3. BSC, JUELICH and CERFACS are working together on this issue and strive to solve it by the final review.
5.3.4. Comparison with the beginning of the project

AVBP could not run on MIC architectures before the project. Therefore we chose to use the code directly on the DEEP Cluster and show the improvement on the SIMPLE, NASA and MERCATO test case. The test cases are ran on MPI only with one thread per MPI. Since the OmpSs branch does not include the vectorisation optimisation, the vectorisation branch is more representative of the progress made compared to the beginning of the project.

Table 31: System setup details for DEEP Cluster (CERFACS)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>intel/14.0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>parastation/intel12-5.0.28</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-unroll -xSSE4.2 -O3 -fp-model precise -fpp -fno-alias -ip -assume byterecl -convert big_endian</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>16</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>Default</td>
</tr>
</tbody>
</table>

The figures below show run time and parallel efficiency for the 2011 version of AVBP and the current one, for the SIMPLE, NASA and MERCATO test cases. For the SIMPLE test case it can be seen that overall run time is improved and faster for the new version, which can be explained by the better vectorisation. Additionally strong scaling improved as the limit of scaling at 16 nodes is not present anymore.

Performance of current AVBP vs. 2011 AVBP for the SIMPLE test case

![Figure 86: Performance of current version of AVBP (without offloading) vs. old version of AVBP, in DEEP for the Simple test case (CERFACS)](image-url)
The same conclusion can be seen on the NASA test case. Overall performance has been substantially improved and the scalability as well at least by a factor two. Parallel performance drops dramatically for the old version of the code after 16 nodes whereas the new version can scale at 80% at 64 nodes.
Figure 89: Parallel efficiency of current version of AVBP (without offloading) vs. old version of AVBP, in DEEP for the NASA test case (CERFACS)

Figure 90: Performance of current version of AVBP vs. old version of AVBP (without offloading), in DEEP for the MERCATO test case (CERFACS)
5.4. Analysis of metrics for WP9

At the moment we are not able to use the extrae lite library with AVBP on MareNostrum. This issue is under investigation and data will be provided ASAP once the problem is solved.

5.5. Conclusions

The work from DEEP allows for more complex and challenging simulations to be tackled using the CFD application AVBP on todays and tomorrows leadership class systems. Important improvements have been made, which have led to a better computational efficiency and scalability. AVBP is currently capable of using MIC architectures with acceptable efficiency. Of course some interrogations remain due to configurations issues; we will strive to solve these problems. Nevertheless, extensive improvements have been made in terms of code modernization and algorithmic analysis, which benefit the whole AVBP community and are also applicable for other codes.

The innovative Cluster/Booster division from DEEP allows for an insightful methodology to handle data analysis and processing while reducing its impact on the computation. Indeed, these temporal data and 3D snapshot dumping methods are particularly hard to scale by their nature (blocking MPI_Allreduce and I/O operations) but are paramount from an user and scientific side. With the implemented offload AVBP users will gain a performance edge on the DEEP System, as soon as the remaining compiler and runtime issues are solved.
6. Task 8.5: High temperature superconductivity (Task leader: CINECA)

This section describes the work done for the implementation of the TurboRVB application on the DEEP Cluster-Booster architecture with the offload facility of OmpSs, and the benchmarks performed on MareNostrum3 in preparation for full testing on the DEEP System. We compare these data with the non-offload version of the code, executed on various architectures such as MareNostrum 3 (both Xeon and Xeon Phi cores) and the Fermi BG/Q architecture at CINECA (PowerPC cores).

6.1. Overview of application structure

In this section we briefly recall the structure of the application and the parallel scaling behaviour. A more detailed analysis is provided in documents D8.1 and D8.2 – the purpose here is to facilitate comparison with the version of the application modified for the DEEP architecture.

6.1.1. Phases

The TurboRVB application can be distinguished in four separate phases (see Figure 92):

1. Initialisation.
2. Evolution of the walkers and calculation of observables.
4. Finalisation.

The first and fourth phases merely set up the program and perform and print final results, while the third phase reassigns the statistical weights to the walkers. From a computational point of view the numerical density is relevant only in the second phase, which advances the Monte Carlo simulation. For some test cases communication may be important in the third phase, but usually the time taken to run the whole program is mostly spent in the second phase (at least 70% of the total). Thus, in the Cluster-Booster environment of DEEP we can adopt a master-worker parallelisation with the bulk of computational effort consumed in the walkers running on the Booster. For a comparison of the time required for the most important program phases on the BlueGene/Q system of CINECA (Fermi) see Figure 94 and Figure 95 (in the figures upgreen and upatable refer to floating point intensive subroutines within the Monte Carlo cycle loop; please refer to earlier deliverables for more information).
6.1.2. Dominant factors

In a typical TurboRVB simulation up to 90% of the total CPU time can be dominated by linear algebra routines, in particular DGEMM for matrix-matrix multiplications, where the matrices are dense with sizes depending on the number of electrons (typically $O(10^2)$). This dependence on the linear algebra library means that there is little scope for scalar optimisations or vectorisation of the main code (see below). We note also that since the matrices involved in these calculations are usually fairly small, the sequential versions of libraries such as MKL can be used in most cases.

6.1.3. Scalability considerations

Quantum Monte Carlo (QMC) programs consist of multiple, weakly-coupled simulations (i.e. the walkers), which are highly scalable, exhibiting both strong and weak parallel scaling behaviour even at very high core counts. For TurboRVB, where it is not possible to assign more than one MPI task per walker, strong scaling implies performing runs with $N_w$ walkers where $N_w$ is an exact divisor of the number of MPI tasks $N$. Weak scaling instead involves running simulations fixing $N=N_w$ for a given lattice size and number of electrons. Given that increasing the number of walkers also increases the accuracy of the simulation results, it is more natural to perform weak scaling experiments but both can be used to examine the parallelisation efficiency of the calculation. As an example of strong scaling we show in Figure 93 the parallel efficiencies of the simulation runs used to generate the data in Figure 94 and Figure 95; the details of the simulation are given in Table 32. The parallel efficiency is high for many core counts, dropping only significantly at 1024 nodes, i.e. 16384 BG/Q cores. (For more benchmarks and a detailed analysis of strong and weak scaling behaviour on various systems, the reader is referred to deliverables D8.1 and D8.2)

Table 32: Experiments setup for the scalability considerations subsection (CINECA)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Strong scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of electrons</td>
<td>72 spin up, 72 spin down</td>
</tr>
<tr>
<td>Lattice size</td>
<td>12X12</td>
</tr>
</tbody>
</table>
Number of walkers | 16384

Parallel efficiency for strong scaling on Fermi

Figure 93: Strong scaling behaviour on the Fermi (BG/Q) system (CINECA)

Performance of the different phases on BG/Q

Figure 94: Impact on run time of each phase of TurboRVB, when running on BlueGene/Q with 16384 walkers (CINECA)
6.2. Code modifications during the project

The modifications required to run TurboRVB on the DEEP prototype can be classified into two categories:

1. Those required to render the program more efficient on an Intel Xeon Phi platform (e.g. changes to improve threading and vectorisation).

2. The changes needed to implement the Cluster-Booster strategy with the offload facility of OmpSs.

In this section we detail the code modifications made to the TurboRVB application in the DEEP project.

6.2.1. Threading

The TurboRVB QMC program available from the developers is a pure MPI program, i.e. the parallelisation is based entirely on MPI calls. Given the modest memory requirements of the program there have been no significant efforts by the developers or others to perform a threaded parallelisation, using for example OpenMP. Since the poor MPI performance of many applications on Xeon Phi is well known, during the project an activity was initiated to produce a threaded version of the application with OpenMP. Initial efforts were severely hampered by the very large number of global variables used within the FORTRAN code which give rise to many dependencies. Before continuing the effort it was decided to embark on a refactoring strategy to redesign the code structure so as to simplify the OpenMP porting, and a number of FORTRAN refactoring tools were evaluated. The most promising of these was judged to be the PHOTRAN plugin for the Eclipse IDE, which has significant functionality and is integrated already into a commonly used programming tool. But despite its promise, it was found that the tools were not able to reduce the manual effort sufficiently to allow the OpenMP porting to be achieved without a significant re-structuring of the entire application and this was not considered feasible within the timeframe of the project.
6.2.2. Vectorisation

No loops were identified which could benefit from improvements to vectorisation. As mentioned above, for most input data sets, a considerable portion of the computational effort is consumed in the linear algebra library routines (e.g. MKL in the case of Intel systems) which are already optimised by the hardware vendors.

6.2.3. Cluster/Booster division

In the case of TurboRVB it was decided to adopt a fairly straightforward Cluster-Booster strategy based on a single master process running on the Cluster node which spawns the walkers as MPI processes on the Booster nodes (i.e. the Xeon Phi nodes). The main Monte Carlo loop is run by each walker and control is returned to the Master only at the end of the run. Thus, virtually all the program time is consumed by the walkers on the Booster nodes, with the master only performing some initial and final file operations (e.g. reading the input files) and spawning the walker tasks. An alternative strategy of using a reverse offload, i.e. running directly on the Xeon Phi and spawning some tasks on the Cluster nodes, was considered but not adopted - such a strategy would be beneficial for processes with high I/O requirements which would run better on the Cluster but we recall that the walkers in TurboRVB perform hardly any I/O (at least in the non-checkpoint version of the code).

The code changes required to perform the Cluster-Booster division were fairly straightforward and consisted of the following modifications:

- Reorganisation of the main part of the program such that all file operations, except rank-local operations (i.e. those done by the walkers), are performed before the DEEP Booster allocation call. An extra input file was also introduced to allow the user to specify the number of Booster nodes (i.e. Xeon Phis) and the number of processes per Booster node to assign to the simulation. The product of the two numbers thus gives the total number of walkers to be used.

- Since OmpSs requires that all allocatable (i.e. dynamic) arrays are allocated before being offloaded, the necessary FORTRAN allocate statements were moved before the Booster allocation call (in the original code the various array allocations were done at various places in the code).

- Insertion of the DEEP Booster allocation call, essentially a wrapped version of the MPI_Comm_spawn command, to make the Booster nodes available for the offload and the OmpSs directives specifying the data dependencies of the spawned, walker tasks.

Addition of OmpSs calls at the end of the program to allow a correct finalisation of the program.

6.2.4. Other optimisations

Since the modifications described above, with the exception of OmpSs allocation calls and the Booster information input, do not actually modify the program logic, it was found useful to insert compiler directives within the source to allow the program to be compiled without the offload facility and so ease the comparison with the MPI-only version.

6.2.5. Summary of changes

Table 33 summarises the changes done to the application in the time frame of the project.
Table 33: Summary of main changes done during the project (CINECA)

<table>
<thead>
<tr>
<th>Threading</th>
<th>Not done due to code complexity.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorisation</td>
<td>No scope for vectorisable loops.</td>
</tr>
<tr>
<td>Offload</td>
<td>Some code modifications and re-structuring to implement OmPss offload</td>
</tr>
<tr>
<td>Other</td>
<td>Pre-processing directives to allow comparison with non-offload version</td>
</tr>
</tbody>
</table>

6.3. Benchmarking

Initial porting of the OmpSs version of the code was performed on the Xeon Phi KNC system at Jülich. Once initial tests had demonstrated to our satisfaction that the porting was successful, as demonstrated by simulation results in agreement with the non-offload version of the program with the same input (within statistical error), the application was transferred to Mare Nostrum 3, which has Xeon Phi partitions. The aim was to test the offload implementation on Mare Nostrum before migrating to the DEEP prototype but due to delays in the availability of the DEEP System, the benchmark tests at BSC were extended and results from the DEEP prototype are not reported here.

Benchmark experiments were performed as follows on Mare Nostrum:

- Comparison with non-offload version on Xeon cores;
- Comparison with original code run in native mode on Xeon Phi cores.
- Offload performance as a function of Xeon Phi nodes;

Note that since here the Mare Nostrum is being used to mimic the future DEEP prototype with its Cluster-Booster separation, very often we will refer to the Xeon processors as “Cluster nodes” and the Xeon Phi devices as “Booster nodes”. A “Booster core” therefore indicates one core of the Xeon Phi device. We note also that in all these runs we do not overload the cores with more than one MPI task so the number of cores also equals the number of MPI tasks.

Table 34: Experiments setup for the benchmarking subsection (CINECA)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of electrons</td>
<td>20 and 100</td>
</tr>
<tr>
<td>Lattice size</td>
<td>5x5 and 10x10</td>
</tr>
<tr>
<td>Other details</td>
<td>10000 MC cycles, lattice generated at start (i.e. not from restart files)</td>
</tr>
</tbody>
</table>

6.3.1. Xeon Phi platforms

For the Mare Nostrum runs with Xeon Phis a fairly small system size for the Quantum Monte Carlo was considered consisting of a 5x5 lattice and a total of 20 electrons and was executed over 10,000 cycles (see Table 34) - we refer to this as the small input set. Although the system size is smaller than what might be used for research purposes, previous experience suggests that the parallel scaling behaviour does not alter significantly for low numbers of electrons, for example (see D8.2 for more information). In some cases it was decided to run over larger systems, using, for example, 100 electrons and a 10x10 lattice (“the large input set”) but unless stated otherwise, the results below refer to the small input set. In all cases the
simulations were started from a lattice generated at the beginning of the run, i.e. restart files from a previous run were not used, as in the runs for the CPU-only cores.

The versions of the system software used (e.g. compilers, OmpSs, runtimes, etc.) are given in Table 35. The runs were performed in batch mode, with LSF batch scripts provided by BSC (after minor modifications).

Table 35: System setup details for Xeon Phi platforms (CINECA)

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel 14.0.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Intel MPI 5.0.1</td>
</tr>
<tr>
<td>Mercurium and nanox versions</td>
<td>OmpSs version 15.06, nanox 0.9a, mcxx 1.99.6</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-g -O2 -mkl=sequential</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>See X axis</td>
</tr>
<tr>
<td>Threads per process</td>
<td>1</td>
</tr>
<tr>
<td>Affinity setup</td>
<td>Default</td>
</tr>
</tbody>
</table>

In the first tests we compare directly the performance of the offload code on a single Xeon Phi with the original code run in native mode on the accelerator. In Figure 96 we show data for both the standard and large system sizes.

We see that for the standard input the performances match very closely those obtained in native mode at all core counts. With the larger system the scaling behaviour is the same, but there is a small performance difference which increases as the number of Xeon Phi cores increases.

**Performance of TurboRVB in Xeon Phis in MareNostrum 3**

![Figure 96: Performance of TurboRVB in Xeon Phis in MareNostrum 3 (CINECA)](chart.png)
We now concentrate on examining the performance of the offload code over multiple Xeon Phis. Weak scaling results for the small input set are shown in Figure 98. We note that for these runs, for memory reasons, it was not possible to use all the cores in the Xeon Phi devices so we show results for 8 and 16 cores per Booster. It is clear from the graph that we don’t see the weak scaling behaviour we observe within a single MIC card, but we need more data with additional input sets to investigate this further.

In the next sets of runs we probe the strong scaling behaviour running over multiple Xeon Phis. The data for the standard input but run over 1000 MC cycles and using 8 cores per Xeon Phi is given in Figure 99. We notice that the strong scaling starts to tail off after about 8 Xeon Phi.
6.3.2. Comparison with cluster architectures without coprocessors (with and without offload to regular CPUs)

Since the offload code is structurally different to the original TurboRVB application, and includes OmpSs tasking, it was decided to test this version on regular processor cores to ensure that the modifications do not significantly alter program results and performance. In Figure 100 and Figure 101 we show the results for MareNostrum 3 (with input data shown in Table 36).

We notice that the weak scaling behaviour is essentially identical but there is a difference in performance, which is almost a constant, i.e. not depending on the number of cores/walkers. We shall investigate further this performance difference later in the analysis section.

Table 36: Experiments setup for the regular processor comparison (CINECA)

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of electrons</td>
<td>200</td>
</tr>
<tr>
<td>Lattice size</td>
<td>20x20</td>
</tr>
<tr>
<td>Other details</td>
<td>1000 MC cycles, lattice generated at start (i.e. not from restart files)</td>
</tr>
</tbody>
</table>
6.4. Analysis of offload performance

In the previous section we saw that there the offload version shows a lower performance than the original TurboRVB code for all but small datasets. Since we see this difference even with the Xeon processor cores, i.e. before trying on the MIC architecture, it makes sense to examine more closely the branching phase of the program, which we recall has many communication calls, before looking at possible issues with Xeon Phi optimisation. We recall that usually the time consumed in this routine is normally fairly small. The branching times for the two sets of runs as a function of number of nodes are shown in Figure 102. It is clear that the communications are much greater for the offload version of the code and are such as to account for the performance difference between the two versions.
6.5. Conclusions

From the above data we can make the following observations:

1. The weak parallel scaling behaviour of the offload version of the TurboRVB code is comparable with that of the original version within Xeon Phi nodes (i.e. in native mode) and also with only Xeon cores. The scaling behaviour is maintained when using multiple Xeon Phi nodes.

2. There is a performance difference between the two versions, on both regular CPUs and Xeon Phis, which varies according to program input. The difference is greater for larger input sizes and occurs in the branching phase of the simulation which contains the bulk of the communications. For small input sizes the difference between the offload and native executions is negligible.

3. Strong scaling over multiple Xeon Phi nodes, i.e. multiple Booster nodes, is also as expected given the limited input set although perhaps can be improved.

The performance difference in point 2, identified to be in the branching subroutine, needs to be investigated further. It was checked during the runs that the task affinities were selected correctly (using appropriate environment options) so the difference presumably is in the OmpSs runtime itself. Aside from this, the behaviour is similar to the original code version for small numbers of booster nodes and, despite the absence of the DEEP prototype we are confident that the offload version of TurboRVB with OmpSs will run successfully on this and similar heterogeneous architectures, although we clearly need to understand and mitigate the increased communication costs in the branching routine for larger input sizes. It should be emphasised that the cluster-booster architecture offers advantages with respect to homogenous, many core systems such as BlueGene/Q. For example, one possibility could be a post-processing phase of the wavefunction on the cluster nodes, which being weakly parallel and memory intensive, would not be appropriate for a BlueGene/Q system.
7. Task 8.6: Seismic imaging (Task leader: CGGVS)

Up to the end of the second year of the project, we have been working on the SRMIP application. Since it was already capable of running efficiently on hybrid Xeon and Xeon Phi clusters, we decided to switch to another application called Reverse Time Migration (RTM). In this deliverable, we will focus on RTM. All details regarding SRMIP can be found on previous deliverables D8.1 and D8.2.

We selected RTM since it is a more advanced seismic imaging algorithm describing more accurately seismic wave propagation in complex geology. As such, it better represents CGG current and future workloads. Moreover, CGG RTM application runs on GPU-accelerated nodes and thus can be used to compare the DEEP concept to a regular GPU cluster.

RTM has the same high level structure and pattern of parallelism as SRMIP: a MPI Master-Worker scheme to distribute the shots and loop parallelism to process a shot. The main difference with SRMIP is the shot processing part: it is much more compute-intensive and it is offloaded on GPU and parallelized with CUDA.

A significant part of the project consisted of porting the CUDA code to C to be able to run on Xeon Phi: we rewrote all CUDA kernels in C, removed the GPU memory allocations and the data transfers between the host and the GPU, and optimized the kernels for the Xeon Phi. Section 7.2 details the choices made during this phase.

Another important difference from SRMIP is that RTM has large I/O needs. To avoid I/O being a bottleneck on the DEEP architecture (or on other Xeon Phi clusters), we remove the I/O during benchmarking. Full RTM performance can easily be extrapolated from the obtained timings by setting a given storage bandwidth. Details on the I/O needs of RTM are given in the next section.

7.1. Overview of application structure

In this section, we present the 3 main phases of RTM and detail factors impacting their performance.

7.1.1. Phases

Like SRMIP, RTM can be decomposed as a collection of shots that can be processed in parallel. The shots are distributed on multiple nodes using a Master-Workers scheme in MPI. The output of one shot is an intermediate image and the final image is the accumulation of all these intermediate images (this accumulation process is called stacking). The processing of one shot contains the following 5 steps on the worker:

- Send shot data to the worker.
- Forward wave propagation.
  - Propagate wavefield from the source from time 0 to T.
  - Save wavefield every 0(1) steps (called checkpointing).
- Backward wave propagation.
  - Propagate wavefield from the receivers from time T to 0.
  - Save wavefield every 0(1) steps (called checkpointing).
- Imaging.
  - For each time t, read forward and backward wavefields corresponding to time t and accumulate contribution in the image.
- Send back the image to the master.

We can decompose the RTM application in the 3 following phases (see Figure 103):
1. Initialization phase: The master reads input data from external storage and sends data common to all shots to all the workers.
2. Master/Worker phase: The master distributes the shots and the workers process them.
3. Finalization phase: The master writes the final image to external storage.

**Figure 103: Phases of RTM (CGGVS)**

### 7.1.2. Dominant factors

The performance of the initialization phase is determined by the read bandwidth to the external storage and the network bandwidth. The performance of the finalization phase is determined by the write bandwidth to the external storage.

The performance of the master/worker phase is determined by the time taken to process a shot on a worker. As detailed in the previous section, shot processing is done by the modelling steps (forward and backward) followed by the imaging step. The modelling time on the worker is the maximum compute time of the wave propagation and the write bandwidth to the scratch space (overlapping of computation and checkpointing). The imaging phase on the worker is determined by the read bandwidth from the scratch space. Overall, performance on the master/worker phase depends on the compute performance and the bandwidth from/to scratch on the worker.

According to the wave propagation performance on a Xeon Phi, the write bandwidth to scratch should be around 300MB/s per Xeon Phi in order to completely overlap the checkpointing time by the computation. This bandwidth is not achievable on the clusters at our disposal for the benchmarking (neither on MN3 nor on the DEEP Cluster). To avoid I/O
being a bottleneck during the benchmarking, we remove the checkpointing of the wavefields in the modelling step and the imaging step. We also remove the finalization phase since there is no more data to be written as output. However, the full RTM application performance could easily be extrapolated from these benchmarks assuming a known scratch bandwidth.

7.1.3. **Scalability considerations**

For all benchmarks in the following sections, we use the same dataset containing 300 shots. We perform a weak scalability studies by keeping the number of shots per worker constant at 4 (see Table 37).

In this section, we analyse the time breakdown per phase and the scalability of each phase in the original GPU accelerated code on CGG internal GPU cluster (see section 7.3.2 for details on the system setup). On Figure 104, we see that the scalability of the master-worker phase is almost perfect. This is not surprising since we are using a Master-Worker scheme where tasks have almost constant processing time and very little data to communicate compared to the compute time. We can also notice that the time spent in the initialization phase is negligible.

<table>
<thead>
<tr>
<th>Scaling</th>
<th>Weak scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of shots</td>
<td>4 shots per worker</td>
</tr>
</tbody>
</table>

**Table 37: Experiments setup for the scalability considerations subsection (CGGVS)**

![Figure 104: Percentage of run time of each phase of RTM, when running on the CGG GPU cluster. Imaging and Finalization phases are not included since these phases cannot be run without I/Os. (CGGVS)](image)

7.2. **Code modifications during the project**

As stated in the introduction of this section, part of the original RTM application from CGG is written in CUDA. Thus, a significant amount of time has been spent porting the kernels from CUDA to C + OpenMP. We detail the work done on the kernels in sections 7.2.1 and 7.2.2. The changes done on the host side removing the CUDA API are detailed in section 7.2.3.
7.2.1. Threading

Most of the CUDA kernels are relatively easy to port to C+OpenMP. They are already written in the form of ‘implicit’ parallel loops. One just needs to replace the implicit CUDA loops by explicit OpenMP parallel loops. As shown in Figure 105 and Figure 106, the scalability is very high (around 90%) up to 60 cores (i.e., without using hyperthreading) except for kernels 1, 4 and 5 which are quickly limited by memory bandwidth. Kernels 2 and 3, which are also very dependent on memory bandwidth, do not show benefit from hyperthreading. Kernels 6, 7 and 8, which have the most complex memory access pattern and are more computationally intensive even show good speedup with hyperthreading.

Those good OpenMP scalability results on the kernels are consistent with the CUDA code which has been designed for a high number of threads.

![Performance of threaded kernels](image)

Figure 105: Performance of the main wave propagation kernels in a Xeon Phi 7120 (CGGVS)
Vectorisation

Since the kernels were originally written in CUDA, they were already written in a form suitable for vectorization. For kernels 1 to 5, we only needed to add a pragma `simd` in the inner loop to make sure the compiler will vectorise. In Figure 107 and Figure 108, we see that all these kernels achieve a significant speedup thanks to vectorization except for kernel 1, which is purely memory bound. Note that we are using single precision so the maximum speedup is in principle 16x. A larger speedup for kernels 2 and 3 might be explained by a different handling of trigonometric functions in serial and vectorised mode.

However kernels 6 to 8 needed more work. Kernels 7 and 8 contain inner loops that cannot be vectorised due to real data dependencies. Thus, they needed to be vectorised at the outer loop level. We were not able to force the compiler to automatically vectorise the outer loop even using pragmas. To achieve this, we rewrote both kernels using the Cilk Array Notation. We use the short vector style, i.e. the size of the Cilk Array is equal to the SIMD width, with which we obtained the best performance. See [2] for details on outer loop vectorization with the Cilk Array Notation and [3] on the short vector style.

We use the same method for kernel 6 to vectorise at the outer loop level. However, we needed one more feature to be able to vectorise. The corresponding CUDA code is using shared memory to change the data layout on the fly to enable vectorization. We replaced it by using the MKL transpose function on small blocks of data.

In all cases, we allocate all the large arrays with a 64B alignment and use compiler annotations in each kernel to declare to the compiler that all those arrays are aligned.
Vectorisation performance of wave propagation kernels

![Vectorisation performance of wave propagation kernels](image1)

Figure 107: Vectorisation performance in a Xeon Phi 7120. Kernels 1 to 5 have no optimized form and kernels 6 to 8 have no original form (CGGVS)

Vectorisation improvement of wave propagation kernels

![Vectorisation improvement of wave propagation kernels](image2)

Figure 108: Vectorisation improvement in a Xeon Phi 7120. Kernels 1 to 5 have no optimized form and kernels 6 to 8 have no original form (CGGVS)

7.2.3. Cluster/Booster division

Since the original code is in CUDA, there is already a separation between the regular code running on the host and the kernel code running on the GPU. We could have replicated this separation for the Cluster/Booster division: run the master code and the host part of the worker code on the Cluster and the kernels on the Booster. However, there are a lot of communications between the host part and the kernel part of the worker. Since on the Booster much larger portions of the code can be offloaded, we can dramatically reduce these communications by running the full code of the worker on the Booster. Moreover with this division, we do not need to use Cluster nodes to drive Booster nodes which is a waste of
resources (both in term of hardware and energy). Using the master-worker scheme to realize the Cluster/Booster division has the additional benefit that this division can be achieved in software using only MPI. The MPI master is launched on a Cluster node and the MPI workers are launched on Booster nodes.

To port the worker code on the Booster, we removed all the CUDA host API calls and as detailed in Section 7.2.1 ported the CUDA kernels to C+OpenMP. Since the host and the kernels are now part of the same address space, we could also remove some duplicated data structures and memory copies.

Since I/O is not efficient on the Booster, we modified the application to remove all I/Os on the workers. Some I/Os were needed by the application (imaging phase) but could not be kept due to insufficient I/O performance on the Cluster (as discussed in Section 7.1.2). Other I/Os, for example in the initialization phase, were replaced by communications from the master.

### 7.2.4. Other optimisations

Additionally to threading and vectorization, we also used software prefetching for some kernels. Indeed, all of our kernels are very dependent on memory bandwidth. In kernels 1 to 5, where the access pattern is linear with unit stride, we already achieve a very high fraction of the peak memory bandwidth (as measured by the stream benchmark). We added software prefetching in kernels 7 and 8 which have large strides and in kernel 6 which access data by blocks. In all cases, we tuned the prefetching distances by an exhaustive search.

### 7.2.5. Summary of changes

Table 38 summarises the changes done to the application in the time frame of the project.

<table>
<thead>
<tr>
<th>Threading</th>
<th>OpenMP parallel for with static load balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vectorisation</td>
<td>Systematic use of pragma simd. Outer loop vectorization with Cilk Array Notation. Alignment of all arrays.</td>
</tr>
<tr>
<td>Offload</td>
<td>Direct offload of workers with MPI.</td>
</tr>
<tr>
<td>Other</td>
<td>Ported CUDA code to C+OpenMP. Modified data input and distribution to remove I/Os on the Booster.</td>
</tr>
</tbody>
</table>

### 7.3. Benchmarking

For the benchmark presented in this section, we use the same setup as described in Table 37.

### 7.3.1. Xeon Phi platforms

In this section, we present experimental results on the MareNostrum 3 cluster. The system setup is detailed in Table 39.

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel Compiler 2015.2.164</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>Intel MPI 4.1.3.049</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O3 -mmic -ftz -fimf-precision=low:sinf,cosf</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>1 MPI rank on the first node</td>
</tr>
<tr>
<td></td>
<td>1 MPI rank per Xeon Phi</td>
</tr>
</tbody>
</table>
Threads per process | 240 threads per MPI worker
Affinity setup        | KMP_AFFINITY=balanced

As shown in Figure 109, parallel efficiency is quite good with more than 85% at 32 nodes.

**Table 40: System setup details for the GPU cluster (CGGVS)**

<table>
<thead>
<tr>
<th>Backend compiler versions</th>
<th>Intel Compiler 12.1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI runtime versions</td>
<td>OpenMPI 1.6.4</td>
</tr>
<tr>
<td>Compilation flags</td>
<td>-O2 -xSSE2 -std=c99 -ftz</td>
</tr>
<tr>
<td>MPI processes per node</td>
<td>2</td>
</tr>
<tr>
<td>GPUs per node</td>
<td>2</td>
</tr>
</tbody>
</table>

As shown in Figure 110, parallel efficiency is very similar on both platforms and always better than 85%. In Figure 111, we see that the runtime ratio between the GPU platform and the Xeon Phi platform is almost constant and equal to the ratio of time needed to process one shot on both platforms. Results are in favour of the GPU platform but it shall not be forgotten that a node on the GPU platform has twice the number of chips that on the Xeon Phi platform. Moreover, the OpenMP parallelization on the workers suffers from too many parallel regions. Using the Vtune profiler, we estimated that performance could be improved by up to 20%.
7.4. Analysis of metrics for WP9

In this section, we analyze the evolution of the number of MPI calls and the fraction of time spent in MPI, with the number of nodes. We use the same experimental setup as in section 7.3.1.

7.4.1. Number MPI Calls

As shown in Figure 112, most of the MPI calls happen in the initialization phase. This number increases with the amount of nodes, although we are in a weak scaling scenario. Indeed, the work in the initialization phase increases with the number of nodes.
Variation of number of MPI calls with the number of nodes, per phase

![Graph showing variation of number of MPI calls with number of nodes, per phase.](attachment:image.png)

Figure 112: Variation of the number of MPI calls with the number of nodes, per phase. Average across ranks. (CGGVS)

### 7.4.2. Fraction of time spent in MPI calls

As shown in Figure 113, most of the time spent in MPI is in the initialization phase. As explained in the previous section, time increases with the number of nodes. For the Master-Worker phase, we can notice that the time spent in MPI becomes significant with 8 or more nodes. This can be explained by the master becoming a bottleneck when distributing data to increase the number of workers.
7.1. Conclusions

During this project, we ported two seismic applications on the DEEP concept. For SRMIP, which was originally a pure MPI application, we added OpenMP parallelism, vectorised and tuned the code running on the Booster. For RTM, which was a MPI + CUDA application, we ported the CUDA part of the code to OpenMP and tuned the kernels for the Xeon Phi. Both applications demonstrated good performance and scalability. In the case of RTM, the DEEP concept is lacking scratch space to run the complete application and only the modelling part of the code (the most important part in terms of compute time and lines of code) was ported. The node-local non-volatile memory of DEEP-ER would be a good fit for the scratch usage of RTM.

Both applications use a master-worker scheme which maps well to the DEEP concept. The master who handles I/O to the global storage and runs a lot of control code runs on a Cluster node and the workers runs on Booster nodes. For master-worker type of applications, the DEEP concept has two advantages:

- First, one can select the optimal ratio of Cluster nodes and Booster nodes for each application. This is not possible in the case of regular accelerated cluster where the number of accelerators per node is fixed. For example in the case of RTM which is very compute intensive, one Cluster node for 8-16 Booster nodes is a good ratio. For SRMIP which is less compute intensive, we would only use 1 or 2 Booster nodes per Cluster node.
- Second, the worker code runs completely inside the Booster in contrast to regular GPU-accelerated cluster where only the kernels run on the accelerator. This removes the need for a lot of regular processors that are only used to drive each accelerator. It reduces power consumption and hardware cost.

One interesting feature of the DEEP concept that we currently not benefit from is the fast interconnect between the accelerators. Indeed, our applications are not built to distribute the processing of a single shot on multiple nodes and thus network requirements are low. However, with the growing dataset size and the limited capacity of high bandwidth memory,
we may need to distribute the wave propagation on multiple accelerated nodes. In that case, we would surely benefit of the high performance network between the Booster nodes.
8. Global conclusions

The DEEP project faced some very challenging situations from the beginning. Its innovative approach does not come without risks, and it has paid the price for them. The delays in hardware and software instabilities – present in every Xeon Phi system tried in the course of the project –, bugs and unforeseen interactions between software components have heavily impacted the amount of results obtained by this work package. Furthermore, the applications team work had its own set of problems – mainly lack of personnel and underestimated complexity of the required changes –, which took their toll and generated delays.

However, the potential of the architecture is evident. It has been shown that the flexibility of the concept allows for very different usage models, as for 6 different applications we have 6 different ways of using the system: Interactive supercomputing with simulation running on the Booster and the steering and visualization parts running on the Cluster (EPFL), coupled models with different characteristics running in different processor architectures (KULeuven), offloading of discrete tasks for dynamic load balancing and improved scalability (CYI), computational part running on the Booster with overlapped I/O and associated reduction operations running on the Cluster (CERFACS), application running completely on the Booster (CINECA), and shot processing on the Booster with image stacking and I/O running on the Cluster (CGG). Of course not all of them obtain the same level of benefit, but this highlights the fact that each application can find a potentially beneficial way of using the system.

Applications that can perform and scale on Xeon Phi without host processors have in DEEP currently the only available platform for it efficiently. Future supercomputers, based on the 2nd generation of Xeon Phi processors will resemble the DEEP System, with self-hosted Xeon Phi nodes. In the meantime, developers of such applications have two options: use traditional clusters with Xeon Phis in each node, and waste resources with idling host processors; or use the DEEP System.

Admittedly the results are not the expected so far. This report lacks in number and size of results, analysis and performance. With the upcoming stabilisation of the DEEP System, and fixing the remaining bugs in the various parts of the software stack, we are confident to show a beneficial impact of the concept in production runs for various applications. There is a lot of potential for 5 of the applications in DEEP:

- Brain simulation aims for interactive supercomputing. The software to support this goes beyond DEEP and its timeframe, but DEEP’s hardware architecture fits this futuristic concept, as Cluster nodes are more adequate for interactive visualization than Booster nodes.
- Climate simulation with EMAC is intrinsically limited due to the lack of scalability of the atmospheric model. However, dynamically offloading tasks to remote accelerators gives the EMAC community the chance to gain an edge in time to solution. It is known that EMAC will not make it to Exascale in its current form, but we should not despise “everyday supercomputing”, even in the Exascale era.
- AVBP is a highly scalable CFD application, with chances to benefit from Exascale computing. However, in production runs, the diagnostics part of the simulation often takes a sizable amount of time, without real usefulness towards calculating the evolution of the simulated problem. Offloading this part – and the associated collective reductions – to other nodes, as well as the 3D snapshots of the current status, can allow the simulation to proceed overlapping without interference. For that purpose, an architecture like DEEP is definitely helpful.
The case of plasma simulation with iPiC3D can also benefit from this concept. Increasing the number of particles per cell allows diminishing the noise of the simulation, which is a typical problem of PiC codes. This is possible by decoupling particles and fields, even though the results presented here do not show its full potential yet.

Lastly, in applications with similar characteristics to seismic imaging, it is possible to use the dynamic ratio of host processors and accelerators. This ratio is fixed in current systems, and host processors are often underutilized, as the heavy lifting is done in the accelerators, which cannot be added to the job on demand.

Furthermore, coupling of different applications has a lot of potential in the DEEP architecture. An example of that would be the post-processing of wave-functions calculated by TurboRVB on the Booster, opening new possibilities for Quantum Montecarlo simulations.

The applications of the DEEP project did not achieve their full potential on the platform, but they are getting close to it and efforts will continue in that direction beyond the end of the project.

The question whether some the above statements require the DEEP concept has not a unique answer. It can be argued that code separation might make sense, but using different processor technology might not. In other words, the diagnostic part of AVBP –for instance– can be offloaded from standard nodes to other standard nodes, effectively achieving the same benefit without relying in differentiated hardware nodes. Whereas this is completely valid, it should be highlighted that this absolutely depends on the characteristics of the application code. At the same time, it is expected that the hardware evolution will iron out some of the rough edges of the concept right now. One example is the performance of communications from current Xeon Phi coprocessors. The performance is clearly lacking, regardless of the underlying network technology used. This was not completely anticipated during the project proposal. However, future iterations of the Xeon Phi coprocessor seem to be able to eliminate –or at least mitigate– this issue. Therefore, some of the problems observed in DEEP will not be present in a future version of the concept. The evolution of the different hardware components, and of the different software parts, will dictate the perfect fit. This evolution will happen gradually, and so will the fitting of the application parts and system parts.

To finish, despite the lack of conclusive results to certify the DEEP architecture as Exascale enabler, it should not be forgotten the impressive performance and scalability improvement achieved by some of the applications participating in the project. That is by itself a step forward towards Exascale, regardless of the future of the platform.
9. Annex: Best practices and recommendation for improvements

Every system has its own particularities. Thus, a best practices guide to help new users achieve optimal performance is often necessary. DEEP is no exception. However, WP8 cannot provide a comprehensive best practices guide, due to the system unavailability at the time of writing this report.

Nevertheless it should be noted that a large part of the effort to benefit from DEEP is optimising the code for Xeon Phi processors. The application developers in DEEP tackled this challenge, and valuable lessons were learnt. Best practice guides for Xeon Phi optimisation have been already extensively covered in the past by other research institutions. Stand out the book by Jeffers and Reinders [4], the PRACE Best Practice Guide for Xeon Phi [5], and the numerous resources provided by Intel [6]. Topics like efficient threading, vectorisation, and enhanced data locality should be central points of any developer targeting Xeon Phi machines. Thus, the lessons learned in DEEP on this regard are no different than those learnt by others using Xeon Phi processors in standard configurations. These lessons have been already explained by external entities, and writing yet another optimisation guide for Xeon Phi does not seem necessary.

What makes DEEP different is the arrangement of Cluster and Booster nodes. Leveraging this idea, with two differentiated hardware parts, working independently from each other, requires some "out of the box" thinking. This is highly application specific, so writing a very detailed guide is not possible. Nevertheless, the following section will sketch the general workflow and explain the usage models used during the project.

Finally, the last section will explain some of the recommendations that have been gathered by WP8 to improve the DEEP concept and have a more efficient architecture and software environment.

9.1. Leveraging DEEP

The effort on code-engineering required to take full advantage of the DEEP concept can vary vastly across applications. However, the methodology to follow should be similar in all the cases. The following paragraphs describe the general logical steps that an application developer should follow to fully benefit from the DEEP architecture.

The first step towards a successful experience on the DEEP System is to carefully analyse the application that shall be ported, to identify its main phases or kernels. An experienced developer can probably straight-forward sketch the structure of the application, with the main phases and the workflow. This "pen-and-paper" sketch will be one of the central points of the process. Starting from such sketch, one should try to recognise those phases running profiling tools with small test-cases. The performance analysis tools Extrae/Paraver and Scalasca are recommended. The obtained profile will serve as a reference for the next steps.

The next step is to clearly quantify the importance of each of the code phases. Many applications have hundreds of kernels, but just a handful of them are the hotspots. Most likely, these are the ones that deserve more attention. It should be noted that in order to identify relevant kernels, the analysis should be done in production-size runs and environments. Only if the problem can be downsized while keeping the relevance of each individual phase representative enough, one can use smaller than production-size experiments. Scalasca is an excellent tool at this point of the process. It should be noted that the performance of many
applications are sensitive to process/threads ratios. The combination with the best performance should be used.

Once the important phases have been verified, the developer should assess how each of them is evolving. With performance analysis tools one should execute a series of scaling experiments (strong or weak scaling, depending on what is relevant for the field of each application). This way the developer will have a dynamic view of how scalable is each phase of the application. At this point, each phase should have its communication pattern thoroughly described. This information is very valuable towards identifying not only current, but future bottlenecks. Likewise, determining how the sizes of different data structures evolve is important, particularly for strong scaling applications.

The limiting factors for each phase must be detected to properly map them to the most appropriate part of the system. Taking a look at the profiles from the performance tools used in previous steps one should be able to tell if communications within a phase are or will become problematic. For phases with significant compute time, it has to be assessed if the kernels are compute bound, memory bandwidth bound, or memory latency bound. This requires a detailed study of each phase. For this kind of studies, the roofline model typically offers a good starting point.

The next step is the first one towards understanding the relationship between different kernels: the data structures used to connect phases have to be described. Important aspects are the kind of data (arrays, structs, objects, etc.), their contiguity in memory, their sizes and their role in the phase (input data, output data, or both). This information will be used later to calculate if the overhead of moving data between parts of the system will be significant or not.

With the information gathered up to now, the developer should have a very detailed view of many different aspects of the application. Before starting modifications, one last step should be considered. Different kernels or phases might be so closely related that considering them as independent from each other might be counterproductive. This is why in some cases clustering of kernels or phases might be interesting, instead of treat them separately. In this way, the separation done in the first steps is refined, leading to better decisions and optimisations later on.

Up to this point all actions aim towards a methodological gathering of information. The next steps will use this information to take informed decisions and implement suitable changes. As a pre-step, it might be advisable to isolate kernels or phases. This is not strictly necessary, but it might ease the optimisation steps that come down the road.

After the preparation steps the developer is in a good position for optimising the most relevant kernel/phases. Depending on the nature of the code different optimisation strategies are possible. All the optimisation efforts should be aimed at the bottlenecks, which can be communications, lack of vectorisation, poor data locality, cache thrashing, etc. Some optimisations are more critical on Xeon Phi than on Xeon. However, the strategies are similar for both architectures, and they apply to both platforms.

The optimisations can have an impact on the previously gathered information. Therefore, after their implementation the whole analysis procedure should be repeated, to update the information with the new situation.

After all these efforts enough data is available to properly map the application into the DEEP System. As a rule of thumb, parts that are highly scalable, threadable and vectorisable can run efficiently on the Booster. Given the sensitivity of the Xeon Phi to code that does not meet these requirements, it is commonly advisable to map code that does not match that description to the Cluster. There are a few extra things to consider. Phases that can be overlapped benefit from an application division, even if both map to the same part of the system. Cluster-to-
Cluster or Booster-to-Booster offloads are possible and supported by the DEEP programming environment. Secondly, the connectivity of the phases is of upmost importance. Assessing the compute time and the communication time between separated parts is a necessary step not to suffer from excessive overhead later on. This is a good moment to consider future developments and use-cases of the application. Examples are online pre- or post-processing, alternative I/O strategies or coupling of models in multi-physic applications. Also, depending on the application structure, the gathered information and the mapping decision, the offload can be direct (Cluster to Booster), reverse (Booster to Cluster), discrete (kernels without communications), OmpSs based, MPI_Comm_spawn based, or a combination of many of these possibilities.

With an optimised code and an informed decision the developer can implement the application division. This might require refactoring the phases to implement clean interfaces between those that are no longer running in the same part of the system.

With a working code-division, the last remaining action is repeating the whole procedure to optimise the division. In this kind of environment an appropriate management of the communications is mandatory. Therefore, the developer should focus on minimising their impact through communication-overlapping techniques. Non-blocking communication, multithreaded communications, tuning the granularity of the tasks, or tuning buffer sizes if possible are all advisable.

9.2. Recommendation for improvements

The DEEP System is the first realisation of the DEEP concept. As such, there are areas that evidently need further work. The sections below highlight some of the recommendations gathered by WP8.

9.2.1. I/O

The DEEP System was never intended to provide strong I/O capabilities. The I/O subsystem attached to it has limited performance due to its size and design. However, this is not a concern since in principle the Cluster can be connected to any kind of state-of-the-art parallel filesystem. On the other hand, the Booster I/O is intrinsically inefficient. There are multiple reasons for it:

- The Booster nodes are connected to I/O through intermediate nodes (the BICs).
- The underlying protocol -NFS- is inherently inefficient for parallel filesystems.
- The I/O traffic shares network interfaces with the computing traffic.
- The I/O system calls are particularly slow on Xeon Phi, due to its bad single-thread performance on non-computing tasks.

In the current hardware realisation offloading I/O from the Booster to the Cluster is mandatory to achieve satisfactory performance. It should not be forgotten that in most cases the I/O offload have extra benefits (like allowing overlapping of I/O with computation and offloading some of the pre-steps required for I/O). However, it is not difficult to imagine situations where doing I/O from the Booster would be desirable -if handled efficiently-. Thus, a faster and scalable I/O infrastructure -from the Booster and from the Cluster-, is advisable for future iterations.
9.2.2. Second generation Xeon Phi

The Xeon Phi coprocessor has outstanding peak performance thanks to its wide vector units, large amount of cores, and 4-way SMT capabilities. However, any kind of code that cannot benefit from these 3 facts at the same time will perform poorly. During the DEEP project it became apparent that unimportant phases in regular processors can become problematic in Xeon Phi coprocessors. Collective communications, message passing of a few kilobytes, or operations with not-strictly-optimal memory access patterns become bottlenecks quite easily. There are two main reasons for this behaviour:

- Low clock frequency (compared to Xeon counterparts).
- In-order cores.

Given the energy requirements of modern supercomputing and processor technology, having a significant bump in clock speed is unlikely in the near future. In fact, the opposite trend can be observed, where 3GHz+ processors were more prominent years ago than they are now. Therefore, this is an issue without immediate solution.

On the other hand, out-of-order cores are known to be better at hiding the effects of suboptimal code. Thus, tasks that require good single-thread performance will behave better in these cores than in in-order cores. The current expectation is that the second generation of Xeon Phi coprocessors, codenamed KNL, with out-of-order cores, will significantly reduce the high latencies observed in KNC coprocessors. Reducing the latency to the minimum is an unavoidable requirement for Exascale, and thus the DEEP concept will never be feasible at that level without an improvement on this area.

9.2.3. Resiliency

On the road to Exascale there are multiple voices raising their concern about the lack of resiliency features in the current software stack. The hardware itself is fairly resilient, but it is becoming more and more clear that with the manufacturing miniaturization the amount of errors that the software have to deal with will steadily increase over the next years.

The techniques to cope with hardware failures are numerous, and often require modifications in the applications code (e.g.: checkpointing). However, the underlying software stack and the system design have to enable these resiliency features, and efficiently support them. The I/O and memory hierarchies are changing at a considerable rate, and these are precisely two areas that might require a tight integration between the system and the software design.

Resiliency has been partly addressed in WP7 with the development of a very scalable RAS system. This allows exhaustive monitoring and pre-emptive maintenance if necessary. However, work closer to the final users is still pending and therefore it is an area for improvement.

9.2.4. Enhanced tools support

It is generally acknowledged that parallel programming is a difficult and error prone task. Even more so when the applications and algorithms have to scale to a large number of threads. Bugs that show up just in complex applications -as opposed to isolated kernels-, and bugs that manifest themselves when running in the multi-thousand-threads range can be difficult to track down.

As part of the DEEP project performance tools have been ported to the architecture. The two ported tools have quite different characteristics. Scalasca is an -relatively- easy to use tool focused on pinpointing scalability issues, whereas Extrae and its associated data visualizer -
Paraver- can be used to identify all sort of performance problems, at the cost of a steeper learning curve. These two tools, together with Intel's Vtune, have been extensively used during the project to address intranode and internode performance issues. Despite this fact, some of the particularities of the architecture call for further performance tool support. In particular, offloading -either via OmpSs or using MPI_Comm_spawn directly- support can be enhanced, as right now it requires analysing at least 2 separate traces.

More troublesome is the lack of parallel debuggers running on the system. The DEEP concept have more chances of succeeding if there is an usable parallel debugger supporting both sides of the system, and able to support intercommunicators created via MPI_Comm_spawn. Currently, having debuggers like Totalview running on the system should be possible. However, support for dynamic process allocation is missing, and therefore any application division debugging will have limited functionality.

9.2.5. OmpSs

OmpSs -both compiler and runtime- improved significantly during the course of the project. As part of the co-design effort, the OmpSs offload was made flexible enough to support many different usage models and features. Reverse offload, offload of large or small kernels or early release of dependencies are examples of that effort.

However, there are a couple of improvements that would have a positive impact for the DEEP users:

- Support for external runtimes. There are cases where the OmpSs offload can be handy, but using a different intranode threading runtime. Intel's OpenMP runtime is popular among HPC application developers, and having an OmpSs offload + OpenMP compatibility is desirable.

- In some cases, remote cores are used as processing units for discrete tasks, just the same as local cores are used in the "traditional" shared-memory task-based model. However, mixing local and remote cores is not transparent to the user. In certain applications it might be desirable to reduce the number of MPI processes per node (because of memory or network bandwidth requirements for instance) but use the otherwise idling cores, together with remote cores, to speedup certain parts.

These additions would make the application development and porting an easier task, and will enable them to run more efficiently on the system.
Bibliography


List of Acronyms and Abbreviations

A

AC/DC: Alternating Current / Direct Current
ADI3 layer: MPICH Abstract Device Interface Version 3
AMR: Adaptive Mesh Refinement
API: Application Programming Interface
ASHRAE: American Society of Heating, Refrigerating and Air-Conditioning Engineers
ASIC: Application Specific Integrated Circuit: Integrated circuit customized for a particular use
ATLANTIS: Project led by the Cyprus Institute
ATOLL: Predecessor of EXTOLL
Aurora: The name of Eurotech’s cluster systems
AVBP: A parallel CFD code for reactive unsteady flow simulations on hybrid grids developed by partner CERFACS

B

BADW-LRZ: Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften.Computing Centre, Garching, Germany
BACNet: Building Automation and Control Network; a protocol used in building automation and monitoring. In DEEP, BACNet is used to manage the cooling and power infrastructure surrounding the DEEP System.
BAR: Base Address Register
BBP: Blue Brain Project
BEINGRID: Business Experiments in GRID: An FP6 EU project
BI: Booster Interface (functional entity)
BIC: Booster Interface Card: Interface card to connect the Booster to the Cluster InfiniBand network
BIC evaluator: A platform consisting of three x86-based nodes equipped with (i) an EXTOLL NIC, (ii) an InfiniBand HCA, (iii) both, EXTOLL NIC and InfiniBand HCA, developed and used only in the DEEP project
BLAS: Basic Linear Algebra Subprograms: Standard application programming interface to publish basic linear algebra libraries
BlueGene/Q: Supercomputing architecture developed by IBM, well known for its energy efficiency, massive parallelism, 5D torus network and wide vector units.
BMC: Baseboard Management Controller
BN: Booster Node (functional entity)
BNC: Booster Node Card: A physical instantiation of the BN
BNC evaluator: Same as EXTOLL evaluator
BoF: Birds of a Feather Session: Informal meeting during a Conference where people can discuss about the topic of their common interest
Booster System: Hardware subsystem of DEEP comprising of BNC, BIC and Intra-Booster network
BoP: Board of Partners for the DEEP project
BRM: Booster Resource Manager: Controlling the DEEP System
BSC: Barcelona Supercomputing Centre, Spain
BSCW: Basic Support for Cooperative Work: Software package developed by the Fraunhofer Society, used to create a collaborative workspace for collaboration over the web
CAABA: Chemistry as a Boxmodel Application
CAN: Controller Area Network
CANBus: Controller Area Network Bus: Vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer.
CARTABLANCA team: Won the RD100 price for the development of high performance software for fluid modelling
CaSToRC: Computation based Science and Technology Research Centre, Cyprus
CAT6: Category 6 network cable
CBP: Cluster-Booster protocol
CEPBA: European Centre for Parallelism, Barcelona, Spain
CERFACS: Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique, Toulouse, France
CFD: Computational Fluid Dynamics
CG: Conjugate Gradient
CGGVS: CGGVeritas Services SA, Paris, France
CINECA: Consorzio Interuniversitario, Bologna, Italy
CMS: Content Management System
CN: Cluster Node (functional entity)
CNC: Computerized Numerical Control
CNR: National Research Council, Italy
CNRS: Centre National de la Recherche Scientifique, Paris, France
COI: Coprocessor Offload Infrastructure
COMP: The Community of OpenMP Users, Researchers, Tool Developers and Providers
COO: Chief Operating Officer
CooLMUC: Prototype at BADW-LRZ with direct warm water cooling
Coordinator: The contractual partner of the European Commission (EC) in the project
CPA: Centrum voor Plasma-Astrofysica, Leuven, Belgium
CPLD: Complex Programmable Logic Device
CPU: Central Processing Unit
CRAC: Computer room air conditioning
CRC: Cyclic Redundancy Check
CRESTA: Collaborative Research into Exascale Systemware, Tools & Applications: EU-FP7 Exascale Project led by the University of Edinburgh.
CSCS: Swiss Center for Scientific Computing, Swiss National Supercomputing Centre
CSIX protocol: Common Switch Interface (CSIX): A physical interface specification between a traffic manager and a switching fabric
CTO: Chief Technology Officer
CUDA: Compute Unified Device Architecture: Parallel computing architecture developed by NVIDIA
CYI: Cyprus Institute, Nicosia, Cyprus

DARPA: Defense Advanced Research Agency
DC: Direct Current (electricity)
DDG: Design and Developer Group of the DEEP project
DEEP: Dynamical Exascale Entry Platform: EU-FP7 Exascale Project led by Forschungszentrum Juelich
DEEP Architecture: Functional architecture of DEEP (e.g. concept of an integrated Cluster Booster Architecture)
DEEP Booster: Booster part of the DEEP System
DEEP Supercomputer: A future Exascale supercomputer based on the DEEP Architecture
DEEP System: The production machine based on the DEEP Architecture developed and installed by the DEEP project
DEISA2: Distributed European Infrastructure for Supercomputing Applications (EU project)
DESCA: Comprehensive, modular consortium agreement for the Seventh Framework Programme (FP7)
DFF: Dense Form Factor
DFG: Deutsche Forschungsgemeinschaft: German research organisation
DGEMM: Double precision General Matrix Matrix multiplication
DGEMV: Matrix-vector multiplication
DHCP: Dynamic Host Configuration Protocol
Dimemas: A performance analysis tool for message-passing programs developed at BSC
DMA: Direct Memory Access
DOE: Department of Energy, US Government
DoW: Description of Work: Annex I of the Grant Agreement
DSL: Domain Specific Language
DVFS: Dynamic voltage and frequency scaling

E
EC: European Commission
ECHAM: ECMWF and Hamburg (Global climate model)
ECL: ExaCluster Laboratory: A collaboration of Intel, ParTec and JUELICH to develop cluster management software for Exascale computing
ECMWF: European Centre for Medium-Range Weather Forecast
EESI: European Exascale Software Initiative (FP7)
EMAC: ECHAM/MESSy (Application coupling together the ECHAM model with the MESSy framework)
EMEA: Europe, the Middle East and Africa: Regional designation used for government, marketing and business purposes
ENCEORE project: ENabling technologies for a programmable many-CORE (EU project)
Energy Efficiency evaluator: Platform used for the investigations of the energy-aware functionality of DEEP, used only in the DEEP project
ENI: Italian Oil and Gas Company ENI, Italy
ENSEEIHT: Ecole Nationale Supérieure d'Electrotechnique, d'Electronique, d'Informatique, d'Hydraulique et des Télécommunications, Toulouse, France
ENSICA: École nationale supérieure d'ingénieurs de constructions aéronautiques, Toulouse, France
EPFL: École Polytechnique Fédérale de Lausanne, Switzerland
eQPACE: European project to develop global communications for the QPACE architecture
ESA: European Space Agency
ESFRI list: List of large scale projects: Defined by the European Strategy Forum on Research Infrastructures

ETP4HPC: European Technology Platform for High Performance Computing
EU: European Union
EUROPLANET: A European Research Infrastructure for Planetary Science (FP7)
Eurotech: Eurotech S.p.A., Amaro, Italy
Exaflop: $10^{18}$ floating point operations per second
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
EXTOLL evaluator: Platform for evaluation of EXTOLL technology, developed and used in the DEEP project

F

FDM: Finite Difference Method
FDR: Fourteen Data Rate: Communication signalling technique of InfiniBand
FFTW: Fastest Fourier Transform in the West
FIFO: First In First Out
FIR: Filter Impulse Response
FLOP: Floating point Operation
FMA: Fused Multiply-Add
FPGA: Field-Programmable Gate Array: Integrated circuit to be configured by the customer or designer after manufacturing

G

GASNet: Global Address Space Networking
Gb: Giga Bit
Gb/s: Giga Bit Per Second
GCM: Global Circulation Model
GCS: Gauss Centre for Supercomputing: The alliance of the three national supercomputing centres in Germany (Garching, Jülich and Stuttgart)
GDDR: Graphics Dual Data Rate RAM. Based on the same storage array as conventional DDR devices it provides a significantly improved access bandwidth (~8x per device). The improved interface also implies lower aggregated capacity.
GDDR5: Graphics Double Data Rate DRAM, version 5 typically used in graphics adapters
GEM: Geospace Environment Modelling
GFlop/s: Gigaflop, $10^{12}$ floating point operations per second
GID: Cell global identifier in the CoreBluron simulator.
Global MPI: MPI allowing communication between the Booster and Cluster part of the DEEP System. Based on the ParaStation process-management and the Cluster-Booster protocol acting as a plug-in for the pscom library. Provides the MPI_Comm_spawn() call used by application processes running on the CNs to start additional processes on the BNs.
GMRES: Generalised Minimum Residual
GPFS: General Parallel File System by IBM
GPU: Graphics Processing Unit
GREEN500 list: Provides rankings of the 500 top most energy-efficient supercomputers in the world

Gridmonitor: Part of the ParaStationV5 cluster suite: A versatile system monitor for Linux-based compute clusters

GRS: German Research School for Simulation Sciences GmbH, Aachen and Juelich, Germany

GSB: “Government Site Builder”: The Content Management System, used for the web site of the DEEP project

H

H4H: Hybrid programming For Heterogeneous architectures (EU project)

HCA: Host Channel Adapter

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

Healthchecker: Part of the ParaStationV5 cluster suite: A test suite to ensure the usability of compute and service nodes within a compute cluster

Helmholtz Association: German research organisation

HMI: Hardware Monitoring Infrastructure

HOPSA: HOlistic Performance System Analysis (EU-Russia FP7 project)

HPC: High Performance Computing

HTc: High critical temperature superconductors

HW: Hardware

I

IA: Intel Architecture

IB: InfiniBand

I²C / I2C: Inter-Integrated Circuit

ICPP: International Conference on Parallel Processing: Yearly conference on parallel and distributed computing

ICT: Information and Communication Technologies

IEEE: Institute of Electrical and Electronics Engineers

IFP: Institut Français pour le pétrole, Rueil-Malmaison, France

IMM: Implicit Moment Method

INFN: Istituto Nazionale di Fisica Nucleare, Italy

INFSO: Information Society

Intel: Intel GmbH Braunschweig, Germany

Intel Xeon® Phi™: Official product name of the Intel Many Core (MIC) architecture processors. The first available Intel Xeon® Phi™ product is code-named Knights Corner (KNC).

Interconnect evaluator: Hardware for interconnect studies on physical and mechanical layer, developed and used in the DEEP project

INP: Institut National Polytechnique, Toulouse, France

I/O: Input/Output

IP: Intellectual Property or Internet Protocol (depending on the context)

IPC: Instructions Per Cycle

iPIC3D: Programming code developed by the University of Leuven to simulate space weather

IPMI: Intelligent Platform Management Interface

IRST: Institute of Scientific and Technologic Research, Trento, Italy
ISC: International Supercomputing Conference: Yearly conference on supercomputing which has been held in Europe since 1986

IT: Information Technology

ITEA-2: Strategic pan-European programme for advanced pre-competitive R&D in software for Software-intensive Systems and Services

J

JSC: Juelich Supercomputing Centre
JUDGE: Juelich Dedicated GPU Environment: A cluster at the Juelich Supercomputing Centre
JUELICH: Forschungszentrum Jülich GmbH, Jülich, Germany
JUQUEEN: Juelich BlueGene/Q: A supercomputer installed at the Juelich Supercomputing Centre
JUST: Jülich Storage Cluster: JUELICHs GPFS file server

K

KB: Kilo Bytes
KNC: Knights Corner: Code name of a processor based on the MIC architecture. The commercial name of this product is Intel Xeon® Phi™.
KNF: Knights Ferry: Intel first available processor based on the MIC
KPP: Kinetic Pre-Processor
KULeuven: Katholieke Universiteit Leuven, Belgium
KVS: Key-Value Space

L

LANL: Los Alamos National Laboratory, USA
LAPACK: Linear Algebra PACKage
LASA: Leuven Centre for Aero & Space Science, Technology and Applications, Brussels, Belgium
LDAP: Lightweight Directory Access Protocol
LES: Large Eddy Simulation
LGFMC: Lattice Green Function Monte Carlo
LHS: Left Hand Side
LinkSCEEM: Linking Scientific Computing in Europe and the Eastern Mediterranean (EU project)
LINPACK: Software library to perform numerical linear algebra calculations used as benchmark
LINUX: A Unix-like computer operating system assembled under the model of free and open source software development and distribution
LMCC: Leuven Mathematical Modelling & Computational Science Centre, Belgium
Lph: Litres per hour
Lpm: Litres per minute
LVDS: Low Voltage Differential Signaling

M

MA: Middle Atmosphere
Maui: Job scheduler for use on clusters and supercomputers
MB: Mega Byte or Mother Board (depending on the context)
**MC:** Monte Carlo
**MCELOG:** Advanced machine check handling on x86 linux
**MECCA:** Module Efficiently Calculating the Chemistry of the Atmosphere
**Mercurium compiler:** OmpSs’ source-to-source compiler
**MESSy:** Modular Earth Submodel System simulation code
**MHD:** Magneto-Hydro Dynamics
**MIC:** Intel Many Integrated Core architecture
**MIC evaluator:** Platform for evaluation of the MIC architectural concept, used only in the DEEP project
**MIC-OS:** Operating System of the MIC architecture
**Mini Booster prototype:** Minimal instantiation of a DEEP Booster used for analysis of the energy-aware functionality, developed and used in the DEEP project
**Mini DEEP System:** A fully featured DEEP System of minimal size comprising the Mini Booster
**MIUR:** Ministry of Education, University and Research, Italy
**MKL:** Intel® Math Kernel Library
**MLNX:** Mellanox Technologies, Ltd., Sunnyvale, California and Yokneam, Israel
**MMM@HPC:** Project of Multiscale materials modelling on high performance computer architectures
**MODULES:** Package for dynamic modification of a user's environment via modulefiles
**MOM:** Machine Oriented Mini Server: a daemon which sends information about a compute node's state
**Mont-Blanc:** European scalable and power efficient HPC platform based on low-power embedded technology: EU-FP7 Exascale Project led by the Barcelona Supercomputing Centre
**MPI:** Message Passing Interface: API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages
**MPI-M:** Max Planck Institute for Meteorology
**MPICH:** Freely available, portable implementation of MPI
**MPSS:** Intel many-core platform software stack. Software bundle to operate Xeon® Phi™ devices
**MRP:** Main RAS Plane
**MSI:** Message signaled interrupt. A method to propagate events through a PCI infrastructure based on writes in the common address space. MSI capabilities are advertised through the configuration space of a peripheral device.
**MSI-X:** An improved version of MSI with more event sources support.
**MTBF:** Mean Time Between Failures
**MQTT protocol:** Message Queue Telemetry Transport. Open message protocol for machine to machine communications. It enables the transfer of telemetry-style data in the form of messages from pervasive devices, along high latency or constrained networks, to a server or small message broker.

**N**

**Nanos++:** OmpSs’ execution runtime
**NASA:** National Aeronautics and Space Administration, Washington, USA
**NIC:** Network Interface Card: Hardware component that connects a computer to a computer network
**NLA:** Network Logical Address
NSF: National Science Foundation, USA

Obsai & CPRI standards committees: Developing wireless network interface standards
OGS: Institute of Oceanography and Experimental Geophysics, Italy
OmpSs: BSC’s Superscalar (Ss) for OpenMP
OpenCL: Open Computing Language to program GPUs
OpenMP: Open Multi-Processing: Application programming interface that support multiplatform shared memory multiprocessing
OpenMP ARB: Open MP Architecture Review Board: a group of leading hardware and software vendors and research organizations that create the OpenMP standard.
OS: Operating System

PAPI: Performance Application Programming Interface
ParaStation Cluster Suite: The Parastation MPI consisting of the MPI library (psmpi), the communication library (pscom) and the process management library (psmgmt)
ParaStation Consortium: Involved in research and development of solutions for high performance computing, especially for cluster computing
ParaStationMPI: Software for cluster management and control developed by ParTec
Paraver: Performance analysis tool developed by BSC
ParTec: ParTec Cluster Competence Center GmbH, Munich, Germany
PBS: Portable Batch System: precursor of Torque
PC: Normally Personal Computer, but in the context of the proposal also Project Coordinator
PCB: Printed Circuit Board: Board used in electronic to mechanically support and electrically connect electronic components
PCI: Peripheral Component Interconnect: Computer bus for attaching hardware devices in a computer
PCIe: PCI Express: Standard for peripheral interconnect, developed to replace the old standards PCI, improving their performance
PFlop/s: Petaflop, $10^{15}$ floating point operations per second
PHY: Physical Network Layer (Chip)
PIC: Particle In Cell
PM: Person Month or Project Manager of the DEEP project (depending on the context)
PMI: Process Management Interface
PMT: Project Management Team of the DEEP project
POD: Plain Old Data
PR: Public Relations
PRACE: Partnership for Advanced Computing in Europe (EU project, European HPC infrastructure)
PRACE-IIIP: PRACE First Implementation Phase (EU project)
Project Coordinator: Leading scientist coordinating and representing the DEEP project
PROSPECT: Promotion of Supercomputing Partnerships for Economic Competitiveness and Technology (registered association, Germany)
Proto-Booster: Minimal instantiation of a DEEP Booster based on early access technologies (EXTOLL FPGA and KNC in PCIe form factor). Developed and used in the DEEP project for software development

Psid: The ParaStation MPI process management daemon executed on each node of the DEEP system

Psmgmt: ParaStation MPI process management library

Psmom: ParaStation MOM is a plugin to the ParaStation process management daemon (psid) responsible for communicating with the central batch system server and executing applications on the nodes

PSU: Power Supply Unit

PUE: Power Usage Effectiveness

Q

QCDOC: Quantum ChromoDynamics On a Chip: Special supercomputer developed by Universities of Edinburgh, Columbia and by IBM

QDR: Quad Data Rate: Communication signalling technique of InfiniBand

QMC: Quantum Monte Carlo

QPACE: Specialised supercomputer for QCD Parallel Computing on CELL processors

QSFP: Quad Small Form-factor Pluggable: hot-pluggable transceiver used for data communications

R

R&D: Research and Development

RAS: Reliability, Availability and Serviceability

RDMA: Remote Direct Memory Access

RHS: Right Hand Side

RMA: Remote Memory Access: A protocol for remote memory access between EXTOLL NICs

RML: Risk management list used in the DEEP project

RTD: Research and Technological Development

RTM: Reverse Time Migration

RX/TX: ReceiverX/TransmitterX

S

SAS: Serial Attached SCSI: point-to-point serial protocol for data movement

SATA: Serial Advanced Technology Attachment: Computer bus interface for connecting host bus adapters to mass storage devices such as hard disk drives and optical drives

SC: International Conference for High Performance Computing, Networking, Storage, and Analysis, organized in the USA by the Association for Computing Machinery (ACM) and the IEEE Computer Society

Scalasca: Performance analysis tool developed by JUELICH and GRS

SCIF: Symmetric Communication Interface. A low level communication layer based on shared memory regions, which are not coherent. Used by MPSS as foundation for any systems or application related communication between Xeon® Phi™ devices and host systems.

SDK: Software Development Kit

SDS: Seismic Data Server
**SI:** Signal integrity: A set of measures of the quality of an electrical signal

**SIMD:** Single Instruction, Multiple Data. Describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously

**SISSA:** International School of Advanced Studies, Trieste, Italy

**SMART:** Monitoring system for computer hard disk drives

**SMBus:** System Management Bus

**SME:** Small and Medium Enterprises

**SMFU:** Shared Memory Functional Unit

**SNMP:** Simple Network Management Protocol

**SOLAIRE Marie Curie RTN:** EU program granting activities within the Research Training Network

**SOTERIA:** Project for the collection, organisation and the use of space physics data aimed at better understanding space weather (EU project)

**SRMIP:** Soubaras-Remez Migration Parallel. Simulation code for seismic imaging used at partner CGGVS

**SS:** Superscalar, programming environment developed by BSC

**SSD:** Solid State Disk

**StarSs:** Generic programming environment developed by BSC

**Stampede:** Supercomputer (Dell PowerEdge C8220 Cluster with Intel Xeon Phi coprocessors) installed at Texas Advanced Computing Center from Univ. of Texas, in the USA. It is nr. 7 in the TOP500 list today

**STRATOS:** PRACE advisory group to foster development of HPC technologies in Europe

**SW:** Software

**SWIFF:** Space Weather Integrated Forecasting Framework, Leuven

**SysFs:** Virtual file system provided by the Linux kernel to export information from devices and drivers to the user.

**T**

**TCO:** Total Cost of Ownership

**TEXT:** Towards Exaflop Applications (EU project)

**TFlop/s:** Teraflop, $10^{12}$ floating point operations per second

**TFTP:** Trivial File Transfer Protocol

**Tier-0, Tier-1, ...:** Different classes of supercomputers ordered by their performance

**TIM:** Thermal Interface Material

**TK:** Task, followed by a number: Term to designate a task inside a work package of the DEEP project

**Torque:** Distributed resource manager providing control over batch jobs and distributed compute nodes

**ToW:** Team of Work Package leaders within the DEEP project

**TP10:** Third Party under Clause 10

**TurboRVB:** Quantum Monte Carlo Software for electronic structure calculations, developed by SISSA

**U**

**UniHD:** University of Heidelberg, Germany

**UniReg:** University of Regensburg, Germany

**UPC:** Universitat Politècnica de Catalunya, Barcelona, Spain

**UPS:** Uninterruptible Power Supply
V
- **VELO**: Virtualized Engine for Low Overhead: An EXTOLL communications channel
- **VELO**: Very Efficient Low-Overhead

VMC: Variational Monte Carlo
VR: Voltage Regulation
VLSI: Very-large-scale integration: Process of creating integrated circuits

W
- **WP**: Work Package

X
- **x86**: Family of instruction set architectures based on the Intel 8086 CPU

Y

Z
- **ZITI Heidelberg**: Institut für Technische Informatik Uni Heidelberg, Germany