### D5.1 Report on initial 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>M9</td>
</tr>
<tr>
<td>Submission date</td>
<td>30/JUN/2016</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, UJI</td>
</tr>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Status</td>
<td>Final</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Roman Iakymchuk (KTH), Enrique S. Quintana-Ortí (UJI), Luis Cebamanos (UEDIN), Dana Akhmetova (KTH), Thomas Gerhold (DLR), Barbara Brandfass (DLR), Christian Simmendinger (T-Systems SfR), Jakub Sistek (Manchester)</td>
</tr>
<tr>
<td>Reviewer(s)</td>
<td>Mark Bull (UEDIN), Vicenç Beltran (BSC)</td>
</tr>
<tr>
<td>Dissemination level</td>
<td>PU 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>09/MAY/16</td>
<td>Prepared template based on iPIC3D</td>
<td>Dana Akhmetova (KTH), Roman Iakymchuk (KTH), Enrique S. Quintana-Ortí (UJI)</td>
</tr>
<tr>
<td>0.2</td>
<td>1/JUN/16</td>
<td>Initial version, including all applications/kernels</td>
<td>Roman Iakymchuk (KTH), Enrique S. Quintana-Ortí (UJI), Luis Cebamanos (UEDIN), Dana Akhmetova (KTH), Thomas Gerhold (DLR), Barbara Brandfass (DLR), Christian Simmendinger (T-Systems SfR), Jakub Sistek (Manchester)</td>
</tr>
<tr>
<td>0.3</td>
<td>3/JUN/16</td>
<td>Revised version: took into account comments from the internal WP5 review</td>
<td>Roman Iakymchuk (KTH), Enrique S. Quintana-Ortí (UJI), Luis Cebamanos (UEDIN), Dana Akhmetova (KTH), Thomas Gerhold (DLR), Christian Simmendinger (T-Systems SfR), Jakub Sistek (Manchester)</td>
</tr>
<tr>
<td>0.4</td>
<td>7/JUN/16</td>
<td>Ready for internal review</td>
<td>Roman Iakymchuk (KTH), Enrique S. Quintana-Ortí (UJI)</td>
</tr>
<tr>
<td>0.5</td>
<td>14/JUN/16</td>
<td>Incorporated reviewers` comments</td>
<td>Enrique S. Quintana-Ortí (UJI), Roman Iakymchuk (KTH), Dana Akhmetova (KTH), Thomas Gerhold (DLR), Barbara Brandfass (DLR), Christian Simmendinger (T-Systems SfR), Jakub Sistek (Manchester)</td>
</tr>
<tr>
<td>0.6</td>
<td>14/JUN/16</td>
<td>Added software release plans and an overview of the Git and Jenkins setups</td>
<td>Luis Cebamanos (UEDIN), Enrique S. Quintana-Ortí (UJI), Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>0.7</td>
<td>15/JUN/16</td>
<td>Changed the bibliography and citation styles</td>
<td>Enrique S. Quintana-Ortí (UJI), Roman Iakymchuk (KTH)</td>
</tr>
<tr>
<td>0.8</td>
<td>16/JUN/16</td>
<td>Minor fixes</td>
<td>Enrique S. Quintana-Ortí (UJI), Roman Iakymchuk (KTH), Jakub Sistek (Manchester)</td>
</tr>
<tr>
<td>1.0</td>
<td>17/JUN/16</td>
<td>Final version</td>
<td>Roman Iakymchuk (KTH), Enrique S. Quintana-Ortí (UJI)</td>
</tr>
<tr>
<td>-----</td>
<td>----------</td>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
</tbody>
</table>
Table of Contents

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

2 Introduction ......................................................................................................................... 8
  2.1 Glossary of Acronyms ....................................................................................................... 8

3 Applications/Kernels ............................................................................................................ 10
  3.1 Ludwig: A parallel Lattice-Boltzmann code for complex fluids .................................... 10
    3.1.1 Introduction ............................................................................................................... 10
    3.1.2 Ludwig workload ........................................................................................................ 10
    3.1.3 Current API combination .......................................................................................... 12
    3.1.4 INTERTWinE Ambition ............................................................................................. 12
    3.1.5 Benchmarks ............................................................................................................... 13
    3.1.6 Evaluation plan .......................................................................................................... 13
    3.1.7 Preliminary Results .................................................................................................... 13
  3.2 iPIC3D: Implicit Particle-in-Cell Code for Space Weather Applications ......................... 16
    3.2.1 Introduction ............................................................................................................... 16
    3.2.2 iPIC3D Workload ..................................................................................................... 16
    3.2.3 Current API Combination ......................................................................................... 18
    3.2.4 INTERTWinE Ambition ............................................................................................. 18
    3.2.5 Benchmarks ............................................................................................................... 19
    3.2.6 Evaluation Plan .......................................................................................................... 19
    3.2.7 Preliminary Results .................................................................................................... 20
  3.3 TAU .................................................................................................................................. 23
    3.3.1 Introduction ............................................................................................................... 23
    3.3.2 Current API Combination ......................................................................................... 23
    3.3.3 INTERTWinE Ambition ............................................................................................. 23
    3.3.4 Evaluation Plan and Benchmarks ................................................................................ 24
  3.4 BAR: Barcelona Application Repository ........................................................................... 26
    3.4.1 Introduction ............................................................................................................... 26
    3.4.2 Benchmark Workload ............................................................................................... 27
    3.4.3 Current API Combination ......................................................................................... 28
    3.4.4 INTERTWinE Ambition ............................................................................................. 28
    3.4.5 Evaluation Plan and Benchmarks ................................................................................ 29
  3.5 Graph-BLAS ..................................................................................................................... 31
    3.5.1 Introduction ............................................................................................................... 31
    3.5.2 Graph-BLAS Workload ............................................................................................. 31
    3.5.3 Current API Combination ......................................................................................... 32
    3.5.4 INTERTWinE Ambition ............................................................................................. 32
    3.5.5 Benchmarks and Evaluation Plan ................................................................................ 33
  3.6 PLASMA: Parallel Linear Algebra Software for Multicore Architectures ........................ 34
    3.6.1 Introduction ............................................................................................................... 34
    3.6.2 Subset of PLASMA for the INTERTWinE project .................................................... 35
    3.6.3 Current API combination ........................................................................................... 36
    3.6.4 INTERTWinE Ambition ............................................................................................. 36
    3.6.5 Benchmarks ............................................................................................................... 37
    3.6.6 Evaluation Plan .......................................................................................................... 37
    3.6.7 Preliminary Results .................................................................................................... 38
  3.7 DPLASMA: Distributed Parallel Linear Algebra Software for Multicore Architectures .... 40
    3.7.1 Introduction ............................................................................................................... 40

Copyright INTERTWinE Consortium 2016
Index of Figures

Figure 1. Disclination network in a shear flow formed in liquid crystal simulation [12]..............10
Figure 2. Left: the lattice decomposed between Message Passing Interface (MPI) tasks. For clarity we show 2D decomposition of a 3D lattice but in practice we decompose in all three dimensions. Halo cells are added to each sub-domain (as shown on the upper right for a single slice) which store data retrieved from remote neighbors in the halo exchange. Lower right: the D3Q19 velocity .........................11
Figure 3. Intra-node performance coexisting MPI and OpenMP...........................................14
Figure 4. Ludwig weak scaling ...............................................................................................14
Figure 5. Ludwig strong scaling ............................................................................................15
Figure 6. iPIC3D simulation of plasma particles interacting with Earth magnetic field lines (in black). ......................................................................................................................16
Figure 7. 3D simulation box decomposed into four processes in a one-dimensional domain decomposition. ................................................................................................................17
Figure 8. Time-stepping of one component in recursive PIC algorithm. .................................17
Figure 9. PIC data movement at individual cell level. .............................................................18
Figure 10. Planned timeline for the iPIC3D software releases corresponding to the API combination. .......................................................................................................................19
Figure 11. Strong scaling test for overall simulation in the GASPI version of iPIC3D [23] ........................................................................................................................................21
Figure 12. Weak scaling test for overall simulation in the GASPI version of iPIC3D [23] ........................................................................................................................................21
Figure 13. Task dependency graph for the Cholesky decomposition benchmark. .................26
Figure 14. Blocked matrix multiply, using blocks A[i][k] and B[k][j] to compute C[i][j] ..........27
Figure 15. A N-Body simulation: different particles interact in a force system. .................27
Figure 16. Tentative timeline for the BAR-Benchmark releases according with each API combination. ..............................................................................................................................29
Figure 17. PLASMA software stack [PLASMA User Guide] ......................................................34
Figure 18. Performance benchmark for PLASMA ....................................................................38
Figure 19. Strong scaling benchmark for PLASMA .................................................................39
D5.1 REPORT ON INITIAL APPLICATION KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 20. Performance benchmark for DPLASMA.................................................42
Figure 21. Weak scaling benchmark for DPLASMA................................................43
Figure 22. Web interface to access the INTERTWinE internal GitLab repository. ....46
Figure 23. Web interface to access the INTERTWinE external GitHub repository...46
Figure 24. Web interface for the INTERTWinE CI tests with Jenkins. .......................47

Index of Tables

Table 1. Developing timeline for the Ludwig application......................................13
Table 2. Planned timeline for TAU-linear-solver-kernel API combinations..............24
Table 3. Planned timeline for TAU-memory-spaces-kernel API combinations...........24
Table 4. Sample test cases for evaluation and benchmarks.................................24
Table 5. Release timeline for the applications/kernels and their API combinations ....45
1 Executive Summary

This document presents an initial plan of the applications/kernels in combination with their initial evaluation and benchmark suite releases for use by WP3 and WP4. These initial releases of applications/kernels will be key to evaluate new or enhanced interoperability features that are made available in the respective runtime implementations.

For each application/kernel, the document offers a brief introduction, describes the current state for the use case from the point of view of API combination, confirms the ambition plans with respect to API combination within the scope of INTERTWinE, and reviews the benchmarks and evaluation plans. In the last section of the document, we offer a summary of software (benchmark) release plans, focusing on the API combinations.
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 one to address the challenging task of programming an Exascale system.

The project will take the applications and kernels provided by project partners and port them to appropriate combinations of APIs. For this purpose, the project will produce initial plans and releases of the applications/kernels for use by WP3 and WP4. When new or enhanced interoperability features are available in the respective runtime implementations, these advances will also be re-implemented and optimized to make use of these advances.

Proceeding in this manner, the project will complete 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 kernel and application optimizations are feed back to WP3 and WP4. The goal of this document is to formally define the initial plans for application/kernels provided by the partners, and present the evaluations and benchmark suite releases.

This document describes in Section 3 the use cases proposed by the project partners in the form of seven applications/kernels: Ludwig, iPIC3D, TAU, BAR, Graph-BLAS, PLASMA, and DPLASMA. For each one of these, the document 1) offers a brief introduction, 2) describes the current state for the use case from the point of view of API combination, 3) defines the ambition plans with respect to API combination within the scope of INTERTWinE, and 4) reviews the benchmarks and evaluation plans. Where possible, the document also characterizes the application/kernel workload and provides some preliminary results based on the current status of the use case.

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>DPLASMA</td>
<td>Distributed Parallel Linear Algebra Software for Multicore Architectures</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>
</tbody>
</table>
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</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>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 Applications/Kernels

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

3.1.1 Introduction
Ludwig is a general purpose parallel Lattice-Boltzmann code capable of simulating the hydrodynamics of complex fluids in 3D. The underlying hydrodynamic model is based on the Lattice-Boltzmann (LB) equation. This itself may be used to study simple (Newtonian) fluids in a number of different scenarios, including porous media and particle suspensions.

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 is a robust and portable code written in ANSI C. It can be used to perform serial and scalable parallel simulations of complex fluid systems based around hydrodynamics via the Lattice-Boltzmann method. Time evolution of modeled quantities takes place on a fixed regular discrete lattice. The preferred method of dealing with the corresponding order parameter equations is by using finite difference.

Users control the operation of the code via a plain text input file; output for various data are available. These data may be visualized using appropriate third-party software (e.g., Paraview).

![Figure 1. Disclination network in a shear flow formed in liquid crystal simulation [12]](image)

3.1.2 Ludwig workload
The LB approach makes use of a regular three-dimensional (3D) lattice (see Figure 2) with discrete spacing $\Delta r$. It also makes use of a discrete velocity space $c_i$, where the $c_i$ are chosen to capture the correct symmetries of the Navier–Stokes equations. A typical choice, used here, is the so-called D3Q19 basis in three dimensions where there is one velocity such that $c_0 \Delta t$ is zero, along with six extending to the nearest neighbor lattice sites, and twelve extending to the next-nearest neighbor sites ($\Delta t$ being the discrete time step). The fundamental object in LB is then the distribution function $f_i(r; t)$ whose moments are related to the local hydrodynamic quantities: the fluid density, momentum and stress. The time evolution of the distribution function is described by a discrete Boltzmann equation:

$$f_i(r + c_i \Delta t; t) - f_i(r; t) = L_{ij} f_j(r; t)$$

Two different stages are identified. Firstly, the right-hand side represents the action of a
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Collision operator $L_{ij}$, which is local to each lattice site and relaxes the distribution towards a local equilibrium at a rate ultimately related to the fluid viscosity. Secondly, the left-hand side represents a propagation step (or streaming step) in which each element $i$ of the distribution is displaced $c_i \Delta t$, that is, one lattice spacing in the appropriate direction per discrete time step [10].

Figure 2. Left: the lattice decomposed between Message Passing Interface (MPI) tasks. For clarity we show 2D decomposition of a 3D lattice but in practice we decompose in all three dimensions. Halo cells are added to each sub-domain (as shown on the upper right for a single slice) which store data retrieved from remote neighbors in the halo exchange. Lower right: the D3Q19 velocity.

To allow utilization of multi-node computing architectures, Ludwig is parallelized using domain decomposition and message-passing communications (MPI). The regular 3D decomposition is illustrated in Figure 2. Each local sub-domain is surrounded by a halo (or ghost) region. Elements of the distribution must be exchanged at the edges of the domains to facilitate the propagation. To achieve the full 3D halo exchange, the standard approach of shifting the relevant data in each coordinate direction in turn is adopted. This requires appropriate synchronization, that is, a receive in the first coordinate direction must be complete before a send in the second direction involving relevant data can take place, and so on. Note that only “outgoing” elements of the distribution need to be sent at each edge. For the D3Q19 model, this reduces the volume of data traffic from 19 to 5 (see Figure 2 lower-right) of the distribution velocity components per lattice site at each edge. In Ludwig, the necessary transfers are implemented using a vector of appropriately strided MPI datatypes for each direction.

It is important to understand the coordinate system used in the computation. This is fundamentally a regular, 3D Cartesian coordinate system. The coordinate system is centered around the (LB) lattice, with lattice spacing $\Delta x = \Delta y = \Delta z = 1$. We will refer, in general, to the lattice spacing as $\Delta x$ throughout, its generalization to three dimensions $x, y, z$ being understood. Lattice sites in the $X$ direction therefore have unit spacing and are located at $x = 1, 2 ... N_x$. The length of the system $L_x = N_x$ with the limits of the computational domain begin $x = 1/2$ and $x = L_x + 1/2$.

This allows us to specify a unit control volume centered on each lattice site $x_i$ being $i - 1/2$ to $i + 1/2$.

Information on the coordinate system, system size and so on is encapsulated in `coords.c`, which also deals with the regular domain decomposition in parallel.
Decompositions may be explicitly requested by the user in the input or computed by the code itself at run time. `coords.h` also provides basic functionality for message passing within a standard MPI Cartesian communicator, specification of periodic boundary conditions, and so on.

Three-dimensional fields are typically stored on the lattice, but are addressed in compressed one-dimensional format. This avoids use of multidimensional arrays in C. This addressing must take account of the width of the halo region required at the edge of each sub-domain required for exchanging information in the domain decomposition. The extent of the halo region varies depending on the application required, and is selected automatically at run time.

3.1.3 Current API combination

Ludwig currently has two different parallelization modes:

- MPI: The executable compiled and linked against the MPI library
- targetDP: Targets data parallel is an abstraction layer of the thread level parallelism which currently supports OpenMP or CUDA. It is currently under continuous development.

Therefore, Ludwig can be run in parallel using uniquely MPI or a combination of either MPI and OpenMP when using only CPUs or MPI and CUDA if we targeting GPUs.

3.1.4 INTERTWinE Ambition

Ludwig is currently a highly optimized application using MPI and OpenMP. However it is possible to investigate other approaches that could lead to an improvement of scalability in future Exascale platforms. Among all the possible options, the following are considered more suitable for Ludwig and other LB based applications.

**GASPI+MPI**

Ludwig currently uses MPI for message passing. However it is worth to explore what a PGAS model approach like GASPI could add. PGAS models and in particular, GASPI are well suited for small messages communications where it can outperform MPI. Hence, it would be interesting to investigate if some performance gain is achieved in those particular cases.

**GASPI+MPI+OpenMP/OmpSs**

We will also investigate the interoperability of GASPI+MPI+OpenMP/OmpSs to be able to exploit the intra-node task parallelization as well as the inter-node communication.

**MPI endpoints + OpenMP/OmpSs/StarPU**

Future implementations of MPI endpoints could help to improve parallelism in some sections of the code, for instance during the halo exchange.

In Table 1 we show the developing timeline in months of effort expected to achieve our ambition plan. Deliverables from WPs 3-4 will be integrated along the project creating a feedback loop between WPs 3, 4 and 5.

<table>
<thead>
<tr>
<th>Combinations of APIs for Ludwig</th>
<th>Months</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-3</td>
</tr>
<tr>
<td>MPI+GASPI</td>
<td></td>
</tr>
<tr>
<td>GASPI+MPI+OmpSs</td>
<td></td>
</tr>
<tr>
<td>GASPI/MPI+OpenMP 4.5</td>
<td></td>
</tr>
</tbody>
</table>
### 3.1.5 Benchmarks
Evaluation of performance is based on the simulation of a binary fluid system. A test case has been previously run on a serial fashion and numeric results stored as the references for testing correctness. Two data sets will be used to stress the different code sections of Ludwig:

- Binary fluid (D3Q19): A 3D binary fluid 19 velocity lattice simulation.
- Binary fluid (D3Q15): A 3D binary fluid 15 velocity lattice simulation.

The grid size present in these test cases can be easily increased to be able to scale Ludwig to large core counts.

### 3.1.6 Evaluation plan
Ludwig prints out statistics of some of the most important sections of the code. We will be looking very carefully to these statistics as they represent different code sections. Results from those sections will be analysed to understand performance improvements or deteriorations:

1. *Propagation kernel* consisting of a set of nested loops performing memory-to-memory copies;
2. *Collision kernel*, which has a strong degree of spatial locality and relies on basic add/multiply operations;
3. *Lattice halos* give information on the communication stages.

Furthermore, a large collection of correctness tests is included within the application package.

To prevent integration problems, Jenkins (a continuous integration tool) has been configured for automatic testing. A battery of correctness tests will be executed after software changes have been committed. This test suite is executed in three stages:

1. **Building**: After new changes have been committed, the application is built to ensure all components are still compilable.
2. **Deployment**: The test suite is deployed to a target platform.
3. **Execution**: A collection of tests are run employing the new code.

Tests will be executed on the ARCHER Cray XC30 supercomputer, the UK national supercomputing facility, although other platforms could also be targeted.

### 3.1.7 Preliminary Results
The current Ludwig implementation uses *targetDP* and MPI. This combination of APIs helps Ludwig to exploit parallelism of modern HPC architectures. Here we present our preliminary results of running Ludwig on different situations. These performance results have been measured on a Cray XC40 with compute nodes made of 2 Haswell 16-core Intel Xeon@2.3 GHz and 128 GB DDR4.

**Intra-node performance**
Since Ludwig is able to operate with MPI and OpenMP is important to understand how the application currently performs and what combination of OpenMP threads and MPI ranks is most appropriate for the given testing platform. Figure 3 shows the time spent of a simulation of size $128^3$ on a fully populated node. At this stage, it is expected that a pure MPI implementation would perform better than any other combination of OpenMP threads and MPI ranks. However there may be situations where a minor
number of MPI ranks would help to improve the scalability and performance of the application.

Figure 3. Intra-node performance coexisting MPI and OpenMP

**Inter-node performance**

We keep the problem size per node fixed that is, measure weak scaling, where the time taken per time step loop would stay constant in the ideal case. This is illustrated in Figure 4. Here can be seen that the CPU time scales almost perfectly for up to 2048 cores. This parallel efficiency will be slowly dropping as we increase the number of cores and nodes in the simulation.

Figure 4. Ludwig weak scaling

We also include strong scaling results, where the problem size is fixed whilst the number of nodes varies. We chose a $256^3$ simulation size in this case. Figure 5 shows how Ludwig exhibits strong scaling up to the largest node count, in the sense that the performance continues to improve. However, the parallel efficiency clearly reduces as the node count increases.
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 5. Ludwig strong scaling
3.2 iPIC3D: Implicit Particle-in-Cell Code for Space Weather Applications

3.2.1 Introduction
Space weather is the study of the processes originating in the sun and propagating through the solar system, with effects on people and technology in space and on the Earth, ranging from auroras in the polar regions to electromagnetic disturbances causing disruptive currents in infrastructure such as power and communication lines. KTH has implemented the massively parallel Particle-in-Cell code, iPIC3D as a C++ program using MPI and OpenMP [18]. It simulates the interaction between solar wind and the Earth magnetic field. Plasma particles from the solar wind are mimicked by computational particles. At each computational cycle, the velocity and location of each particle are updated, the current and charge density are interpolated to the mesh grid, and Maxwell’s equations are solved. The magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. One of the iPIC3D use cases, plasma particles interacting with the Earth’s magnetic field, is depicted in Figure 6.

The iPIC3D code is used in production by several groups in European Universities, such as KU-Leuven and the University of Pisa, all of which will benefit from the INTERTWinE results.

![Figure 6. iPIC3D simulation of plasma particles interacting with Earth magnetic field lines (in black).](image)

3.2.2 iPIC3D Workload
The workload of iPIC3D mainly depends on the size of the computational grid and the number of computational particles. The workflow of simulations by iPIC3D consists of approximately thousands of time steps, each of which carries out four major steps sequentially: interpolation of particles to grid, integration of Maxwell’s equations on the grid, interpolation of fields to particles, and integration of particle equation of motions.

In iPIC3D each particle is represented by a "struct" data type and all the particles by an array of these structures; both the field and a cell are also represented by a struct. The latter contains also a list or an array of all particles associated to it; all the cells form a grid. iPIC3D takes a user-defined 2D or 3D simulation box as the domain and decomposes the domain into a group of sub-domains. These sub-domains are divided among processors using a generic Cartesian virtual topology. Particles are split over the processors based on their location relative to the sub-domain. The computation workload and memory cost are therefore split over these processors. For example, in
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 7 a simulation box of 4x5x2 = 40 cells is decomposed and assigned to four processors. Each processor takes up a subdomain of 1x2x5 cells.

![Figure 7. 3D simulation box decomposed into four processes in a one-dimensional domain decomposition.](image)

There are mainly three scenarios where communication between adjacent processors is required: field interpolation to calculate charge density (rho) and current density (J) at each grid node, Maxwell field solver to calculate electric field E and magnetic field B at each grid node, and particle mover to move exiting particles from source processor to destination processors.

![Figure 8. Time-stepping of one component in recursive PIC algorithm.](image)

Figure 8 illustrates three time steps of one recursive component – the set of progressively smaller cubes. In the iPIC3D formulation, we apply the following principle: the 4D space is spanned by 3D data and simulation time steps are tiled. We also consider the scenario where particles move between cells.
3.2.3 Current API Combination

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. iPIC3D is used with HDF5, a data model, library, and file format for storing and managing data, which supports an unlimited variety of data types, and is designed for flexible and efficient I/O and for high volume and complex data. The iPIC3D MPI communication is dominated by synchronous point-to-point communication, occurring from communication of particles and ghost cells among neighbor processes, and by global reductions resulting from solving two linear systems every simulation time step.

3.2.4 INTERTWinE Ambition

Thanks to the efforts in the DEEP and EPiGRAM projects, iPIC3D is an early adopter of both MPI+OmpSs and MPI+GASPI programming models, and thus it can drive requirements in the early stages of the project [22][23]. In addition, iPIC3D supports the MPI+OpenMP programming model. Further developments of interest are to use MPI endpoints with OpenMP and MPI+GASPI with OpenMP tasks, OmpSs, or StarPU for node-level parallelism. Thus, starting from the early stage of the project, we focus on a combination of distributed and shared memory programming models, except for MPI+GASPI, which is counted as an intermediate phase. Hence, at first, we will work on an extension to MPI+OpenMP (MPI_THREAD_FUNNELED in the current version) by considering the MPI_THREAD_MULTIPLE model, where multiple threads may call MPI at once with no restrictions. Then, we will move towards task-based parallelism by implementing versions of iPIC3D with MPI/GASPI and OpenMP/OmpSs/StarPU.

We prioritize the combination of APIs as follows:

1. MPI + OpenMP
   a) Extension to MPI_THREAD_MULTIPLE hybrid model
   b) Use of MPI + OpenMP tasks (MPI_THREAD_MULTIPLE model)
2. MPI + OmpSs
3. GASPI + MPI
4. GASPI + OpenMP
5. GASPI + OmpSs
6. GASPI/MPI + StarPU
7. MPI endpoints + OpenMP
3.2.5 Benchmarks

One typical use case of iPIC3D is to simulate magnetic reconnection in the Earth’s magnetotail (GEM Modeling Challenge). A typical configuration of a large problem size uses one million grids and ten billion particles. On a Cray XC40 machine, such setup of simulation runs on 8,192 MPI processes for 24 hours and generates 25 TB of data.

In order to evaluate the performance of the iPIC3D code on a variety of target systems, we propose two benchmark inputs to be utilized with the iPIC3D code to solve two different problems:

- Particle-dominated regime simulation. In this test problem, we use a relatively large number of particles (up to 1,000 per cell) and the most computationally-expensive part of the iPIC3D code results in the particle mover. This test simulates kinetic turbulence in a two-dimensional space with 100 Whistler waves in initially uniform plasma.

- Field solver dominated regime simulation. In this test, we use a relatively small number of particles (27 per cell) and the most computationally-expensive part of the PIC code results in the Maxwell field solver. This test simulates magnetic reconnection in a three-dimensional space starting from Harris current sheet configuration.

As configuring, compiling, and running the iPIC3D code on several machines, followed by result verification and analysis take a lot of time and energy, and without a benchmarking environment all these procedures have to be carried out manually, we use Jenkins for automatic testing on a set of our benchmarks.

3.2.6 Evaluation Plan

During all of the stages of the development and the verification of interoperability issues, we will provide an automatic test suite, which will be run on a regular basis. This test suite will be composed of three components:

- Build test suite using cmake to test whether the iPIC3D code was built properly.
- Execution test suite to verify the iPIC3D execution as well as the correctness of its results.

In addition, we plan to add a performance test suite, which will include both weak and strong scaling test cases, as a separate test suite in order to detect possible performance regressions. iPIC3D is able to scale to very large number of nodes, e.g. 512K, delivering roughly 80% efficiency.

Tests will be executed on the Beskow Cray XC40 Supercomputer of the PDC Supercomputing Center at KTH and other platforms could also be targeted.

The number of iterations will be fixed to 20 in the linear solver to ensure a fair comparison although, in a real simulation, the number of iterations depends on the...
speed of convergence. In each test, the execution time will be normalized by the time spent in the initial iPIC3D implementation.

For the strong scaling tests, one-dimensional decomposition on X-axis will be used, resulting topology 15*1*1, 30*1*1, 60*1*1, 120*1*1, 240*1*1, etc. A fixed number of cells and particles will be used in each simulation.

For the weak scaling tests, one-dimensional decomposition on X-axis will be used as well (15*1*1, 30*1*1, 60*1*1, 120*1*1, 240*1*1, etc.). The simulation size will be decreased proportionally to the number of used processes.

3.2.7 Preliminary Results
The first MPI+GASPI implementation of iPIC3D was developed within the EPIGRAM project. In this pilot porting, two GASPI communication modes were tested, one was based on gaspi_read (so called "blocking" version) and one was based on gaspi_write ("non-blocking" version). The non-blocking GASPI communication in halo exchange, in the field solver, performed faster than the same implementation using gaspi_read in the blocking way.

The blocking GASPI reduction operation in the linear solver GMRES was also tested. The GASPI reduction operation performed faster with the same number of iterations in the linear solver. However, at that time the developers could not avoid using additional GASPI barrier calls in the linear solver, while the corresponding MPI reduction operation in the linear solver did not require a barrier. Thus, this was not fully a fair comparison, as it was necessary to ensure the interoperability of GASPI and MPI without additional MPI or GASPI barriers.

The GASPI implementation was tested in all three types of the iPIC3D communications (halo exchange in the field solver, reduction operation in the linear solvers, and particle communication). The strong scaling and weak scaling tests were performed. Figure 11 and Figure 12 show the performance results of the pure MPI implementation and hybrid implementation using both MPI and GASPI in a standard plasma simulation, obtained by the authors. The strong scaling test showed a relative (to 24 processes) parallel speedup of 3.78 with pure MPI implementation and a speedup of 3.80 with hybrid (MPI+GASPI) implementation on 480 processes. The hybrid implementation and pure MPI implementation showed very similar performance with differences ranging from 0.12% to 2.07%. The weak scaling test also showed very similar performance between the two implementations, with differences ranging from 0.05% to 5.32%. The experiments were carried out using up to 480 processes, as the initialization of GASPI processes requires a large value of ulimit on their system (in their test environment, the old Lindgren supercomputer at KTH, ulimit was equal to 8,000 and its value could be set only by system administrators of the supercomputer; with this ulimit value, the maximum number of GASPI processes that could be initialized was 480).

Within the DEEP project, OmpSs was integrated with the iPIC3D code, although there were additional changes in iPIC3D such as OpenMP parallelisation of particle processing and moments accumulation, inclusion of subroutines to guarantee data locality of dynamically allocated vectors and matrices, extensive use of intrinsic calls to load vectors into the Xeon Phi VPU registers, use of hints to the compiler for automatic vectorisation, or use of an array of structures for representation of particles with a following transpose on the fly to a structure of arrays to enable vectorisation.

The iPIC3D code was split in three computing phases and two so-called Cluster-Booster communication phases as it was shown that such a computing decomposition suits best the numerical methods built in the iPIC3D code.
The iPIC3D particle solver is deployed in the so-called Booster architecture (Xeon Phi) and contains two phases ("particle mover" phase and a "moment gathering" phase), and the iPIC3D field solver is deployed in the Cluster architecture (Xeon) and contains one phase. For testing purposes the executions with the particle solver launched on additional nodes of the Xeon Cluster were performed.

Once the particle solver was launched, it triggered the offloading of the field solver to the Xeon Cluster, performing a so-called reverse offload. Two methods were considered to perform the offloading of the field solver: using the OmpSs runtime extended during the DEEP project and using the MPI_Comm_Spawn call from MPI. OmpSs extended the functionalities of OpenMP, but the former was not fully compatible with the later. OmpSs did not support some of the pragmas introduced in the iPIC3D code. To use this offloading method, the developers commented out all the OpenMP pragmas aside from #pragma omp task and #pragma omp for. The code was then compiled with the Mercurium compiler. The offload was performed by a call to the field solver at the interior of a #pragma omp task device (MPI) call. OmpSs took this section of the code and offloaded the calculation to the Xeon processor. All the objects...
required by the field solver needed to be sent through the Cluster-Booster communicator from the Xeon Phi to the Xeon. However, OmpSs did not support the offload of C++ objects. A solution to this problem would be to serialize in the starting process and de-serialize in the spawned processes all the necessary objects, but their initial tests showed that this task was too complex. Instead, only the argv and argc parameters were offloaded to the Cluster, and each Xeon processor recreated the full initialization procedure itself. This procedure was tested multiple times but led to complications without clear benefits in this particular code division. Therefore, it was decided to continue the development of the code using the second offloading approach.

To compare the Xeon Phi platforms with classic Xeon-only clusters the developers performed the runs using three different processor allocations: 1. Offload from MIC to Xeon (m2x): the application was launched on the Xeon Phi coprocessors and the fields were offloaded to the Xeon processors, 2. Offload from Xeon to Xeon (x2x): the application was launched on the Xeon nodes and the field solver was offloaded to additional Xeon processors. 3. No offload: the code was not split in two branches. The code ran as a classical parallel application on the Xeon cluster using only one global communicator. This was the baseline for the benchmarking procedure.

The division Cluster-Booster did not affect the run time of the application when the offload was performed on the Xeon processors. However, when the code used the Xeon Phi coprocessors the application had a runtime 4 to 5 times slower than the case without offload. It is stated in the deliverable that the problem was not associated either with the Cluster-Booster division, nor with the communications, but with the introduction of the Xeon Phi coprocessors.
3.3 TAU

3.3.1 Introduction
TAU is a hybrid unstructured flow solver for compressible flow, based on the solution of the Navier-Stokes equations, and developed by DLR [11][21]. It is a world-wide recognized tool with an excellent reputation. TAU is routinely used in the German aircraft industry and at many universities, as well as within DLR and other European research centers.

Currently, development work is on a new software design, with one focus being to implement a flexible layout for future architectures. Another focus is the scalability of next generation implicit methods to be integrated in this new solver.

3.3.2 Current API Combination
The TAU solver is parallelised with MPI, using a cell vertex oriented decomposition of the grids. For typical industrial applications with grid sizes in range of 20 to 80 million points a parallel efficiency of more than 80% is reached on up to 1,000 to 2,000 cores, even when using a 3 or 4 level geometrical multigrid scheme. However, for larger numbers of processes, the scalability decreases, because the overlap region begins to dominate the number of points in the partition, which is especially true for the coarse multigrid levels. In order to improve this situation even further DLR worked on an OpenMP+MPI or GASPI parallelisation under the German national project GASPI [14].

The new flow solver development is multi-threaded within single domains and can use either MPI or GASPI for the network communication. The threading model in the new solver implemented so far is based on a static task distribution mechanism for the explicit solver. First scalability benchmarks available are very promising compared to TAU legacy results.

3.3.3 INTERTWinE Ambition
DLR’s ambition in this project is to investigate possible parallelisation strategies for the implementation of next generation implicit methods in the new solver framework. In particular, node local scalability of implicit methods and the possibility of using task based programming models for these methods will be evaluated. To this end it is planned to implement an appropriate prototypic evaluation kernel using OmpSs or OpenMP tasks in combination with MPI or GASPI for inter-node communication.

The first implementation extracts a kernel of the implicit method from the TAU code, which is based on pure MPI parallelisation. A task model will be introduced for the evaluation of OmpSs/OpenMP on the node level. An alternative evaluation of StarPU is planned only if unforeseen drawbacks of OmpSs necessitate a change of the API for continuing the project work. In a later development stage a kernel extension is planned for using GASPI alternatively to MPI communication for the evaluation of the scalability on the system level.

The main task of T-Systems SfR will be the development of a highly scalable threading and communication model as well as corresponding open-source kernels for the next generation CFD solver of the DLR. It is anticipated that the currently employed flat threading model of this next generation CFD solver will not be suitable for the upcoming next generation systems with deep and fragmented memory hierarchies (extensive cache hierarchies, 3D stacked memory or hybrid memory cube (HMC), DRAM and NVRAM). We believe that task graph models, such as OmpSs or StarPU, will be a very good match for these architectures. However, these task graph models ultimately will have to operate on a global system scale, rather than the currently predominantly used node local view. Due to its focus on asynchronous one-side dataflow notifications, the GASPI API will be an excellent match for this anticipated global extension of the task graph model of OmpSs. T-Systems SfR hence will
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

concentrate its efforts on the DLR next-generation solver hence on the co-design between application kernel and the integration of GASPI/OmpSs.

Summarizing the collaborative work of T-Systems SfR and DLR, the following API combinations, along with the corresponding software releases (see Table 2 and Table 3), are planned:

1) OmpSs/GASPI or OpenMP/GASPI (OmpSs on system level as well on node level)
2) OmpSs/MPI or OpenMP/MPI
3) StarPU/GASPI as fall-back position only

<table>
<thead>
<tr>
<th>DLR: linsolve-kernel</th>
<th>Month:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>MPI</td>
<td></td>
</tr>
<tr>
<td>MPI + OpenMP</td>
<td></td>
</tr>
<tr>
<td>MPI + OpenMP/OmpSs</td>
<td></td>
</tr>
<tr>
<td>GASPI + OpenMP/OmpSs</td>
<td></td>
</tr>
<tr>
<td>Evaluation Report</td>
<td></td>
</tr>
<tr>
<td>Final Report</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Planned timeline for TAU-linear-solver-kernel API combinations.

<table>
<thead>
<tr>
<th>T-Systems: memory-spaces-kernel</th>
<th>Month:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>GASPI/mpi + OpenMP</td>
<td></td>
</tr>
<tr>
<td>MPI Endpoints</td>
<td></td>
</tr>
<tr>
<td>GASPI/mpi + OmpSs</td>
<td></td>
</tr>
<tr>
<td>Evaluation Report</td>
<td></td>
</tr>
<tr>
<td>Final Report</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Planned timeline for TAU-memory-spaces-kernel API combinations.

3.3.4 Evaluation Plan and Benchmarks

When a first version of the evaluation kernels (see 3.3.3) is available efficiency and scalability of the task-based programming model will be evaluated using application test cases for relevant geometries and grid sizes typically used for TAU. Different grids with different grid sizes for such geometries are available to the public (see e.g. AIAA CFD Drag Prediction Workshop – NASA). We will select at least three different grids of different size (see Table 4) to use each of them for a series of strong scaling benchmarks for evaluation of the scalability of the application with the proposed API combinations.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Grid points</th>
<th>Volume elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>NACA64A010</td>
<td>20,000</td>
<td>16,000</td>
</tr>
<tr>
<td>DLR-F6</td>
<td>2,000,000</td>
<td>5,000,000</td>
</tr>
<tr>
<td>DLR-F6 DPWII</td>
<td>15,800,000</td>
<td>39,000,000</td>
</tr>
</tbody>
</table>

Table 4. Sample test cases for evaluation and benchmarks
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

During kernel development the continuous integration tool Jenkins will be used for automated build and correctness tests.

Project specific task-parallel patterns which arise in design and implementation of the TAU-kernels can be released under a BSD license provided the corresponding pattern is sufficiently generic to warrant broader interest.
3.4 BAR: Barcelona Application Repository

3.4.1 Introduction

BSC provides a set of benchmarks centralized in a public repository. The BSC Application Repository (BAR) contains applications developed internally at the center and also by third partners. Among this set of applications are **Cholesky** factorization, **matrix multiplication**, **heat** and the **N-Body** benchmark.

The **Cholesky** factorization [8][3] computes $A = LL'$, with $A$ being an $N \times N$ symmetric positive definite (SPD) matrix and $L$ lower-triangular. The implementation in OmpSs divides $A$ into blocks of $B \times B$ elements, which can be further subdivided into sub-blocks of $T \times T$ elements. Both single and double precision are supported.

The **matrix multiplication** package contains three implementations of the matrix multiplication algorithm. The first one implements the classic multiplication which divides $A$, $B$, $C$ in blocks elements (see Figure 14); a second implementation implements Strassen's algorithm, which performs 7 matrix multiplications and 18 matrix additions of submatrices to compute the matrix product; and the Winograd version implements the Winograd's variant of Strassen's algorithm, which requires 7 matrix multiplications and 15 matrix additions of submatrices to accomplish the matrix product. The latter two algorithms use Morton layout of $A$, $B$ and $C$.

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. The application generates a PPM image upon completion.

![Figure 13. Task dependency graph for the Cholesky decomposition benchmark.](image)

The **matrix multiplication** package contains three implementations of the matrix multiplication algorithm. The first one implements the classic multiplication which divides $A$, $B$, $C$ in blocks elements (see Figure 14); a second implementation implements Strassen's algorithm, which performs 7 matrix multiplications and 18 matrix additions of submatrices to compute the matrix product; and the Winograd version implements the Winograd's variant of Strassen's algorithm, which requires 7 matrix multiplications and 15 matrix additions of submatrices to accomplish the matrix product. The latter two algorithms use Morton layout of $A$, $B$ and $C$.

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. The application generates a PPM image upon completion.
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 14. Blocked matrix multiply, using blocks $A[i][k]$ and $B[k][j]$ to compute $C[i][j]$.

Figure 15. A N-Body simulation: different particles interact in a force system.

Finally the **N-Body** simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body. A familiar example is an astrophysical simulation in which each body represents a galaxy or an individual star, and the bodies attract each other through the gravitational force; or, as in Figure 15, where different bodies interact with each other in the presence of gravity. N-body simulation arises in many other computational science problems as well. For example, protein folding is studied using N-body simulation to calculate electrostatic and van der Waals forces. Turbulent fluid flow simulation and global illumination computation in computer graphics are other examples of problems that rely on N-body simulation.

Apart for the aforementioned benchmark the repository has other kernels that will be suitable to use as initial tests: N-Queens, Sparse LU, Perlin Noise, Conjugate Gradient (CG) or QR factorization are some of them.

### 3.4.2 Benchmark Workload

Each different benchmarks comes with its own list of command line options that allow to change the workload for a given execution. The nature of this set of kernels is to be used for testing purposes and, therefore, this feature in essential. Programmers may launch execution with a small workload to check correctness or medium/large workloads in order to test performance.

The Cholesky decomposition benchmark control the workload used in a given execution by means of the matrix size command line option.
The Heat diffusion benchmarks come with different input files in order to define the input size for a given execution. Users may change the input file by means of the command line option:

```
$.heat <input-file> [<result-file>]
```

Here, the input file is the output image size and can be one of the following (ordered from smaller to bigger):

- test-0128.dat
- test-0256.dat
- test-0512.dat
- test-1024.dat
- test-2048.dat

The matrix multiply kernel computes a blocked matrix multiplication and the workload can be set at command line using the –nb flag option. This option specifies the number of blocks for the input square matrices:

```
$.matmul –nb 12
```

The block size is predetermined by the kernel to 128x128 double precision floating points. Then, an input size of –nb 12 will give the following configuration:

```
:Matrix size (blocks): 12x12
:Block size: 128x128
:Matrix size (total): 1536x1536
```

Finally the N-body kernel uses two command line options to determine the workload. One sets the number of particles to be used, and a second one the maximum number of steps is going to execute before finalize:

```
$.nbody <num-particles> <steps> <verbosity>
```

Verification files are provided for the following configurations:

- 4096 particles – 10 steps
- 8192 particles – 10 steps
- 8192 particles – 20 steps

3.4.3 Current API Combination
All mentioned kernels are written in C and are parallelized with the OmpSs programming model. They are mainly based in the exploitation of task dependences and, although the main intent of these benchmarks is to be executed by an OmpSs runtime, they can be easily ported to OpenMP (tasking model).

3.4.4 INTERTWinE Ambition
The main purpose of the suite is to serve as examples/benchmarks to test the functionality of the different interoperability proposals of the project. They will be ideal examples when developing the resource manager component but they can also be used in evaluating other aspects of the project.
During the project it may be also interesting to port those simple benchmarks to other task-based runtime systems (e.g. StarPU). The Heat diffusion and N-Body kernels are also candidates to support MPI/GASPI as well and they can be used in combination with OmpSs. The N-Body kernel may also exploit easily three levels of HPC layer: use GASPI across cluster nodes, OmpSs in host architecture, and potentially use CUDA/OpenCL to exploit accelerator devices. The following list is ordered according with the project’s priorities with respect to the use of multiple APIs:

1. OmpSs + OpenMP (MKL) → Cholesky, Matrix Multiplication
2. OmpSs + CUDA/OpenCL → N-Body
3. OmpSs + GASPI → N-Body
4. StarPU + OpenMP (MKL) → Cholesky, Matrix Multiplication
5. StarPU + CUDA/OpenCL → N-Body
6. StarPU + GASPI → N-Body
7. OmpSs + MPI (end-points) → N-Body, Heat diffusion
8. OmpSs + GASPI + CUDA/OpenCL → N-Body

Figure 16 shows the proposed timeline to deliver each API combination. The expectation is to start porting our benchmarks to other programming models using (i) communication libraries to have versions running in clusters, (ii) accelerator programming models, to have versions running in heterogeneous environment, and (iii) verifying our use cases by replacing our original task-based programming model by a different one (e.g. OmpSs → StarPU). The tentative timeline also considers two optional experiments: extending the distributed version of the N-Body kernel (OmpSs + GASPI) in order to use accelerator programming models (CUDA and OpenCL). The final decision will depend on the project’s requirements.

In order to guarantee the correct behaviour of the current implemented version, as well as the future versions that still need to be implemented, we have created a set of tests already included in the repository. The verification mechanism has been automated using the Jenkins web-based system which perform nightly builds and execute this set of verification tests.

3.4.5 Evaluation Plan and Benchmarks

The nature of the application is to serve as a set of independent benchmarks to evaluate the features of OmpSs applications. In the context of this project we plan to use them as vehicle to test the new interoperability features specified in it. The evaluation plan is divided in different phases.
First we plan to port some of these benchmarks (Cholesky and Matrix Multiply) to use the MKL math library. MKL is implemented on top of the Intel OpenMP runtime system so we can study the interaction of its execution in terms of resource consumption. The evaluation of the performance in this phase will be based on strong scaling measurements, as the problem size should not be an issue. We will compare several combinations in the number of threads used by each runtime system (including oversubscription) but the main goal will be to better understand how the thread binding impacts the performance and how the different runtimes involved in the execution can coexist when sharing these resources.

Exploring the possibilities of using an accelerator programming model (CUDA/OpenCL) is the natural behaviour of OmpSs, but it is also a good opportunity to analyse the effect of using helper threads and their impact on performance. We plan to use N-Body as a use case to implement a task-based and an accelerator version. In these tests the performance will also be evaluated using the strong scaling methodology, and the main goal is to figure out how the helper threads used to offload the work to accelerator devices may impact the performance of worker threads running in the same architecture.

A third phase will validate previous analysis with the StarPU programming model. In this stage we plan to port Cholesky, Matrix Multiplication and N-Body to the StarPU programming model and verify the conclusions from previous steps.

In all the previous cases we will run in a single node environment and there is no expectation to scale up to a large amount of cores. The usability of these small benchmarks is expected to be higher than complex applications due the simplicity of the code. All the implementations consist of a few files, which make them ideal candidates.

The last phase will introduce MPI/GASPI libraries. In this phase we plan to port kernels to these programming models to study the effect of communication. N-Body will use GASPI and MPI (end-points) and the Heat diffusion kernel will be also be ported to use MPI (end-points) mechanisms. As one of the final objectives, we will analyse the interaction of GASPI using accelerator pinned memory as data targets.

In this last phase we plan to complete a more complex study and we will probably combine different performance measurements (strong/weak scaling, variation of the granularity in the parallelisation approach, etc.). As the effect of the communication will have an impact on performance, the problem size and granularities may also impact on it.

Network communication can play an important role in the application behaviour when increasing the number of cores. We cannot foresee the results of the execution but at this point we do not expect any limitation imposed by the benchmark implementations.
3.5 Graph-BLAS

3.5.1 Introduction

Computation with large-scale graphs (combinatorial computing or combinatorics) is crucial for modern Big Data analytics. While graph computations are often a source of poorly scalable parallel algorithms, due to their irregular nature and low computational intensity, many graph operations exhibit ample coarse-grained parallelism, which can be uncovered by exploiting the duality between graphs and sparse matrices [5][15].

For example, sparse matrix-matrix multiplication (SpMM) is connected to graph contraction, peer pressure clustering, breadth-first search from multiple-source vertices, recursive formulations of all-pairs shortest paths algorithms, matching algorithms, cycle detection, etc. Sparse matrix-vector multiplication (SpMV) can be applied to page ranking, breadth-first search (BFS), shortest path algorithms (Bellman-Ford), and generation of minimum spanning trees (Prim's algorithm). Finally, clustering, partitioning, community detection, and anomaly detection are all problems that require the solution of a (sparse) linear system, which in turn relies on SpMV.

The basic functionality necessary to deal with large-scale graphs is provided, among many other libraries, by the following packages:

- **Combinatorial BLAS**: Basic sparse linear algebra operations ([http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/](http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/)). This is an extensible graph library offering a small and powerful collection of sparse linear algebra primitives specifically targeting graph analytics. The software is released under the MIT license.

- **HLib**: Linear algebra operations on hierarchical matrices ([http://www.hlib.org](http://www.hlib.org)). This is a program library for hierarchical matrices that provides the functionality necessary to construct these structures (cluster trees, block cluster trees, low-rank matrices, etc.) along with discretization functions to fill the structures, arithmetic algorithms that perform approximate matrix operations, and several other auxiliary routines. HLib is free for non-commercial research and non-commercial teaching by users who are willing to sign a license agreement.

- **ILUPACK**: Solution of large-scale sparse linear systems via iterative Krylov subspace methods with preconditioning ([http://www.icm.tu-bs.de/~bolle/ilupack/](http://www.icm.tu-bs.de/~bolle/ilupack/)). ILUPACK implements a variety of multilevel ILU preconditioners for general real and complex matrices as well as real and complex symmetric (Hermitian) positive definite systems. ILUPACK relies on so-called inverse-based ILUs, a class of ILU decompositions that monitor the growth of the inverse triangular factors during the computation of the preconditioner. ILUPACK is freely available for scientific (non-commercial use), while users who want to integrate ILUPACK into commercial products need to negotiate a commercial license.

3.5.2 Graph-BLAS Workload

The basic kernels for the combinatorial BLAS, SpMV and SpMM, present a straightforward parallelization that decomposes the operation into multiple independent suboperations. In particular, by simply partitioning the first operand involved in the multiplication in both kernels into blocks of rows, and assigning each one of these row-blocks to be computed by a different resource (i.e., node/processor/core), all the suboperations can proceed in parallel. While other partitionings/distributions of the workload are possible, these require special care to deal with race conditions, though they may result in less communication and higher scalability. Thus, the preferred parallelization strategy depends on the test cases that are selected for the evaluation and the target architecture.

For HLib we will select the computation of the LU factorization of a hierarchical matrix, which is a fundamental kernel for hierarchical linear algebra upon which many other
operations, such as the solution of hierarchical linear systems, rely on. The workload for this particular factorization can be decomposed into a directed acyclic graph (DAG) that reflects the dependencies between suboperations as arcs between nodes (tasks). The number of nodes and dependency pattern of the graph, as well as the granularity of the tasks (computational cost), depends on the problem data, but can be adjusted to vary the amount of concurrency/dependencies in the operation.

The task-parallel versions of ILUPACK decompose the problem (graph associated with the sparse linear system) into a collection of tasks that form a DAG with a binary tree structure. The direction of the dependencies in the DAG differs for the computation of the preconditioner (bottom-up) and its application (both bottom-up or top-down, depending on the type of triangular system that is being solved). The number of tasks is usually adjusted to equal or double the number of computational resources.

### 3.5.3 Current API Combination

Except for the Combinatorial BLAS package, which was conceived from its beginning as a message-passing collection of routines, none of the previous libraries was originally designed with a parallel execution in mind. In spite of this, there exist prototype extensions of these libraries that address this problem, though they only target one parallel API only. In more detail:

- **Combinatorial BLAS** is an MPI-based parallel library featuring a reduced set of C++ sparse array-based primitives for graph analytics and data mining. The library is designed for distributed memory platforms, but also runs in multicore servers. The supported operations include, among others, SpMM and SpMV on a semiring defined by two generic operators (+,*).

- **HLib** is a library written in C that relies on BLAS and LAPACK (http://www.netlib.org) to perform lower-level dense algebraic operations and extract (limited levels of) parallelism for multicore servers. There exist a task-parallel implementation of these codes based on Intel TBB for multicore platforms [24] that is integrated into the commercial version of the HLib library (http://www.hlibpro.com).

- **ILUPACK** is a sequential package, with routines written in Fortran and C. During the last years, in collaboration with others, the group at UJI has developed several versions of ILUPACK for multicore platforms that exploit task-parallelism via OpenMP, OmpSs, or MPI; as well as data-parallel versions for NVIDIA CUDA graphics processing units (GPUs); see [2] for reference.

### 3.5.4 INTERTWinE Ambition

The codes in Combinatorial BLAS, HLib and ILUPACK present several technical difficulties in order to be leveraged in analysis of the API interoperability due to their design and license type. To overcome these problems, we will develop new, simplified versions of the codes that can be used in the framework of the INTERTWinE project, providing a close functionality while maintaining the parallel operation of the codes:

- **Combinatorial BLAS**. We will re-implement a few basic graph kernels such as SpMV and SpMM using algorithmic variations that are especially appropriate for computations with large-scale graphs [5].

- **HLib**: We will prepare basic matrix factorization routines, such as Cholesky or LU (Gaussian elimination), for hierarchical matrices that reflect the algorithmic approach in HLib [17].

- **ILUPACK**: We will implement a simplified ILU0 version of a preconditioned CG method underlying ILUPACK. We expect that our ILU0 solver will overcome some of the scalability problems that are inherent to ILUPACK and can, therefore, be leveraged to perform large-scale tests.

We plan to investigate the combinations of APIs in the following prioritized list:
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

1. MPI+OpenMP and MPI+OmpSs for the ILU0 preconditioned CG solver. This combination will be made available in M09.
2. OpenMP+MKL and OmpSs+MKL for matrix factorization routines involving hierarchical matrices. This combination will be made available in M16.
3. GASPI+OpenMP, MPI+OpenMP and MPI+OmpSs for SpMV and SpMM. This combination will be made available in M28.

3.5.5 Benchmarks and Evaluation Plan

For experimentation, we plan to use on synthetic data sets produced with the recursive matrix (R-MAT) and block two-level Erdos-Renyi (BTER) generators [7][16]. These tools produce graphs with properties of large-scale real-world applications.

In more detail, [7] describes R-MAT, a simple model, which can be used to rapidly generate realistic graphs, capturing the essence of each graph in only a few parameters. The model can trivially generate weighted, directed and bipartite graphs; and it subsumes the Erdos-Renyi model as a special case; it can match the power law behaviors, as well as the deviations from them. On the other side, the BTER model can be tuned to capture two fundamental properties: degree distribution and clustering coefficients. The latter is particularly important for reproducing graphs with community structure, such as social networks. [16] proposes a graph generator based on this model that can be used for benchmarking purposes and provide idealized degree distributions and clustering coefficient profiles that can be tuned for user specifications.

We will provide tests for all codes that can be used to easily verify the correctness of the results. In addition, we will include strong and weak scalability tests for the basic combinatorial BLAS kernels SpMV and SpMM. For the remaining cases, i.e. HLib and ILU0, we will provide tests that can be scaled to a size that fits a single server and a moderate-scale cluster, respectively.

Jenkins will be used to provide an automatic build-up of the library as well as test runs.
3.6 PLASMA: Parallel Linear Algebra Software for Multicore Architectures

3.6.1 Introduction

Many numerical simulations lead to solving systems of linear algebraic equations. The matrices in these problems can be sparse or dense, SPD or non-symmetric square, or even present a general-shape such as in linear least squares problems. Dense linear algebraic subroutines are basic building blocks of many of these algorithms, and libraries such as BLAS and LAPACK, which provide their efficient implementations, are among the most widely used software packages.

The PLASMA library [20] provides utilities to solve systems of linear equations with SPD as well as general dense matrices, and linear least squares problems. The main purpose of PLASMA is to address the shortcomings of LAPACK on multicore processors, multi-socket systems of multicore processors and recent manycore processors with shared memory. PLASMA also allows solving eigenvalue and singular value problems. For solving systems of equations, PLASMA uses Cholesky, QR, and LU factorizations. It supports both real and complex arithmetic in single and double precision.

Three distinctive features that allow PLASMA to achieve higher performance compared to LAPACK are: (i) the implementation of tile algorithms, (ii) the application of tile data layout and (iii) the use of dynamic scheduling. PLASMA represents matrices as collections of tiles. A tile is a square block of an original matrix. For example, a matrix of size 1000 x 1000 elements can be represented as a collection of 4 x 4 tiles, each of size 250 x 250. It is preferred to have small tile sizes to ensure that tiles entering a basic operation, such as the matrix-matrix multiplication, fit entirely into the cache memory associated with one computation core. The computation is represented as a DAG, where nodes of the graph are computational tasks and edges are data dependencies [9][6].

PLASMA is a library implemented in C. The interface and naming conventions of PLASMA are derived from LAPACK, and can be called from many languages. The software stack underlying PLASMA consists of QUARK, BLAS, CBLAS, LAPACK, and NetLib LAPACK C wrapper (see Figure 17). BLAS and LAPACK routines are used as the “kernels” executed sequentially on the tiles within the tile-based algorithms.

![Figure 17. PLASMA software stack](image)
3.6.2 Subset of PLASMA for the INTERTWinE project
PLASMA is a large project, and in order to evaluate developments in WP3 and WP4, it is necessary to define a suitable subset of functionalities we will focus on within INTERTWinE. This subset will be based on solving systems of linear equations. A factorization of the dense matrix is the critical algorithm in this context. In particular, the Cholesky factorization is used for SPD matrices, the LU factorization with partial (row) pivoting is used in the case of general matrices, and the QR factorization based on Householder reflections is used for matrices of general shape and full rank. In addition, different runtimes will be tested only in real double precision (64-bit floating point arithmetic). Inheriting conventions of the LAPACK library, names of these functions start with an initial letter ‘d’.

3.6.2.1 Solving systems with SPD matrix (dposv)
The algorithm is based on the Cholesky decomposition. In particular, the linear system

\[ Au = b \]

is to be solved with \( A \) being a dense square symmetric positive definite matrix, \( b \) the given right-hand side vector, and \( u \) the unknown solution vector. The algorithm proceeds as follows

1. Cholesky factorization of the matrix into triangular factors

\[ A = LL^T, \]

where \( L \) is a lower-triangular matrix. This step corresponds, in the LAPACK naming style, to the operation \texttt{dpotrf}.

2. Substitute the factorization and solve system with a triangular matrix

\[ Lv = b. \]

This step corresponds to the \texttt{dtrsm} operation from BLAS.

3. Find the solution by solving

\[ L^Tu = v, \]

which corresponds to another BLAS \texttt{dtrsm} operation, with a transpose flag.

3.6.2.2 Solving least-squares problems (dgels)
The algorithm is based on the QR factorization of a matrix \( A \), which is defined for any shape of the matrix. The algorithm proceeds as

1. QR-factorization of matrix \( A \),

\[ A = QR \]

Here \( Q \) is an orthogonal matrix and \( R \) is an upper triangular matrix. As Householder reflections are used, matrix \( Q \) is not constructed explicitly and only the reflectors are stored. This operation corresponds to \texttt{dgeqrf} in LAPACK.

2. Application of the transpose of \( Q \) on the right-hand side vector,

\[ c = QTb \]

This operation is performed by the \texttt{dormqr} routine.

3. Solving the transformed system of linear equations

\[ Ru = c \]

by back-substitution. This corresponds again to a BLAS \texttt{dtrsm} operation.

3.6.2.3 Solving systems with general square matrices (dgesv)
This algorithm is based on the LU factorization of a matrix \( A \) with partial pivoting to improve numerical stability. In particular

1. Factorize matrix \( A \) with matrix of row permutations \( P \) as

\[ PA = LU \]
where $L$ is a unit lower triangular matrix and $U$ is an upper triangular matrix. This corresponds to LAPACK dgetrf function.

2. Apply permutations $P$ to the right-hand side vector
   $$d = Pb.$$ This corresponds to the dlaswp function from LAPACK.

3. Substitute the factorization and solve systems with triangular matrices
   $$Lv = d.$$ This step corresponds to the dtrsm operation from BLAS.

4. Find the solution by solving
   $$Uu = v,$$ which corresponds to another BLAS dtrsm operation.

The following table presents a summary of PLASMA functions selected for regular testing of developments of INTERTWinE as well as for subsequent performance evaluations:

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dposv</td>
<td>Solving linear systems with SPD matrices</td>
<td>The algorithm is based on Cholesky factorization of the matrix (dpotrf) and two back-substitutions with a triangular matrix (dtrsm).</td>
</tr>
<tr>
<td>Dgels</td>
<td>Solving least-squares problems</td>
<td>The algorithm is based on QR factorization of the matrix (dgeqrf), application of the transpose of $Q$ to the right-hand side vector (dormqr), and back-substitution with the triangular matrix $R$ (dtrsm).</td>
</tr>
<tr>
<td>Dgesv</td>
<td>Solving systems with general square matrices</td>
<td>The algorithm is based on LU factorization with partial pivoting (dgetrf), permuting right-hand side vector by the given permutation matrix (dlaswp), and two back-substitutions with a triangular matrix (dtrsm).</td>
</tr>
</tbody>
</table>

### 3.6.3 Current API combination

The dynamic scheduling of tasks in PLASMA, as for the version 2.8.0 of the library, is based on the QUARK (QUEuing And Runtime system for Kernels). This version will be used as the starting point for a version based on top of the task dependent functionality of OpenMP 4.0 for dynamic scheduling, which is currently under development and should be finished by the end of 2016.

### 3.6.4 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 4.0. The repository of the project is at [https://bitbucket.org/icl/plasma](https://bitbucket.org/icl/plasma)

After finishing this transition, the subset of routines for solving systems of linear equations (dposv, dgels, and dgesv) will be also ported to OmpSs and 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.

A tentative timeline for these studies is the following:

1. PLASMA + OpenMP (M12)
2. PLASMA + OmpSs (M15)
3. PLASMA + StarPU (M18)
4. PLASMA performance in different runtimes (M21)
5. Coexistence of PLASMA with OpenMP, OmpSs, and StarPU (M36)

3.6.5 Benchmarks
PLASMA distribution contains a large set of routines for testing. These tests use randomly generated matrices and they evaluate the correctness of the results by suitable normalized thresholds on the relative norm of the residual. This methodology is well established already from LAPACK and it is based on rigorous theoretical considerations about rounding errors developed in the frame of numerical analysis [13]. These tests are compiled as an integral part of each build of PLASMA and provide a convenient feedback for the correctness of the underlying routines.

For the purpose of the INTERTWinE project, a python script has been devised which executes the target subset of routines. The tests in PLASMA include the high-level routines dposv, dgels, and dgesv, as well as their lower-level building blocks (dportf, dgeqrf, dgetrf, dtrem, dormqr).

Building of the library as well as running the tests has been automated using the Jenkins web-based system, with nightly builds and execution of these tests.

To evaluate the interoperability, dedicated benchmarks targeted at combining PLASMA with different runtime systems in applications will be devised. For example, calling a PLASMA function from each thread of an OpenMP parallel block and/or from each MPI process will be studied.

Apart from small correctness tests, PLASMA already contains a large number of timing routines for evaluating performance. Benchmarking will be based on these programs for testing dposv, dgesv, and dgels routines. In particular, we define the following benchmarks relevant for the developments in WP4.

3.6.5.1 Performance benchmark
A common benchmark to be used also in the scope of INTERTWinE evaluates the performance of the routines in dependence on the dimension of the matrix. The performance is obtained as the theoretical number of operations of the standard (i.e. LAPACK) version of the algorithm divided by the time the routine requires for its completion. These data provide a basis for the plots of performance in Gflops/s with respect to matrix dimension (see Figure 18).

3.6.5.2 Scalability benchmark
Another relevant benchmark for comparing the efficiency of the scheduling strategies of the different runtime systems performs a strong scaling test. In this benchmark, the size of the matrix is fixed, as well as the size of tiles in the matrix. The number of cores is increased from 1 to the number of available cores. One can see the deviation of the parallel algorithms from the ideal performance given by a multiple of number of cores and theoretical performance of a single core (see Figure 19).

3.6.6 Evaluation Plan
The performance and scalability benchmarks defined in the previous section will be evaluated for OpenMP, OmpSs, and StarPU, once the implementation of the subset of functions from PLASMA is available. They will be compared also with the current version of PLASMA based on QUARK.
Performance in Gflop/s will be also the basic measure for evaluating the interoperability and coexistence of the runtimes.

### 3.6.7 Preliminary Results

As a starting point, the proposed benchmarks have been run with the current version of PLASMA (2.8.0) based on the QUARK runtime system. The node used for testing contains two Intel Xeon Haswell processors, with 12 cores each, i.e. 24 cores in total. Theoretical peak performance of the node is 960 Gflop/s. The Intel C compiler was used and BLAS and LAPACK functions are those provided by MKL.

First, the performance benchmark was run for the three algorithms. Results are presented in Figure 18. All problems were run using 24 threads, with tile size 250 and inner block size 50. Average value from 2 runs is used.

![Figure 18. Performance benchmark for PLASMA](image)

The performance plots in Figure 18 present a typical behaviour, where, for the smaller matrix sizes, the performance is limited by the speed of fetching data from the main memory. For matrix size at around 7000, the performance curves flatten, indicating saturation of the computing units, and the computations become bounded by the computing speed of the processors. For this setup, dposv reaches about 60\% of the theoretical peak performance, while dgels and dgesv reach around 50\% of this value.

The second benchmark evaluates strong scalability of these algorithms. Based on the previous benchmark, matrix size 12,000 was chosen for this experiment, as the performance already reaches the peak for all tested algorithms. Tiles and internal blocking are of the same size as in the previous test, 250x250 and 50, respectively.

In the strong scaling test, the matrix size is fixed as well as size of the tiles, and only the number of threads is varied from 1 to 24 (the full node).

In Figure 19, dependence of performance on the number of threads is plotted. The dashed line shows the theoretical peak performance, and deviation of the algorithms from this ideal scaling can be assessed from these plots.
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

Figure 19. Strong scaling benchmark for PLASMA

One can see that the algorithms almost reach the theoretical peak performance for a single core, while they gradually lose scalability, finishing at about 60% of the peak for dposv and around 50% for dgels and dgesv.
3.7 DPLASMA: Distributed Parallel Linear Algebra Software for Multicore Architectures

3.7.1 Introduction
DPLASMA (Distributed Parallel Linear Algebra Software for Multicore Architectures) [1] is a task-based implementation of dense linear algebra routines, suitable for distributed heterogeneous systems. DPLASMA is based on the state-of-the-art PaRSEC runtime [19].

PaRSEC is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. An algorithm is expressed as a DAG of tasks with labelled edges designating data dependencies. PaRSEC efficiently handles data movements by scheduling the data transfers (between nodes as well as to and from storage entities) in order to maximize the overlap between the computation and communications, while seamlessly integrating available accelerators to the application, by managing all data transfers, coherency and execution of tasks.

DPLASMA implements some of the tile algorithms of PLASMA (described in the previous section) in the JDF format, which is one of the high-level languages for expressing DAG-based computations for PaRSEC. From a JDF file, the source-to-source compiler of PaRSEC generates a source file in C, which is then compiled using a standard C compiler with the MPI support [4].

The software stack underlying DPLASMA consists of PaRSEC, PLASMA, BLAS, CBLAS, LAPACK and NetLib LAPACK C wrapper.

3.7.2 Subset of DPLASMA for the INTERTWinE project
Similarly to the approach to PLASMA, we also use only a subset of functions of DPLASMA for evaluating developments of INTERTWinE. The primary target will be testing developments of the underlying PaRSEC runtime. Again, we will focus on solving systems of linear equations in real double precision (64 bit floating point arithmetic). Details of the algorithms are described in the section about PLASMA.

3.7.2.1 Solving systems with SPD matrix (dposv)
The algorithm is based on the Cholesky factorization (dpotrf), followed by two triangular solves (dtrsm).

3.7.2.2 Solving least-squares problems (dgels)
The algorithm is based on the QR factorization (dgeqrf), followed by application of the Q matrix to the right-hand side vector (dormqr) and a triangular solve (dtrsm).

3.7.2.3 Solving systems with general square matrices (dgesv)
This algorithm is based on the LU factorization with partial pivoting (dgetrf). Applications of row permutations (dlaswp) and two triangular solves (dtrsm) follow.

3.7.3 Current API Combination
DPLASMA is implemented on top of the PaRSEC runtime. Although PLASMA is a dependency of DPLASMA, only the sequential core BLAS algorithms for operations on tiles from PLASMA are used and no interoperability with QUARK is needed. Since the underlying data interchange in PaRSEC is based on MPI, DPLASMA needs to be compiled with a compiler with MPI support.

3.7.4 INTERTWinE Ambition
The team at the University of Manchester collaborates closely with the Innovative Computing Laboratory at UTK, where both PaRSEC and DPLASMA have been developed. Since DPLASMA is based on the data flow description of PaRSEC, it will be mainly used for testing developments within this runtime.
As a starting point, coexistence of DPLASMA and PaRSEC in distributed memory applications based on MPI will be studied.

Next, interoperability of PaRSEC with shared memory task-based models on nodes will be tested. The first focus will be on PaRSEC+OpenMP combination.

Finally, coexistence of DPLASMA and PaRSEC with OmpSs and StarPU will be studied and evaluated.

The proposed timeline for these developments is the following:

1. DPLASMA/PaRSEC + MPI (M12)
2. DPLASMA/PaRSEC + OpenMP (M15)
3. DPLASMA/PaRSEC + OmpSs (M18)
4. DPLASMA/PaRSEC + StarPU (M21)
5. Coexistence of DPLASMA/PaRSEC in applications with MPI, OpenMP, OmpSs, and StarPU (M36)

3.7.5 Benchmarks

DPLASMA distribution contains a large set of routines for testing. These tests use randomly generated matrices and they evaluate the correctness of the results by suitable normalized thresholds on relative norm of the residual, similarly to PLASMA. These tests are compiled as an integral part of each build of DPLASMA, and provide a convenient feedback for the correctness of the underlying routines.

For the purpose of the INTERTWinE project, a python script has been devised which executes the target subset of subroutines. The tests in DPLASMA include the lower-level building blocks of the solve routines, i.e. dportf, dgeqrf, dgetrf, dtrem, dormqr.

Building of the library as well as running the tests has been automated using the Jenkins web-based system, with nightly builds and execution of these tests.

To evaluate the interoperability, dedicated benchmarks targeted at combining DPLASMA/PaRSEC with MPI and shared memory runtime systems in applications will be devised. As an example, shared memory PLASMA routines will be used on tiles of DPLASMA in place of the serial core BLAS routines.

For larger performance tests, we will use two benchmarks with similar functionality as those proposed for PLASMA, however, this time in a distributed memory environment.

3.7.5.1 Performance benchmark

A common dense linear algebra benchmark to be used also for DPLASMA evaluates the performance of the routines in dependence on the dimension of the matrix. The performance is obtained as the theoretical number of operations of the standard (i.e. LAPACK) version of the algorithm divided by the time the routine requires for its completion. These data provide a basis for the plots of performance in Gflops/s with respect to matrix dimension (see Figure 20).

3.7.5.2 Scalability benchmark

Another relevant benchmark for a distributed memory environment is a weak scaling test, in which the size of the local submatrix on each computational node is fixed, as well as the size of tiles in the matrix. The number of cores is increased proportionally to the matrix size, and the performance is monitored. One can evaluate the deviation of the parallel algorithms from the ideal performance given as a product of the number of cores and performance of a single core (see Figure 21).

3.7.6 Evaluation Plan

The performance and scalability benchmarks defined in the previous section will be regularly run for DPLASMA, and they will serve for evaluation of developments within the underlying PaRSEC runtime performed in WP4 and for studying the interoperability
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

of PaRSEC with shared-memory runtime systems. Results will be compared against the current version of PaRSEC and DPLASMA.

Benchmarks will use the Cholesky, QR, and LU factorization functions (dpotrf, dgeqrf, and dgetrf), which are at the heart of the three target solution algorithms.

3.7.7 Preliminary Results

As a starting point, the proposed benchmarks have been run with the current version of DPLASMA and PaRSEC (Git commit fcf8d761063c2770744243850162616358d6c32c in the INTERTWinE-WP5 repository). The functions of DPLASMA were run on a distributed memory machine with nodes composed of two 12-core Intel Xeon Haswell processors. Theoretical peak performance of a node is 960 Gflop/s. Intel C compiler and MPI library were used and BLAS and LAPACK functions are provided by the MKL.

First, the performance benchmark was run for the three factorizations. Results are presented in Figure 20. All problems were run using 4 nodes, i.e. 96 cores, with tile size 250 and inner block size 50. The runs were using hybrid computation, with 4 MPI processes (one per each node).

![Figure 20. Performance benchmark for DPLASMA](image)

The performance plots in Figure 20 show that for the smaller matrix sizes, the performance is limited by the speed of fetching data from the main memory and communication among processes. For matrix size at around 20000, the performance curves flatten. For this setup, dpotrf reaches about 60% of the theoretical peak performance, while dgeqrf about 50%, and dgetrf around 40% of this value.

The second benchmark evaluates weak scalability of these factorizations. Matrix size 20000 and 4 nodes were chosen as the reference, since performance already reached the peak for all tested algorithms. Tile size and internal blocking are the same as in the previous test, 250 x 250 and 50, respectively.

In the weak scaling test, the local matrix size per node is fixed as well as the size of the tiles, and the global matrix size and the number of threads vary. In our case, the largest problem used 16 nodes (i.e. 384 cores) and the dimension of the corresponding matrix was 40000.

Figure 21 presents the dependence of performance on the number of CPU cores. Dashed line shows the theoretical peak performance, and deviation of the algorithms from this ideal scaling can be assessed.
One can see that for this setup, the factorizations slightly deviate from the ideal scaling, especially for more than 192 cores. However, these are preliminary results, and better results can be expected after tuning the algorithm parameters, as indicated by results e.g. in [4].
## 4 Software Release Plans

This section summarizes the project timeline from the perspective of software releases for each application/kernel and its API combination (see Table 5).

<table>
<thead>
<tr>
<th>Application/Kernel</th>
<th>API combinations</th>
<th>Ready for month</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Ludwig</strong></td>
<td>1. MPI+GASPI</td>
<td>1. M10-12</td>
</tr>
<tr>
<td></td>
<td>2. GASPI+MPI+OmpSs</td>
<td>2. M13-15</td>
</tr>
<tr>
<td></td>
<td>3. GASPI/MPI+OpenMP 4.5</td>
<td>3. M19-21</td>
</tr>
<tr>
<td></td>
<td>4. MPI Endpoints+OmpSs/StarPU/OpenMP</td>
<td>4. M22-24</td>
</tr>
<tr>
<td><strong>2 iPIC3D</strong></td>
<td>1. MPI+OpenMP</td>
<td>1. M9-10</td>
</tr>
<tr>
<td></td>
<td>2. MPI+OmpSs</td>
<td>2. M12</td>
</tr>
<tr>
<td></td>
<td>3. GASPI+MPI</td>
<td>3. M15</td>
</tr>
<tr>
<td></td>
<td>4. GASPI+OpenMP</td>
<td>4. M15</td>
</tr>
<tr>
<td></td>
<td>5. GASPI+OmpSs</td>
<td>5. M18</td>
</tr>
<tr>
<td></td>
<td>6. GASPI/MPI+StarPU</td>
<td>6. M21</td>
</tr>
<tr>
<td></td>
<td>7. MPI Endpoints+OpenMP</td>
<td>7. M24</td>
</tr>
<tr>
<td><strong>3 TAU</strong></td>
<td>Linolvable kernel:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. MPI</td>
<td>1. M6</td>
</tr>
<tr>
<td></td>
<td>2. MPI+OpenMP</td>
<td>2. M9</td>
</tr>
<tr>
<td></td>
<td>3. MPI+OpenMP/OmpSs</td>
<td>3. M12</td>
</tr>
<tr>
<td></td>
<td>4. GASPI+OpenMP/OmpSs</td>
<td>4. M18</td>
</tr>
<tr>
<td></td>
<td>Memory-spaces kernel:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. GASPI/MPI+OpenMP</td>
<td>5. M9</td>
</tr>
<tr>
<td></td>
<td>6. MPI Endpoints</td>
<td>6. M18</td>
</tr>
<tr>
<td></td>
<td>7. GASPI/MPI+OmpSs</td>
<td>7. M18</td>
</tr>
<tr>
<td><strong>4 BAR</strong></td>
<td>Matrix multiplication</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. OmpSs+OpenMP(MKL)</td>
<td>1. M9</td>
</tr>
<tr>
<td></td>
<td>2. StarPU+OpenMP(MKL)</td>
<td>2. M12</td>
</tr>
<tr>
<td></td>
<td>Cholesky</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. StarPU+OpenMP(MKL)</td>
<td>3. M12</td>
</tr>
<tr>
<td></td>
<td>N-Body</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4. MPI+OmpSs</td>
<td>4. M9</td>
</tr>
<tr>
<td></td>
<td>5. OmpSs+CUDA/OpenCL</td>
<td>5. M18</td>
</tr>
<tr>
<td></td>
<td>7. StarPU+CUDA/OpenCL</td>
<td>7. M21</td>
</tr>
<tr>
<td></td>
<td>8. StarPU+GASPI</td>
<td>8. M24</td>
</tr>
<tr>
<td></td>
<td>9. MPI Endpoints+OmpSs</td>
<td>9. M28</td>
</tr>
<tr>
<td></td>
<td>Heat diffusion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10. MPI Endpoints+OmpSs</td>
<td>10. M28</td>
</tr>
<tr>
<td><strong>5 Graph BLAS</strong></td>
<td>ILU0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. MPI+OpenMP</td>
<td>1. M9</td>
</tr>
<tr>
<td></td>
<td>2. MPI+OmpSs</td>
<td>2. M9</td>
</tr>
<tr>
<td></td>
<td>Hierarchical matrix factorization</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. OpenMP+MKL</td>
<td>3. M16</td>
</tr>
<tr>
<td></td>
<td>4. OmpSs+MKL</td>
<td>4. M16</td>
</tr>
</tbody>
</table>
### Table 5. Release timeline for the applications/kernels and their API combinations

<table>
<thead>
<tr>
<th>Category</th>
<th>Applications/Kernels</th>
<th>API Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SpMV, SpMM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5. GASPI+OpenMP</td>
<td>5. M28</td>
</tr>
<tr>
<td></td>
<td>6. MPI+OpenMP</td>
<td>6. M28</td>
</tr>
<tr>
<td></td>
<td>7. MPI+OmpSs</td>
<td>7. M28</td>
</tr>
<tr>
<td>6</td>
<td>PLASMA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. PLASMA+OpenMP</td>
<td>1. M12</td>
</tr>
<tr>
<td></td>
<td>2. PLASMA+OmpSs</td>
<td>2. M15</td>
</tr>
<tr>
<td></td>
<td>3. PLASMA+StarPU</td>
<td>3. M18</td>
</tr>
<tr>
<td>7</td>
<td>DPLASMA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1. DPLASMA/PaRSEC+MPI</td>
<td>1. M12</td>
</tr>
<tr>
<td></td>
<td>2. DPLASMA/PaRSEC+OpenMP</td>
<td>2. M15</td>
</tr>
<tr>
<td></td>
<td>3. DPLASMA/PaRSEC+OmpSs</td>
<td>3. M18</td>
</tr>
<tr>
<td></td>
<td>4. DPLASMA/PaRSEC+StarPU</td>
<td>4. M21</td>
</tr>
</tbody>
</table>
5 Git and Jenkins Setups

5.1 Git repositories

In the frame of MS15 the project members have prepared a structure for the project shared Git software repositories composed of two main repositories: project internal Git repository and project public Git repository.

**Project internal Git repository (GitLab)**

This Git repository is hosted at the UEDIN internal GitLab server accessible at https://git.ph.ed.ac.uk/icebaman/intertwine-WP5. To access to this repository, all non-UEDIN members have an active visitor account created by the IT support team of UEDIN.

![Figure 22. Web interface to access the INTERTWinE internal GitLab repository.](image)

**Project external Git repository (GitHub)**

By PM18 (for MS17) INTERTWinE will create project public external Git repository that will be updated with new versions of software releases as they are produced. The use of this repository will allow a smooth transition from a private to a public repository.

![Figure 23. Web interface to access the INTERTWinE external GitHub repository.](image)

5.2 Jenkins setup

A Linux Virtual Machine with Jenkins has been deployed at an internal server with a public IP. This way, all project members can access the server and manage their application/kernel specific continuous integration (CI) tests. For all applications/kernels, the CI test suite consists of the build and correctness tests.
D5.1 REPORT ON INITIAL APPLICATION/KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES

The virtual machine (VM) contains four build executors, which allow the compilation of multiples applications/kernels at a given time. The Jenkins server can be reached at https://intertwine-ci.epcc.ed.ac.uk and only INTERTWinE members can access and manage it.

In order to avoid checking out of sections of the code by other members, the Jenkins Git plugin has been configured to use sparse checkout.

Figure 24. Web interface for the INTERTWinE CI tests with Jenkins.
6 References


D5.1 REPORT ON INITIAL APPLICATION KERNEL PLANS, EVALUATIONS AND BENCHMARK SUITE RELEASES


