D5.2 Interim report on application/kernel plans, evaluations and benchmark suite releases

<table>
<thead>
<tr>
<th>Project Acronym</th>
<th>INTERTWinE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Title</td>
<td>Programming Model INTERoperability ToWards Exascale</td>
</tr>
<tr>
<td>Project Number</td>
<td>671602</td>
</tr>
<tr>
<td>Instrument</td>
<td>Horizon 2020</td>
</tr>
<tr>
<td>Thematic Priority</td>
<td>FETHPC</td>
</tr>
<tr>
<td>Due date</td>
<td>M18</td>
</tr>
<tr>
<td>Submission date</td>
<td>31/MAR/2017</td>
</tr>
<tr>
<td>Project start date</td>
<td>01/OCT/2015</td>
</tr>
<tr>
<td>Project duration</td>
<td>36 months</td>
</tr>
<tr>
<td>Deliverable lead organization</td>
<td>KTH</td>
</tr>
<tr>
<td>Version</td>
<td>1.1</td>
</tr>
<tr>
<td>Status</td>
<td>Final</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Roman Iakymchuk (KTH), Maria Barreda Vaya (UJI), Rocio Carratala Saez (UJI), Enrique Quintana-Orti (UJI), Dana Akhmetova (KTH), Jakub Sistek (Manchester), Luis Cebamanos (EPCC), David Stevens (Manchester), Thomas Gerhold (DLR), Barbara Brandfass (DLR) and Xavier Teruel (BSC)</td>
</tr>
<tr>
<td>Reviewer(s)</td>
<td>George Beckett (EPCC), Olivier Aumage (Inria), Valeria Bartsch (Fraunhofer), Mark Bull (EPCC)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Public</td>
</tr>
</tbody>
</table>
## Version History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Comments, Changes, Status</th>
<th>Authors, contributors, reviewers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>08/FEB/17</td>
<td>Initial structure of the document</td>
<td>Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>0.2</td>
<td>03/MAR/17</td>
<td>Merged short reports on GraphBLAS, iPIC3D, PLASMA, and Ludwig</td>
<td>Maria Barreda Vaya (UJI), Enrique Quintana-Orti (UJI), Dana Akhmetova (KTH), Roman Iakymchuk (KTH), Jakub Sistek (Manchester), Luis Cebamanos (EPCC)</td>
</tr>
<tr>
<td>0.3</td>
<td>06/MAR/17</td>
<td>Merged short reports on TAU linsolv_kernel; enhanced introduction</td>
<td>Barbara Brandfass (DLR), Thomas Gerhold (DLR), Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>0.4</td>
<td>07/MAR/17</td>
<td>Updated the Ludwig part</td>
<td>Luis Cebamanos (EPCC)</td>
</tr>
<tr>
<td>0.5</td>
<td>08/MAR/17</td>
<td>Updated the PLASMA part; completed executive summary, introduction, and section on API; enhanced the TAU part</td>
<td>Jakub Sistek (Manchester), David Stevens (Manchester), Roman Iakymchuk (KTH), Barbara Brandfass (DLR)</td>
</tr>
<tr>
<td>0.6</td>
<td>08/MAR/17</td>
<td>Ready for internal review</td>
<td>Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>0.7</td>
<td>15/MAR/17</td>
<td>Updated the BAR section</td>
<td>Xavier Teruel (BSC)</td>
</tr>
<tr>
<td>0.8</td>
<td>27/MAR/17</td>
<td>Addressed reviewers’ comments</td>
<td>Xavier Teruel (BSC), Maria Barreda Vaya (UJI), Enrique Quintana-Orti (UJI), Dana Akhmetova (KTH), Jakub Sistek (Manchester), Luis Cebamanos (EPCC), Barbara Brandfass (DLR), Thomas Gerhold (DLR), Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>0.9</td>
<td>28/MAR/17</td>
<td>Enhanced introduction, summary, and section 2</td>
<td>Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>0.95</td>
<td>30/MAR/17</td>
<td>One more round of reviewing; enhanced executive summary; updated the GraphBLAS part concerning the HierMatFact (OpenMP + MKL) version released on M18</td>
<td>Roman Iakymchuk (KTH), Olivier Aumange (Inria), George Beckett (EPCC), Jakub Sistek (Manchester), Luis Cebamanos (EPCC), Barbara Brandfass (DLR), Maria Barreda Vaya (UJI), Enrique Quintana-Orti (UJI)</td>
</tr>
<tr>
<td>1.0</td>
<td>31/MAR/17</td>
<td>Final version</td>
<td>Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>1.1</td>
<td>27/JUL/17</td>
<td>Revisions made in light of feedback from project reviewers</td>
<td>Roman Iakymchuk (KTH); Mark Bull (EPCC); Olivier Aumange (Inria); George Beckett (EPCC); Jakub Sistek (Manchester), Luis Cebamanos (EPCC); Barbara Brandfass (DLR); Maria Barreda Vaya (UJI); Enrique Quintana-Orti (UJI)</td>
</tr>
</tbody>
</table>
Table of Contents

1 Executive Summary ................................................................. 7

2 Introduction ............................................................................. 9
  2.1 Glossary of Acronyms ......................................................... 9

3 Prioritized API Combinations .................................................. 11

4 Co-design cycle: A WP5 Role ..................................................... 13
  4.1 Collaboration with WP3 .......................................................... 13
  4.2 Collaboration with WP4 .......................................................... 14
    4.2.1 Applications/Kernels using the Directory/Cache Service .......... 14
    4.2.2 Applications/Kernels using the Resource Manager Service .......... 15
  4.3 Summary .............................................................................. 15

5 Applications/Kernels ................................................................ 16
  5.1 Ludwig: A parallel Lattice-Boltzmann code for complex fluids .......... 16
    5.1.1 Introduction .................................................................... 16
    5.1.2 INTERTWinE Ambition .................................................... 16
    5.1.3 GASPI + MPI .................................................................. 17
    5.1.4 Multithreaded MPI + OpenMP threads ................................. 24
    5.1.5 Analysis ....................................................................... 26
  5.2 iPIC3D: Implicit Particle-in-Cell Code for Space Weather Applications .... 27
    5.2.1 Introduction .................................................................... 27
    5.2.2 INTERTWinE Ambition .................................................... 28
    5.2.3 Multithreaded MPI + OpenMP threads ................................. 28
    5.2.4 Multithreaded MPI + OpenMP tasks ..................................... 34
    5.2.5 Analysis ....................................................................... 38
  5.3 TAU-kernel linsolv ................................................................. 39
    5.3.1 Introduction .................................................................... 39
    5.3.2 INTERTWinE Ambition .................................................... 39
    5.3.3 MPI + OpenMP Implementation (threads) ............................. 39
    5.3.4 MPI + OpenMP Implementation (explicit tasks) .................... 43
    5.3.5 Analysis ....................................................................... 46
  5.4 BAR: Barcelona Application Repository .................................... 47
    5.4.1 Introduction .................................................................... 47
    5.4.2 INTERTWinE Ambition .................................................... 48
    5.4.3 Correctness .................................................................... 48
    5.4.4 Evaluation environment .................................................... 49
    5.4.5 Performance results .......................................................... 49
    5.4.6 Portability ..................................................................... 52
    5.4.7 Discussion of Further Collaboration with WP4 ....................... 52
  5.5 Graph-BLAS ........................................................................... 52
    5.5.1 ILU0: MPI + OmpSs/OpenMP ............................................ 52
    5.5.2 HGraphBLAS HierMatFact: MKL + OmpSs/OpenMP .............. 59
  5.6 PLASMA: Parallel Linear Algebra Software for Multicore Architectures ... 68
    5.6.1 Introduction .................................................................... 68
    5.6.2 INTERTWinE Ambition .................................................... 68
    5.6.3 PLASMA + OpenMP tasks ................................................ 69

6 Updated Software Release Plans ............................................... 74

7 Summary .................................................................................. 76
Index of Figures

Figure 1: INTERTWinE co-design cycle. ................................................................. 13
Figure 2: 3D domain decomposition in Ludwig. .................................................... 16
Figure 3: Lattice subdomain where the internal section represents the real lattice and the
external region the halo sites [14]. ........................................................................ 18
Figure 4: Setting up MPI and GASPI environments in Ludwig. ................................. 18
Figure 5: GASPI pointers to GASPI segments in the YZ plane. ................................. 19
Figure 6: GASPI Write and Notify in the YZ plane. .................................................. 20
Figure 7: Bandwidth and message size on ARCHER. Data obtained using the OSU
benchmarks [13]. ........................................................................................................ 21
Figure 8: Data transfer size by each process at the end of 192^3 lattice size simulation.
................................................................................................................................. 21
Figure 9: Strong scaling results of a 192^3 lattice simulation on ARCHER. ................. 22
Figure 10: Total loop time comparison in Ludwig for a 192^3 lattice size simulation. .... 22
Figure 11: Performance difference on a single fully populated (24 cores) node on
ARCHER. ....................................................................................................................... 23
Figure 12: Inter-node total loop time comparison. ...................................................... 23
Figure 13: MPI and OpenMP initialization. ............................................................... 24
Figure 14: Pseudocode of the halo exchange section in Ludwig. ............................... 24
Figure 15: Code snippet of the halo exchange code using MPI + OpenMP. ................. 25
Figure 16: Performance and scaling results of a 192^3 lattice simulation on 4 ARCHER
nodes. The solid line represents the speed up achieved. ........................................ 25
Figure 17: Performance and scaling result of a 192^3 lattice simulation on 8 ARCHER
nodes. The solid line represents the speed up achieved. ........................................ 26
Figure 18: Computational steps in iPIC3D. ............................................................... 28
Figure 19: OpenMP threading in an MPI code. ......................................................... 29
Figure 20: Code snippet of communication between faces in Halo exchange in iPIC3D
with MPI_THREAD_MULTIPLE using four OpenMP threads per MPI process. ......... 31
Figure 21: Code snippet of Halo exchange in iPIC3D (communication between faces)
with MPI_THREAD_SINGLE. ...................................................................................... 31
Figure 22: Weak scaling tests for the four different simulations of the original and new
versions of iPIC3D. ....................................................................................................... 34
Figure 23. Weak scaling tests for the four different simulations of the original and
"MPI_THREAD_MULTIPLE + OpenMP tasks" versions of the iPIC3D code. ............. 37
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 24: LU decomposition; MPI + OpenMP (threads) .......................................................... 40
Figure 25: Point implicit; MPI + OpenMP (threads) ................................................................. 41
Figure 26: Jacobi; MPI + OpenMP (threads) ........................................................................... 41
Figure 27: Gauss-Seidel; MPI + OpenMP (threads) ................................................................. 42
Figure 28: symmetric Gauss-Seidel; MPI + OpenMP (threads) ............................................... 42
Figure 29: Relative performance; MPI + OpenMP (threads) ..................................................... 43
Figure 30: Parallel speedup for all methods; MPI + OpenMP (threads) .................................. 43
Figure 31: LU decomposition; MPI + OpenMP (tasks) ............................................................ 44
Figure 32: Point implicit; MPI + OpenMP (tasks) ................................................................. 45
Figure 33: Jacobi; MPI + OpenMP (tasks) .............................................................................. 45
Figure 34: Relative performance; MPI + OpenMP (tasks) ....................................................... 46
Figure 35: Parallel speedup for LU, PNT and JAC; MPI + OpenMP (tasks) ......................... 46
Figure 36: Matrix multiplication grain size analysis ................................................................. 50
Figure 37: Matrix multiplication strong scaling with MKL sequential (left) and MKL parallel (right). ........................................................................................................ 50
Figure 38: Cholesky Factorization strong scaling with MKL sequential (left) and MKL parallel (right). ........................................................................................................ 51
Figure 39: N-Body strong scaling pure OmpSs (left) and MPI + OmpSs (right) .................. 51
Figure 40: Algorithmic formulation of the the PCG method. Here, T_{max} is an upper bound on the relative residual for the computed approximation to the solution. .................. 53
Figure 41: Mapping of a TDG to 4 MPI ranks (R0-R3) with 2 OmpSs threads per rank. ........................................................................................................ 54
Figure 42: Execution time per PCG iteration for the Laplace A400 problem for different configurations, using 1 leaf per core, in MPI + OmpSs (left) and MPI + OpenMP (right). ........................................................................................................ 56
Figure 43: Execution time per PCG iteration for the Laplace A400 problem for different configurations, using 2 leaves per core, in MPI + OmpSs (left) and MPI + OpenMP (right). ........................................................................................................ 57
Figure 44: Execution time per PCG iteration for different Laplace problems and different configurations, using 1 leaf per core, in MPI + OmpSs (left) and MPI + OpenMP (right). ........................................................................................................ 57
Figure 45: Execution time per PCG iteration for different Laplace problems problems and different configurations, using 2 leaves per core, in MPI + OmpSs (left) and MPI + OpenMP. ........................................................................................................ 58
Figure 46: Blocked Right-Looking algorithm (BRL) for solving an LU factorization. .... 59
Figure 47: Performance of the task-parallel H-LU factorization using OmpSs in the Intel ES-2603v3 server. The notation in the legend specifies the programming model used for parallelizing the code and the number of cores used in each case. The X axis corresponds to the percent of dispersion of the matrices used for the tests (0%, 25%, 50% or 75%) and the Y axis refers to the amount of GFLOPS obtained. .................. 66
Figure 49: Performance of the QR factorization for different matrix sizes (ZGEQRF function). Theoretical peak performance is 3053 Gflop/s. ......................................................... 71
Figure 50: Performance of the Cholesky factorization for different matrix sizes (ZPOTRF function). Theoretical peak performance is 3053 Gflop/s. ......................................................... 72
Figure 51: Performance of the LU factorization for different matrix sizes (ZGETRF function): 8 threads per panel factorization. Theoretical peak performance is 3053 Gflop/s.

Index of Tables

Table 1: API extensions to the prioritised API combinations.................................................. 14
Table 2: Possible parallelization strategies for the phase of communication between faces......................................................................................................................... 30
Table 3: Combinations of number of OpenMP threads per MPI process (NT) and number of MPI processes per node (PPN)................................................................................................................. 40
Table 4: Matrices employed in the experimental evaluation, where $nz$ accounts only for the non-zeros in the upper triangular part of the matrix.................................................. 55
Table 5: Number of iterations required for convergence for each problem instance and number of leaves. .................................................................................................................................................................................. 55
Table 6: Configurations for the experimental evaluation of the H-LU factorization........ 63
Table 7: Release timeline for the applications/kernels and their API combinations. .... 75
1 Executive Summary

Based on the experience gained during the early months of the project and taking into account reviewers’ recommendations at the Project Check Meeting (on July 1st, 2016), the Work Package 5 team has developed a prioritisation of combinations of parallel models and interfaces (collectively referred to as the APIs), to ensure project resources are effectively employed to achieve the project’s ambitions.

The prioritisation identifies four key attributes that justify investing effort into particular API combinations, as follows:

- APIs that have significant uptake in existing application software (most notably MPI and OpenMP) are very important, as experience tells us these will remain significant in the timeframe of the first Exascale supercomputers and beyond;
- APIs that enable greater levels of parallelism (in particular, combining threading/tasks with distributed-memory APIs) to be exposed within algorithms are important, as greater parallelism is the most promising route to scaling applications for future generations of supercomputers;
- API combinations to validate and drive the functionality of the project’s Directory/Cache concept of supporting pure task models on distributed systems;
- API combinations to validate and drive the functionality of the project’s Resource Manager concept, which will allow applications using different runtimes or APIs to adapt successfully to heterogeneous hardware with their performance inhomogeneity and will avoid over- and under-subscription for resources.

A number of interoperability experiments have been undertaken in the first half of the project, which have contributed to the project team's experience of interoperability opportunities and issues, as well as influenced and helped the WP3 and WP4 teams, through the project's co-design cycle.

Exposing greater (or more layers of) parallelism is a promising technique to make topical applications Exascale-ready. Experiments to add threading or task-based parallelism to Ludwig, iPIC3D, and Tau have yielded some success, which has been reflected in Best Practice Guides issued by the wider project team, though have also highlighted significant issues both in the complexity and effort required to add such parallelism to introduction applications and in realising significant speed-up as a result of such changes. This experience has been relayed to WP3 and WP4 teams for further investigation.

An experiment (with Ludwig) to reduce the level of necessary synchronisation in the very common halo-exchange mechanism, which is used to communicate state between neighbouring processes in a domain-decomposed application has had mixed success. As with adding more parallelism, specialised effort is required to eliminate synchronisation points from the code, which will be conveyed to the supercomputing community through up-coming iterations of several Best Practice Guides. Further, the specific strategy to replace MPI message passing with GASPI has highlighted incompatibility in the way the two APIs handle data in messages, which will be investigated further in the second half of the project.

In preparation for the first full prototype of the Directory/Cache and the Resource Manager concepts, a benchmarking suite has been developed using BAR kernels, which is important to monitor and to showcase performance strengths and weaknesses.

Early in the project, we learned that development on the QUARK API was to cease and, responding to this, we have taken part in a revision to the PLASMA library, to replace QUARK by OpenMP tasks. Profiling suggests that the change of API has not impacted the performance of PLASMA, though has led to a simpler, easier to understand implementation. This work has concluded our involvement with QUARK within the project.
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

The software release plan has been updated to reflect progress in the first 18 months of the project. Progress remains on track and upcoming experiments are well-placed to build on the experience gained so far.
2 Introduction

The INTERTWinE project aims to improve the interoperability between different programming models to make the most of the first Exascale systems. Applications running on these systems will have to deal with a massively parallel and heterogeneous architecture with a complex memory hierarchy. Today, there is no single programming model that can deal with the challenging task of programming such a system in an efficient way. However, by carefully combining current programming models and parallel libraries, we can leverage the best features of each of the models to address the challenging task of programming an Exascale system.

In Work Package 5, the team takes the applications and kernels provided by project partners and ports them to appropriate combinations of APIs. For this purpose, the Work Package produces the revised software release plans and some of the actual software releases of the applications/kernels for use by WP3 and WP4; the initial plans and releases were previously described in Deliverable D5.1 [7]. When new or enhanced interoperability features are available in the respective runtime implementations, relevant advances will be re-implemented and optimized to assess these new interoperability features.

Proceeding in this manner, the project completes the co-design cycle, producing an initial set of applications to drive the requirements in WP3 and WP4; ensuring that the work done in these two work packages is used to make improvements to the applications and kernels; and ensuring that kernels and applications optimizations are fed back to WP3 and WP4. The goal of this document is to revise the initial plans for application/kernels provided by the partners, and present the interim evaluations and benchmark suite releases.

This document presents in Section 3 a prioritized API combinations list that aims to sharpen the focus of the project as well as to address the reviewers’ comments from the Project Check Meeting in PM09. Section 4 provides highlights of the co-design aspects. Section 5 outlines the findings and lessons learnt from the API combination development on the partners’ applications/kernels: Ludwig, iPIC3D, TAU, BAR, Graph-BLAS, and PLASMA. For each one of these, the document 1) outlines a given application/kernel; 2) justifies the API combinations choice; 3) summarizes the key implementation details, clearly stating the parts of applications being modified; 4) describes the preliminary performance results for the use case from the point of view of API combination; 5) provides analysis of the performance results with respect to the evaluation criteria; 6) discusses interoperability opportunities/ issues, further improvements, and collaboration with both WP3 and WP4. Section 6 revises the initial software release plans, while Section 7 draws conclusions.

2.1 Glossary of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BAR</td>
<td>Barcelona Application Repository</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subprograms</td>
</tr>
<tr>
<td>BFS</td>
<td>Breath First Search</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate Gradient (method)</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>D/C</td>
<td>Directory/Cache</td>
</tr>
<tr>
<td>DPLASMA</td>
<td>Distributed Parallel Linear Algebra Software for Multicore Architectures</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>GASPI</td>
<td>Global Address Space Programming Interface</td>
</tr>
<tr>
<td>GMRES</td>
<td>Generalized Minimal Residual (method)</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>iPIC3D</td>
<td>implicit Particle-in-Cell 3D Code for Space Weather Applications</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Linear Algebra Package</td>
</tr>
<tr>
<td>LB</td>
<td>Lattice Boltzmann</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>PLASMA</td>
<td>Parallel Linear Algebra Software for Multicore Architectures</td>
</tr>
<tr>
<td>RM</td>
<td>Resource Manager</td>
</tr>
<tr>
<td>SpMM</td>
<td>Sparse Matrix Multiplication</td>
</tr>
<tr>
<td>SpMV</td>
<td>Sparse Matrix-Vector Multiplication</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
3 Prioritized API Combinations

INTERTWinE has identified more than 20 relevant API combinations, covering the most popular and most widely used parallel constructs in applications from both academia and industry. Combining this amount of different API combinations with the number of target applications/kernels could result in more than 60 software releases. Such a large number of API combinations and even larger collection of the corresponding software releases has been highlighted by the reviewers at the Project Check Meeting in PM09.

Since the number of combinations of parallelization schemes is rather large, prioritization of these combinations should be done. This will allow other WPs to use the results of this WP even if not all combinations have been implemented.

On the way to prioritize the current extensive list of API combinations, we have identified the most common combinations of programming models used in industry:

- MPI + OpenMP, usually in master-only style, where all MPI calls occur outside of OpenMP parallel regions;
- Combinations of threading APIs inside a node, typically occurring where the application code uses a different API (e.g. OpenMP) from that used inside numerical libraries (e.g. Posix threads, Intel TBB, StarPU, PaRSEC).

Furthermore, we have identified those API combinations that could be exploited by industry as follows:

- MPI + OpenMP with the MPI multiple style, when multiple OpenMP threads make concurrent calls to the MPI library;
- MPI + GASPI, which allows effective and efficient asynchronous one-sided communication in key kernels, which cannot be achieved with MPI one-sided routines;
- MPI or GASPI + OpenMP tasks, which can reduce synchronization both inside and between nodes to the minimum actually required by the application's data dependencies, and be tolerant of performance unpredictability in future hardware due to complex memory systems and power-saving features.

Taking all the above-mentioned factors into account, our study of API combinations relevant to industry, our analysis of the interoperability between various programming models in Deliverable D3.1 [5], and the effects on (prioritized) API combinations in Deliverable D3.2 [6], we propose the following prioritized list of API combinations:

1. **MPI + OpenMP threads.** This combination covers the two de facto standards for distributed and shared memory architectures. MPI applications can be extended with OpenMP threads to improve their scalability on large core counts and reduce the memory footprint of the MPI runtime by reducing the number of communication buffers. We study this combination following the four district styles, depending on if and how multiple OpenMP threads make MPI library calls. In addition, INTERTWinE is focused on applying MPI endpoints to address some of the problems with using the MPI multiple style (see Deliverable D3.1 [5]).

2. **MPI + OpenMP tasks/OmpSs.** In addition to the benefits of MPI + OpenMP threads, using a task-based programming model – such as OpenMP 4.0 or OmpSs – with MPI can potentially improve overlapping of computation and communication phases and address load imbalance caused by performance inhomogeneity in the hardware. Furthermore, there is a close connection between OmpSs and OpenMP as recommendations and improvements to OmpSs are often highly relevant to OpenMP, and can be proposed to the OpenMP ARB. Using OmpSs is convenient, since we have a runtime implementation available in the project to experiment with, but any lessons learned will transfer directly to OpenMP tasks.

3. **GASPI + OpenMP threads.** Threads are the suggested way to handle
parallelism within nodes. The first insight is that the GASPI API, and its implementation GPI2, are thread safe and allow each thread to post requests and wait for notifications independently and without additional synchronization. Moreover, as threading is orthogonal to the GASPI communication model, any approach to threading is supported.

4. **GASPI + OpenMP tasks/OmpSs.** As there is no previous experience of mixing both OpenMP tasks or OmpSs and GASPI on the same application we plan the porting of several applications to gain further insights into this API combination. We believe that this combination offers the opportunity to minimise synchronisation as much as possible without requiring support for task scheduling across nodes.

5. **GASPI + MPI.** GASPI aims at interoperability with MPI in order to allow for incremental porting of applications. There are two aspects to interoperability: memory management and communication management.

6. **GASPI + StarPU.** The API combination GASPI + StarPU will be tackled with the help of the Directory/Cache service, which is being developed in WP4.

7. **Resource Manager** for shared memory. In order to help validating the functionality of the Resource Manager developed in WP4, we consider the following list of API combinations:
   a. PLASMA + OmpSs/StarPU;
   b. OmpSs/StarPU + OpenMP (MKL);
   c. OmpSs/StarPU + CUDA/OpenCL.

For more details, including justifications and motivations, regarding each of the prioritized API combinations, we would refer to Deliverables D3.1 [5] and D3.2 [6].

During the formalization of this prioritized API combination list, we had multiple discussions with the leaders of WP3 and WP4, and with the developers of the Directory/Cache (D/C) and the Runtime Manager (RM). The D/C allows task-based runtime systems to efficiently run distributed applications on top, while being able to consistently manage data stored in distributed memory and in local caches. The RM aims to coordinate the access to CPU and GPU resources between different runtime systems and APIs, to avoid both over- and under-subscription situations. More detailed information on the D/C and the RM can be found in Deliverables D4.2 [3] and D4.3 [4], respectively.

Therefore, the proposed prioritized list of API combinations not only represents all the efforts and interests within the project, but also covers most interesting and widely applied combinations in academia and industry.
4 Co-design cycle: A WP5 Role

The INTERTWinE co-design cycle comprises three work packages, namely WP3, WP4, and WP5, as depicted in Figure 1. Hence, this work package has the responsibility of completing the co-design cycle, by:

- producing an initial set of applications and kernels to drive requirements in WP3 and WP4
- validating the outputs of WP3 and WP4 by the applications and kernels via
  - testing extensions to programming models and new implementations provided by WP3
  - testing new tools developed in WP4
  - ensuring and reviewing the individual application delivery timescales, taking into account the work timescales in WP3 and WP4
- ensuring that the results of the kernel and application interoperability test runs and their optimizations are fed back to WP3 and WP4.

![Figure 1: INTERTWinE co-design cycle.](image)

4.1 Collaboration with WP3

One of the key points in collaboration with WP3 is to provide information and results on the interoperability test runs to enhance the Best Practice Guides (BPGs). We will provide these updates to the BPGs, which target the prioritized API combinations, while preparing the final version of the corresponding software releases, which also target the same API combinations. These software releases will be delivered to the public as packages, including the following information:

- Updated Best Practice Guide
- Tutorial code
- An application kernel-extract and/or benchmark
- Applications/kernels implementations
  - Brief explanation to each implementation, highlighting the parts of application/kernel that were modified and explaining the interoperability strategy employed
- Brief description regarding industrial/academic relevance of each API combination
The other aspect of our collaboration with WP3 is our suggestions to the programming models extensions. Our current suggestions, which are based on software releases described in this report, are listed in Table 1.

<table>
<thead>
<tr>
<th>API combinations</th>
<th>Comments</th>
<th>API extension recommended / issue raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI+OpenMP</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>
| MPI+OmpSs          | Direct connection tested with GraphBLAS       | • Current coupling not performant, anticipate improvement with new implementation with DLL
• Recommendation on Programming Objects (see Section 6.5 in Deliverable D3.2) |
| GASPI+OpenMP       | Not yet tested within INTERTWinE             |                                                                                                         |
| GASPI+OmpSs        | Not yet tested within INTERTWinE             |                                                                                                         |
| GASPI+MPI          | Tested with Ludwig                           | Issue raised on packing/unpacking of derived MPI datatypes, investigation of GPI extension to help message contention is ongoing, consulting on Ludwig code is ongoing |
| GASPI+StarPU       | Not yet tested within INTERTWinE             |                                                                                                         |
| Resource Manager for shared memory | Not yet tested within INTERTWinE |                                                                                                         |

Table 1: API extensions to the prioritised API combinations.

4.2 Collaboration with WP4

In order to verify the functionality of the Directory/Cache, we will use applications/kernels that couple the following programming models combinations: MPI + OmpSs, GASPI + OmpSs, and GASPI + StarPU. To validate the functionality of the Resource Manager, we created a separate item with a sub-list of API combinations, mainly targeting shared-memory architectures with heterogeneous resources such as CPUs plus GPUs.

Below we provide in detail the collaboration with WP4 using the concrete examples of applications with the corresponding API combinations:

4.2.1 Applications/Kernels using the Directory/Cache Service

The Directory/Cache service manages data in distributed memory. It allows the task-based models OmpSs, StarPU and PaRSEC to run on distributed memory hardware with either MPI or GASPI as the underlying communication layer. The applications were chosen to represent all three task-based programming models that will be coupled to the D/C:

- BAR (to test the combinations: MPI + OmpSs and GASPI + OmpS);
- Chameleon (to test the combinations: MPI + StarPU and GASPI + StarPU);
- DPLASMA (to test the combination: MPI + PaRSEC and GASPI + PaRSEC).

It will be very interesting to compare and contrast this approach to the tasks + explicit communication approach. While some applications may perform and scale well with distributed tasks, others may require explicit communication. The D/C, which is
developed in WP4, allows different message-passing interfaces (specially, GASPI and MPI) to be tested, to support a fair comparison of different approaches.

In terms of applications, the numerical library DPLASMA uses the PaRSEC programming model; StarPU is one of the underlying task-based programming model for the mathematical library Chameleon. Mathematical libraries are strong candidates to be linked in applications operating on large execution systems. Chameleon is at the heart of applications in collaboration with Airbus and CEA, which plan to reach Exascale in the coming years. Thus a performant and scalable cluster-wide version of StarPU is an important research topic.

The BAR suite has been chosen to show that a transparent use of the D/C is possible.

### 4.2.2 Applications/Kernels using the Resource Manager Service

We list below candidates for using the INTERTWinE Resource Manager service

- **the BAR benchmarks are considered as the most suitable candidates for evaluating both the D/C and the RM. For instance, we will work with new BLAS libraries: BLASSs on top of the OmpSs programming model and the Chameleon on top of StarPU. In these libraries we will implement/use the offload APIs (native/OpenCL) proposed in the RM component. As a second step we need to swap out the default BLAS library from the BAR benchmarks, replacing MKL with BLASSs or Chameleon, so that we can test how the RM overcomes the issue of two different runtimes competing for node-level resources.**

- **the GraphBLAS applications can benefit from the D/C and the RM. Concretely, the execution of the HGraphBLAS HierMatFact application using OmpSs + multithreaded MKL reported oversubscription problems, which can be solved using BLASSs or Chameleon, and the RM (Pause/Resume API) in collaboration with WP4. The ILU0 and SPMV/SPMM applications analyze the interoperability when we combine distributed-memory and shared-memory programming models, so they can be useful to test the D/C;**

- **after finalizing the PLASMA + OmpSs release, the code will be enhanced to support the RM. This will be done in a close collaboration with WP4, for which it can provide a mechanism for verification. For PLASMA, that will enable running multiple instances concurrently within one CPU as well as running it next to other applications. Enhancing the PaRSEC runtime system to support the Directory/Cache service is under development, also in collaboration with WP4. Once finished, it should be straightforward to use DPLASMA + PaRSEC for evaluation of this service.**

### 4.3 Summary

The set of the suggested programming models combinations as well as the applications for validating both the D/C and the RM was crystalized in cooperation with WP3 and WP4. This was a big step towards enhancing collaboration in the frame of the INTERTWinE co-design cycle. We will strengthen this collaboration further over the remaining period of the project:

- **via active dialogs during the integration of the D/C and the RM into applications/kernels;**
- **during the actual validation and verification stages of the D/C and the RM functionalities;**
- **during the co-design cycles of enhancing applications/kernels as well as both the D/C and the RM;**
- **during monthly co-design meetings.**

More detailed information regarding the **INTERTWinE co-design cycle**, in particular our definition of the co-design cycle and the entire view on the collaboration between these three work packages, can be found in the updated version of Deliverable D3.2 [6].
5 Applications/Kernels

In this section, we present results from the first wave of implementation experiments, as explained in D5.1. We provide details of insight into the implementation/porting, as well as analyzing the results that have been obtained.

5.1 Ludwig: A parallel Lattice-Boltzmann code for complex fluids

5.1.1 Introduction

Ludwig is a versatile code for the simulation of Lattice-Boltzmann (LB) models in 3D on cubic lattices [12]. Some of the problems that could be simulated with Ludwig include detergency, binary fluids in porous media, mesophase formation in amphiphiles, colloidal suspensions, and liquid crystal flows. Broadly, the code is intended for complex fluid problems at low Reynolds numbers, so there is no consideration of turbulence, high Mach number flows, high density ratio flows, and so on.

Ludwig uses an efficient domain decomposition algorithm, which employs the Lattice Boltzmann method to iterate the solution on each subdomain. The domain decomposition is carried out by splitting a three-dimensional lattice into smaller lattices on subdomains and exchanging information with adjacent subdomains. For each iteration, Ludwig uses MPI for communications with adjacent subdomains, using a technique commonly referred to as halo exchange.

In the original implementation of the Ludwig halo exchange, the number of messages sent and received by each MPI process is reduced as much as possible. Each subdomain needs to exchange data with its 26 neighbors in 3 directions (X, Y, Z) to continue with the solution of the problem. This means that synchronization between the different planes is required.

5.1.2 INTERTWinE Ambition

Ludwig is a highly optimized Lattice-Boltzmann application that uses MPI for communication between subdomains. One of the performance problems with this type of application is the amount of communication and synchronization required to obtain the solution of the problem and therefore our main objective is to reduce the impact of the communication sections by overlapping with computation. There are multiple ways of achieving this goal, however in INTERTWinE we believe that interoperability between different APIs will be a key to achieve maximum performance on future Exascale systems, exploiting the advantages of the different programming models. Based on this, we have chosen the following API combinations that we believe will help to achieve our goals:
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

- **GASPI + MPI.** GASPI introduces the PGAS model which will allow us an efficient asynchronous communication.
- **MPI + OpenMP (threads).** The introduction of OpenMP threads capable of making calls to MPI (the MPI Multiple mode) will allow the possibility of having multiple communications happening at the same time instead of a single thread making calls to the MPI library for communication.
- **MPI + OpenMP (tasks).** The use of tasks combined with MPI introduces a new communication model able to avoid bulk synchronization since multiple independent tasks will be able to freely run during the execution of Ludwig.
- **MPI + OmpSs.** OmpSs is also a task-based programming model, however within INTERTWInE we have the ability to integrate OmpSs runtime with the MPI library and, therefore, experiment with new features and prototype ideas – such as integrating features – that would allow threads to efficiently yield MPI blocking calls.
- **MPI endpoints + OpenMP (threads).** As it stands, MPI does not provide any mechanism that allows additional MPI ranks to be generated and assigned to threads. The addition of MPI endpoints enable a greater degree of interoperability between MPI and other programming models since a set of threads could perform MPI calls with each thread using a different endpoint.

The study of these API combinations on Ludwig will help us to understand how scientific MPI based applications, employing the Lattice-Boltzmann method, would be affected by combining multiple APIs and, therefore, how their performance could be increased on future Exascale systems.

5.1.3 GASPI + MPI

5.1.3.1 Implementation details

Ludwig is a practical and efficient domain decomposition algorithm which employs the LB method to iterate the solution on each subdomain. As mentioned above, the original implementation of Ludwig uses MPI for parallel communications where each subdomain is controlled by one MPI process. To coordinate the solution, certain communication between adjacent subdomains is required after each iteration. This is done by creating halos around the dimensions of the subdomain, i.e. extending the dimension of the subdomain by one lattice point in each direction. This can be seen in Figure 3.
After each time step, MPI processes will have to communicate a 2D plane of m (D3Qm) velocities to their adjacent MPI processes. Since each plane shares some sites with the other planes, the exchange of information in each direction should be synchronized before continuing with the execution. One of the advantages of porting the whole application to GASPI is that it could help to reduce the synchronization between subdomains. However, scientific applications like Ludwig normally contain a few sections where the MPI communication is required and porting the entire application to GASPI could become a challenging task. Therefore, in this section we will explain the process of porting Ludwig’s main halo exchange routines from MPI to GASPI.

**Setting up the environment:** When running in mixed-mode, GASPI is able to detect at runtime the MPI environment and use it for its own environment. Here, we want the GASPI environment to have exactly the same ranks and number of processes as MPI. We have achieved this by initializing and finishing the GASPI environment right after the MPI one as shown in Figure 4.

```c
int main(int argc, char ** argv) {
    /* MPI Init */
    MPI_Init(&argc, &argv);
    /* GASPI Init*/
    GASPIERROR(gaspi_proc_init(GASPI_BLOCK));

    // .......

    /* MPI Finalize*/
    MPI_Finalize();

    /* GASPI Finalize*/
    GASPIERROR(gaspi_proc_term(GASPI_BLOCK));

    return 0;
}
```

**Figure 4:** Setting up MPI and GASPI environments in Ludwig.
D5.2 INTERIM REPORT ON APPLICATION KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

and where GASPIERROR is a user-defined macro that checks and prints any error related to a GASPI operation.

Replacing MPI with GASPI The halo exchange routine responsible for exchanging data between neighbor subdomains uses non-blocking MPI and MPI derived datatypes. MPI derived datatypes allow us to specify non-contiguous data in a convenient manner and yet treat it as if it was contiguous.

GASPI requires the creation and later on use of the so-called GASPI segments. In our case we have created a GASPI segment per plane and direction. Therefore, since we have 3 planes and 2 directions per plane, we will require 6 different GASPI segments. The size of the segments is defined as twice the size of buffer to be sent since we will use the same segment to send and receive data from neighbor subdomains.

```c
int YZ_size = lb->ndist*NVEL*ny*nz;

/* Segment sizes is exactly twice the size of the buffer. */
const gaspi_size_t seg_size = 2 * YZ_size * sizeof(double);

/* segment ids */
const gaspi_segment_id_t seg_id_YZ_L = 0;
const gaspi_segment_id_t seg_id_YZ_R = 1;

gaspi_pointer_t gptr_YZ_L, gptr_YZ_R;

/* pointer to the right */
GASPIERROR(gaspi_segment_ptr(seg_id_YZ_L, &gptr_YZ_L));
double* ptr_YZ_L = (double*)gptr_YZ_L;

/* pointer to the left */
GASPIERROR(gaspi_segment_ptr(seg_id_YZ_R, &gptr_YZ_R));
double* ptr_YZ_R = (double*)gptr_YZ_R;
```

Figure 5: GASPI pointers to GASPI segments in the YZ plane.

For purposes of clarity, Figure 5 shows the GASPI pointer creation only in the YZ plane. Each created segment is assigned with an independent id number. In the YZ plane, the data is already contiguous in memory and therefore a simple copy directly from the buffer that contains the data to a GASPI segment is straightforward. However, since Ludwig uses MPI datatypes, more complicated layouts of the data exist for other planes and it is necessary to unpack the MPI datatypes and copy the data contiguously into a GASPI segment. Once the data has been sent and notified we need to recover the data back from the GASPI segment to the original buffer to be able to continue with normal execution of Ludwig. This can be seen in Figure 6. To differentiate the data sent and received in a GASPI segment, we will use an offset variable.
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

```c
/* copy data to GASPI segments */
mempcpy(ptr_YZ_L, pt2, YZ_size*sizeof(double));
mempcpy(ptr_YZ_R, pt3, YZ_size*sizeof(double));

gaspi_notification_id_tnotif_id_L = (gaspi_notification_id_t)0;
gaspi_notification_id_tnotif_id_L = (gaspi_notification_id_t)15;
gaspi_notification_id_tnotif_id_R = (gaspi_notification_id_t)2;
gaspi_notification_id_tnotif_id_R = (gaspi_notification_id_t)20;

/* Send and notify */
GASPIERROR(gaspi_write_notify( seg_id_YZ_L, 0, cart_neighb(BACKWARD,X),
                             seg_id_YZ_L, offset, offset,notif_id_L,
                             notif_val_L, 0, GASPI_BLOCK));

GASPIERROR(gaspi_write_notify( seg_id_YZ_R, 0, cart_neighb(FORWARD,X),
                             seg_id_YZ_R, offset, offset,notif_id_R,
                             notif_val_R, 1, GASPI_BLOCK));

wait_or_die(seg_id_YZ_L, notif_id_L, notif_val_L);
wait_or_die(seg_id_YZ_R, notif_id_R, notif_val_R);

/* pointer to the location of the data within the gaspi segment */
double* recv = ptr_YZ_L+YZ_size;
double* recv2 = ptr_YZ_R+YZ_size;

/* copy back the data from the gaspi segment */
mempcpy(recvbuf_f_R, recv, YZ_size*sizeof(double));
mempcpy(recvbuf_f_L, recv2, YZ_size*sizeof(double));
```

Figure 6: GASPI Write and Notify in the YZ plane.

5.1.3.2 Performance

We have carried out a series of performance tests on ARCHER, see Appendix 8.1 for hardware details. All simulations have been run 5 times on fully populated nodes, i.e. using 24 MPI/GASPI processes per node.

The time taken to transfer a message depends on the network latency and bandwidth. The latency is independent of the size of the message being sent, but dependent of the MPI implementation and network use. Here we have use the Cray MPI implementation. Figure 7 shows the measured bandwidth against the message size. The bandwidth is low at very small message sizes because the time spent to send each message is dominated by the latency. As soon as the message size is increased over 0.2 Mbytes, the bandwidth quickly rises to the maximum allowed by the fabric interconnect.

Strong scaling tests The strong scaling results of the halo exchange time demonstrate little difference between the pure MPI implementation compared to the GASPI + MPI implementation. However, the GASPI-MPI combination seems to incur an extra overhead. This can be seen in Figure 9.
We have also measured the amount of data required to be sent and received from each process at the end of each iteration in $192^3$ lattice size. This is represented in Figure 8.

Figure 7: Bandwidth and message size on ARCHER. Data obtained using the OSU benchmarks [13].

Figure 8: Data transfer size by each process at the end of $192^3$ lattice size simulation.
These results indicate that the performance of the GASPI + MPI version of Ludwig is closer to the MPI version when the package size used in the halo communication is large, i.e. when smaller core counts are used for a given lattice size.

We have also measured the total time that the application spends at each time step of the main loop. Figure 9 shows the time difference between the loop time in GASPI + MPI and MPI implementations. As it can be seen, the performance difference is relatively low between each other.

We also wanted to investigate the intra-node performance difference, this is, the performance of both Ludwig’s implementations on a single ARCHER node. We have done this by keeping fixed the total number of processes (24) used on a fully populated node (24 cores) and varying the lattice size employed in the simulations, see Figure 11 and Figure 12.
It is interesting to highlight that as it happened in the inter-node performance results, the GASPI performance seems to get closer to MPI one when the lattice size is larger. This is, when the message size is actually larger.

Similarly, we have also represented the total loop time comparison between both implementations. As we have seen in Figure 9, the total performance difference is very small although the pure MPI implementation is always slightly more efficient.

In this investigation, we have implemented the halo swap section of Ludwig, which uses pure MPI-1, in GASPI. Performance difference between them seems to indicate that the MPI implementation is still faster than the GASPI one. However, it is also important to highlight that the MPI implementation is probably highly optimized compared to the GASPI one. Nevertheless, the results obtained here seem to indicate that GASPI’s performance is best for large message sizes.

Although we have managed to eliminate some synchronization points while porting Ludwig to GASPI, the code has not yet been optimized to fully exploit GASPI by
leveraging the thread-safe use of multithreading parallelism that allows each thread to post requests and wait for notifications.

5.1.4 Multithreaded MPI + OpenMP threads

5.1.4.1 Implementation details
To be able to eliminate at least part of the synchronization between different planes in a Ludwig simulation, we intend to replace the MPI Send/Recv communication pattern implemented in the halo exchange section with a multithreaded MPI and OpenMP approach. To do so, we first need to initialize MPI in thread multiple mode.

```c
int provided;
MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
if (provided < MPI_THREAD_MULTIPLE) {
    printf("ERROR: The MPI library does not have full thread support\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

**Figure 13:** MPI and OpenMP initialization.

There are 4 options of MPI thread support:

- **MPI_THREAD_SINGLE** -- Only one thread will execute;
- **MPI_THREAD_FUNNELED** -- The process may be multi-threaded, but only the main thread will make MPI calls;
- **MPI_THREAD_SERIALIZED** -- The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time;
- **MPI_THREAD_MULTIPLE** -- Multiple threads may call MPI, with no restrictions.

In this work, we have only focused on the MPI_THREAD_MULTIPLE mode because it allows us to create independent threads, which can also make calls to the MPI routines. In order to select a thread level higher than MPI_THREAD_SINGLE, a programmer should set the environment variable, specifying that MPI_THREAD_MULTIPLE will be used; this may vary from implementation to implementation.

The structure of halo exchange routine in each direction (X, Y and Z) in Ludwig can be simplified to:

```c
for in i,j,k,z
    unpack_data_to_buffer(i,j,k,z)

MPI_receive_from_neighbors()

MPI_send_to_neighbors()

for in i,j,k,z
    pack_data_from_buffer(i,j,k,z)
```

**Figure 14:** Pseudocode of the halo exchange section in Ludwig.

In a traditional MPI + OpenMP hybrid approach, one would create parallel regions only in the packing/unpacking loops, i.e. where the parallelism can be found. However, in MPI_THREAD_MULTIPLE we can also allow those threads to make MPI calls and therefore it would only be required to generate one large parallel region. The large parallel region will contain not only loop parallelism but also MPI communications. This has been
represented in Figure 15 where MPI messages depend on the thread ID to differentiate multiple MPI calls within the parallel region.

```c
#pragma omp parallel private(...) firstprivate(...)
{
    #pragma omp for collapse(4)
    {
        for in i,j,k,z
            unpack_data_to_buffer(i,j,k,z)
    }
    MPI_receive_from_neighbours(thread_id_tag)
    #pragma omp for collapse(4)
    {
        for in i,j,k,z
            pack_data_from_buffer(i,j,k,z)
    }
}
```

**Figure 15:** Code snippet of the halo exchange code using MPI + OpenMP.

For performance reasons, we have added the `collapse()` OpenMP directive on the nested for loops.

### 5.1.4.2 Performance

Performance tests have also been carried out on ARCHER, see Appendix 8.1 for hardware details.

Figure 16 and Figure 17 illustrate the halo exchange performance of running Ludwig in hybrid mode (MPI + OpenMP) in two different thread modes, `MPI_THREAD_SINGLE` (SINGLE) and `MPI_THREAD_MULTIPLE` (MULTIPLE). These configurations have been run on 4 and 8 nodes on ARCHER. Figure 16 shows performance and scaling results running on 4 ARCHER nodes.

![Figure 16](image.png)

**Figure 16:** Performance and scaling results of a 192^3 lattice simulation on 4 ARCHER nodes. The solid line represents the speed up achieved.

These results suggest that although the performance of Ludwig is still better running on pure MPI (only 1 OpenMP thread per PE), the `MPI_THREAD_MULTIPLE` mode is considerably faster (over 2x) than the `MPI_THREAD_SINGLE` mode. Similarly, we also show the performance on 8 ARCHER nodes in Figure 17 with similar conclusions.
D5.2 INTERIM REPORT ON APPLICATION KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 17: Performance and scaling result of a 192^3 lattice simulation on 8 ARCHER nodes. The solid line represents the speed up achieved.

5.1.5 Analysis

5.1.5.1 Correctness
For testing the correctness of the new implementations, we have compared the field distribution data that is written out at the end of a simulation, with our reference data, which has previously been run using the original Ludwig. The `diff` command between the two files should report no difference. Furthermore, the Jenkins server (see Deliverable D5.1 [7] for more details) has been configured to run correctness tests right after every single code commit to the Git repository. This has helped us to track down bugs introduced during the porting process.

5.1.5.2 Portability
We have ensured full standard compliance of the code. This will greatly aid portability to any other HPC system, assuming a Linux system, provided with an MPI and GASPI library installation.

The code has been successfully compiled with GCC, Intel and Cray compilers.

5.1.5.3 Usability
Porting Ludwig to use a hybrid GASPI-MPI API combination has proven to be challenging due to the rather big differences between the two message-passing approaches. The Best Practice Guide [11] has been of great help to understand how both APIs can co-exist in the same application.

The transition to an MPI + OpenMP multithreaded implementation of Ludwig has been slightly smoother. This is probably due to the experience gained during the GASPI + MPI process and the fact that we were already more familiarized with MPI + OpenMP approaches.

5.1.5.4 Suitability for Exascale
Our performance tests have demonstrated that although a pure MPI implementation is still the most efficient approach, OpenMP threads and GASPI messages open a door to create more parallelism which will be essential to exploit Exascale systems. Longer simulations with larger core counts could help us to understand if these API combinations could potentially be beneficial.
5.1.5.5 Discussion and Further Enhancements

The original version of Ludwig, like many other MPI applications, uses MPI datatypes. That soon became a problem for the porting process since GASPI works on segments of data. This means that we had to unpack the data used by MPI datatypes, copy the data required to a GASPI segment, send and then repack the data. We believe this packing-unpacking was the major burden for Ludwig's performance. In order to improve the interoperability with a flat MPI programming model, GASPI will introduce a novel allocation policy for segments where data and GASPI notifications can be shared across multiple processes on a single node. To that end and within WP3, GASPI will use System V or POSIX shared memory for storing notifications such that any incoming one-sided GASPI notification will be visible node-locally across all node-local ranks. The shared notifications should be used with GASPI segments that are using shared memory provided by the applications, such as MPI windows, in interoperability mode.

Instead of node locally packing/unpacking datatypes, the implementation then will publish its respective rank-local datatype layout and will subsequently and merely notify the availability of rank-local data for node-local reading. Data for remote nodes can be aggregated across multiple node-local ranks; however, this may re-introduce a level of synchronization between multiple node-local ranks. As the GASPI notifications will be globally visible on the remote target node all the corresponding remote processes running on that node will be able to see and extract their communication parts. All these features are planned to be releases in the new version of GASPI by mid-July 2017 and, therefore, will be tested in Ludwig.

5.2 iPIC3D: Implicit Particle-in-Cell Code for Space Weather Applications

5.2.1 Introduction

iPIC3D is a Particle-in-Cell (PIC) code for the simulation of space plasmas in space weather applications during the interaction between the solar wind and the Earth’s magnetic field. The iPIC3D code was initially written entirely in C++ with MPI C bindings and consists of approximately 10,000 lines of code, but now makes use of hybrid MPI + OpenMP.

The magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. Hence, the iPIC3D code is based on the implicit moment algorithm, using the numerical discretization of Maxwell’s equations and particle equations of motion that allows simulations with large time steps and grid spacing when compared to common PIC codes, but still retaining the numerical stability. Plasma particles from the solar wind are mimicked by computational particles. At each computational cycle, the velocity and location of each particle is updated, the current and charge density are interpolated to the mesh grid, and Maxwell’s equations are solved. Figure 18 depicts these computational steps. The communication kernel comprises several files for the communication of the particles’ values and quantities defined on the computational mesh.
5.2.2 INTERTWinE Ambition

Starting from the early stage of the project, we have focused on a combination of distributed and shared memory programming models. Hence, at first, we have worked on an extension to MPI + OpenMP (with the single-threaded MPI model using MPI_THREAD_FUNNELED) by considering the MPI_THREAD_MULTIPLE level, where multiple threads may call MPI at once with no restrictions. This combination particularly will help to parallelize the currently sequential calls to MPI functions by OpenMP threads and, therefore, enhance the overall performance of iPIC3D; both OpenMP threads and tasks are used in this scenario. In addition to these combinations, we plan to explore MPI endpoints with OpenMP as MPI endpoints aim to address some problems with the MPI multiple mode: remove threading restrictions in MPI and facilitate high-performance communication between multithreaded OS processes.

The iPIC3D MPI communication is dominated by non-blocking point-to-point communication, occurring from communication of particles and ghost cells among neighbouring processes (halo exchange), and by global reductions resulting from solving two linear systems every simulation time step. In order to reduce the communication burden in iPIC3D, we aim at replacing the MPI communication with the GASPI asynchronous one-sided communication on the communication critical parts of the code such as halo exchange in the field solver. Thus, we verify the interoperability of MPI and GASPI on those parts of the code. We expect improved performance due to the ability of GASPI to overlap computation and communication more effectively than MPI. This step also has another purpose – to be a transition stage for large scientific codes in moving from MPI to PGAS programming models like GASPI.

Then, on top of the MPI + GASPI implementation, with the latter being more favourable in coupling with task-based runtimes, we will employ task-based parallelism provided by OpenMP v4.5, OmpSs, and StarPU runtimes in order to overlap communication of particles with computations of particles trajectories. For the comparison purpose, we also plan to provide independent MPI + OmpSs/StarPU releases. An important stage in this process would be the analysis and comparison of these implementations using different runtimes with respect to the established evaluation criteria such as programmability (e.g. easy to couple), scalability, efficiency, and suitability for Exascale.

5.2.3 Multithreaded MPI + OpenMP threads

In order to parallelize the currently sequential calls to MPI functions by OpenMP threads (with the single-threaded MPI model using MPI_THREAD_FUNNELED), we consider here the MPI_THREAD_MULTIPLE level, where multiple threads may call MPI at once with no restrictions. We particularly stress the interoperability and work splitting among MPI and OpenMP in the widely used halo exchange and particle mover parts of iPIC3D. To stress these two parts, we consider two different simulation regimes:
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

- Field solver dominated regime. Here, a relatively small number of particles (27 per cell) is used. The most computationally expensive part of the iPIC3D code results in the Maxwell field solver. This test, for instance, simulates magnetic reconnection;
- Particle dominated regime. It is characterised by a relatively large number of particles (1,000 per cell). The most computationally expensive part of the iPIC3D code results in the particle mover. This test, for instance, simulates kinetic turbulence

5.2.3.1 Implementation details

The original version of the iPIC3D code is based on the MPI and OpenMP programming models without multithreading support in an MPI library. In order to enable multiple threads in the iPIC3D code to call MPI functions simultaneously, the MPI initialization routine has been changed: in utility/MPIdata.cpp the function MPI_Init() was replaced by MPI_Init_thread() (see Figure 13). In Multiple style, it is more natural for threads to communicate only their “own” data, and the user has fewer concerns about synchronising threads.

Figure 19 illustrates two cases of the MPI and OpenMP programming models interoperability:
- Multiple OpenMP threads cannot call MPI functions (on the left side of the figure);
- Multiple OpenMP threads may call MPI functions (on the right side of the figure).

In iPIC3D, halo exchange consists of three communication phases: between faces, between edges, and between corners. The first phase (communication between faces) includes six sequential calls to MPI_Irecv() and six sequential calls to MPI_Isend(). With the usage of MPI_THREAD_MULTIPLE, it is possible to parallelize each set of these calls. As there are only six independent serial calls to the non-blocking MPI receive functions and six to the non-blocking MPI send functions. And therefore, with taking into account that the number of CPUs in a node of a supercomputer is often a power of two, there are several possible strategies for the parallelization of the region (with two, four and eight threads) – they all were implemented as the “if-then” conditional construct. Table 2 describes these approaches.
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Table 2: Possible parallelization strategies for the phase of communication between faces.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No MPI_THREAD_MULTIPLE</td>
<td>6 MPI_Irecv() and 6 MPI_Isend() are called sequentially.</td>
</tr>
<tr>
<td>1 MPI proc. with 2 OpenMP threads</td>
<td>2 OpenMP threads call MPI_Irecv() in parallel and MPI_Isend() in parallel.</td>
</tr>
<tr>
<td>1 MPI proc. with 4 OpenMP threads</td>
<td>4 OpenMP threads call MPI_Irecv() in parallel and MPI_Isend() in parallel, and thereafter 2 OpenMP threads do the same with MPI_Isend() and MPI_Irecv().</td>
</tr>
<tr>
<td>1 MPI proc. with 8 OpenMP threads</td>
<td>6 OpenMP threads call MPI_Irecv() in parallel and MPI_Isend() in parallel.</td>
</tr>
</tbody>
</table>

For the sake of brevity, Figure 20 illustrates only the phase of communication between faces in Halo exchange in iPIC3D with enabled OpenMP thread support in MPI with four OpenMP threads, so with MPI_THREAD_MULTIPLE four threads call MPI functions in parallel. The cases with two and eight threads have same pattern. Figure 21 shows how it was in original iPIC3D, without MPI_THREAD_MULTIPLE, so threads cannot call an MPI function, and therefore all MPI function calls have to be sequential. As there are six possible parallel calls to the MPI_Irecv() functions and another six to the MPI_Isend() functions, four calls to MPI_Irecv() are executed by the threads in parallel and then the remain two calls to MPI_Irecv() are done in parallel. Same strategy is applied to MPI_Isend(). Here "#pragma omp critical" was used to ensure that one particular MPI function is executed by only one thread and does not hurt anything. We do not need several threads to execute the same one MPI function with the same arguments, but instead, with the "if(id_thread==...)", we assign corresponding parts of the work to right threads.

```c
#pragma omp parallel default(shared) private(id_thread,nthreads)
id_thread = omp_get_thread_num();
nthreads = omp_get_num_threads();
...
if (nthreads==4){
  if(id_thread==0)
    if(left_neighborX != MPI_PROC_NULL && left_neighborX != myrank ){
      #pragma omp critical
      MPI_Irecv(&vectorMPI_Isend(&vector[1][1][0][1][1], 1, yzFacetype, left_neighborX,tag_XR, comm, &reqList[recvcnt]);
      recvcnt++;
      communicationCnt[0] = 1;}
  if(id_thread==1)
    if(right_neighborX != MPI_PROC_NULL && right_neighborX != myrank ){
      #pragma omp critical
      MPI_Irecv(&vector[nx-1][1][0][1][1], 1, yzFacetype, right_neighborX,tag_XL, comm, &reqList[recvcnt]);
      recvcnt++;
      communicationCnt[1] = 1;}

  #pragma omp barrier
  if(id_thread==0){
```

Copyright INTERTWinE Consortium 2017
5.2 INTERIM REPORT ON APPLICATION KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 20: Code snippet of communication between faces in Halo exchange in iPIC3D with MPI_THREAD_MULTIPLE using four OpenMP threads per MPI process.

Figure 21: Code snippet of Halo exchange in iPIC3D (communication between faces) with MPI_THREAD_SINGLE.

The phase of communication between edges also consists of six calls to the MPI_Irecv() functions and six to the MPI_Isend() functions. The phase of communication between corners uses two calls to MPI_Irecv() and two calls to MPI_Isend(). In order to parallelize these two phases, the same approach has been used as for the phase of communication between faces.

5.2.3.2 Performance
In this section we discuss the performance results from the weak scaling tests. Tests were performed on the Beskow supercomputer, see Appendix 8.2 for hardware details. To compare the original version of the iPIC3D code with the new version, based on MPI threading support, we used two standard simulation cases called GEM 3D and
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Magnetosphere 3D. In addition, we used two different data sizes/regimes for both simulation cases: field solver and particle dominated regimes.

Each of four test cases (two version of codes for two simulation cases) has been executed on the increasing number of cores from 32, 64, 128 and up to 256 cores. Here the presented results of the new version of the code are for the cases with two, four and eight OpenMP threads. In order to ensure a fair comparison, the number of iterations in the linear solver was fixed to 20, although in a real simulation the number of iterations depends on the speed of convergence.

Figure 22 shows the results of the weak scaling tests for each of the four cases. Three-dimensional decomposition of MPI processes on X-, Y- and Z-axes was used, resulting in different topologies of MPI processes, each having two, four, and then eight threads in addition. The total number of particles and cells in a simulation are calculated from numbers of cells and numbers of particles per cell in the X, Y and Z directions: $n_{xc}n_{yc}n_{zc}n_{pcelx}n_{pcely}n_{pcelz}$ and $n_{xc}n_{yc}n_{zc}$, respectively. Thus, for example, for the particle dominated Magnetosphere 3D simulation on 32 cores (2x2x2 MPI processes x 4 OpenMP threads), there were used 27x106 particles and 30x30x30 cells, and the simulation size increased proportionally to the number of processes.

![GEM 3D simulation](image)

<table>
<thead>
<tr>
<th>Simulation time ($)</th>
<th>32 cores</th>
<th>64 cores</th>
<th>128 cores</th>
<th>256 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (no OpenMP)</td>
<td>2615.97</td>
<td>2787.15</td>
<td>3208.42</td>
<td>3552.88</td>
</tr>
<tr>
<td>Original (2 threads)</td>
<td>2172.05</td>
<td>2345.01</td>
<td>2500.25</td>
<td>3774.86</td>
</tr>
<tr>
<td>Original (4 threads)</td>
<td>1807.83</td>
<td>2091.65</td>
<td>2246.28</td>
<td>3700</td>
</tr>
<tr>
<td>Original (8 threads)</td>
<td>1553.48</td>
<td>1847.41</td>
<td>2797.08</td>
<td>3652.5</td>
</tr>
<tr>
<td>New hybrid (2 threads)</td>
<td>2342.38</td>
<td>2638.76</td>
<td>2854.3</td>
<td>3291.24</td>
</tr>
<tr>
<td>New hybrid (4 threads)</td>
<td>2053.54</td>
<td>2415.56</td>
<td>2718.89</td>
<td>2958.96</td>
</tr>
<tr>
<td>New hybrid (8 threads)</td>
<td>1706.13</td>
<td>2319.99</td>
<td>2632.78</td>
<td>2987.96</td>
</tr>
</tbody>
</table>

Copyright INTERTWinE Consortium 2017
b) GEM 3D simulation (particle dominated regime)

<table>
<thead>
<tr>
<th></th>
<th>32 cores</th>
<th>64 cores</th>
<th>128 cores</th>
<th>256 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (no OpenMP)</td>
<td>3255.67</td>
<td>3309.29</td>
<td>3358.88</td>
<td>3381.59</td>
</tr>
<tr>
<td>Original (2 threads)</td>
<td>2552.53</td>
<td>2727.45</td>
<td>2761.93</td>
<td>2793.06</td>
</tr>
<tr>
<td>Original (4 threads)</td>
<td>2308.89</td>
<td>2317.21</td>
<td>2416.56</td>
<td>2533.37</td>
</tr>
<tr>
<td>Original (8 threads)</td>
<td>2168.34</td>
<td>2200.58</td>
<td>2265.33</td>
<td>2378.5</td>
</tr>
<tr>
<td>New hybrid (2 threads)</td>
<td>2575.49</td>
<td>2742.53</td>
<td>2759.48</td>
<td>2764.32</td>
</tr>
<tr>
<td>New hybrid (4 threads)</td>
<td>2314.27</td>
<td>2334.61</td>
<td>2447.81</td>
<td>2479</td>
</tr>
<tr>
<td>New hybrid (8 threads)</td>
<td>2182.11</td>
<td>2232.32</td>
<td>2245.64</td>
<td>2461.75</td>
</tr>
</tbody>
</table>

Magnetosphere 3D simulation (field solver dominated regime)

<table>
<thead>
<tr>
<th></th>
<th>32 cores</th>
<th>64 cores</th>
<th>128 cores</th>
<th>256 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (no OpenMP)</td>
<td>3338.57</td>
<td>3508.7</td>
<td>3695.67</td>
<td>4102.58</td>
</tr>
<tr>
<td>Original (2 threads)</td>
<td>2792.88</td>
<td>3335.81</td>
<td>3605.78</td>
<td>3751.63</td>
</tr>
<tr>
<td>Original (4 threads)</td>
<td>2619.14</td>
<td>3260.18</td>
<td>3473</td>
<td>3654.68</td>
</tr>
<tr>
<td>Original (8 threads)</td>
<td>2406.89</td>
<td>2578.97</td>
<td>3160.18</td>
<td>3723.64</td>
</tr>
<tr>
<td>New hybrid (2 threads)</td>
<td>2920.83</td>
<td>2994.34</td>
<td>3233.17</td>
<td>3388.98</td>
</tr>
<tr>
<td>New hybrid (4 threads)</td>
<td>2741.56</td>
<td>2853.59</td>
<td>2904.02</td>
<td>3090.3</td>
</tr>
<tr>
<td>New hybrid (8 threads)</td>
<td>2533.21</td>
<td>2792.3</td>
<td>2891.21</td>
<td>2982.47</td>
</tr>
</tbody>
</table>
5.2.4 Multithreaded MPI + OpenMP tasks

5.2.4.1 Implementation details

The use of task-based programming models is becoming more and more common. Task-based approach refers to designing a program in terms of “tasks” – a logically discrete section of work to be done. Then these tasks are selected by a task scheduler, which dynamically assigns the tasks to threads that are idle and can execute them. The details of task scheduling to hardware are hidden from programmers, so the task-scheduling runtime becomes responsible for efficient execution of a task-based program. Thus, developers just focus on implementing their algorithms in terms of tasks, what decreases programming effort and gives portability across different hardware. Most likely, this separation of concerns is the main reason why task-based programming models are becoming more important. In addition, there are other performance-related benefits from using the task-based approach. In order to port the version of iPIC3D with MPI (MPI_THREAD_MULTIPLE) + OpenMP threads to the task-based parallelism, the existing...
for-loops in the particles/Particles3DComm.cpp and particles/Particles3D.cpp files were re-written with the use of “#pragma omp task”.

In general, there are two options of how to change for-loops to enable tasking:

1) Single task producer:

```c
void foo(float* x, float* y, float a, int n)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++)
                #pragma omp task
                {
                    y[i] = y[i]+a*x[i];
                }
        }
    }
}
```

This is a common pattern as it is employed for tasks with dependences. Here the “#pragma omp single” directive allows only one thread to execute the for-loop, while the rest of the threads in the team wait (the implicit barrier) at the end of the “single” construct. At each iteration, a task is created.

2) Parallel task producer:

```c
void bar(float* x, float* y, float a, int n)
{
    #pragma omp parallel
    {
        #pragma omp for
        {
            for (int i = 0; i < n; i++)
                #pragma omp task
                {
                    y[i] = y[i]+a*x[i];
                }
        }
    }
}
```

This approach was used for iPIC3D as there were no task dependencies in the for-loops in iPIC3D. Here the task creation loop is shared among the threads in the team.

### 5.2.4.2 Performance

In this section we discuss the performance results from the weak scaling tests. Tests were performed on the Beskow supercomputer, see Appendix 8.2 for hardware details. For the performance experiments, we employed the same simulation cases and the identical configuration files with the same decomposition of MPI processes as in Section 5.2.3.2.
GEM 3D simulation (field solver dominated regime)

Simulation time (s)

<table>
<thead>
<tr>
<th>Core</th>
<th>32 cores</th>
<th>64 cores</th>
<th>128 cores</th>
<th>256 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (no OpenMP)</td>
<td>2615.97</td>
<td>2787.15</td>
<td>3208.42</td>
<td>3552.88</td>
</tr>
<tr>
<td>Original (2 threads)</td>
<td>2172.05</td>
<td>2345.01</td>
<td>2500.25</td>
<td>3774.86</td>
</tr>
<tr>
<td>Original (4 threads)</td>
<td>1807.83</td>
<td>2091.65</td>
<td>2246.28</td>
<td>3700.84</td>
</tr>
<tr>
<td>Original (8 threads)</td>
<td>1553.48</td>
<td>1847.41</td>
<td>2797.08</td>
<td>3652.5</td>
</tr>
<tr>
<td>New hybrid (2 threads)</td>
<td>2715.41</td>
<td>3049.25</td>
<td>3521.53</td>
<td>3781.9</td>
</tr>
<tr>
<td>New hybrid (4 threads)</td>
<td>2366.73</td>
<td>2887.28</td>
<td>3195.01</td>
<td>3661.53</td>
</tr>
<tr>
<td>New hybrid (8 threads)</td>
<td>2016.55</td>
<td>2725.52</td>
<td>3074.62</td>
<td>3652.98</td>
</tr>
</tbody>
</table>

GEM 3D simulation (particle dominated regime)

Simulation time (s)

<table>
<thead>
<tr>
<th>Core</th>
<th>32 cores</th>
<th>64 cores</th>
<th>128 cores</th>
<th>256 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (no OpenMP)</td>
<td>3255.67</td>
<td>3309.29</td>
<td>3358.88</td>
<td>3381.59</td>
</tr>
<tr>
<td>Original (2 threads)</td>
<td>2552.53</td>
<td>2727.45</td>
<td>2761.03</td>
<td>2793.06</td>
</tr>
<tr>
<td>Original (4 threads)</td>
<td>2308.89</td>
<td>2317.21</td>
<td>2416.56</td>
<td>2533.37</td>
</tr>
<tr>
<td>Original (8 threads)</td>
<td>2168.34</td>
<td>2200.58</td>
<td>2265.33</td>
<td>2378.5</td>
</tr>
<tr>
<td>New hybrid (2 threads)</td>
<td>3240.58</td>
<td>3385.68</td>
<td>3437.66</td>
<td>3540.7</td>
</tr>
<tr>
<td>New hybrid (4 threads)</td>
<td>2921.74</td>
<td>2962.21</td>
<td>3130.75</td>
<td>3140.93</td>
</tr>
<tr>
<td>New hybrid (8 threads)</td>
<td>2929.26</td>
<td>3087.37</td>
<td>3185.74</td>
<td>3314.67</td>
</tr>
</tbody>
</table>
**Figure 23.** Weak scaling tests for the four different simulations of the original and "MPI_THREAD_MULTIPLE + OpenMP tasks" versions of the iPIC3D code.
Figure 23 shows that adding OpenMP tasking introduces a runtime overhead, which leads to poorer performance of the hybrid version on small numbers of cores. In the particle dominated regimes (b) and d) in Figure 23, on 32 cores, the new hybrid version with two, four and eight threads per one MPI process is 27-35% slower than the original version with the corresponding number of threads. In the field solver dominated regimes (a) and c) in Figure 23, on the large number of cores (256), the new implementation with two, four and eight threads per one MPI process perform mostly better than the original implementation.

5.2.5 Analysis

5.2.5.1 Correctness
For the multithreaded MPI with OpenMP threads/tasks versions of iPIC3D, we verified the correctness of the delivered results, e.g. the total energy, by comparing the computed values with the ones yielded from the original runs of iPIC3D on the same input data sets. We automatized the process of verifying correctness of our modifications by employing the Jenkins CI tests. So, after each modification to the code, we have an immediate opportunity to check that the modified version of iPIC3D delivers the same results as its original version.

5.2.5.2 Portability
The code can be easily ported and run on the conventional CPU clusters, presumably with a Linux OS, equipped with MPI installation, supporting multithreading, and OpenMP with tasks of version 3.x and higher.

5.2.5.3 Usability
Porting iPIC3D to multithreaded MPI with OpenMP tasks/threads required considerable effort. There are two reasons for that: 1) the code is rather complex, so requires considerable amount of time to understand as well as to identify regions where multithreading could be beneficial; 2) the changes were required to be made to the field solver, more precisely the halo exchange part, probably the most complex part of the code. However, for experienced users, as we became now, that would yield much less effort.

5.2.5.4 Suitability for Exascale
As the performance experiments indicate, the version of iPIC3D with multithreaded MPI and OpenMP (see Figure 23) tasks still needs to be improved, while the version with multithreaded MPI and OpenMP threads (see Figure 22) has potential for Exascale. We plan to study another approaches for porting OpenMP thread-based codes to tasking to improve the performance. In order to confirm our suggestions, we plan to conduct large experiments.

5.2.5.5 Discussion and Further Enhancements
iPIC3D code already supports a pure MPI version and a conventional hybrid MPI + OpenMP version with the single-threaded MPI model. Our focus here was on the latter hybrid model, where we employed two API combination scenarios: the first was to enable multiple OpenMP threads to call MPI functions simultaneously; the second was to utilize the OpenMP tasking model on top of the first scenario. Both API interoperability scenarios showed promising results for codes that often occur in scientific simulations such as halo exchange.

In the field solver dominated regime, where there is a little overlap of communication and computation, on the large number of cores (256+ cores) both our new implementations were faster than the original hybrid version.

Our ongoing work with respect to these two implementations is focused on extending the preliminary results. In particular, we are studying, with the help of the Performance Optimization and Productivity (POP) Center of Excellence in Computing Applications,
the interoperability and performance issues of the multithreaded MPI + OpenMP tasks combination.

## 5.3 TAU-kernel linsolv

### 5.3.1 Introduction

TAU-kernel linsolv implements several (iterative) methods to find an approximate solution of the linear system $Ax = b$, where $A$ is a sparse block matrix of dimension $N$ (number of grid points), with square blocks $A_{ij}$ of dimension $V$ (usually $V = 6$).

Available solution methods are:

- **(PNT) Point implicit:**
  \[
  x_i^{(1)} = A_{ii}^{-1}b_i, \quad i = 1, ..., N
  \]

- **(JAC) Jacobi:**
  \[
  x_i^{(k+1)} = A_{ii}^{-1}\left(b_i - \sum_{j=1, j\neq i}^N A_{ij}x_j^{(k)}\right), \quad i = 1, ..., N
  \]

- **(GS) Gauss-Seidel:**
  \[
  x_i^{(k+1)} = A_{ii}^{-1}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} - \sum_{j=i+1}^N A_{ij}x_j^{(k)}\right), \quad i = 1, ..., N
  \]

- **(SGS) Symmetric Gauss-Seidel:**
  \[
  x_i^{(k+1)} = A_{ii}^{-1}\left(b_i - \sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} - \sum_{j=i+1}^N A_{ij}x_j^{(k)}\right), \quad i = 1, ..., N, N, ..., 1
  \]

As a preliminary step for all methods, the LU decomposition of the diagonal blocks of $A$ is calculated (LU), and used for the solution of the small systems $A_{ij}x_i$.

In TAU these methods are used to construct a preconditioner for the Runge-Kutta scheme which does not require to solve the linear system exactly. Usually only a few iterations $k$ are performed.

**MPI parallelization** The parallelization is based on a domain decomposition of the computational grid. Matrix $A$ is decomposed row-wise according to the mapping of grid points to subdomains. Each subdomain is assigned to one MPI process and is first solved as an individual problem. For JAC and GS, the halo part of the approximate solution is then communicated after each completed sweep; for SGS, after each forward and backward sweep; for LU and PNT, only local data is involved, so no MPI communication is needed. Note that in parallel GS and SGS are no longer exact Gauss-Seidel methods and the convergence rate depends on the number of subdomains used.

### 5.3.2 INTERTWinE Ambition

The ambition is to use the TAU-kernel linsolv to investigate different parallelization strategies for DLR’s next generation solver development as a way of validating the interoperability enhancements proposed by the INTERTWinE project. It is planned to use task-based programming models (OpenMP or OmpSs) in combination with MPI or GASPI for inter-node communication. By using a hybrid parallelization approach we hope to achieve better scalability due to minimized communication needs as well as better load balancing capabilities. First, we focus on the MPI + OpenMP implementation.

### 5.3.3 MPI + OpenMP Implementation (threads)

#### 5.3.3.1 Implementation details

For LU, JAC and PNT, all iterations $i = 1 ... N$ are completely independent and are, thus, parallelized using `#pragma omp for` directives. For GS and SGS, iterations over rows
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

\( i \) have been manually split into \( nthread \) consecutive chunks (i.e. local subdomains), each thread then calculates and updates only its own part of \( x \). MPI communication is done by only one thread inside a master or single region.

5.3.3.2 Performance evaluation

The performance tests were conducted on the DLR SCART HPC-cluster. Intel Parallel Studio XE 2016 (compiler, MPI Library and OpenMP runtime) was used for the tests.

Strong scaling tests were performed for the DLR-F6 test case (~ \( 2 \cdot 10^6 \) grid points) using 1 to 12 compute nodes (i.e. 20 to 240 cores) and different combinations of MPI processes and OpenMP threads. For LU, PNT and JAC, also different scheduling strategies (static, dynamic, guided) with varying chunk sizes have been tested.

![Figure 24: LU decomposition; MPI + OpenMP (threads).](image)

Figure 24 - Figure 28 show results for the MPI + OpenMP version for four different combinations (see Table 3) of number of OpenMP threads per MPI process (NT) and number of MPI processes per node (PPN), as well as for the original pure MPI version (mpi) with 20 processes per node. Subfigures on the left (a) show the absolute method runtimes, subfigures on the right (b) the relative performance of the hybrid version compared to the original pure MPI version, which is given as the ratio of the pure MPI runtime to the MPI + OpenMP runtime. For the MPI + OpenMP version, \texttt{schedule(guided)} has been used for LU, PNT and JAC, as this was the overall best scheduling strategy found during the tests.

<table>
<thead>
<tr>
<th>NT</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 3: Combinations of number of OpenMP threads per MPI process (NT) and number of MPI processes per node (PPN).

Regarding the different settings used for the MPI + OpenMP version, the best performance was achieved with 4 processes per node and 5 threads per process (NT5 PPN4). There is no real difference in runtime for LU (Figure 24) compared to the original.
pure MPI version, for PNT (Figure 25) this setting is about 2-4% faster than MPI. For JAC (Figure 26) it is about 4-6% slower than MPI on 1 to 5 nodes and about 2-9% faster on 7 to 12 nodes. For GS (Figure 27) and sGS (Figure 28) this setting generally runs about 3-16% slower than MPI on small numbers of nodes (up to 7). On 8 or more nodes it is about 3-6% faster than MPI for GS and about 1-3% for sGS.

The setting with 2 processes and 10 threads (NT10 PPN2) shows almost the same performance as NT5 PPN4 for LU and PNT. Not quite as good but still very similar results were produced for GS and sGS. For JAC the increase in runtime is more obvious.

The setting with 5 processes and 4 threads (NT4 PPN5) shows an increase in runtime compared to NT5 PPN4 and NT10 PPN2 for all methods. It is also always slower than the pure MPI version. With this setting, one of the processes and its corresponding threads uses resources from both sockets, which causes some NUMA overhead.

The worst performance was observed with 1 process per node and 20 threads per process (NT20 PPN1). This setting suffers a lot from NUMA overhead and runs significantly slower than all other tested variants.

Figure 25: Point implicit; MPI + OpenMP (threads).

Figure 26: Jacobi; MPI + OpenMP (threads).
Additional performance tests on up to 20 nodes have been conducted for the setting with 5 threads and 4 processes (NT5 PPN4) and schedule(guided). Figure 29 shows the relative performance of the hybrid version compared to the pure MPI version for all methods, which is given as the ratio of the pure MPI runtime to the MPI + OpenMP runtime. Figure 30 shows the parallel speedup for all methods (a) for the pure MPI version and (b) the hybrid MPI + OpenMP version.

It can be seen that for the methods involving no communication (LU and PNT) the hybrid version offers no real advantage compared to the original pure MPI version. For the methods involving communication (JAC, GS and SGS) the hybrid method generally performs better than the original version on larger number of nodes.
5.3.4  MPI + OpenMP Implementation (explicit tasks)

5.3.4.1  Implementation details
The #pragma omp for parallelization of methods LU, PNT and JAC has been modified to use explicit tasks. Three different task based versions have been implemented and tested:

<table>
<thead>
<tr>
<th>V1: parallel task producer</th>
<th>V2: single task producer</th>
<th>V3: taskloop construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp for</td>
<td>#pragma omp single</td>
<td>#pragma omp taskloop</td>
</tr>
<tr>
<td>for(i = 0; i &lt; n; i++)</td>
<td>for(i = 0; i &lt; n; i++)</td>
<td>for(i = 0; i &lt; n; i++)</td>
</tr>
<tr>
<td>#pragma omp task</td>
<td>#pragma omp task</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

5.3.4.2  Performance evaluation
The performance tests were conducted on the DLR SCART HPC-cluster, see Appendix 8.3 for hardware details. Intel Parallel Studio XE 2016 (compiler, MPI Library and OpenMP runtime) was used for the tests.
Strong scaling tests were performed for the DLR-F6 test case (~2 \cdot 10^6 grid points) using 1 to 20 compute nodes (i.e. 20 to 400 cores) and different combinations of MPI processes and OpenMP threads. For LU, PNT and JAC, also different scheduling strategies (static, dynamic, guided) with varying chunk sizes have been tested.

For the *single task producer* version very poor parallel performance was observed. In the best case this version showed an increase in runtime of more than 200% compared to the original pure MPI version. For this version each tasks consists only of a single iteration. Considering that the threaded version with a chunk size of 1, i.e. schedule(static/dynamic, 1), also performed very poorly, this results isn’t entirely unexpected. Further investigations concerning the performance of the single task producer pattern using more iterations per tasks are necessary.

Figure 31 - Figure 33 show results for the *parallel task producer* (a) and *taskloop* (b) versions for four different combinations (see Table 3) of number of OpenMP threads per MPI process (NT) and number of MPI processes per node (PPN), as well as for the original pure MPI version (mpi) with 20 processes per node. For the parallel task producer version, schedule(guided) has been used, as this was the overall best scheduling strategy found during the tests.

Regarding the different combinations of threads and processes, the setting with 4 processes per node and 5 threads per process (NT5 PPN4) achieved the best performance for all methods, both for the *parallel task producer* and the *taskloop* version.

The setting with 2 processes and 10 threads (NT10 PPN2) shows almost the same performance for methods LU (Figure 31) and PNT (Figure 32) and a small increase in runtime for method JAC (Figure 33).

With 5 processes and 4 threads (NT4 PPN5) a clear increase in runtime compared to NT5 PPN4 and NT10 PPN2 for all methods becomes visible. By far the worst performance was achieved with 1 process per node and 20 threads per process (NT20 PPN1). Both NT4 PPN5 and NT20 PPN1 always run slower than the original pure MPI version.
For all tested combinations of threads and processes the taskloop version always achieved better performance than the parallel task producer version. For setting NT5 PPN4 the parallel task producer version shows an increase in runtime of 15-21% for LU and PNT and 8-12% for JAC compared to the taskloop version.

Compared to the original pure MPI implementation, the parallel task producer version always runs slower for methods LU and PNT (about 6-20%). For method JAC it is about 5-12% slower than MPI on 1 to 9 nodes and about 3-11% faster on 13 or more nodes.

Figure 34(a) shows the relative performance of the taskloop version (NT5 PPN4) compared to the pure MPI version (PPN20), given as the MPI to MPI + OpenMP (taskloop) runtime ratio, for all methods. In general there is no clear difference in performance for LU and PNT, although for the taskloop version performance tends to slightly decrease with the number of used nodes. For JAC the taskloop version runs about 3-6% slower on small numbers of nodes (up to 5) and about 2-20% faster on 7 or more nodes.
Figure 34: Relative performance; MPI + OpenMP (tasks).

Figure 34(b) shows the relative performance of the taskloop version (NT5 PPN4) compared to the MPI + OpenMP (threads) version (NT5 PPN4, schedule(guided)), given as the MPI + OpenMP (threads) to MPI + OpenMP (taskloop) runtime ratio, for all methods. For LU both versions achieve very similar results. For PNT the taskloop version runs about 2-20% slower. Only for method Jac the task based version manages to outperform the threaded version, but the maximum decrease in runtime is only about 2%.

Figure 35 shows the parallel speedup for all methods (a) for the pure MPI version and (b) the hybrid MPI + OpenMP taskloop version. Only for method Jac the task based implementation offers an advantage compared to the original pure MPI implementation, especially on larger number of nodes.

5.3.5 Analysis

5.3.5.1 Correctness

For methods PNT and Jac correctness of the hybrid MPI + OpenMP implementations has been verified by comparing the computed approximate solution $x$ with the one yielded by the original pure MPI implementation for the same input data, i.e. using the
same test case, partitioning method and number of subdomains/processes. The correctness has been verified for each single run of the kernel.

For the OpenMP parallelization of methods GS and sGS algorithmic changes were necessary, hence the convergence rate depends not only on the number of MPI processes used but also upon the number of threads used. As a consequence, the calculated approximate solutions differ from the ones produced by the original pure MPI implementation, when more than one thread is used. It has been verified that each run of the hybrid version using the same input data and the same number of threads produces the same solution. Additionally some convergence tests for GS and SGS with a very small test case have been conducted to ensure that both the original MPI version and the hybrid version converge to the same solution.

5.3.5.2 Portability
Given that all necessary software, libraries and runtimes (C compiler, thread safe MPI, OpenMP, NetCDF) are available, the code can be easily ported to conventional multicore platforms. It is expected that for running the code efficiently, individual investigations for each particular system are necessary, e.g. to find the right combination of processes per node and threads per processes.

5.3.5.3 Usability
Having had only little experience in using OpenMP (none with the concept of explicit tasks), quite some time has been spent studying the OpenMP standard and related literature. The actual modification of the original code then required reasonable effort. Learning how to run hybrid MPI + OpenMP jobs on a HPC cluster, how to use process and thread pinning, finding the best thread scheduling strategy and figuring out which combinations of threads per process and process per node are most efficient, again took some considerable amount of time.

5.3.5.4 Suitability for Exascale
The conducted performance tests indicate that the hybrid MPI + OpenMP parallelization is better suited for Exascale than the pure MPI parallelization. For larger core counts, the hybrid implementation scales clearly better than the original implementation for all methods involving communication and not worse for the others (see Figure 30 and Figure 35).

5.3.5.5 Discussion of further enhancements
The original pure MPI implementation as well the MPI + OpenMP versions of TAU-kernel linsolv currently use a bulk synchronous communication pattern for the halo exchange. In order to allow for an overlap of computation and communication, we are going to investigate other, more asynchronous communication strategies using OpenMP dependent tasks.

5.4 BAR: Barcelona Application Repository

5.4.1 Introduction
This set of benchmarks is composed by four different applications that we consider representative in the field of High Performance Computing. The Matrix multiplication benchmark implements the classic multiplication which divides A, B, C in blocks elements and computes for each of them the local multiplication, propagating these partial results to neighbors’ blocks. The Cholesky factorization computes $A = LL'$, with $A$ being an NxN symmetric positive definite (SPD) matrix and $L$ is a lower-triangular matrix. The Heat diffusion benchmark is an implementation of an iterative solver for heat distribution. There are three user-selectable algorithms: Jacobi, Gauss-Seidel and Red-Black. And finally, the N-Body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body.
Each different benchmark comes with its own list of command line options that allow changing the workload for a given execution. The nature of this set of kernels is to be used for testing purposes (especially for the initial verification and benchmarking of the Resource Manager and Directory/Cache implementations) and, therefore, this feature is essential. Programmers may launch execution with a small workload to check correctness or medium/large workloads in order to test performance.

5.4.2 INTERTWinE Ambition

The API combinations we have chosen from the BAR benchmarks cover four different cases:

- Benchmarks using a pure task-based programming model: OmpSs or StarPU.
- Benchmarks using two different task-based programming models: OmpSs + MKL and StarPU + MKL.
- Benchmarks using a communication library and a task-based programming model: MPI + OmpSs, GASPI + OmpSs, and GASPI + StarPU.
- Benchmarks extending initial implementation with the use of accelerators: OmpSs + CUDA, OmpSs + OpenCL, StarPU + CUDA, and StarPU + OmpSs.

Benchmarks using a pure task-based programming model can be used as the baseline. In addition, they will be able to run over a cluster/distributed version of the benchmarks. Benchmarks using two different task-based runtime systems will be candidates to verify the Resource Manager. In this situation, it is expected that the access to underlying resources will be not orchestrated and thus will generate problems of oversubscription (with the correspondent performance penalty). In the following subsections, we present some results using the matrix multiplication and the Cholesky benchmark implemented with OmpSs + MKL. In these cases we have the host application parallelized with OmpSs and the mathematical library parallelized with OpenMP or TBB. When activating the parallel version of MKL we expect to measure the effect of oversubscription.

The combination of a task-based programming model with a communication library will be interesting in order to compare with the performance of the pure task-based programming model application when running over the cluster/distributed version of the runtime. For example, we can compare the performance of a pure OmpSs application, running on top of the OmpSs @ cluster implementation with the performance obtained with the MPI + OmpSs.

The cluster/distributed version of the task-based programming models will run on top of the Directory/Cache implementation. For that, we will need the support of a Global Address Space mechanism in order to execute in these runtime systems.

The future extension of the set of benchmarks with accelerator versions (CUDA and OpenCL) are scheduled due the need of task 4.3: Runtime Systems for accelerators. We expect to apply the same ideas of the Resource Manager when using GPUs.

5.4.3 Correctness

The application repository contains different configurations that can be used to test the algorithm correctness. In some cases the verification will be done through the comparison with a sequential execution. In other cases the repository contains an output file that can be used to test the final result.

The set of benchmarks include as a Makefile’s mechanism a target (check), which executes and reports test results for each of them. This mechanism is used in the Jenkins job to periodically check the suite correctness. A possible output for this check (if everything went well) is:

```
$make check -s
Cholesky (Original+MKL) - sequential: successful
Cholesky (OmpSs+MKL) with 1 thread/s: successful
```
### 5.4.4 Evaluation environment

The experiments presented in this section were run on the MinoTauro system, see Appendix 8.4 for hardware details. The operating system installed in MinoTauro is the Red Hat Enterprise Server. Each version of the benchmarks is compiled, according with its requirements, using:

- **OmpSs** uses the Mercurium compiler 2.0.0 ((distributed version) 59b8c8ba 2017-03-09 20:48:44 +0100), using as a native compiler gcc 4.9.1, and as a support library Nanos++ RTL 0.12a (git master f79a104 2017-02-27 11:52:32 +0100 developer version).
- **StarPU** uses the StarPU runtime library version 1.2 with the Hardware locality package version 1.7.
- **MPI** uses the MPI library 5.1.3.181 and the Intel compiler icc 16.0.2 20160204, in all cases sources have been compiled using the option –show, which provides compiler and linker flags.
- **MKL** uses the version 11.3.2

The purpose of the evaluation is to briefly present some results and introduce the benchmark features. These benchmarks are not designed to be used or tested with a high number of nodes/CPUs but as validation and debugging tools.

### 5.4.5 Performance results

In the analysis of granularities shown in Figure 36 we compute the multiplication of two matrices of 2048x2048 double precision floating-point numbers (doubles). All the executions have been done with 12 threads (maximum number of CPUs in the node) using OmpSs + MKL sequential. In the experiment we change the decomposition of the matrices from blocks of 16x16 doubles up to blocks of 1024x1024 of doubles. The results showed a good grain size trade-off between 64x64 and 512x512 doubles (having the optimal value in 128x128). Below these block sizes range the application performance is degraded by the overhead. Above them performance is degraded by the lack of parallelism, because the number of blocks in these cases is too low.
In Figure 37 (left) we also present a strong scaling analysis of the OmpSs parallelized version using a sequential version of MKL. In the plot we can verify that benchmark scales almost linearly. For this test we have used blocks of 128x128 doubles (optimal value obtained in the previous test). The total matrix size is 4096x4096 doubles. In this version of matrix multiplication we have to take into account that we can lose some performance due to the impact of data locality, since tasks computing the same block in the matrix may execute in different CPUs.

In the same figure (right) we can see the same OmpSs application but this time linked with the parallel version of MKL (in this specific case using the.gnu threaded version). Both versions use the sequential execution as the baseline. Results show how as we increase the number of threads we start losing performance compared with the MKL sequential as we increase the number of threads. This is because threads from OmpSs and threads from MKL are competing one with each other for the CPU usage. Trying to play with the different options that each programming model provides in order to control the number of threads (NX_SMP_WORKERS and MKL_NUM_THREADS) does not help due to the fact that MKL binds its threads to the CPU mask provided by the calling thread (see D4.1 Resource Manager interoperability requirements for further information).
The Cholesky factorization presents similar results to those seen in the previous plots. Figure 38(left) presents strong scaling results of a blocked Cholesky using MKL sequential math services to compute inner blocks (potrf, trsm, syrk and gemm). Figure 38(right) also presents strong scaling results for the Cholesky kernel but this time using a parallel version of the MKL library (verifying the same effects than in the previous test). As in the previous example, both versions use the sequential execution time as the baseline.

![Cholesky Factorization](image)

**Figure 38**: Cholesky Factorization strong scaling with MKL sequential (left) and MKL parallel (right).

Ideally these types of problems, combining different task-based runtime systems, can be used joint with the Resource Manager component in order to coordinate the use of the CPUs and avoid the oversubscription effect shown in previous figures.

![N-Body](image)

**Figure 39**: N-Body strong scaling pure OmpSs (left) and MPI + OmpSs (right).

Figure 39 shows the results executing N-Body using the Jacobi kernel. The figure in the left shows the pure OmpSs (no message passing library). Scalability is almost linear reaching a speed-up of 10 with 12 threads. Figure in the right shows a hybrid version of the N-Body application dividing the initial configuration in several slices (once per node). Each node computes its local portion and sends the result to its correspondent neighbor in the MPI rank.
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Ideally, a pure OmpSs version of the application could be adapted in order to send/receive portions of memory without explicit message routine calls but relying instead on a virtual global address space which allows determining where the data is located and automatically move it. This approach simplifies the way programmers write distributed memory applications by hiding the burden of dealing with memory movements back and forward. This is the main purpose of the Directory/Cache component.

At short- and mid-term we will also release versions of these kernels using CUDA and/or OpenCL. The idea of these versions will be to validate and extend the concepts of the Resource Manager using accelerator devices (e.g. GPUs). Versions using OmpSs will be also implemented using StarPU and versions using MPI will be also implemented using GASPI. This variety of programming model will allow us to validate the Resource Manager and the Directory/Cache’s ideas over multiple scenarios. Finally, we will also provide, at long-term, an implementation of the Heat diffusion benchmark using the endpoints approach.

5.4.6 Portability

The suite can be easily ported to other platforms that fulfill the requirements of having installed: OmpSs, MKL, StarPU and MPI. Different provided kernels are intended on different API combinations.

We have included within the repository a configure.sh script that initializes all environment variables used during the build process. Users may configure the path for OmpSs runtime, MKL library, Hardware locality and StarPU runtime.

The Makefile build system also relies on the use of the mpicc compiler and its associated flags --showme:compile and --showme:link. In such environments in which these flags are not available (e.g. Intel Compiler), user should provide the correspondent options in order to compile and link MPI applications.

5.4.7 Discussion of Further Collaboration with WP4

The purpose of this set of micro benchmarks is to use them as initial tests for the Resource Manager (those using two task-based runtime systems or the future versions based on accelerators) and the Directory/Cache (those using a pure task-based runtime system and the correspondant hybrid version using a task-based programming model with a communication library) both components developed in the context of WP4.

5.5 Graph-BLAS

5.5.1 ILU0: MPI + OmpSs/OpenMP

5.5.1.1 Introduction

The solution of linear systems of equations is a ubiquitous linear algebra problem arising in many scientific and engineering computing applications. In most of these cases, the coefficient matrix of the system is large and sparse (i.e., a majority of the entries are zeros, and the nonzero pattern presents an irregular distribution). In this scenario, iterative methods, enhanced with some sort of preconditioner to accelerate the convergence of the solver, are often especially effective. For symmetric positive definite (s.p.d.) systems, the preconditioned Conjugate Gradient (PCG) method, accelerated with a simple preconditioner based on a incomplete LU (ILU) factorization, is an iterative solver with a wide spectrum of applicability. ILU0 is a specialized yet simpler ILU-type preconditioner that enforces the same sparsity pattern in the triangular factors as that present in the coefficient matrix of the system. Figure 39 illustrates a simplified version of the PCG solver underlying ILUPACK. The most challenging operations in this algorithm are the computation of the preconditioner (S1), before the iteration commences, and its application at each iteration (S9). The remaining computations are
basic linear algebra operations. For details on sparse linear systems, iterative solvers, and preconditioning, see [8].

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Compute the preconditioner ( A \rightarrow M \approx LU )</td>
<td></td>
</tr>
<tr>
<td>2. Initialize ( x_0, r_0, z_0, d_0, \beta_0, r_0 )</td>
<td></td>
</tr>
<tr>
<td>3. ( k := 0 )</td>
<td></td>
</tr>
<tr>
<td>4. ( \textbf{while} \ (\tau_k &gt; \tau_{\text{max}}) )</td>
<td></td>
</tr>
<tr>
<td>5. ( w_k := \text{Ad}_k )</td>
<td></td>
</tr>
<tr>
<td>6. ( \rho_k := \beta_k d_k^T w_k )</td>
<td></td>
</tr>
<tr>
<td>7. ( x_{k+1} := x_k + \rho_k d_k )</td>
<td></td>
</tr>
<tr>
<td>8. ( r_{k+1} := r_k - \rho_k w_k )</td>
<td></td>
</tr>
<tr>
<td>9. ( z_{k+1} := M^{-1} r_{k+1} \approx U^{-1} L^{-1} r_{k+1} )</td>
<td></td>
</tr>
<tr>
<td>10. ( \beta_{k+1} := \tau_{k+1}^T z_{k+1} )</td>
<td></td>
</tr>
<tr>
<td>11. ( \alpha_k := \beta_{k+1} / \beta_k )</td>
<td></td>
</tr>
<tr>
<td>12. ( d_{k+1} := z_{k+1} + \alpha_k d_k )</td>
<td></td>
</tr>
<tr>
<td>13. ( \tau_{k+1} := | r_{k+1} |_2 )</td>
<td></td>
</tr>
<tr>
<td>14. ( k := k + 1 )</td>
<td></td>
</tr>
</tbody>
</table>

**Iterative PCG solve**

- (SPMV) 
- (DOT product) 
- (AXPY) 
- (AXPY) 
- Apply preconditioner 
- (DOT product) 
- (AXPY-like) 
- (2-norm)

**Figure 40:** Algorithmic formulation of the PCG method. Here, \( \tau_{\text{max}} \) is an upper bound on the relative residual for the computed approximation to the solution.

Exploiting the relationship between sparse matrices and adjacency graphs, nested dissection can be recursively applied to permute a sparse matrix, yielding a collection of diagonal blocks that are linked to certain subgraphs and separators. Moreover, the hierarchy of subgraphs and separators fixes the order in which the diagonal blocks have to be factorized. This process renders a Task Dependency Graph (TDG) with the structure of a tree, where the subgraphs occupy the leaves and the separators correspond to the internal nodes.

In order to improve the concurrency of the computation of the preconditioner, the permuted matrix can be disassembled into one submatrix per leaf of the TDG. Thus, the factorizations of the leading blocks of these four submatrices can proceed in parallel, while the non-leaf blocks are needed to solve the dependencies of the ancestor tasks. This process continues traversing the dependency tree, until the root task factorizes its local submatrix.

The application of the preconditioner requires the solution of two triangular systems, corresponding to the lower and the upper incomplete triangular factors. The TDG for the former triangular system presents the same structure and dependencies as that associated with the computation of the preconditioner. In the latter triangular solve, the structure is preserved but dependencies are reversed, pointing top-down from the root to the leaves. Therefore, concurrency increases/decreases as we move towards/away from the leaves.

In the other kernels of the PCG, the matrix is also disassembled and the vectors are partitioned in a conformal manner. With this formulation all these computations only involve the leaves of the TDG and, therefore, can be computed fully in parallel, except for the dot products, which require an atomic addition (reduction) of the values locally computed in each leaf.

**5.5.1.2 INTERTWinE Ambition**

The ambition is to use ILU0 to investigate different parallelization approaches to detect problems which can be solved in the INTERTWinE project, and also to test the interoperability improvements suggested by the project.

The combination of MPI + OmpSs (or OpenMP tasks) permits parallel execution on a cluster of multicore processors. It also allows us to exploit the characteristics of
OmpSs/OpenMP to employ task-parallelism within the nodes and benefit from the true asynchronous execution in the operations. Therefore, we hope to achieve better scalability by using a hybrid parallelization approach, because it will minimize the communication needs and will improve the load balancing capabilities.

5.5.1.3 Implementation Details
The MPI + OmpSs/OpenMP version requires an initialization where the root process distributes the data corresponding to (the leaves of) the subtrees among the MPI ranks. The MPI + OmpSs/OpenMP version of ILU0 is then divided into a sequence of interleaved OmpSs (or OpenMP) and MPI stages, with the former ones computing the tasks internal to the subtrees local to the MPI ranks, and the latter requiring communication between MPI ranks. In particular, the computation of the preconditioner comprises only one stage of each type, but its application in the loop body of PCG has two OmpSs (or OpenMP) stages per iteration because the TDG is traversed twice. Figure 41 illustrates the initial distribution for a TDG with 8 leaves, together with a scheme of the execution of the two stages in the preconditioner computation. In that example, the OmpSs (or OpenMP) threads process the tasks within the bottom two levels, with no MPI communication involved. For the top two levels, the OmpSs (or OpenMP) threads remain inactive and it is the MPI ranks that are in charge of processing the tasks. The dot operations also exhibit the same two stages: On the leaves, the OmpSs (or OpenMP) threads accumulate their local subvectors, and an atomic reduction is then applied to compute the reduction inside each MPI rank. These local values are then reduced using an MPI collective primitive. The remaining vector computations of the PCG iteration operate in the bottom level only and, therefore, are computed by OmpSs (or OpenMP) threads with no MPI communication involved.

![Figure 41](image)

Figure 41. Mapping of a TDG to 4 MPI ranks (R0-R3) with 2 OmpSs threads per rank.

5.5.1.4 Correctness
This application computes the solution of a sparse linear system $A x = b$, where $A$ is an $n \times n$ sparse s.p.d. coefficient matrix, $b$ is a vector containing the $n$ independent terms, and vector $x$, of the same dimension as $b$, is the sought-after solution. In order to tackle this problem, the application implements the PCG method with an ILU0 preconditioner. The result is correct if the method converges in a certain number of iterations. The
convergence implies that the relative residual $||A x_k - b||$, where $x_k$ denotes the approximate solution after $k$ iterations, satisfies the user-defined bound.

The input data provided to the application corresponds to the finite difference discretization of a 3D Laplace problem; see [1] and Table 4 for details.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Dimension ($n$)</th>
<th>#non-zeros ($nz$)</th>
<th>Density (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A100</td>
<td>1,000,000</td>
<td>3,970,000</td>
<td>3.97E-4</td>
</tr>
<tr>
<td>A159</td>
<td>4,019,679</td>
<td>16,002,873</td>
<td>9.90E-5</td>
</tr>
<tr>
<td>A200</td>
<td>8,000,000</td>
<td>31,880,000</td>
<td>4.98E-5</td>
</tr>
<tr>
<td>A252</td>
<td>16,003,008</td>
<td>63,821,520</td>
<td>2.49E-5</td>
</tr>
<tr>
<td>A318</td>
<td>32,157,432</td>
<td>128,326,356</td>
<td>1.24E-5</td>
</tr>
<tr>
<td>A400</td>
<td>64,000,000</td>
<td>255,520,000</td>
<td>6.23E-6</td>
</tr>
</tbody>
</table>

Table 4: Matrices employed in the experimental evaluation, where $nz$ accounts only for the non-zeros in the upper triangular part of the matrix.

The right-hand side vector is initialized with all entries equal to 1, and the iteration commences with the starting guess $x_0 = 0$. This selection of the parameters ensures the convergence of the method. The number of iterations required for convergence depends on the problem instance and the number of leaves induced by the partitioning that exposes task-parallelism (see next section). Table 5 reports the iteration count for different combinations of these two parameters. Due to the nondeterminism of the task-parallel reduction operations, there may be slight variations on these numbers between different executions of the same code.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Leaves</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>A100</td>
<td></td>
<td>108</td>
<td>124</td>
<td>126</td>
<td>126</td>
<td>128</td>
<td>132</td>
<td>132</td>
<td>132</td>
</tr>
<tr>
<td>A318</td>
<td></td>
<td>298</td>
<td>328</td>
<td>336</td>
<td>338</td>
<td>343</td>
<td>353</td>
<td>351</td>
<td>351</td>
</tr>
</tbody>
</table>

Table 5: Number of iterations required for convergence for each problem instance and number of leaves.

5.5.1.5 Performance

The experiments were performed using the IEEE754 double-precision arithmetic on MareNostrum, see Appendix 8.5 for hardware details.

In the following we analyse the performance of three parallel versions of the PCG+ILU0 solver: one based on MPI that can handle several leaves per MPI rank (hereafter, referred to as MPI-only); and two alternative variants that combine either MPI + OmpSs or MPI + OpenMP, also capable of processing several leaves per MPI rank, but which do so via OmpSs/OpenMP threads internally to each node.

The MPI + OmpSs code was compiled using Mercurium C/C++ (2.0), with the OpenMPI (1.8.1) flags -showme:compile and -showme:link. The MPI + OpenMP version employed gcc (4.9). The MPI-only variant was compiled with the same version of OpenMPI. Other software included OmpSs (16.06), and ParMetis (4.0.2) for the graph reorderings. In the executions with the MPI-only version, we spawned one MPI rank per core (i.e., 16 per node). For MPI + OmpSs and MPI + OpenMP, we tested distinct combinations of MPI
ranks and OmpSs/OpenMP threads, with the numbers of ranks multiplied by the number of threads always being equal to 16 per node.

Several preliminary experiments revealed that, in general, the best performance was obtained when splitting the sparse matrix via nested dissection to generate a task dependency graph (TDG), with the structure of a binary tree, and a number of leaves that equals or doubles the number of cores. Therefore, for simplicity, in the following we analyze only these two cases. In order to assess the performance of the parallel MPI + OmpSs and MPI + OpenMP versions of PCG+ILU0, we also evaluated different combinations of MPI ranks and OmpSs/OpenMP threads per node (configurations). These experiments revealed that, for almost all cases, the best option employs two MPI ranks per node with 8 OmpSs/OpenMP threads per rank. In the following experiments, we will adopt this distribution, which mimics the internal socket/core architecture of the servers.

5.5.1.5.1 Strong Scalability

We first evaluate the strong scalability of the parallel solvers. Figure 42 and Figure 43 show the execution time per iteration, using one and two leaves per core, of the three versions of the PCG solve for the A400 problem (MPI-only, MPI + OmpSs and MPI + OpenMP) as the resources are increased from 16 cores/1 node to 256 cores/16 nodes. In general, as expected, there is a decrease in the iteration time as the number of cores grows.

If we compare the implementations, the results demonstrate that the MPI + OmpSs/OpenMP variants consistently outperform the MPI version (with no underlying OmpSs/OpenMP runtime system), by a margin that is around 5-10%. Moreover, there is a slight difference between the cases with one or two leaves per core that is enlarged with the number of cores, revealing the TDG with one leaf per core as the best choice for 32 or more cores. The reason is that, as the amount of computational resources grows, the additional concurrency explicitly exposed by further splitting the computational load (sparse matrix/adjacency graph) does not compensate the overhead that is introduced for this particular (moderate) problem dimension.

![Figure 42: Execution time per PCG iteration for the Laplace A400 problem for different configurations, using 1 leaf per core, in MPI + OmpSs (left) and MPI + OpenMP (right).](image-url)
5.5.1.5.2 Weak Scalability

The next experiment aims to provide an evaluation of weak scaling for the parallel solvers. Unfortunately, for the PCG+ILU0 solver it is not possible to generate an instance of the Laplace problem with a computational complexity that grows exactly in proportion to the number of resources. To approximate this scenario, we set the number of non-zeros of the sparse matrix \((nz)\) to be roughly proportional to the number of cores. However, we emphasize that \(nz\) only offers an estimation of the computational cost, as other factors such as the fill-in/quality of the preconditioner may play a relevant role.

Figure 44 and Figure 45 report the performance of the parallel implementations of the PCG+ILU0 solve (per iteration) for the different matrices in Table 4. These results show that the execution times grow with the number of cores/problem dimension. The reason is that the number of actual floating-point arithmetic operations per iteration increases faster than \(nz\).

Comparing the three implementations, the MPI + OmpSs and MPI + OpenMP versions outperform the MPI variant; and the difference between the cases with one or two leaves per core also grows with the number of cores.
5.5.1.6 Portability
The code can be easily ported to other platforms equipped with implementations of MPI and OmpSs/OpenMP. The application is intended for clusters of multicore processors, and the current implementation cannot exploit graphics accelerators that may be present in the platform.

5.5.1.7 Usability
The effort to port the code to the new API combinations was considerable. The parallelization that only relied on MPI required a re-formulation of the PCG-ILU0 solver in order to explicitly expose task-parallelism. However, that parallel version assumed a direct 1:1 mapping between leaf nodes of the TDG and MPI ranks. This solution was not appropriate for the parallel MPI + OmpSs and MPI + OpenMP versions of PCG-ILU0 solver, and required a substantial rewrite of certain parts of the parallel code.

5.5.1.8 Suitability for Exascale
On a cluster equipped with recent multicore technology and a fast interconnect, the current PCG-ILU0 solver only scales to about 256 cores. The reason for this behavior is twofold. First, as the number of leaves in the TDG grows to expose further task-level parallelism, there is an associated increase in the number of floating-point arithmetic operations (flops) required by the PCG-ILU0 solver. Second, this additional cost (overhead) is mostly required to process the tasks that lie in the intermediate levels of the TDG, a part of the solver with a more reduced degree of concurrency that deteriorates the parallel performance of the PCG-ILU0 solver. We are currently working with a more sophisticated preconditioner that concentrates a larger fraction of the flops in the leaves. We expect this version to offer a higher degree of scalability.

5.5.1.9 Discussion, Further Enhancements, and Collaboration with WP3
These approaches extract task-parallelism by splitting the sparse matrix into multiple levels, yielding a directed acyclic graph, with the form of a binary tree, where the nodes represent tasks, the arrows indicate data dependencies, and most computational work is performed in the leaf tasks. This graph is then traversed from bottom-up for the computation of the preconditioner and one of the triangular solves during its application, and top-down for the second triangular solve. In principle, the tree can be expanded into further levels to expose any number of tasks and, therefore, degree of concurrency. However, doing so yields different preconditioners and, from a certain depth, incurs into a significant overhead. In general, the best compromise is to generate up to two leaves per core, to allow the OmpSs/OpenMP scheduler to optimize the computation. The experimental results confirm this assert for configurations with a reduced number of nodes, where the overhead is compensated by the OmpSs/OpenMP optimization. For unstructured matrices, the OmpSs/OpenMP runtime system accelerates the computation in most scenarios, due to the irregularity of the node sizes.
5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

These implementations combine the MPI and OmpSs/OpenMP programming models, with the best solution corresponding to a configuration that maps one MPI rank per socket and a number of OmpSs/OpenMP threads which fills the complete socket, mimicking the internal architecture of the cluster nodes. With these parameters, the new MPI + OmpSs/OpenMP version of ILU0 outperforms the initial implementation for clusters, which was based on MPI and could only process one leaf per rank. Comparing the results obtained using OmpSs and those achieved with OpenMP tasks we can conclude that there are negligible differences between them.

This study can benefit WP3, because it analyses several aspects of the interoperability between MPI + OmpSs/OpenMP.

5.5.2 HGraphBLAS HierMatFact: MKL + OmpSs/OpenMP

5.5.2.1 Introduction

Hierarchical matrices (abbreviated as H-matrices) provide an efficient numerical tool to store an n×n dense matrices in compressed form, using only O(nclog(n)) elements, where the parameter c can be tuned to control the accuracy of the approximation. Furthermore, common linear algebra operations, such as matrix addition, matrix-matrix multiplication and matrix factorizations, can be (approximately) computed in H-arithmetic, with a cost of O(nlog(dn)) floating-point operations (flops), for a small constant d (see [9]).

Hierarchical matrices (H-Matrices) can be viewed as a problem that stands in-between the highly-parallel dense matrix factorization and their memory-bound sparse counterparts. For H-Matrices in particular, it is thus essential to exploit the problem parallelism via a task-parallel approach, but also to extract loop-level parallelism for certain suboperations/tasks, via e.g. a multi-threaded library such as Intel MKL.

The aim of this application is to compute an LU factorization of an H-Matrix. This decomposition can be computed, e.g., via a right-looking algorithm (see Figure 46), which is the blocked algorithm that has classically been used for obtaining high performance LU decompositions.

\begin{verbatim}
Require: A ∈ R^{n×n}
1: for k = 1, 2, . . . , n do
2:   A_{kk} = L_{kk}U_{kk}
3: for j = k + 1, k + 2, . . . , n do
4:   U_{kj} := L_{kk}^{-1}A_{kj}
5: end for
6: for i = k + 1, k + 2, . . . , n do
7:   L_{ik} := A_{ik}U_{kk}^{-1}
8: end for
9: for i = k + 1, k + 2, . . . , n do
10:  for j = k + 1, k + 2, . . . , n do
11:   A_{ij} := A_{ij} - L_{ik}U_{kj}
12: end for
13: end for
14: end for
\end{verbatim}

Figure 46: Blocked Right-Looking algorithm (BRL) for solving an LU factorization.

The operations that appear in the BRL algorithm correspond to three basic linear algebra building blocks (or computational kernels):

- LU factorization.
- Triangular system solve (with upper triangular factor or lower triangular factor).
- Matrix-matrix multiplication.

The algorithm for obtaining the LU decomposition of H-Matrices (H-LU) is a straightforward generalization of the BRL algorithm for the LU factorization that leverages the hierarchical structure of those matrices, adapting the procedure to different block sizes.
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Concretely, the kernels operating on dense blocks correspond to building blocks from LAPACK or the Level-3 BLAS (as implemented, e.g., in Intel MKL), namely:

- routine_GETRF from LAPACK for the LU factorization.
- routine_TRSM from BLAS for solving Triangular Systems.
- routine_GEMM from BLAS for the matrix product.

5.5.2.2 INTERTWinE Ambition

The main goal of the project is to analyse and develop codes adapted to future Exascale computing systems by exploiting interoperability. Concretely, merging MKL multithread with OmpSs or OpenMP is the interoperability combination that has been studied in this application, in order to take advantage of the vast amount of thread concurrency which characterizes those systems. These combinations will allow to achieve a good parallel efficiency on multicore processors exploiting coarse-grain task-based and also fine-grain loop-based parallelism.

5.5.2.3 Implementation Details

In order to achieve a good parallel efficiency, three different approaches have been analysed when implementing parallel versions of the H-LU factorization.

The simpler approach to parallelize the BRL algorithm for the H-LU factorization is to rely on a multi-threaded implementation of the BLAS kernels, by linking to MKL multi-threaded building blocks. However, this fork-join approach can only exploit a reduced degree of concurrency and incurs a considerable overhead due to the synchronizations at the end of each BLAS kernel.

A second approach is to exploit the parallelism inside the individual loops of the BRL algorithm, adding OpenMP parallel loop pragmas. This alternative often provides considerable more concurrency than the previous approach, although it still cannot exploit the parallelism between operations in different loops (e.g., from distinct iterations of the algorithm).

The third approach, which is the one that offers the best parallel efficiency, relies on a task-based parallel implementation of the algorithm via OmpSs, which employs a runtime to leverage the actual concurrency of the algorithm. In this case, each invocation to one of the three types of kernels that appear in the BRL algorithm is identified as a task, including the dependencies, which are evaluated by the runtime in order to create a task dependency graph which is used to schedule the tasks to the cores taking into account the dependencies.

Although OmpSs + MKL multithread is the proposed interoperability combination, it has not been possible to test it. The problem when combining OmpSs + MKL multithread lies in that the default configuration of OmpSs does not recognize the topology of the machine and, as a consequence, the system maps all the threads to the same physical core (the only one that is detected), yielding the oversubscription problem. In our view, this problem could be tackled using the Resource Manager implemented within the INTERTWinE project. Therefore, the experiments reported in this document are obtained using OmpSs or OpenMP + MKL sequential.

5.5.2.4 Correctness

This application computes the LU-factorization of an H-Matrix (abbreviated as H-LU) exploiting task-parallelism [2]. We provide two H-Matrices of size 600x600 to test the application. Those matrices are partitioned into 3 levels of sizes 600x600, 300x300 and 100x100 respectively, one of them having 0% of null blocks and the other one 25%. The files “H-Matrices/matrix_600_3_00_0.txt” and “H-Matrices/matrix_600_3_25_0.txt” store the two example H-Matrices values that can be used to run and test the application.

For the H-Matrix “H-Matrices/matrix_600_3_00_0.txt”, the LU factorization should show in the output (apart from size, execution time and GFLOPS): “Values[360000] =
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

6.0045285200588364e+02 and, for “H-Matrices/matrix_600_3_25_0.txt”, it should show: “Values[350000] = 6.00442731918694e+02”. Those values can be used to check the correctness of the H-LU execution.

Moreover, we provide an Octave/Matlab code for generating H-Matrices to test the application with them. In order to obtain a correct LU-factorization, the users of that application must provide some parameters when the code starts running:

- File to store H-matrix: name of the file (e.g. ‘file_name.txt’) where the H-matrix data will be stored.
- File to store null information: name of the file (e.g. ‘file_null_info_name.txt’) where the information on null blocks of the H-matrix will be stored.
- Size: size of the H-matrix (n), which also corresponds to the size of the first level of the H-matrix.
- Dispersion degree: percentage (between 0 and 100) of null blocks of the matrix whose elements will be set to zero.
- Number of levels.
- Dimension of each level: size of each of the previously fixed number of levels of the H-matrix.

In order to take profit of MKL performance, the blocks of the matrix should not be smaller than 100.

Our goal is to expose the performance benefits of using a task-parallel programming model such as OmpSs for the solution of linear algebra operations on H-matrices. We do not pursue the development of a mature library for this purpose that competes with other implementations. For this reason, our experiments in this section are designed to assess the scalability of our codes, in a simplified yet practical scenario. For this evaluation, we employ H-matrices comprising dense and null blocks, but without low-rank blocks. From the point of view of a task-parallel execution, dealing with low-rank blocks requires special numerical kernels. These kernels simply change the implementation and costs of the tasks operating on these blocks, but have no other effect on the task-based parallelization effort.

Moreover, we have also developed a simple loop-parallel OpenMP version to offer a comparison between it and the OmpSs task-based parallel code performance.

5.5.2.5 Performance

The experiments were performed using IEEE double precision arithmetic, on a server equipped with two Intel E5-2603v3 sockets, each with a 6-core processor (1.6 GHz), and 32 Gbytes of DDR3 RAM. Our codes were linked with Intel MKL (composer_xe_2011_sp1) for the BLAS kernels, a modified version of the routine for the LU factorization in the legacy version LAPACK that avoids pivoting, and OmpSs (version 16.06). The OpenMP version employed gcc (version 5.3).

The performance of the task-parallel H-LU factorization depends on that of the building blocks, which are computed in our implementation via calls to the tuned routines in Intel MKL. Note that, as the parallelism is extracted by the runtime, our code for the H-LU factorization does not require a multi-threaded implementation of these building blocks.

Figure 47 reports the GFLOPS (billions of flops per second) attained by the four building blocks (LU factorization without pivoting, upper and lower triangular solve, and matrix-
matrix multiplication) using a single core of the target platform. In all cases, we used square operands of dimension \( t_s \). As could be expected, the highest performance rates are attained by the matrix-matrix multiplication kernel (DGEMM). The reason is that, unlike the LU and triangular solvers, this operation does not present data dependencies.

The figure also reveals that the asymptotic performance for DGEMM is around 12.1 GFLOPS. This value is relevant because the DGEMM kernel dominates the cost of the H-LU factorization by a large margin. Furthermore, the problem size \( t_s \) in the experiment in Figure 8 is related with that of the leaf blocks of the block cluster tree. For example, for an H-matrix with leaf blocks of dimension \( t_s = 1000 \), we can expect that an execution of the H-LU factorization, using a single core, proceeds at the rate reported for kernel DGEMM and that problem size in the figure. A multiplication of the asymptotic rate with the number of cores employed for a task-parallel execution of the H-LU factorization thus offers an upper bound on the highest performance rate that we can observe in a parallel execution.

It is important to realize that the evaluation of the building blocks was performed using data already stored in the processor cache. Especially for the smallest problems, the GFLOPS rate is much lower if the data has to be fetched from the main memory as part of the execution. However, in the scenario occurring during the H-LU factorization, the operands to a task are the results from a previous task and, therefore, can be expected to reside in the higher levels of the memory hierarchy. Thus, the GFLOPS rates in the figure are those that we can expect in a practical execution of the H-LU factorization routine.

For the evaluation of the task-parallel H-LU factorization, we generated two H-matrices, of dimension \( n=5,000 \) and \( 10,000 \). The matrix entries were randomly distributed following a normal distribution in \((0, 1)\). To avoid numerical difficulties, the matrix was enforced to be diagonally dominant. For each case, we varied the number of levels \( (n_l) \) and granularity of the blocks in each level (cases A and B) as displayed in the following table. Finally, we performed experiments for four different ratios of null blocks: 0% (full matrix), 25%, 50% and 75%.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( n_l )</th>
<th>Block granularity in each level</th>
</tr>
</thead>
</table>
### Table 6: Configurations for the experimental evaluation of the H-LU factorization.

<table>
<thead>
<tr>
<th>5K</th>
<th>2</th>
<th>5K, 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>5K, 500, 100</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5K, 2.5K, 1,250, 250</td>
</tr>
<tr>
<td>10K</td>
<td>2</td>
<td>10K, 500</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10K, 500, 100</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10K, 1K, 500, 100</td>
</tr>
</tbody>
</table>

The following graphics report (see Figure 48 with its six sub-figures a-f) the GFLOPS rates attained both by the MKL+OmpSs version and the MKL+OpenMP version of the H-LU factorization routines on the Intel 12-core server. The top of the performance line (limit for the y-axis) in all plots is set at 144 GFLOPS, which roughly corresponds to the highest practical performance that we could expect using 12 cores, each delivering about 12 GFLOPS for DGEMM.
Matrix 5000x5000 (3 Levels)

Matrix 5000x5000 (4 Levels)
D5.2 INTERIM REPORT ON APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Matrix 10000x10000 (2 Levels)

Matrix 10000x10000 (3 Levels)
This evaluation offers some insights:

- The GFLOPS rates grow with the number of cores in most cases for both parallelization alternatives. In general, the task-parallel OmpSs approach outperforms the simpler loop-parallel OpenMP variant. An exception to this is the small problem with 2 levels for which the OpenMP solution provides a higher GFLOPS rate for any number of cores.

- The performance differences between the two parallelization alternatives tends to become larger, in favour of OmpSs, when the concurrency is reduced. This occurs when the ratio between the factorization cost and number of cores is small, taking into account the rate of nonzeros. A clear example is visible for the small problem with 2 levels where the performance of the OpenMP version shows a significant drop when the dispersion is shifted from 25% to 50%. The OmpSs also suffers a performance reduction for this particular case, but only for the execution with 12 cores.

Our experience with matrix factorizations identified a relevant performance bottleneck for two programming model combinations: OmpSs + MKL and StarPU + MKL. In particular, in these two cases we identified a drop in the scalability of the solution due to oversubscription of some of the hardware cores, as a result of an incorrect mapping of the threads to the physical resources.

5.5.2.6 Portability

The code can be easily ported to other platforms equipped with a sequential implementation of the BLAS and OmpSs/OpenMP. The application is intended for multicore processors, and the current implementation cannot exploit graphics accelerators that may be present in the platform.
5.5.2.7 Usability

Our goal when designing the H-LU code was to offer a prototype of that factorization over H-matrices to evaluate the performance that OmpSs could achieve for that type of factorization [2].

The main problem was to avoid overlapping Program Objects in OmpSs, which forced us to force a logical partition of every leaf block into blocks of the smallest size on the H-matrix.

Moreover, due to the operations that have to be performed to obtain the factorization, we had to use a special storage format named Block Data Layout (BDL) [10] instead of the common Column Major Order (CMO) which would have been easier to implement. That storage format implies that each leaf block involved in a certain operation has to be logically partitioned into blocks of size equal to smaller of the blocks sizes used in the operation.

After solving the storage format problems, the implementation using OpenMP has not required any significant effort. It was developed mainly by adding parallel for loops pragmas.

In general, we expect that the Resource Manager may tackle some of the interoperability problems identified for the HMat application (see the Best Practice Guide for writing OpenMP/OmpSs/StarPU + MKL interoperable programs) by controlling the number of spawned threads and their binding to the hardware (physical) cores in a flexible and efficient manner.

5.5.2.8 Suitability for Exascale

With care, the prototype code for H-LU can be applied to compute the factorization of a very-large scale matrices, with no a priori limit. In other words, in principle there is no restriction on the scalability of the code. However, from the application’s point of view, the generation and use of such large-scale problem instances may not be representative of real problems. Furthermore, the current implementation of Hlib is restricted to operate on a single node, which restricts the dimension of the problems to those which fit in the target server.

5.5.2.9 Discussion, Further Enhancements, and Collaboration with WP3 and WP4

As soon as oversubscription issues are solved, the OpenMP/OmpSs + MKL Multithread combination will be developed and tested by using the Resource Manager. We hope to obtain a remarkable benefit from combining both levels of parallelism.

Moreover, in respect of specific improvements of the code, the new version of OmpSs (and Nanos) will avoid overlapping Program Objects restrictions. This will lead on being able to adapt the partition of the H-Matrix and grain of the operations tasks according to the specific structure instead of being forced to apply a logical partitioning of each block into the smallest block size of the matrix. By controlling the grain of the tasks and the degree of concurrency, we aim to achieve a better parallel efficiency.

WP4 developments will allow us to combine OpenMP/OmpSs + MKL Multithread and to avoid overlapping Program Objects restrictions of OmpSs. Taking advantages of this will improve the performance of the described application and will serve both to WP4 (for testing the new issues) and WP3 (for analysing the interoperability aspects).
5.6 PLASMA: Parallel Linear Algebra Software for Multicore Architectures

5.6.1 Introduction
The Parallel Linear Algebra Software for Multicore Architectures (PLASMA) is a parallel library for numerical linear algebra with dense matrices. PLASMA offers a collection of routines for solving linear systems of equations and least squares problems. With respect to functionality, PLASMA follows the structure used by LAPACK and BLAS libraries. Four different versions are provided for most subroutines, related to different data precisions (double complex, double real, single complex, single real).

PLASMA uses matrices stored as square blocks called tiles, in which data are stored consecutively in memory. The heart of PLASMA consists in implementing algorithms working on these tiles. These algorithms prescribe an order of sequential kernels operating on one or several tiles, which form sequential tasks. PLASMA relies on concurrent runtime scheduling of these tasks. The concept is closely related to the idea of expressing computation through a task graph, often referred to as the DAG (Directed Acyclic Graph), and the flexibility of exploring the DAG at runtime.

PLASMA can be split into the routines with the tile algorithms and the sequential kernels. The sequential kernels in PLASMA map to simple calls to BLAS routines, to LAPACK routines, or may require custom implementations derived specifically for tile algorithms (e.g. in the case of the QR factorization).

5.6.2 INTERTWinE Ambition
The team at the University of Manchester collaborates closely with the ICL at UTK, where PLASMA and QUARK have been developed. As a consequence, the University of Manchester is contributing to the development of the new PLASMA version based on task-based runtime of the OpenMP 4.5. The repository of the project is at https://bitbucket.org/icl/plasma

After finishing this transition, the subset of routines for solving systems of linear equations has been also ported to OmpSs, and it is being ported to StarPU runtime systems. This will allow a performance comparison of the different runtimes for PLASMA, but will also provide building blocks for studying coexistence of different runtime systems in one application. Performance of these routines will be evaluated for versions based on QUARK, OpenMP, OmpSs, and StarPU.

Coexistence of PLASMA with applications based on OpenMP, OmpSs and StarPU will be studied after these developments.

The API combinations selected for INTERTWinE include:

1. **PLASMA + OpenMP**
   Motivation:
   - Simpler portability and interoperability through the widely adopted and standardized OpenMP API.

2. **PLASMA + OmpSs (prioritized in collaboration with WP3 and WP4)**
   Motivation:
   - Potential for employing the Resource Manager of INTERTWinE into PLASMA allowing flexible running of PLASMA only on a subset of cores and several instances of PLASMA;
   - Generating a version of PLASMA on top of OmpSs for testing coexistence in applications.

3. **PLASMA + StarPU**
   Motivation:
   - Allowing PLASMA to go beyond the shared memory realm to distributed memory environments;
4. PLASMA performance in different runtimes
Motivation:
  • Comparison of the efficiency of the different runtime systems for PLASMA.

5. Coexistence of PLASMA with OpenMP, OmpSs, and StarPU
Motivation:
  • As a numerical library, it is necessary to verify that PLASMA can be smoothly combined with applications written in different APIs.

5.6.3 PLASMA + OpenMP tasks
PLASMA has recently been ported from its own runtime system, QUARK, to OpenMP tasks with dependencies. A significant part of this transition has been performed within the scope of the INTERTWinE project. The primary motivations behind this transition have been simplifying the maintenance of the PLASMA library, and better interoperability with OpenMP applications as well as the Intel Math Kernel Library (MKL). On the other hand, QUARK has been optimized for linear algebra algorithms and, therefore, the more general tasking in OpenMP may potentially lead to loss of performance in some circumstances. An important step of the transition is, therefore, to evaluate the performance and verify that it has not been unduly affected by the switching of runtime system.

5.6.3.1 Implementation details
The structure of parallel execution in PLASMA is quite straightforward. After some preprocessing steps, an OpenMP parallel section is opened. The following is an example for a function for performing the QR-factorization. PLASMA uses the scheme with a single task producer due to the data dependencies among the tasks.

```c
// asynchronous block
#pragma omp parallel
#pragma omp master
{
  ...
  plasma_omp_pzgeqrf(A, ...);
  ...
}
// implicit synchronization
```

The `plasma_omp_pzgeqrf()` function is responsible for running a tile-based QR factorization algorithm. It contains explicit loops over the tiles (i.e. matrix sub-blocks) of the matrix A and calls sequential kernels to operate on these tiles. In case of QR factorizations, the four required kernels are called zgeqrt, zunmqr, ztsqrt, and ztsmqr:

```c
void plasma_pzgeqrf(plasma_desc_t A, ...)
{
  for (int k = 0; k < imin(A.mt, A.nt); k++) {
    core_omp_zgeqrt(...);
    for (int n = k+1; n < A.nt; n++) {
      core_omp_zunmqr(...);
    }
    for (int m = k+1; m < A.mt; m++) {
      core_omp_ztsqrt(...);
    }
  }
  ...
}
```
The tasks are created inside these kernels; see for example the `core_omp_zgeqrt()` kernel performing the sequential QR factorization of a tile.

```c
void core_omp_zgeqrt(m, n, A, lda, T, ib, ...) {
    #pragma omp task depend(inout:A[0:lda*n]) \
        depend(out:T[0:ib*n])
    {
        // Call the sequential kernel.
        int info = core_zgeqrt(A, T, ...);
    }
}
```

The QUARK-based version (in PLASMA 2.8.0) of the tile-based algorithm looks similar:

```c
void plasma_pzgeqrf(plasma_desc_t A, ...) {
    for (int k = 0; k < imin(A.mt, A.nt); k++) {
        QUARK_CORE_zgeqrt(...);
        for (int n = k+1; n < A.nt; n++) {
            QUARK_CORE_zunmqr(...);
        }
        for (int m = k+1; m < A.mt; m++) {
            QUARK_CORE_ztsqrt(...);
            for (int n = k+1; n < A.nt; n++) {
                QUARK_CORE_ztsmqr(...);
            }
        }
    }
}
```

However, the tasks are generated and inserted differently by QUARK:

```c
void QUARK_CORE_zgeqrt(Quark *quark, ..., 
                   m, n, A, lda, T, ib, ...)
{
    QUARK_Insert_Task(quark, CORE_zgeqrt_quark, task_flags,
                     sizeof(int),                        &m,     VALUE,
                     sizeof(int),                        &n,     ARG,
                     &A,                                    ARG,
                     &lda,                                  ARG,
                     &T,                                    ARG,
                     &ib,                                   ARG,
                     NULL)                                ARG;
```
5.6.3.2 Performance

Employing OpenMP tasks in PLASMA leads to a simpler code, and avoids the need to maintain an application-specific runtime system; however, we must consider the possibility of reduced computational performance. Following the conversion we have performed a series of performance benchmarks for one-sided factorizations, namely QR factorisation, Cholesky factorisation, and LU factorisation. These benchmarks have been performed with complex matrices in double precision. The performance has been measured for square matrices, with size ranging from 1000 to 20000, using 68 cores of the Intel Xeon Phi 7250 processor (also known as Knights Landing). The version of PLASMA based on OpenMP is 17.01, while the QUARK-based version is 2.8.0. Intel icc compiler version 16.0.3 was used for these tests with the default OpenMP runtime. We also compare the performance of the two generations of PLASMA with LAPACK, using the threaded MKL BLAS library version 11.3.3. No significant tuning of internal parameters of PLASMA such as the tile size was performed here. Nevertheless, the same value was consistently used for both PLASMA generations. In the presented results, the tiles were of the size 300 x 300 for all matrix sizes.

![Graph showing performance of QR factorization](image)

**Figure 49:** Performance of the QR factorization for different matrix sizes (ZGEQRF function). Theoretical peak performance is 3053 Gflop/s.
Figure 50: Performance of the Cholesky factorization for different matrix sizes (ZPOTRF function). Theoretical peak performance is 3053 Gflop/s.

Figure 51: Performance of the LU factorization for different matrix sizes (ZGETRF function): 8 threads per panel factorization. Theoretical peak performance is 3053 Gflop/s.

The benchmarks reveal, that for the QR and Cholesky factorizations, there is no loss of performance from the OpenMP based version in comparison to the QUARK-based version. For LU factorization, the performance of the OpenMP version is somewhat lower; however, this seems to be related to changes in the underlying tile algorithm rather than the change of runtime system. Another reason for the lower performance of the LU factorization is the fact that the algorithm assigns priorities to tasks allowed in OpenMP 4.5. However, the Intel icc compiler did not support this feature in version 16.0.3, and the priorities were commented out from the source code for these tests as well as in the released version. The challenging part of this algorithm is the partial pivoting, which synchronizes operations on tiles within a single panel, and this issue will be the subject of further optimization.

5.6.3.3 Correctness
The PLASMA library contains a set of dedicated correction testers. They are typically based on random matrices and include some corner cases (matrices of size 1, matrices...
of a single row, etc.). During the change of the runtime system from QUARK to OpenMP tasks, correctness of all routines has been continuously checked by these tests.

5.6.3.4 Portability
Portability and interoperability were the two main reasons for moving PLASMA from its own runtime system QUARK to the OpenMP standard. The codes are now simply compiled using a C compiler with the OpenMP support. It is now much simpler to obtain good parallel performance in a portable manner. The new codes can run on any system with support of OpenMP version 4.5, such as NUMA nodes or members of the Intel MIC family. PLASMA relies on tasks with dependencies, and it has recently introduced the use of priorities of tasks into some of its algorithms.

5.6.3.5 Usability
As PLASMA had already been based on tasks with dependencies, it has been quite straightforward to port it from QUARK to the OpenMP tasks. This is also due to similarities of these runtime systems. It is simpler to implement new algorithms based on OpenMP tasks compared to development with QUARK. In fact, this has been the main goal of the transition to this runtime.

5.6.3.6 Suitability for Exascale
PLASMA is aiming at shared memory architectures. So rather than reaching Exascale as a standalone application, the goal of PLASMA is maximizing performance on one compute node. This makes it an important building block of larger applications aiming at Exascale performance. As an example, one could use PLASMA on subdomain level within a large domain decomposition solver running over a distributed memory system. Towards Exascale, it is increasingly important to exploit parallel resources of one node as efficiently as possible.

5.6.3.7 Discussion, Further Enhancements, and Collaboration with WP4
Given the task-based structure of the library, porting PLASMA from the QUARK runtime system to OpenMP has been reasonably straightforward, and did not require significant changes to the structure of the code. Performance tests have revealed that the new version is equally performant as the one based on QUARK in most cases. The lower performance in the case of LU factorization has become a matter of further research.

On the other hand, like PLASMA based on QUARK, PLASMA based on OpenMP still has a limited flexibility in assigning hardware resources to different function calls. In particular, it does not provide to the user enough control to run different routines on a prescribed set of cores.

This limitation can be overcome by using the native offload mechanism of the INTERTWinE Resource Manager developed in WP4. For this reason, parts of PLASMA have been ported to OmpSs. This version will allow flexible partitioning of the resources into subsets of cores and running PLASMA functions only on such subsets. Hence, for this reason, this API combination has been selected as a prioritized combination suitable for testing the Resource Manager.
6 Updated Software Release Plans

This section summarizes the revised project timeline from the perspective of software releases for each application/kernel and its API combination (see Table 7). The highlighted API combinations in bold font indicate those software releases that have been implemented. The table also contains indicators to software releases that are considered as potential candidates to verify the functionalities of the D/C and the RM. For instance, all BAR benchmarks are considered as candidates to validate the RM, while the D/C will be verified by DPLASMA/PaRSEC.

<table>
<thead>
<tr>
<th>Application/Kernel</th>
<th>API combinations</th>
<th>Ready for month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ludwig</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1. MPI + GASPI</td>
<td>1. M13</td>
</tr>
<tr>
<td></td>
<td>2. MPI + OpenMP 4.0</td>
<td>2. M16</td>
</tr>
<tr>
<td></td>
<td>3. MPI + OpenMP 4.5 tasks</td>
<td>3. M19</td>
</tr>
<tr>
<td></td>
<td>4. MPI + OmpSs</td>
<td>4. M20</td>
</tr>
<tr>
<td></td>
<td>5. GASPI + OpenMP/OmpSs</td>
<td>5. M21</td>
</tr>
<tr>
<td></td>
<td>6. MPI Endpoints + OmpSs</td>
<td>6. M27</td>
</tr>
<tr>
<td></td>
<td>7. MPI Endpoints + StarPU</td>
<td>7. M27</td>
</tr>
<tr>
<td></td>
<td>8. MPI Endpoints + OpenMP</td>
<td>8. M27</td>
</tr>
<tr>
<td>iPIC3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1. MPI + OpenMP threads</td>
<td>1. M9</td>
</tr>
<tr>
<td></td>
<td>2. MPI + OpenMP tasks</td>
<td>2. M10</td>
</tr>
<tr>
<td></td>
<td>3. MPI + GASPI</td>
<td>3. M19</td>
</tr>
<tr>
<td></td>
<td>4. MPI + OmpSs</td>
<td>4. M20</td>
</tr>
<tr>
<td></td>
<td>5. MPI + GASPI + OpenMP</td>
<td>5. M21</td>
</tr>
<tr>
<td></td>
<td>6. MPI + GASPI + OmpSs</td>
<td>6. M21</td>
</tr>
<tr>
<td></td>
<td>7. MPI + GASPI + StarPU</td>
<td>7. M24</td>
</tr>
<tr>
<td></td>
<td>8. MPI + StarPU</td>
<td>8. M24</td>
</tr>
<tr>
<td></td>
<td>9. MPI Endpoints + OpenMP</td>
<td>9. M27</td>
</tr>
<tr>
<td>TAU</td>
<td>Linolv kernel:</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1. MPI</td>
<td>1. M6</td>
</tr>
<tr>
<td></td>
<td>2. MPI + OpenMP threads</td>
<td>2. M15</td>
</tr>
<tr>
<td></td>
<td>3. MPI + OpenMP tasks</td>
<td>3. M17</td>
</tr>
<tr>
<td></td>
<td>4. GASPI + OpenMP</td>
<td>4. M40</td>
</tr>
<tr>
<td></td>
<td>5. GASPI + OmpSs</td>
<td>5. M33</td>
</tr>
<tr>
<td></td>
<td>6. MPI + OmpSs</td>
<td>6. M33</td>
</tr>
<tr>
<td></td>
<td>Memory-spaces kernel:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7. GASPI + OpenMP</td>
<td>7. M20</td>
</tr>
<tr>
<td></td>
<td>8. MPI + OpenMP</td>
<td>8. M40</td>
</tr>
<tr>
<td></td>
<td>9. MPI + OmpSs</td>
<td>9. M24</td>
</tr>
<tr>
<td></td>
<td>10. GASPI + OmpSs</td>
<td>10. M24</td>
</tr>
<tr>
<td></td>
<td>11. MPI Endpoints</td>
<td>11. M24</td>
</tr>
<tr>
<td>Application/Kernel</td>
<td>API combinations</td>
<td>Ready for month</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>5</td>
<td><strong>Graph BLAS</strong>&lt;br&gt;ILU0**&lt;br&gt;1. MPI + OpenMP&lt;br&gt;2. MPI + OmpSs**&lt;br&gt;<strong>Hierarchical matrix factorization (candidate to validate the RM)</strong>&lt;br&gt;3. OmpSs + MKL&lt;br&gt;4. OpenMP + MKL**&lt;br&gt;<strong>SpMV, SpMM</strong>&lt;br&gt;5. GASPI + OpenMP&lt;br&gt;6. MPI + OpenMP&lt;br&gt;7. MPI + OmpSs**</td>
<td>1. M15&lt;br&gt;2. M15&lt;br&gt;3. M16&lt;br&gt;4. M18&lt;br&gt;5. M28&lt;br&gt;6. M28&lt;br&gt;7. M28</td>
</tr>
<tr>
<td>6</td>
<td><strong>PLASMA</strong>&lt;br&gt;1. PLASMA + OpenMP (candidate to validate the RM)<strong>&lt;br&gt;2. PLASMA + OmpSs (candidate to validate the RM)</strong>&lt;br&gt;3. PLASMA + StarPU**</td>
<td>1. M17&lt;br&gt;2. M19&lt;br&gt;3. M21</td>
</tr>
<tr>
<td>7</td>
<td><strong>DPLASMA</strong>&lt;br&gt;1. DPLASMA/PaRSEC + MPI (candidate to verify the D/C)<strong>&lt;br&gt;2. DPLASMA/PaRSEC + OpenMP&lt;br&gt;3. DPLASMA/PaRSEC + OmpSs&lt;br&gt;4. DPLASMA/PaRSEC + StarPU</strong></td>
<td>1. M19&lt;br&gt;2. M21&lt;br&gt;3. M24&lt;br&gt;4. M27</td>
</tr>
</tbody>
</table>

**Table 7:** Release timeline for the applications/kernels and their API combinations.
7 Summary

This deliverable provided insight into the progress on the development of applications and kernels made during the first half of the project (PM1—PM18). In particular, we 1) demonstrated various detailed approaches for combining different APIs in each application/kernel; 2) presented validation strategies for each application/kernel with the corresponding performance results, following the earlier defined evaluation criteria [7]; 3) identified some of the interoperability issues, e.g. oversubscription for resources while using multiple runtimes at once, that had arisen during this development process; 4) highlighted possible strategies for solving these issues, for instance, by employing the INTERTWinE Resource Manager service; 5) documented all these interim findings and lessons learnt.

In conjunction to this development, we revised our initial software releases plans. We aimed to cluster software releases with respect to certain API combinations, e.g. MPI + OpenMP, in order to make some of them available for the usage by WP4 in their developments of the Directory/Cache and the Resource Manager.

We have also sharpened the focus of the work package, following the reviewers’ recommendations at the Project Check Meeting in PM09, by prioritizing the current list of API combinations. In addition, we identified potential candidates for verification of the functionalities of the Directory/Cache and the Resource Manager.

Following the co-design cycle, we have already engaged the WP3 and WP4 leaders, the co-design manager, and the developers of the Directory/Cache and the Resource Manager in the above-mentioned activities within WP5. This collaboration will become even closer once we will start exploiting the Directory/Cache and Resource Manager services in our applications and kernels.
**References**


Appendix: INTERTWinE Computing Infrastructure

This section outlines the INTERTWinE computing infrastructures, which are mainly in house architectures that were used in our experiments presented in this document.

8.1 ARCHER

ARCHER is the latest UK National Supercomputing Service provided by the ARCHER partners such as EPCC and the University of Edinburgh. ARCHER is a Cray XC30 system, which compute nodes contain two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors. Each of the cores in these processors can support 2 hardware threads (Hyperthreads). Within the node, the two processors are connected by two QuickPath Interconnect (QPI) links.

8.2 Beskow

Beskow is a Cray XC system at the PDC Center for High Performance Computing at the KTH Royal Institute of Technology. Beskow is based on Intel Haswell processors and Cray Aries interconnect technology. There are a total of 9 racks in two rows with a theoretical peak performance of nearly 2 petaflops. The system is named after the famous Swedish author and illustrator Elsa Beskow, who produced many delightful children's books.

8.3 DLR SCART HPC-cluster

The DLR SCART HPC-cluster is equipped with Ivy-Bridge processors (Intel Xeon Processor E5-2680 v2, dual-socket, 10 cores per socket, 2.8GHz) connected with InfiniBand QDR (Intel True Scale Fabric).

8.4 MinoTauro

The MinoTauro system is placed at BSC. The cluster is based on Bull blades containing each one 2 Intel E5649 (6-Core) processor at 2.53 GHz with 24 GB of main memory. Each node in the cluster is equipped with a 250 GB SSD (Solid State Disk) as local storage and they are connected through InfiniBand QDR (40 Gbit/s bandwidth each) to a non-blocking network.

8.5 MareNostrum

MareNostrum is a large-scale computing infrastructure at Barcelona Supercomputing Center (BSC). This platform connects 3,056 compute nodes via an Infiniband Mellanox FDR10 network. Each node contains two Intel Xeon E5-2670 processors for a total of 16 cores per server (2.6 GHz). The nodes employed in these experiments were also equipped with 64 Gbytes of DDR3 RAM.