D3.2
Runtime systems (OmpSs, OpenStream) and communication libraries (GPI, MPI): Advanced implementation customized for ExaNoDe architecture, interconnect, operating system

Workpackage:	3	Enablement of Software Compute Node
Author(s):	Valeria Bartsch, Carsten Lojewski	FHG
	Antoniu Pop	UOM
	Paul Carpenter, Babis Chalios, Antonio J. Peña, Kyunghun Kim	BSC
	Andrea Bartolini, Francesco Conti	ETHZ
Authorized by	Paul Carpenter	BSC
Reviewer	Paul Carpenter	BSC
Reviewer	Giuliano Taffoni	INAF (ExaNeSt WP2 Applications lead)
Reviewer	Manolis Marazakis	FORTH
Dissemination Level	Public (PU)	

Date	Author	Comments	Version	Status
2017-07-18	Antonio Pena	Initial MPI Version added	V0.1	Draft
2017-07-21	Carsten Lojewski	GPI Version added	V0.2	Draft
2017-08-28	Babis Chalios, Antonio Pop	OmpSs, OpenStream and parallel runtime support section added	V0.3	Draft
2017-08-30	Antonio Pena	Updated MPI version, including measurements	V0.4	Draft

2017-08-31	Andrea Bartolini	Power Management added	V0.5	Draft
2017-09-06	Valeria Bartsch	Executive Summary, Introduction, Concluding Remarks added	V0.6	Final
2017-09-26	Paul Carpenter, Giuliano Taffoni, Manolis Marazakis, Antoniu Pop, Valeria Bartsch, Paul Carpenter, Antonio Pena, Andrea Bartolini	Reviewer's comment and suggestions added	V0.7	Including some of the reviewers comments

Executive Summary

In this deliverable, we describe the runtime systems (OmpSs and OpenStream) and communication libraries (GPI and MPI) being adapted to the ExaNoDe hardware. These runtime systems and libraries will provide standard and portable programming interfaces so that an application can take advantage of the unique system characteristics of the ExaNoDe prototype without needing to optimize the application for the specific UNIMEM APIs defined in D3.6 [1] and D3.7 [2].

All runtime systems and communication library have started to integrate UNIMEM which provides non-coherent load–store and RDMA access to any other remote node. The integration with UNIMEM will allow applications to transparently benefit from UNIMEM using the above-mentioned runtimes and communication libraries. OpenStream, MPI and GPI are being directly coupled with the UNIMEM API, while OmpSs uses the underlying MPI layer to indirectly couple with UNIMEM. To ease the integration effort an emulation library is being used allowing tests on a standard x86_64 SMP system without the need to have the prototype hardware available on site. Tests using the emulation library have been successful. In the third year the partners plan to use ARM+FPGA prototypes to test their system integration.

The following limitations of the UNIMEM library have been found when customizing the implementations to UNIMEM and are under discussion with FORTH:

- Cooperation between FORTH and the other partners (particularly FHG) resulted in an extension to the UNIMEM API functionality. Until July 2017, the prototypes had a bug when using more than one buffer allocation registered with the communication hardware, and no memory registration API was not available. There was also an issue that the UNIMEM API incorrectly specified that only one buffer could be registered at a time. This prevented registering user-provided memory buffers preventing in turn a low-latency zero-copy approach. This bottleneck should now be resolved in the newest UNIMEM software and needs to be tested.
- A parallel startup mechanism like mpi_run or gpi_run is not yet available and standard tools and scripts cannot be used on UNIMEM. In addition, environment settings/variables and command line arguments must be communicated to the remote node and setup correctly before a process inside a parallel topology can start. We are in the process of defining an interface that fulfils the requirements for GPI and MPI to start up remote processes.

In addition to the integration with UNIMEM all runtime systems and communication libraries also start to integrate FPGA support e.g. based on experience from previous projects. E.g. BSC has participated in the AXIOM project, UoM is participating in the EcoSCALE project.

Finally, this deliverable describes other runtime support, specifically regarding thermal and power management and runtime libraries for performance-critical primitives:

- The ExaNoDe hardware does not provide hardware mechanisms to control power consumption, so the power and thermal control in the scope of the project will directly control the frequency of cores to optimise the power reduction while minimizing the application performance loss. With the MPI profiling tool in the reference application QuantumESPRESSO a DVFS (Dynamic Voltage Frequency Scaling) based power capping approach has been tested and has shown competitive results with respect to hardware based power and thermal control mechanisms.
- Dynamic load balancing has been implemented as a dynamic load balancing library on top of UNIMEM. It relies on remote atomic operations provided by UNIMEM for which an emulation library has been developed which is integrated with the FORTH RDMA emulation library.

These technologies will be made available and potentially integrated into the optimized implementations of GPI, OmpSs, OpenStream and MPI.

Table of Contents

Table of Figures

List of abbreviations

Term	Definition
ACPI	Advanced Configuration and Power Interface
API	Application Programmer Interface
APP	
BW (MPI)	Busy Waiting MPI
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DoA	Description of the Action
DSA	Dynamic Single Assignment
DVFS	Dynamic Voltage and Frequency Scaling
EAW	Energy-Aware MPI Wrapper
ECED	Edge and Coherence-Enhancing Anisotropic Diffusion filter
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSP	First Step Problem
GAS	Global Address Space
GASNet	Global Address Space Networking
GASPI	Global Address Space Programming Interface
GPL	GNU General Public License
GPU	Graphics Processing Unit
GSAS	Global Shared Address Space
HLS	High-Level Synthesis
IB (MPI)	Interrupt-based MPI
ILP	Integer Linear Programming
IMC	
ISP	*i*-th Step Problem
LLC	
MCTP	(Fraunhofer's) Multicore Thread Package
MPI	Message Passing Interface
MPSD / MPMD	Multiple Program Single/Multiple Data
NUMA	Non-Uniform Memory Access
OFI	OpenFabrics Interface
OS	Operating System
OTC	Optimal Thermal Controller
PE	
PGAS	Partitioned Global Address Space
PID	Proportional–Integral–Derivative (controller)
PMPI	MPI Profiling interface
PMU	
PoC	Proof of Concept (prototype)
QE–CP	Quantum ESPRESSO Car–Parrinello
RAPL	(Intel) Running Average Power Limit
RDMA	Remote DMA (Direct Memory Access)
RTM	Reverse Time Migration
SIMD	Single Instruction Multiple Data
SPSD / SPMD	Single Program Single/Multiple Data
SMP	Symmetric Multiprocessor
TDP	Thermal Design Power

TMC	Thermal-aware Task Mapper and Controller
UDP	User Datagram Protocol
VMR	Virtual Memory Region

1 Introduction

The ExaNoDe project is developing a unique HPC system architecture based on the UNIMEM architecture, which is also the basis for the related projects EUROSERVER [3], ExaNeSt [4] and EuroEXA [5]. A system that implements UNIMEM consists of a number of computational nodes connected through a custom network. Each node typically contains multiple processing cores, which communicate amongst themselves using coherent shared memory as provided by the hardware. Distinct nodes communicate using UNIMEM's global address space (GAS), which provides non-coherent load–store and RDMA access to any other remote node. The UNIMEM hardware architecture is exposed to user space via the Global Shared Address Space (GSAS), user-space RDMA, mailbox and remote allocator APIs defined in D3.6 [1] (which was due in project month 12).

For easier programming, the application developers will be provided with standard and portable programming interfaces through the runtime systems and communication libraries described in this deliverable. This approach allows applications to take advantage of the characteristics of the ExaNoDe system architecture and UNIMEM architecture, without them having to be ported to a specific API and without the application developer needing to understand in detail the associated performance tradeoffs.

Section 2 describes the work done on integrating the task-based programming models OmpSs and OpenStream with UNIMEM. In addition work done to support FPGA programming has been included in the section as well as the choice of mini-application to test the programming model with.

OmpSs is a task-based programming model that extends OpenMP with new directives for asynchronous parallelism and heterogeneous devices such as GPUs and FPGAs. In ExaNoDe, the cluster implementation of OmpSs runtime system Nanos6 is being leveraged as the basis for efficient runtime support for offloading tasks across nodes on the UNIMEM architecture with the help of the underlying MPI communication API. OmpSs already supports offloading of tasks to FPGAs, using High-Level Synthesis (HLS), and it is being ported to the Xilinx UltraScale+ FPGA in the AXIOM Project [6]

OpenStream is a task-based data-flow programming model also implemented as an extension to OpenMP, and designed for efficient and scalable data-driven execution. OpenStream has explicit dependencies in the source program marked using streams. Compile-time transformations map each task's memory accesses to private input and output buffers. The OpenStream runtime system controls memory allocation, task placement and RDMA memory transfers between tasks. OpenStream is supporting OpenCL to exploit FPGAs and is integrating the EcoSCALE [7] High-Level Synthesis (HLS) toolchain.

Section 3 describes the integration of the communication APIs GPI and MPI in the ExaNoDe prototype. In addition work done to support FPGA programming has been included in the section as well as the choice of mini-application to test the programming model with.

GPI is an open-source communication library that implements the GASPI standard PGAS API. It provides a portable and lightweight API that leverages remote completion and one-sided RDMA-driven communication, both being efficiently supported by the UNIMEM architecture. UNIMEM dependent module of GPI have been identified, integrated with UNIMEM and integrated with an emulation framework to socket layer of UNIMEM, the software has been tested on the remote prototype. A setup of a small test system consisting of Xilinx Ultrascale+ FPGAs and ARM 64-bit in one package is foreseen to build up the necessary FPGA support.

MPI is the standard message-passing API supported by all serious HPC systems and employed by the vast majority of scientific applications. Efficient support for MPI is mandatory for any HPC system or prototype, and MPI support is an important output from ExaNoDe WP3 that is needed by the ExaNeSt project and will be further optimized in the EuroEXA project. The OmpSs integration of UNIMEM will be based on MPI. The coupling will be done with the low-level network interface OFI on which the MPI implementation MPICH is built.

Section 4 describes thermal and power management. These technologies will be made available and potentially integrated into the optimized implementations of GPI, OmpSs, OpenStream and MPI.

The runtime systems and communication libraries are being prototyped and developed using (a) remote access to the multi-board prototype hosted at FORTH in Crete, which provides functional verification on real hardware, and (b) software emulation of the UNIMEM APIs using a software layer provided by FORTH and UOM. The latter provides the ability to perform substantial development work on a local machine.

The runtime systems and communication library will be tested and evaluated using the mini-applications from WP2 (from D2.1 [8]) as indicated in Table 1.

Table 1: Comparison of runtime systems and communication libraries

	MPI	**GPI-2**	**OmpSs (clusters)**	**OpenStream**
Programming model	Message passing	PGAS	Tasks with argument directionality (input/output)	Tasks with explicit dependencies specified using streams
Data visibility	Local to MPI process	Global	Global	Global
Mapping work to nodes	Manual	Manual	Runtime system	Runtime system
Language type	API	API	Language extension (Pragmas)	Language extension (Pragmas)
Execution style	MPMD	MPMD	SPSD / SPMD	SPSD / SPMD
Inter-node communication	Explicit (message passing)	Explicit (one-sided asynchronous)	Implicit (runtime system based on argument directionality)	Implicit (runtime system based on streams)
Work scheduling	Manual	Manual	Runtime system	Runtime system
Base language(s)	C, C++, FORTRAN	C, FORTRAN	C, FORTRAN, CUDA	C
WP2 Mini-app	All	GPI test suite, separate stencil kernel	MiniFE	HydroC, MiniFE and NEST

2 Runtime systems

The work on two task-based runtime systems, OmpSs and OpenStream, is presented in this section. Both OmpSs and OpenStream extend the programming language (C, FORTRAN or CUDA in case of OmpSs, C in case of OpenStream) with pragmas. The internode communication is implicit. OmpSs and OpenStream will exploit UNIMEM in their cluster implementation.

2.1 OmpSs

This section was contributed by BSC.

This section presents the contributions of BSC related to the OmpSs programming model and the Nanos runtime system for distributed execution. In Section 2.1.1 we discuss the latest features of OmpSs which enable more opportunities for exploiting parallelism at the programming model level. Section 2.1.2 presents the distributed memory version of Nanos6, the new implementation of the OmpSs programming model. Section 2.1.2.1 presents the memory model of the distributed memory Nanos6, Section 2.1.2.2 describes the execution model, showing how tasks can be offloaded to nodes of the cluster transparently to the programmer, Section 2.1.2.3 discusses the design of the communications layer of the runtime system and finally, in Section 2.1.2.4 we show some initial results from popular linear algebra kernels ported to the OmpSs programming model.

2.1.1 Introduction to OmpSs-v2

OmpSs [9] is a task-based parallel programming model aimed to provide scalability and malleability without significant programming effort. OmpSs-v2 [10] is an extension of the programming model, initiated in the INTERTWinE [11] Project that increases the scalability of applications by integrating more efficiently nested tasks, a natural way to decompose a bigger problem in finer-grain computational tasks, with task dependencies.

2.1.1.1 Fine-grained release of dependencies across nesting levels

Task-based programming models that support dependencies and nesting normally require a the invocation of a synchronisation primitive at the end of the task, e.g. an OpenMP *taskwait*

```
#pragma omp task depend(inout:a,b) weakwait   //Task T1
{
  a++; b++;
  #pragma omp task depend(inout: a)           //Task T1.1
  a += ...;
  #pragma omp task depend(inout: b)           //Task T1.2
  b += ...;
}

#pragma omp task depend(in: a)                //Task T2
... = ... + a + ...;

#pragma omp task depend(in: b)                //Task T3
... = ... + b + ...;
```

Figure 1: Fine-grained release of dependencies using the weakwait construct of OmpSs-v2

pragma, which blocks the task until all its subtasks have finished. This is required in order to preserve the correct semantics of dependencies across tasks. Prior work in the INTERTWinE Project introduced the OmpSs-v2 *weakwait* clause to *task* pragma. The *weakwait* clause implicitly inserts a taskwait after the execution of the task, which allows the runtime system to understand that no more subtasks are going to be created and the dependencies of the task that do not need to be enforced any more can be released incrementally.

For example in Figure 1 we have a code snippet with task T1 that depends on variables *a* and *b* and has two subtasks T1.1 which depends on *a* and T1.2 which depends on *b*, task T2 which depends on *a* and task T3 which depends on T3. Without the weakwait construct T1 would need to include a *#pragma omp taskwait* at the end of the task body. T2 and T3 would wait until the completion of T1 which would happen only after T1.1 and T1.2 have finished. With the weakwait clause once the body of T1 exits only the live dependencies of T1 need be enforced, i.e., if T1.1 has not yet finished the dependency from T1 to T2 becomes a dependency from T1.1 to T2, so that T2 can start even if T1.2 has not finished yet. This allows the discovery of more parallelism dynamically.

In the previous example, in order to release dependencies this way task T1 needs to finish execution. However, it might be useful to release dependencies while the body of the parent task is still executing, e.g., the task knows that it will use some data only at the beginning. In order to enable this functionality OmpSs-v2 introduces a new directive:

#pragma omp release depend(…)

which releases all the dependencies in the list of the depend clause.

2.1.1.2 Weak dependencies

Section 2.1.1.1 presents how OmpSs-v2 allows the early release of dependencies from inner to outer nesting-levels in a fine-grained fashion. However, where nesting is used, it is likely that the outer nesting levels define dependencies in a coarser granularity. Even if some elements of the depend clause of the outer task is only needed by its subtasks, its execution and eventually the creation of the subtasks will be deferred and discovery of parallelism, suspended.

OmpSs-v2 extends the depend clause with the *weakin*, *weakout* and *weakinout* dependency types. Semantically, these types define dependencies equivalent to the non-weak types. When a task declares weak dependencies, though, it signifies that it will not access itself the data, only its subtasks will do, hence the task is allowed to start its execution, which will allow it to create the subtasks.

As a result, early release of dependencies and weak dependencies, together in action can potentially result in increased parallelism discovery while expressing the applications using nesting which is very natural for a large number of problems.

2.1.2 Nanos6 runtime system on distributed memory

OmpSs-v2 is implemented in Nanos6 the successor of the Nanos++ runtime system. The choice to implement a new runtime system, rather than implementing OmpSs-v2 as extensions in Nanos++, is guided from the requirements of backwards compatibility for OmpSs applications as well as better maintainability of the Nanos6 codebase in comparison with Nanos++.

Nanos6 provides a new version for the distributed memory runtime implementation in the ExaNoDe project, which incorporates the features of the OmpSs-v2 programming model and introduces a novel memory model, task offloading mechanism and communication layer.

2.1.2.1 Nanos6 memory model

The distributed memory version of OmpSs developed in the ExaNoDe Project provides a Partitioned Global Address Space (PGAS) model abstraction layer for the memory view of the system. This OmpSs memory layout is general-purpose and applicable to an implementation for any cluster, but it enables future work, in ExaNoDe or EuroEXA, to take advantage of the UNIMEM shared memory architecture. The OmpSs memory model presents the distributed physical address space of the nodes involved in the computation as a single address space which is accessible by every compute node of the cluster. As a result, on conventional clusters that require physical memory copies among nodes of the cluster, the programmer does not need to explicitly program these data transfers, as they are handled by the Nanos6 distributed memory runtime using MPI. The current implementation will target UNIMEM via the UNIMEM-optimized MPI library. We will consider the potential benefit of future optimizations to use the native UNIMEM API to eliminate the data transfers on UNIMEM platforms, while maintaining software compatibility with traditional distributed memory clusters.

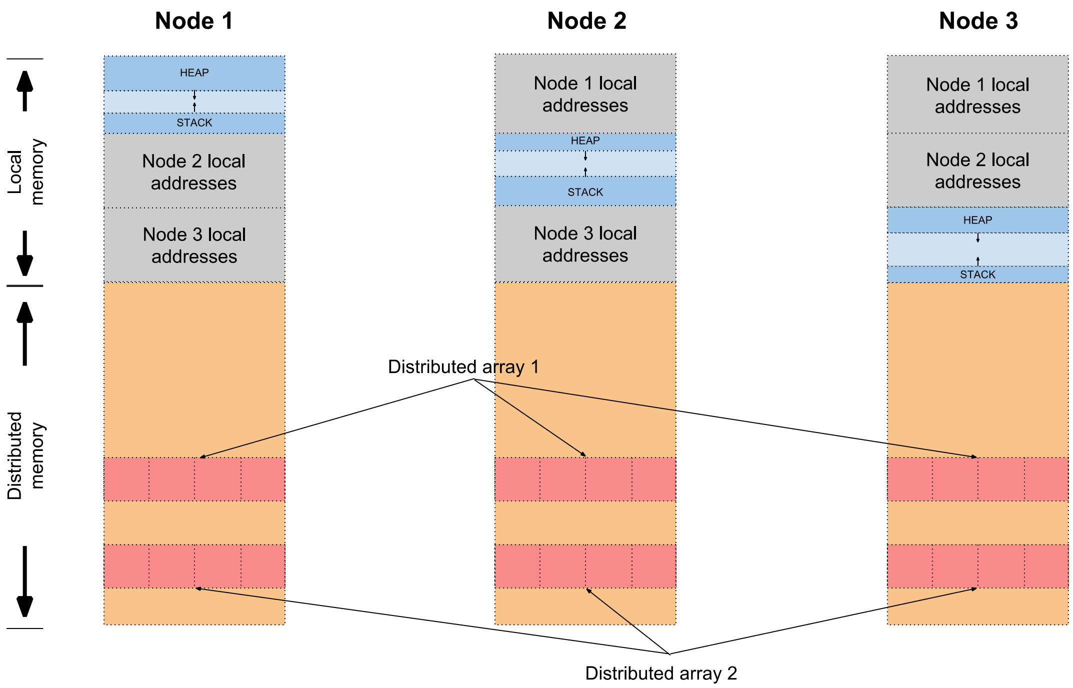

Figure 2: Virtual Memory (VM) address space representation of cluster nodes managed by Nanos6.

Figure 2 shows the layout of the virtual memory of the cluster nodes managed by the Nanos6 runtime system. During initialization Nanos6 maps in every node a virtual memory region (VMR) large enough to handle the maximum memory requirements of the OmpSs application. The starting address of these VM regions is the same on every node. This is necessary in order to facilitate the transfer of data across nodes without having to apply address translation across nodes. Memory requests are served through custom allocators of the Nanos6 runtime system. Subsequently, Nanos6 divides each VMR into two distinct regions, which have different allocation semantics.

The lower addresses of the VMR are reserved for conventional local memory allocations, i.e., stack and normal heap allocations. Nanos6 divides this set of addresses equally among the nodes of the cluster. This means that every address within this region is used to store the local data of one particular node of the cluster. The rest of the nodes of the cluster use these addresses whenever they need to bring local data of the said node, temporarily. This simplifies the process of moving data around the cluster, since it eliminates the need for address translation.

The higher addresses of the VMR are reserved for *distributed allocations*. An allocation from this memory region is implemented inside the runtime system as a collective operation across all nodes of the cluster. Figure 3 describes the operation of a distributed allocation. Firstly, the whole distributed array is allocated in every cluster node at exactly the same memory range $[ptr, ptr + size)$. Subsequently, each node becomes the *home node* of one part of the array. This means that by default, the latest produced data of a subrange of the array will be stored in its home node. If during execution, a range $[subrange\_ptr, subrange\_ptr + size)$ needs to be used by a task that is scheduled on a node different than its home node, a memory transfer will be initiated from the home node of the subrange. When a node fetches a range of data from its home node it uses the same range of addresses as its home node does. Those virtual addresses are available also locally, since during the allocation of the array these addresses were allocated on every node of the cluster. In this way, Nanos6 does not have to do address translation when it moves distributed data across the cluster nodes. The way an array is distributed to home nodes is controlled by the programmer who can choose the distribution policy. Information about the distribution policy of arrays can be used later by the Nanos6 scheduler in order to make decisions based on locality criteria. Thus the distribution policy is meant to be chosen according to the access patterns of the application.

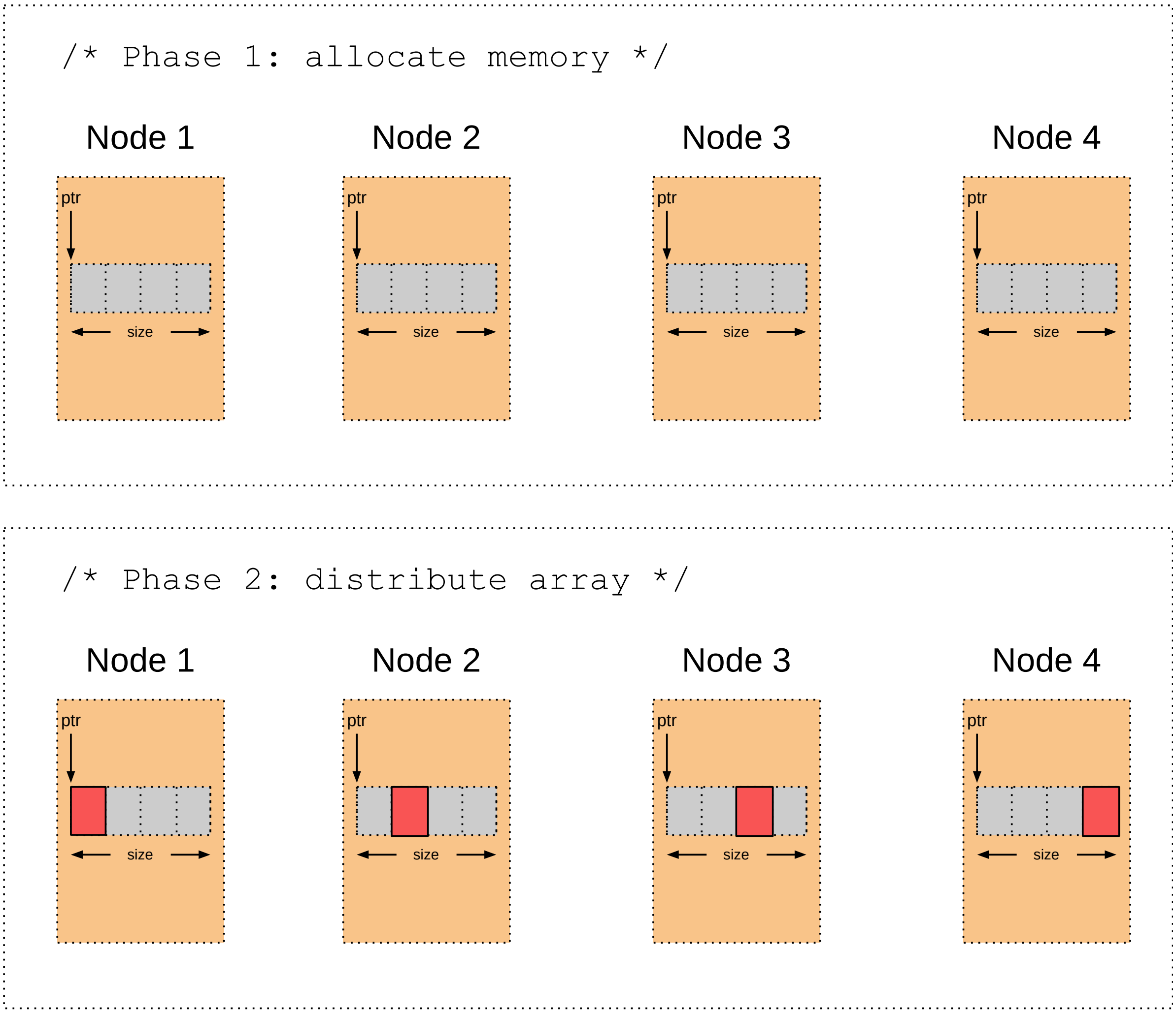

Figure 3: Distributed allocation in Nanos6 is a collective operation. The array is allocated first on all nodes and then logically distributed across them

2.1.2.2 Nanos6 execution model

The memory model is coupled with the task-parallel semantics of OmpSs for defining computations. The programmer defines tasks i.e., computational units that operate on ranges of data located on the address space.

Nanos6 uses a master–slave architecture. The OmpSs application begins executing on the *master* node, similarly to the shared-memory flavour of the runtime. The code is executed serially and whenever a `#pragma omp task` directive is encountered a new task is created and becomes available for concurrent execution once its dependencies are resolved. When running on distributed memory, the scheduler of Nanos6 can also decide to offload tasks to *slave*, or else *remote*, nodes once they are ready for execution, i.e., all their strong dependencies have been resolved.

During execution, the scheduler takes decisions regarding the node onto which the task should be offloaded. Before a remotely-executed task executes its body function, the runtime system copies any non–node-local data to the node that the task will execute on. The programmer needs to declare all the dynamically allocated data that the task uses and the way the task will handle them using the dependencies clauses: *in(), out(), inout(), weakin(), weakout()* and *weakinout().* When executing on distributed memory, in addition to declaring the dependencies among tasks, these clauses provide the necessary information about data transfers that must be performed by the runtime before executing a task.

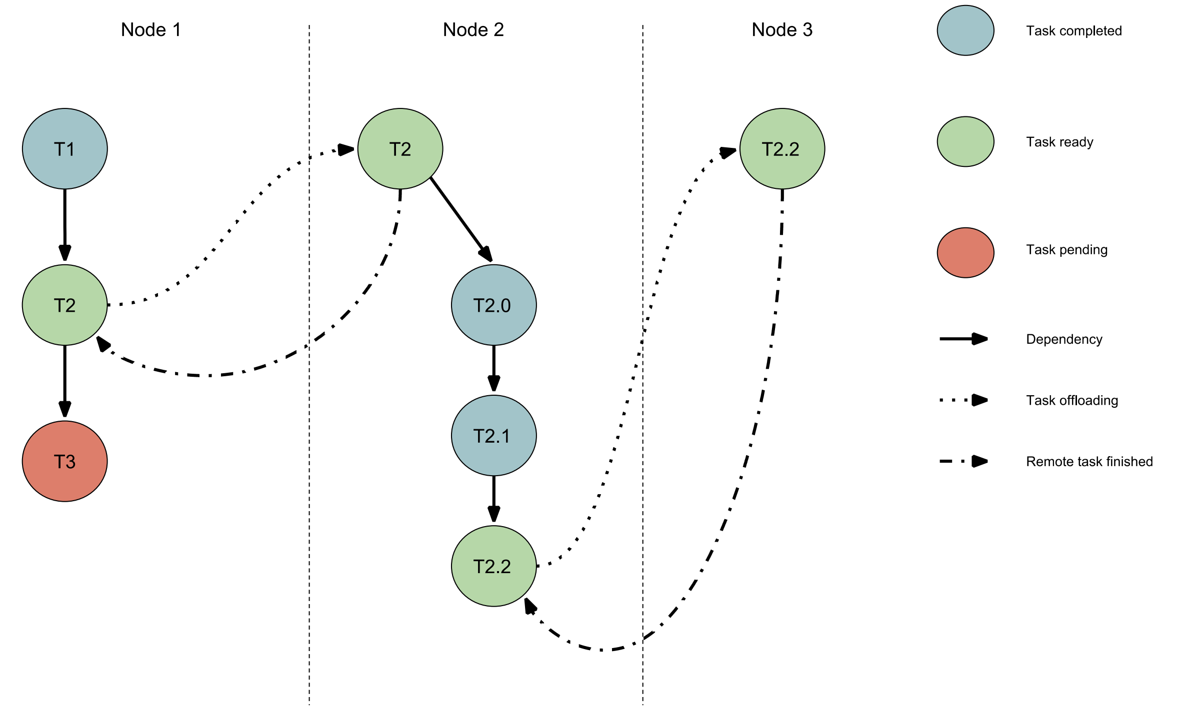

Figure 4: Nanos6 task offloading. A ready task can be offloaded to a remote node. All tasks with a dependency on the offloaded task will wait until the offloaded signals its completion.

Figure 4 presents an example of the execution model of Nanos6 for distributed memory systems. In this example, when task T2 becomes ready for execution, the scheduler decides to offload it to Node 2. The original task is marked as an *offloaded* task and it remains in the memory of Node 1 so that the dependencies within Node 1 are preserved. Task T3 on Node 1 has a dependency on T2 and as a result it will not be ready until the T2 is marked as complete. This will happen once the *remote* T2 sends a message to the offloaded T2 signaling its completion. Along with the task T2, Node 1 sends to Node 2 information regarding the location of all the data that T2 takes as input (*in()* and *inout()* dependencies). Once the access information for all the input arguments of the remote task T2 on Node 2 is received the task T3 is ready for execution. In addition, the remote T2 creates three subtasks. The first two are executed locally, but T2.2 is offloaded by the scheduler from Node 2 to Node 3. The parent task T2 will not be marked as complete until the remote T2.2 finishes. When T2.2 on Node 3 finishes it sends a message to Node 2 along with access information about all the output dependencies i.e., *out()*

and *inout()*. This information is then propagated from Node 2 to Node 1. At this point T3 can start execution, knowing the location of all the output accesses of T2. This example shows how Nanos6 uses the dependency system to propagate information regarding the location of all the data of the OmpSs application. This scheme allows us to handle all the data transfers without the need of a software directory, which simplifies the design and implementation and minimizes the amount of communication among the cluster nodes.

2.1.2.3 Communication Layer

The implementation of Nanos6 requires communication among the cluster nodes for exchanging *command* and *data transfer* messages. Command messages include all the messages for offloading tasks, synchronization of nodes, sending information regarding the location of data and initiating data transfers. Data transfer messages are used to transfer data regions among nodes.

The communication layer of Nanos6 operates as an abstraction layer that decouples the rest of the components of the runtime system from the actual library that is used to implement the actual network transfers. This design is very modular since it allows the network communication layer to be transparently implemented on top of different libraries and allows the user to choose the most desirable implementation at runtime.

For ExaNoDe we have implemented the communication layer of Nanos6 on top of standard MPI. This provides compatibility with all HPC systems that implement the MPI standard, making it a very appealing choice. In particular, the port of MPI to the UNIMEM architecture will allow Nanos6 to run on any UNIMEM platform without modifications. In future work, in ExaNoDe or EuroEXA we will consider the benefit of eliminating the data transfer messages using the native UNIMEM API, while maintaining software compatibility with traditional distributed memory clusters.

2.1.2.4 Preliminary results

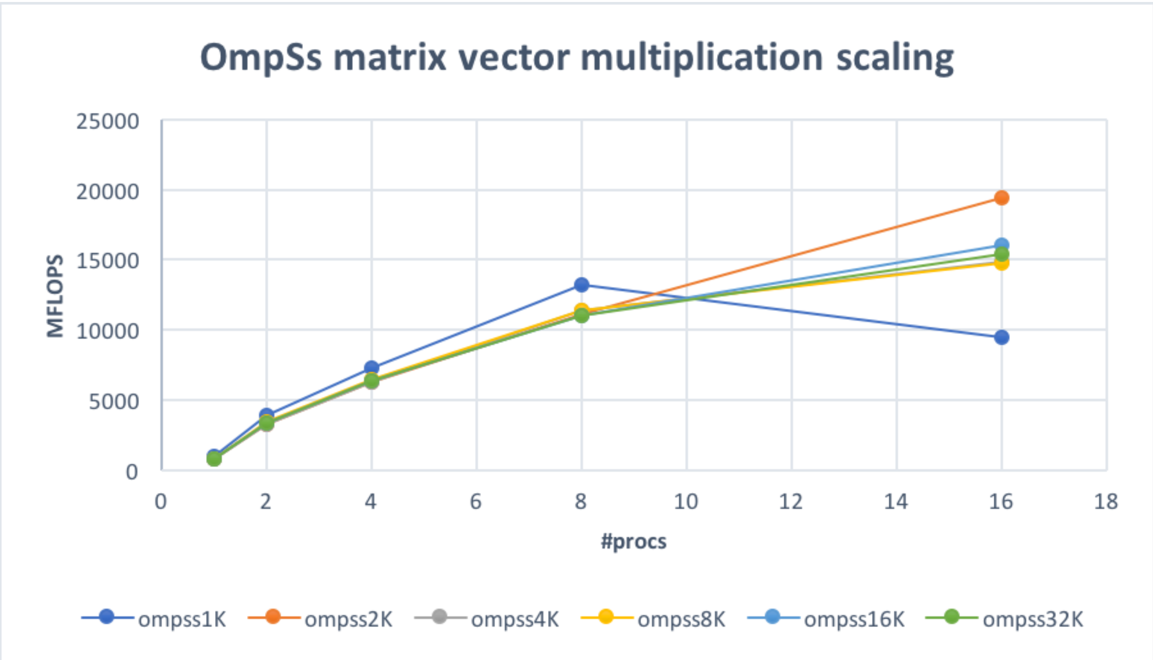

Figure 5: Scaling of a matrix vector multiplication operation implemented in OmpSs with Nanos6 Clusters.

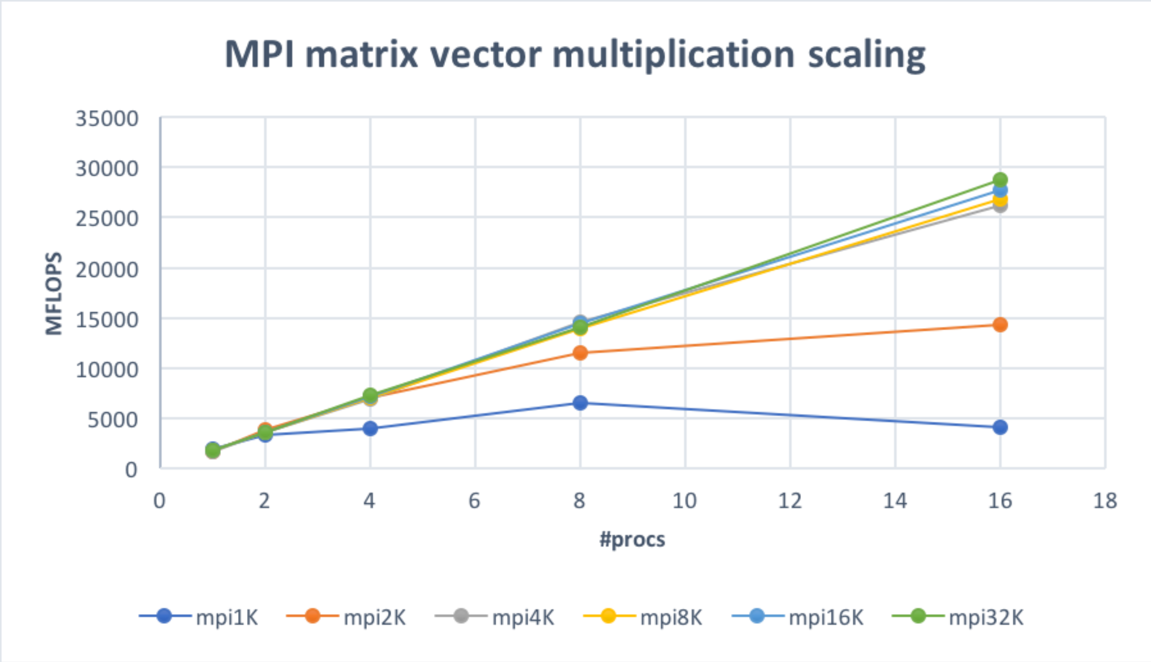

Figure 6: Scaling of a matrix vector multiplication operation implemented in MPI

We have performed an evaluation of the initial implementation of Nanos6 using various BLAS kernels ported to OmpSs-v2, and compared them with the equivalent MPI implementations. Figure 5 and Figure 6, respectively, show the scaling of a matrix–vector multiplication operation in OmpSs and MPI measured on the MareNostrum 4 supercomputer. The results show that, compared with MPI, Nanos6 currently faces scalability issues when the problem sizes increase. This could be attributed to various issues, e.g. the scheduler implementation of Nanos6 or overheads related to the offloading of tasks to nodes and caching data to remote nodes. We are currently investigating these bottlenecks with the assistance of Extrae and Paraver, which are the tracing and performance analysis tools that have been developed from BSC and are being integrated in Nanos6.

2.2 OpenStream

This section was contributed by UOM.

2.2.1 Introduction to OpenStream

OpenStream [12] is a task-parallel, data-flow programming model implemented as an extension to OpenMP. It is designed for efficient and scalable data-driven execution; shared-memory programming is allowed for fast prototyping, essentially following the OpenMP syntax, but additional information must be provided by the programmer, using a dedicated syntax, in order to take advantage of OpenStream optimizations. In particular, OpenStream enables programmers to express arbitrary dependence patterns, which are used by the runtime system to exploit task, pipeline and data parallelism. Each data-flow dependence is semantically equivalent to a communication and synchronization event within an unbounded FIFO queue. Pragmatically, in the original shared-memory instantiation, this is implemented by compiling dependences as accesses to task buffers dynamically allocated at execution time: writes to streams result in writes to the buffers of the tasks consuming the data, while read accesses to streams by consumer tasks are translated to reads from their own, task-private buffers.
Compared to the more restrictive data-parallel and fork–join concurrency models, task-parallel models enable improved scalability through load balancing, memory latency hiding, mitigation of the pressure on memory bandwidth, and as a side effect, reduced power consumption.

Currently developed at UOM, OpenStream further takes advantage of the information provided by programmers on task dependences to aggressively optimize memory locality through dynamic task and data placement.

2.2.2 Exploiting UNIMEM in OpenStream

OpenStream relies on a private-by-default strategy for handling communication between tasks, which means that despite a shared-memory view from the programmer's perspective, communication is more akin to message-passing than to concurrent shared-memory communication. This is made possible by requiring programmers to provide additional information on how data is accessed within tasks. This information is used at compile time to generate the appropriate modifications to memory accesses to achieve Dynamic Single Assignment (DSA). OpenStream tasks compute on data available in input buffers and write data in output buffers, each belonging to a unique task reading from them. This data-flow execution model is a perfect match for the UNIMEM memory model, providing a straightforward mapping of communication on top of RDMA and minimizing the reliance on global atomics. Furthermore, the privatization of data communicated between tasks is the key to enable the runtime to fully control the locality of memory allocation and of task placement. OpenStream relies on the inter-node atomics provided in the UNIMEM memory model to implement low-level runtime algorithms, such as dynamic load balancing, inter-node synchronization and locality-aware scheduling and memory allocation. This is further discussed in Section 4.2.

Further optimization of the behavior of the OpenStream runtime will be possible if UNIMEM permits RDMA and atomics to be used within the same memory regions. This has been one of the key challenges to port OpenStream as it has required splitting the data-structures used for managing memory and task placement across separate memory regions while ensuring that data and meta-data remain coherent.

The development of concurrent data-structures and algorithms on memory models that do not provide sequential consistency is patently error-prone and time consumming. Testing poses significant challenges as errors may only manifest when specific interleavings of memory operations occur – behavior which can be impossible to exhibit on emulation or on prototypes where the timing is substantially different than the target hardware. This problem is further compounded when the memory model is not uniform across all execution units as node-local behavior will differ from inter-node behavior. We expect that a substantial number of issues are likely to become apparent when executing on increasingly advanced hardware that allows more memory interleavings to occur.

For example, we expect that the behavior of atomic operations provided in UNIMEM will have an impact on our work-stealing load-balancing algorithm if there is an asymmetry between the success rates of local and remote atomic operations. Such an asymmetry would introduce a bias in the way our algorithm works, which in turn would translate into poor work distribution across the machine. As we further discuss below, this is critical for OpenStream programs as we do not assume an initial distribution of data and work across the machine.

2.2.3 Implementation

Communication is automatically managed by the OpenStream runtime system. Part of the work is done at compilation time, by privatizing all data dependences between tasks and introducing runtime hooks for setting up remote memory operations. This step enables the runtime to determine which data is locally available and which data needs transferring, then initiate the memory transfers and determine when all data required for execution is finally available. As no worker is waiting for data to be transferred, a good load balance will ensure that computation and communication is fully overlapped.

Memory and work placement are driven by two main algorithms [13] [14]. The main mechanism for nodes and workers to acquire work is randomized work-stealing. This allows to balance the workload across compute resources, but may be inefficient with respect to the amount of communication it generates as tasks are randomly acquired by workers across the entire machine. Dependence-aware memory allocation and work-pushing allow to reduce the amount of communication required by moving tasks to nodes that will require the least data movement.

Multiple node startup is managed directly within the OpenStream loader rather than relying on an external tool, and uses the *remoteFork* facility. The startup procedure instantiates a core process on each node and initialises the OpenStream communication and scheduling data structures, then allows the local processes to set up a team of worker threads on each node. There is no initial distribution of work as OpenStream relies primarily on hierarchical work-stealing for load-balancing. As soon as a node is ready to start executing tasks, it starts attempting to steal work from neighboring nodes.
In later stages of execution, once data is distributed across the machine, locality-aware work-pushing will complement work-stealing by sending tasks that require remote data to be executed on the node where most of their inputs are located. While this approach helps reduce the amount of data movement overall, it may lead to poor load-balance if it is used on its own as it will have a tendency to concentrate data and work on a subset of nodes. Randomized work-stealing is therefore still required to ensure that computation is reliably distributed across all nodes, further enabling the possibility of seamlessly bringing new nodes online during execution.

2.2.4 ExaNode Mini-app

Due to the availability of the HydroC mini-app in multiple parallel programming models, including a C+OpenMP version, this has been the primary target for porting to OpenStream. The initial translation from OpenMP parallel loops to OpenStream tasks relying on shared memory communication was straightforward and yields identical performance results on uniform shared memory multi-core platforms. This compatibility behavior is allowed in OpenStream to facilitate porting efforts, however, this version cannot be compiled directly to execute on multiple UNIMEM nodes.
In a second step, the OpenStream implementation was converted from shared-memory data-parallel execution to pure data-flow, where tasks communicate exclusively through privatized streams. This step was complicated by the frequent use of shared memory pointer arithmetic when communicating partial results between different computation phases of HydroC, but it is essential to enable multi-node execution and to allow showcasing the advantages of UNIMEM RDMA communication overlapping computation.
The porting effort has now shifted towards optimizing the data-flow implementation, in particular focusing on eliminating over-synchronization between computation steps, and towards integration in the UNIMEM emulation framework and the physical prototypes.

2.2.5 Towards FPGA integration

To exploit the Field-Progammable Gate-Arrays (FPGAs) that represent the bulk of the computational power in the ExaNoDe system, we have made the decision to extend OpenStream with OpenCL support, enabling programmers to specify multiple versions for the work function of each task, written either in C (possibly wrapping code in other sequential

languages) or in OpenCL. These versions are managed by the OpenStream dynamic scheduler, which decides at runtime on the best target for each kernel.
Currently, OpenStream is able to schedule the different kernel versions across CPU cores and multiple, heterogeneous OpenCL devices (e.g., CPU cores + AMD APU + discrete GPU) and we are currently optimizing the scheduler heuristics for dynamically adapting the target device based on the observed cost of communication and compute time on each available resource for the different types of tasks.
This preliminary work has cleared a path to the next step, which is to integrate the OpenStream environment with the EcoSCALE high-level synthesis toolchain to take advantage of the Xilinx Ultrascale+ FPGAs that will be available on the ExaNoDe system. The design of the current implementation was chosen to maximise the flexibility of the OpenStream framework and we expect to be able to integrate FPGAs in the OpenStream resource model and scheduler.

2.3 Parallel runtime support

This section was contributed by UOM.

2.3.1 Introduction

In order to maximize the efficiency of execution, both in terms of performance and energy, and to exploit fully the massive parallelism provided by the ExaNoDe architecture, it is essential to optimize performance-critical aspects of the runtime. In particular, UOM is focusing on dynamic load balancing through work-stealing, dynamic scheduling for memory locality and synchronization.

2.3.2 Optimized runtime support

UOM has ported the current state-of-the-art implementation [14] of work-stealing dynamic load-balancing based on Chase and Lev's algorithm for intra-node load balancing, as well as the fastest hybrid barrier synchronization implementation [15] for a single node. This first step is essential even with the new UNIMEM memory model because these algorithms are very sensitive to latency and therefore cannot rely on a uniform view of the memory.
In a second step, UOM has implemented a functional unoptimized work-stealing library on top of UNIMEM for inter-node load balancing, which is integrated with the intra-node algorithm in the form of hierarchical work-stealing, whereby work is sought in widening neighbourhoods. This implementation relies on the remote atomic operations provided by UNIMEM, and for which UOM has developed an emulation layer that integrates with FORTH's RDMA emulation library. Furthermore, to minimize the overheads incurred by memory transfers between nodes, UOM has developed locality-aware allocation and scheduling optimizations that deliver above 94% locality and up to 99% locality and 5× speedup over hierarchical work-stealing on 24 nodes [14]. While this study was conducted on a classical NUMA machine, the results are likely to translate into similar locality benefits, albeit with new tradeoffs that will require further investigation, on an ExaNoDe platform once ported.

3 Communication Libraries

Two communication libraries, the message passing MPI and the partitioned global address space library GPI, will use UNIMEM. Compared to the runtime systems the inter-node communication with the UNIMEM communication will be explicit. Thus a direct coupling between MPI/GPI components and UNIMEM becomes center-stage in this section. For GPI the focus is on one-sided asynchronous messages, whereas the MPI work will be foucessed on message passing.

3.1 GPI

This section was contributed by FHG.

3.1.1 Introduction to GPI

The Fraunhofer GPI (Global Address Space Programming Interface) open-source communication library is an implementation of the GASPI standard [16], freely available to application developers and researchers. GASPI stands for Global Address Space Programming Interface, and it is a Partitioned Global Address Space (PGAS) API that aims to provide extreme scalability, high flexibility and failure tolerance for parallel computing environments.

GASPI aims to initiate a paradigm shift from bulk-synchronous two-sided communication patterns towards an asynchronous communication and execution model. It leverages remote completion and one-sided RDMA-driven communication in a Partitioned Global Address Space. The asynchronous communication enables a perfect overlap between computation and communication. The main design idea of GASPI is to have a lightweight API ensuring high performance, flexibility and failure tolerance. More details about GPI can be found in deliverable D3.1 or on the GPI web page (http://www.gpi-site.com/gpi2/).

3.1.2 Exploiting UNIMEM in GPI

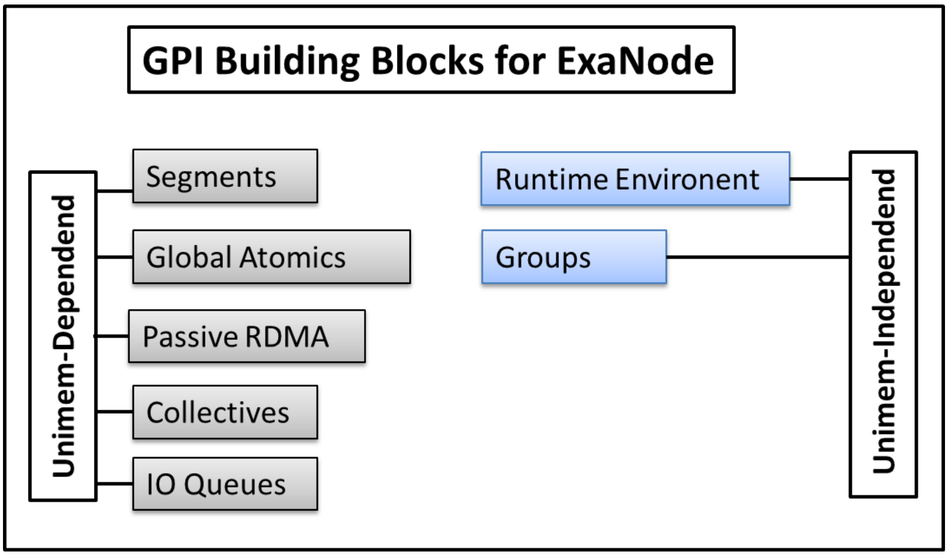

Figure 7: PI Building blocks for ExaNoDe architecture support

The UNIMEM independent modules (Runtime Environment and GPI Groups) can be developed/ported without any knowledge of the final hardware characteristics and interface descriptions of the ExaNode/UNIMEM architecture. Both modules are able to run over a secondary network using TCP/IP for data exchange. This makes it possible to start early implementations of these components within the ExaNoDe project. The communication interface for these modules will be RDMA over Sockets, one of the transport layers of UNIMEM.

Both independent modules have now been ported to the aarch64 architecture and can used on top of socket-interfaces of UNIMEM.

All UNIMEM-dependent modules, such as pinned memory segments, global atomics (and related memory areas), passive RDMA, collectives and one-sided reads and writes managed by IO-queues are hard to implement without detailed knowledge of the behaviour of the interconnect interface. Therefore an emulation library has been developed that implements most of the current functionality as described in [17]. This emulation library allows early tests on a standard x86_64 SMP system without the need to have real prototype hardware available on site. The remote system provided has proven not to be stable enough to run integration tests. All dependent modules have been implemented on top of the emulation framework and early tests were successful.

Some design decisions made by Forth for the user level RDMA Interface of UNIMEM will not allow GPI Applications to run directly on this layer without re-compilation and re-coding. In addition to that, the prototype systems and the available UNIMEM libraries are not yet stable enough for developments like GPI or practical tests and benchmarks. To ensure that a working GPI version for the ExaNoDe architecture is ready at the end of the project, we have ported the dependent GPI Modules from our emulation framework to the socket layer of UNIMEM. At the current stage of the project it is crucial to be able to start gpi_run. Here further work on UNIMEM needs to be done (as pointed out in Section 3.1.5.

This workaround allows us to further improve the individual GPI-Submodules and to continue the developing process without time-consuming delays (**Erreur ! Source du renvoi introuvable.**). A running GPI communication library is the basis for the development of one-sided micro benchmarks and mini-applications on top of the ExaNoDe hardware.

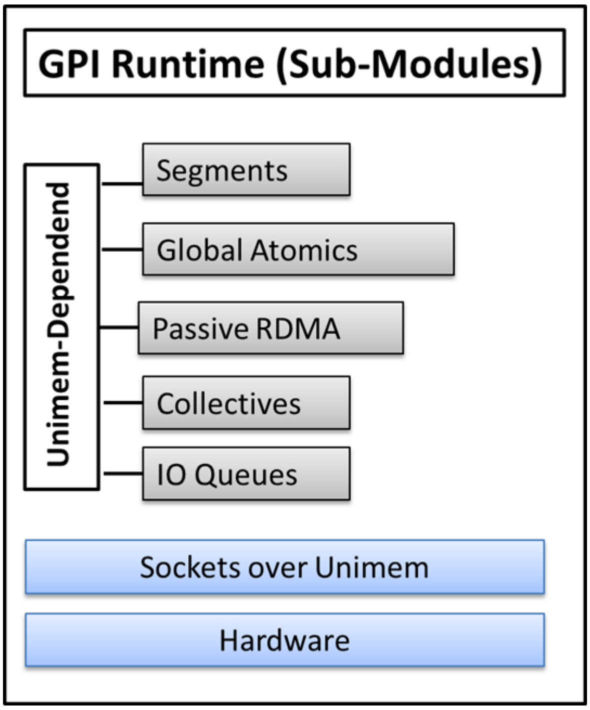

Figure 8: GPI BuildingBlocks of UNIMEM

3.1.3 Design of preliminary software implementation

The following sub-chapters are describing the implementation of dependent GPI Modules (**Erreur ! Source du renvoi introuvable.**) on top of UNIMEMs socket layer.

3.1.3.1 GPI Segments

GPI Segments can be created and deleted by using standard system routines like e.g. malloc and free. There are no size or alignment constraints on these segments as we have today with RDMA memory segments on most other interconnects. The allocation time will also be fast as we can use copy on write (cow) mechanisms compared to pre-pinned continuous memory segment allocations.

3.1.3.2 GPI IO-Queues

GPI applications trigger one-sided RDMA reads and writes by placing communication tokens into GPI IO-Queues. The status of a single token or a group of tokens can be determined at any time by calling a wait operation on a given queue. The wait operation returns a status array filled with the status of all completed read and write operations at that time. To enable non-blocking functionality for all worker threads within posting and wait calls, a background communication thread will be spawned internally. This special thread takes care of all the ongoing communications on all GPI queues and fills up the status arrays. As the communication takes place on top of sockets, the thread does not have to poll for io-operations or completions. The operating system can schedule this thread when data can be sent or peer data have received for one of the active queues.

3.1.3.3 GPI Collectives

In a first design GPI Collectives can be implemented by using internal GPI Segments and IO-Queues as described in Sections3.1.3.1 and **Erreur ! Source du renvoi introuvable.**3.1.3.2.

3.1.3.4 GPI Global Atomics

GPI currently defines two operations for Global Atomics: Atomic increment and atomic compare and swap (cas). With these two atomic operations in place, global spinlocks can be implemented which can be used to protect global data-structures and variables. As the current semantic for GPI Global Atomics require the immediate return of the previous values, standard IO-Queues cannot be used due to the separation of posting and wait calls. Instead a special IO-Queue will be implemented internally for atomics that combines and interlocks the posting and wait calls.

3.1.3.5 GPI Passive RDMA

Passive RDMA operations cannot be fully offloaded. They need some support from the Operating System so that passive waiting (sleeping) processes/threads can be informed when matching communication data is available. To implement this kind of data transport a special passive IO-Queue as described in 3.1.3.2 will be implemented. For this special IO-Queue the background thread will not fill up any completion arrays. Instead it will trigger one of the system calls like select, poll or epoll to inform waiting worker threads (waiting in GPI_PASSIVE_RECEIVE) about available data. The location, status and size of the data is returned to the caller directly from GPI_PASSIVE_RECEIVE.

3.1.3.6 Parallel Process Startup

A parallel startup mechanism like mpi_run or gpi_run is still not available yet and standard tools and scripts (e.g. process startup via ssh) cannot be used on UNIMEM. UNIMEM-Processes need to have some kind of parent->child relationship to inherit access rights to memory segments. In addition, environment settings/variables and command line arguments must be communicated to the remote node and setup correctly before a process inside a parallel topology can start. Here we are still in the process of defining an interface that fulfills the requirements for GPI and MPI to start up remote processes.

3.1.4 Suitable ExaNode Mini-app

Beside the GPI test suite a stencil kernel application (such as an BQCD simplified kernel) will be implemented to demonstrate the strength of overlapped and offloaded data communication on ExaNoDe/UNIMEM.

3.1.5 Current Status and Limitations

Current Unimem Limitations for single-sided Commmunication (GPI-2) are:

- Single memory segment: all GPI- based applications are using at least two or more memory segments during runtime to switch between communication buffers and computation segments (double buffering approach for asynchronous programs). This limitation has been present in the emulation library but is resolved in the UNIMEM software and still needs to be tested.
- Size limitation of the memory segment: 256mb. This size is much too small for real world applications, which typically use up to 16GB to 32GB per node, doing overlapped computations and communications.
- Single communication channel: at least two communication channels are needed to implement overlapped communication: One to post current IO-operations on and another one to poll for previous posted IO-operations as described above.
- Size limitation per IO-operation: 8mb-1byte: for large communication sizes this will produce a lot of overhead (several IO-operations). Stencil code algorithms might be able to run on such a system without any changes, however especially modern machine learning applications might need to use 32-64MB.
- Single RDMA status request: it would be much more efficient to request the status of an array of communication-identifier at once. Single RDMA status requests produce a lot of context switches and other overhead.
- The RDMA functionality should be implemented mostly in userspace and not in kernelspace.
- Atomic operations in UNIMEM cannot use the same memory segments as RDMA IO-operations. Since most of the GPI based applications run atomics and RDMA operations out of the same memory segment, this limitation does not allow pre-compiled GPI-2 binaries to run on top of UNIMEM.
- Atomic operations on UNIMEM need some kind of relationship between affected processes. Access-tokens are distributed over a special startup mechanism which is not compatible with mpi_run or gpi_run. Here a startup procedure similar to mpi_run or gpi_run is needed to establish a working environment like on any other computing system today.
- The remote UNIMEM prototype systems are not yet stable enough to do intensive tests. To get significant performance data for evaluation, a stable UNIMEM environment is needed.

The above mentioned limitations will be discussed with the FORTH group.

Figure 9 and Figure 10 show first measurements on the remote UNIMEM prototype, namely the GPI-2 bandwidth and latency. The data have been taken on the TRENZ board with the UNIMEM software stack (kernel driver and user-level library). For the latency and bandwidth measurement for each measured point the average of 1000 samples has been taken. For latency a classical ping-pong mechanism was employed, completion time is the time that the pong has been received. For the bandwidth measurement the completion has been signalled by a signal received by UNIMEM. The data values are consistent with the values reported in the MPI implementation in Section 3.2.3. However our measurements start with smaller message sizes. The latency and bandwidth can be compared with measurements taken with Infiniband FDR a few years back both for GPI-2 (and MVAPICH2-1.9). The plots can be found at the GPI web page[1] . The bandwidth for Infiniband FDR saturates at messages sizes about 4kbytes at a bandwidth of 6000 MB/s. The bandwidth is about a factor 50 higher than measured on the Trenz board. The latency of small messages with Infiniband FDR is about 1microsec for 2bytes up to about 2microsec for 2kbytes. This is about a factor of 250 faster than the latency measured with GPI over UNIMEM on the Trenz board. Further optimized capabilities with the GPI implementation are going to be implemented during the coming months.

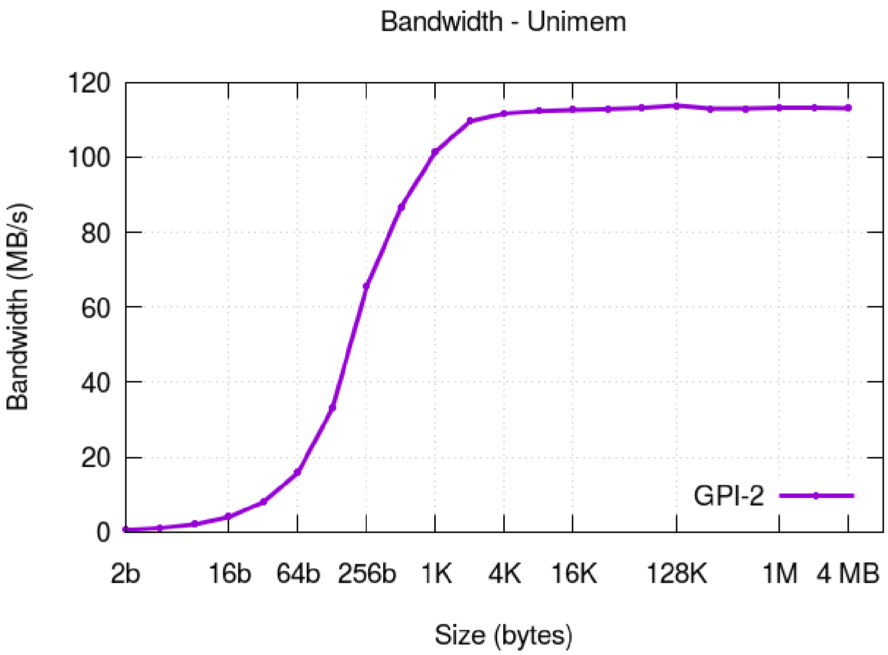

Figure 9: GPI-2 bandwidth on UNIMEM Prototype system (Sockets over Unimem)

[1] GPI web page: http://www.gpi-site.com/gpi2/benchmarks/

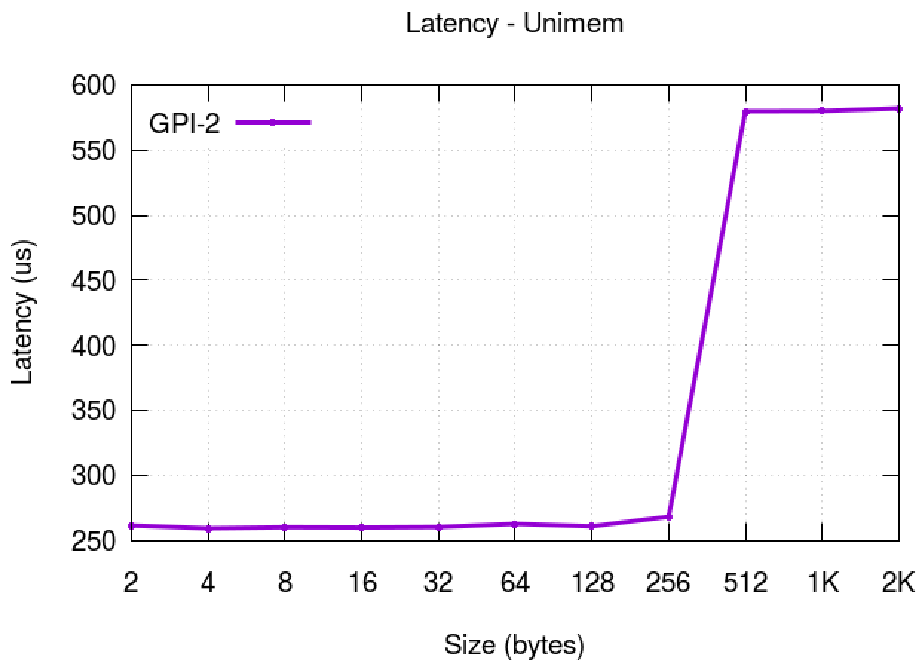

Figure 10: GPI-2 latency on the UNIMEM Prototype system (Sockets over UNIMEM)

3.1.6 FPGA Prototype System

The Exanode System drags it compute performance out of Field-Programmable-Gate-Arrays (FPGA) to be competitive with state-of-the-art architectures configured with e.g. GPUs or other Accelerators. We are currently in the phase of setting up a small test-system consisting of Xilinx Ultrascale+ FPGAs and ARM 64bit cores in one package. This platform will be used to implement a GPI-Interface that is able to offload compute kernels to the FPGA and to monitor the external program execution. We will also evaluate different development environments for these FPGA kernels to be able to select the best workflow that integrates optimal into the GPI Build-Environment.

3.2 MPI

BSC has proposed to the Consortium that the high-level architectural design of the MPI port over UNIMEM should lie on the recently-emerged OpenFabrics Interfaces (OFI)[2], an open generic low-level networking standard for HPC. This is in accordance with current efforts in the major MPI implementations (Intel MPI, MPICH, Open MPI). The effort is performed to overcome the well-known performance limitations of TCP, so we expect to improve upon an MPI over TCP implementation and, when finished, offer minimal overhead with respect to direct use of UNIMEM.

3.2.1 State-of-the-Art MPICH

Currently MPICH, the MPI implementation decided to be the primary target in this project, is undergoing a major code rewriting on its Channel layer, moving from CH3 to the new CH4, with major improvements on scalability and latency. Part of this effort is aimed at better exploiting HPC networking capabilities, by providing full communication semantics to the low-level network interface. This enables highly efficient MPI communications on top of OFI. Currently in alpha 2 version and already passing most of the wide MPICH test suite on x64

[2] https://ofiwg.github.io/libfabric

architectures, a stable release is foreseen to be announced in November 2017 during the SC17 conference.

3.2.2 MPI over UNIMEM Architecture

BSC is developing an OFI port (called a "provider") on top of the UNIMEM API. Figure 11 shows the major high-level architectural components of the state-of-the-art MPICH implementation described in the above section, and the light blue box represents the main component that BSC is developing: the UNIMEM OFI provider. Figure 12 depicts a future more optimised implementation of the UNIMEM version of MPI, with optimised collective communications that extend the OFI API. At this point there is no expectation of having to modify any layers of the MPI stack above OFI.
Developing an OFI provider instead of an integrated solution has the advantage that the UNIMEM OFI provider may be usable by other MPI implementations and potentially even other runtimes implementing the OFI API. BSC is currently targeting development under MPICH because of its current know-how and established contacts with the developing group, and the portability premise will be checked using Open MPI on a more advanced stage.

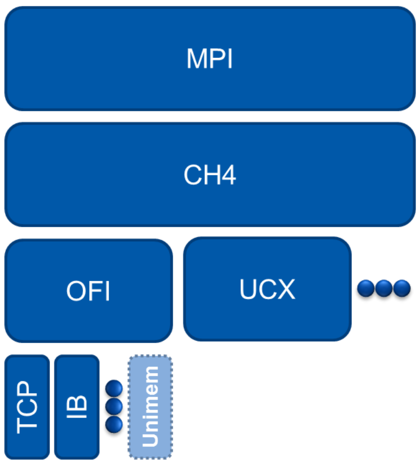

Figure 11: State-of-the-art MPICH design, showing UNIMEM OFI provider

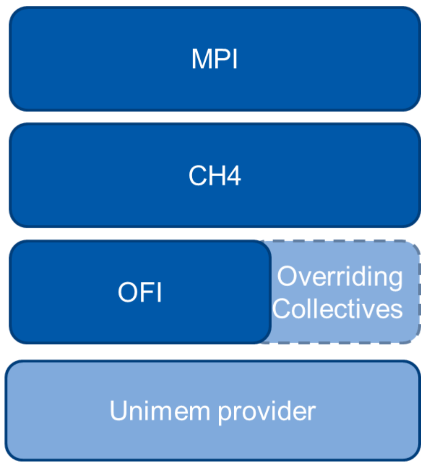

Figure 12: Future optimized UNIMEM MPI implementation with overriding collectives

3.2.3 Development Approach and Current Status

For practical reasons, rather than starting the implementation of the UNIMEM OFI provider from scratch, BSC chose to start development based on the OFI TCP/sockets provider, and progressively replace TCP/sockets communication with communication over the native UNIMEM API. This allows incremental development, with full functionality always provided by TCP/sockets, while taking advantage of improved performance for the features that have been optimized natively over the UNIMEM API.

Currently the OFI provider is able to transfer MPI_Send data payloads over native UNIMEM. Discussions between BSC and FORTH resulted in improvements to the UNIMEM API, which will benefit MPI and the other runtimes/communication libraries:

(1) Until July 2017, the prototype provided only a single buffer allocation registered with the communication hardware. This was fixed in July 2017, and the change was communicated point-to-point to the engineer at BSC when the issue was discussed at the end of August 2017. This will improve performance for MPI and the other runtimes

and communication libraries by the end of the project, but the change arrived too late to incorporate into the codebase discussed in this deliverable.

(2) The previous version of the UNIMEM API, which was based on the cDMA hardware block, had no memory registration API, which prevents registering user-provided memory buffers. An updated version of the API takes advantage of new features of the zDMA hardware block on the UltraScale+ SoC. This changes the semantics of the UNIMEM APIs, and the more powerful APIs will result in a zero-copy MPI implementation, improving MPI usability and potentially performance. This requires some redesign of the UNIMEM OFI provider to take advantage of the new APIs. This does not affect the schedule for delivery of the final optimized MPI library by M36.

These limitations are under continuing discussion with FORTH. In addition, there are still issues with the stability of the remote UNIMEM prototype. To get significant performance data for evaluation, a stable UNIMEM environment is needed.

3.2.4 Preliminary results

Figure 11 and Figure 12 show our first performance results on the Juno prototype, comparing the use of MPICH over an OFI TCP provider with our prototype of the UNIMEM OFI provider (note that TCP support is in turn implemented over UNIMEM). To avoid system noise, every measurement represents the average of 50 round-trip message exchanges. These experiments are executed 30 times and the average is represented.

Figure 11 shows roundtrip latency for small message sizes. We can see that the pure TCP sockets provider exposes in general lower latency than our under-development UNIMEM provider. We expect to overcome this limitation once we move from TCP-based notification to using the Mailbox functionality for this purpose. The figure shows that this penalty is overcome by the much faster data payload transfers for sufficiently large transfers, starting at 128 KB. We have not yet identified the reasons for the high latency exposed by the TCP provider at 1 KB data payload.

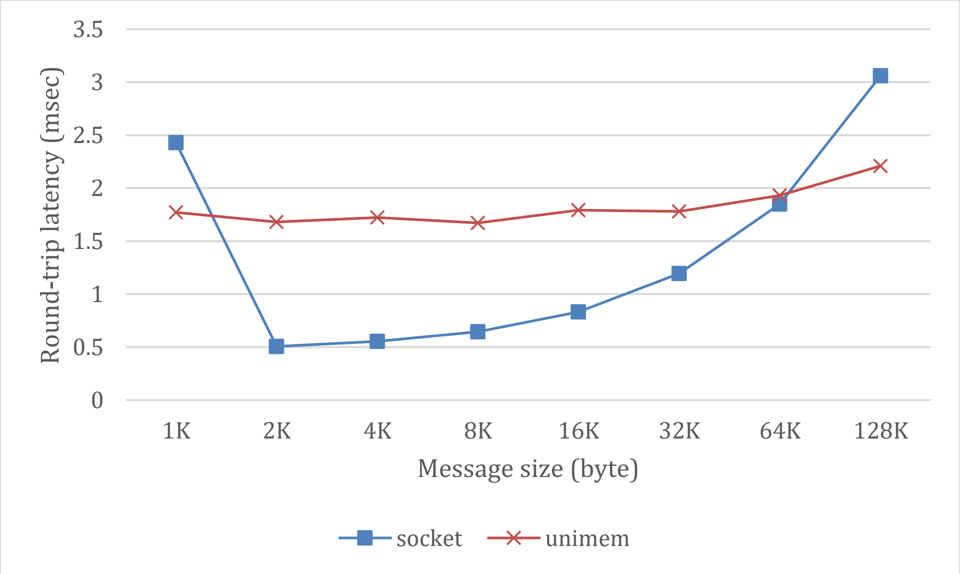

Figure 11: Roundtrip latency for small message sizes.

Figure 12 shows one-way throughput (defined as 1 / roundtrip latency * 2) for large message transfers. We can see how TCP flattens at 100 MB/s with a 2 MB data payload while the UNIMEM provider is consistently over 3x faster, yielding up to about 370 MB/s. Note that TCP implementations over low-level high-performance networking APIs are widely known to pose significant overheads due to factors such as additional memory copies, context switches to Operating System kernel calls, and the intrinsics of the TCP protocol itself.

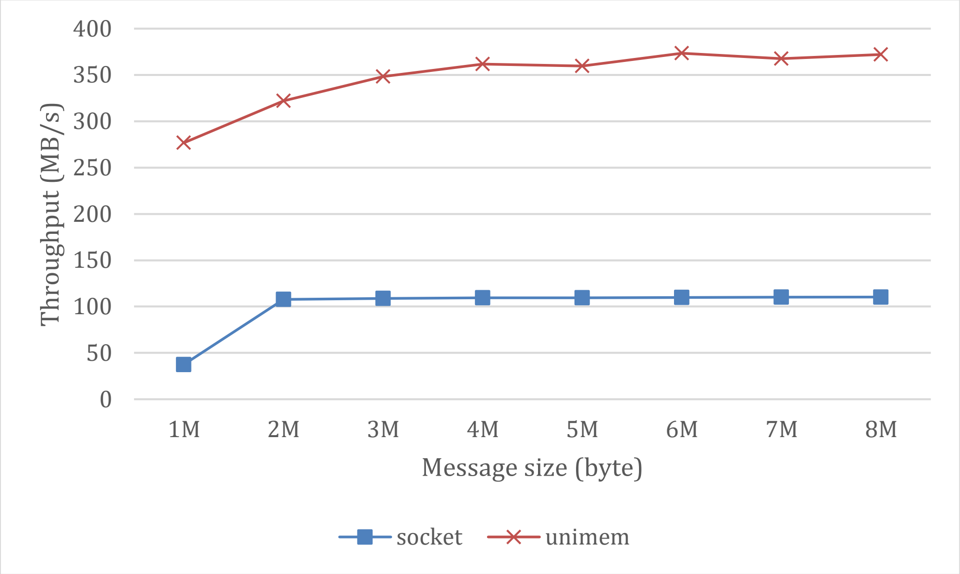

Figure 12: Throughput for large data payloads.

In summary, the preliminary UNIMEM implementation of MPI has higher latency than the baseline MPI over sockets, for small messages, but this penalty is overcome for messages larger than about 64 KB). The preliminary UNIMEM implementation achieves greater than 3x the bandwidth for messages over about 2 MB, reaching 370 MB/s.

We have not yet been able to understand the precise reasons for the excessive latency, but we intend to improve the implementation using the new version of the UNIMEM RDMA API, which makes use of the zDMA hardware block, and has slightly different semantics (providing user-level initialization of RDMA transfers), which will avoid the need for the MPI library to allocate specific buffers for RDMA transfers, enabling a zero-copy implementation. This will improve performance but it potentially requires major modifications to the MPI implementation. We will also make use of the Mailbox API. This is expected provide a complete and high-performance OFI provider on top native UNIMEM only, covering the entire MPI 3.1 Standard.

3.3 Further Requirements of the runtimes and communication models on the underlying platform

The runtime systems and programming models have specified their requirements on the underlying UNIMEM system in their respective sections. Here we collect the list of requirements:

3.3.1 Requirements of OmpSs

OmpSs will use the underlying MPI programming model which will then connect to UNIMEM. Therefore the requirements of OmpSs itself within the ExaNoDe project on UNIMEM are modest, however the MPI requirements need to be incorporated.

3.3.2 Requirements of OpenStream

- OpenStream uses UNIMEM directly. To work efficiently OpenStream requires the support of RDMA and atomics by UNIMEM.
- For OpenStream it is important to be able to test on larger and advanced hardware systems to study the impact of UNIMEM on the OpenStream work stealing algorithm which ensures load balancing. OpenStream expects that an asymmetry can occur due to the success rates of local and remote atomic operations. OpenStream will complement the work-stealing algorithms by sending tasks that require remote data to be executed on the node where most of their inputs are located to reduce communication overall.

3.3.3 Requirements of GPI

- Single memory segment: all GPI- based applications are using at least two or more memory segments during runtime to switch between communication buffers and computation segments (double buffering approach for asynchronous programs). This limitation has been present in the emulation library but is resolved in the UNIMEM software and still needs to be tested.
- Size limitation of the memory segment: 256mb. This size is much too small for real world applications, which typically use up to 16GB to 32GB per node, doing overlapped computations and communications.
- Single communication channel: at least two communication channels are needed to implement overlapped communication: One to post current IO-operations on and another one to poll for previous posted IO-operations as described above.
- Size limitation per IO-operation: 8mb-1byte: for large communication sizes this will produce a lot of overhead (several IO-operations). Stencil code algorithms might be able to run on such a system without any changes, however especially modern machine learning applications might need to use 32-64MB.
- Single RDMA status request: it would be much more efficient to request the status of an array of communication-identifier at once. Single RDMA status requests produce a lot of context switches and other overhead.
- The RDMA functionality should be implemented mostly in userspace and not in kernelspace.
- Atomic operations in UNIMEM cannot use the same memory segments as RDMA IO-operations. Since most of the GPI based applications run atomics and RDMA operations out of the same memory segment, this limitation does not allow pre-compiled GPI-2 binaries to run on top of UNIMEM.
- Atomic operations on UNIMEM need some kind of relationship between affected processes. Access-tokens are distributed over a special startup mechanism which is not compatible with mpi_run or gpi_run. Here a startup procedure similar to mpi_run or gpi_run is needed to establish a working environment like on any other computing system today.
- The remote UNIMEM prototype systems are not yet stable enough to do intensive tests. To get significant performance data for evaluation, a stable UNIMEM environment is needed.

3.3.4 Requirements of MPI

- MPI requires more than a single buffer allocation registered with the communication hardware to be able to provide a low-latency zero-copy approach.
- MPI requires a memory registration API to be able to register user-provided segments.

3.4 Suitable ExaNoDe Mini-apps

All ExaNoDe mini-apps considered in D2.2 *(Report on the ExaNoDe mini-applications)* [18] are either implemented in MPI or they have MPI versions: Abinit, BQCD, HydroC, KKRnano, MiniFE and NEST. The four chosen applications (BQCD, HydroC, KKRnano and MiniFE) are therefore suitable for evaluation of the MPI port. Moreover, the MPI port will be developed in consultation with the application partners of ExaNeSt and will be made available to them for implementation and performance evaluation using production scientific applications, in particular those from INAF and INFN.

4 Power and thermal control

Building on the definition of the thermal capping optimal policy, we focused on the realization of the power capping problem which ease the need of extracting the thermal model from the target architecture.

Several approaches in the literature have proposed mechanisms to constrain the power consumption of large-scale computing infrastructures. These can be classified into two main families. Approaches in the first class use predictive models to estimate the power consumed by a job before its execution. At job scheduling time this information is used to allow into the system jobs that satisfy the total power consumption budget. Hardware power capping mechanism like RAPL (Running Average Power Limit)are used to ensure that the predicted budget is respected during all the application phases and to tolerate prediction errors in the job's average power consumption estimation [19] [20] [21]. Approaches in the second class distribute a slice of the total system power budget to each active computing element. The per-compute element power budget is ensured by mean of hardware power capping mechanism like RAPL. The allocation of the power consumption budget to each compute nodes can be done statically or dynamically [22] [23] [24]. It is goal of the run-time to trade off power reduction with application performance loss. GEOPM implement a plugin for power balancing to improve performance in power constraint systems reallocating power on sockets involved in the critical path of the application.
Authors in [25] quantitatively evaluated RAPL as a control system in term of stability, accuracy, settling time, overshoot, and efficiency. They evaluate only the proprieties of RAPL mechanism without considering other power capping strategies and how can vary application workload.

State-of-the-art mechanism relies on a hardware mechanism to directly control the power consumption. In the ExaNoDe architecture this feature is not present and as consequence we constrain the power consumption by directly controlling the frequency of the cores. In the following section, we focus on the comparison with state-of-the-art related work, which has been done on an Intel platform, in order to compare with Intel RAPL.

4.1 HPC Architectures

HPC systems are composed of tens to thousands computational nodes interconnected with a low-latency high-bandwidth network. Nodes are usually organized in sub-clusters allocated at execution time from the system scheduler according to the user request. Sub-clusters have a limited lifetime, after which resources are released to the system scheduler. Users request resources through a batch queue system, where they submit applications to be executed. Even a single node can be split in multiple resources shared among users. The single indivisible units in a HPC machine are CPU, memory and possibly accelerators (GPGPU, FPGA, Many-core accelerator, etc.).
HPC applications typically use the Single Program Multiple Data (SPMD) execution model, where the same application executable is instanced multiple times on different nodes of the cluster; each instance works on a partition of the global workload and communicates with other instances to orchestrate subsequent computational steps. For this reason, a HPC application can be seen as the composition of several tasks executed in a distributed environment which exchanges messages among all the instances. Achieving high-performance communication on distributed applications in large clusters is not an easy task. The Message-Passing Interface (MPI) runtime responds to these demands by abstracting the level of network infrastructure using a simple but high-performance interface for communication that can scale up on thousands of nodes.

HPC machines are extreme energy consumers, and server rooms require a proportioned cooling system to avoid overheating situations. The extreme working conditions of this kind of machines brings a lot of inefficiencies in terms of energy and thermal control, that turn in computational performance degradation. Hardware power managers are becoming a fundamental to controlling power utilization using different strategies to reduce energy waste and, at the same time, assure a safe thermal environment.

4.2 Power Management in HPC Systems

Nowadays, operating systems can communicate with different hardware power managers through an open standard interface called Advanced Configuration and Power Interface (ACPI) [26]. In this work, we focus on ACPI implementation of Intel architecture, since most HPC machines (more than 86% in [27]) are based on Intel CPUs. Intel implements the ACPI specification defining different component states which a CPU can use to reduce power consumption. Today's CPU architectures are composed of multiple processing elements (PE) which communicate through a network subsystem that interconnect PEs, Last Level Cache (LLC), Integrated Memory Controllers (IMC) and other uncore components. Intel architecture optimizes ACPI using different power saving levels for cores and uncore components. The ACPI standard defines P-states to select DVFS operating points targeting the reduction of active power, while defines C-States the idle power levels. In our work, we consider only P-states to manage DVFS control knob, this because HPC applications do not manifest idle time during the execution.

Intel P-States show in Figure 13, defining a number of levels which are numbered from “0” to “*n*” where “*n*” is the lowest frequency and “0” is the highest frequency with the possibility to take advantage of Turbo Boost technology. Turbo Boost is an Intel technology that enables processors to increase their frequency beyond the nominal via dynamic control of clock rate.
The maximum turbo frequency is limited by the power consumption, thermal limits and the number of cores that are currently using turbo frequency. Since Haswell, Intel cores allow independent per-core P-State.

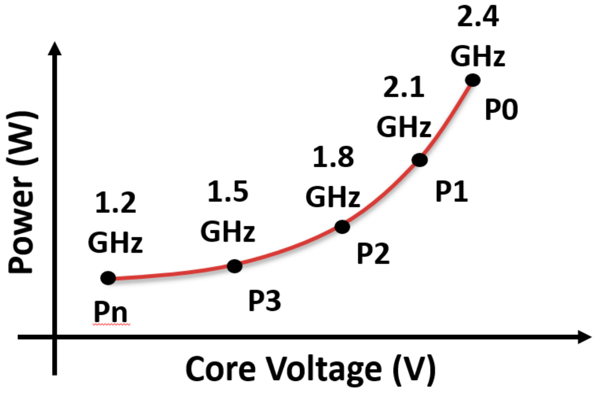

Figure 13: DVFS mechanism

Intel Power Management Driver: Intel P-States are managed by a power governor implemented as a Linux kernel driver. By default, on Linux system, Intel architectures are managed by a kernel module called “intel_pstate”. This driver implements a Proportional–Integral–Derivative (PID) feedback controller. The PID controller calculates an error value every 10 ms as the difference between a desired setpoint and the measured CPU load in that period. The PID controller acts to compensate this error by adapting the P-State value.
The PID internal parameters are defined with default values by the Intel driver but can be customized by the system administrator.

Inside “intel_pstate” driver only two governors are implemented: “powersave” (default) and “performance”. We will not describe in detail the operations of these governors because it is outside the scope of this work, but from a practical point of view, “performance” always maintains the CPU at maximum frequency while “powersave” can choose a different level depending of the machine workload. Hence, “powersave” tries to achieve better energy efficiency while “performance” tries to achieve the best performance at the expense of higher energy consumption.

Linux Power Management Driver: The “intel_pstate” driver does not support a governor that allows users to select per-core fixed frequency. Differently, the default power management driver of Linux “acpi-cpufreq” does it.
“acpi-cpufreq” is similar to Intel driver but implement a large set of governors which implement different algorithms. The available governors are:

1. **Powersave**; this governor differently from Intel driver, runs the CPU always at the minimum frequency.
2. **Performance**: runs the CPU always at the maximum frequency.
3. **Userspace**: runs the CPU at user specified frequencies.
4. **Ondemand**: scales the frequency dynamically according to current load. It is equivalent to the “powersave” governor of Intel driver [28].
5. **Conservative**: similar to ondemand but scales the frequency more gradually.

In our work, we use “userspace” governor to select fixed frequencies for all the duration of our benchmarks.

4.3 Hardware Power Controller

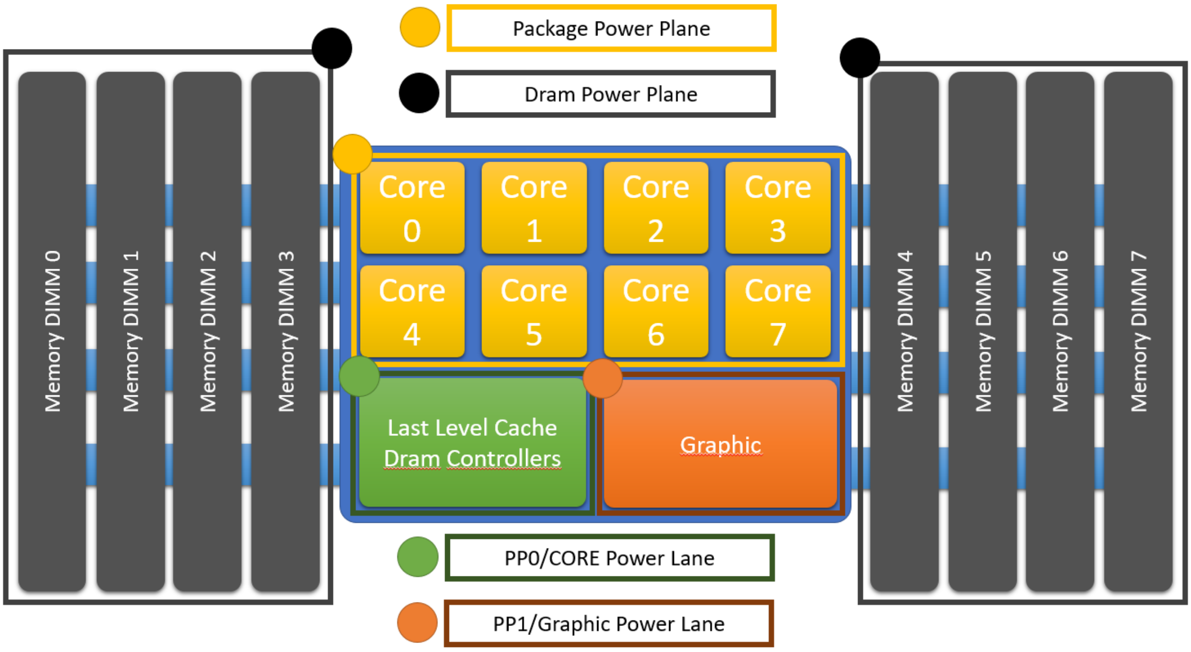

Figure 14: RAPL power domain

Today's CPU architectures implement reactive hardware controller to maintain the processor always under an assigned power budget. The hardware controller tries to maximize the overall performance while constraining the power consumption and maintaining a safe silicon temperature. Intel architectures implement in its CPU a hardware power controller RAPL depicted in Figure 14. RAPL is a control system, which receives as input a power limit and a time window. As consequent, RAPL continuously tunes the P-states to ensure that the limit is respected in the specified time window. RAPL can scale down and up core's frequencies when the power constraint is not respected overriding the selected P-states. RAPL power budget and time window can be configured writing a Machine Specific Register (MSR) on the CPU. Maximum and minimal values for both power budget and time window are specified in a read-only architectural register. Values for both power and time used in RAPL are represented as multiple of a reference unit contained in a specific architectural register. At the machine start-up, RAPL is configured using thermal design power (TDP) as power budget with a 10ms time window. RAPL also provides 32bit performance counters for each power domain to monitor the energy consumption and the total throttled time. RAPL implements four power domains which can be independently configured:

1. **Package Domain**: this power domain limits the power consumption for the entire package of the CPU, this includes cores and uncore components.
2. **DRAM Domain**: this power domain is used to power cap the DRAM memory. It is available only for server architectures.
3. **PP0/Core Domain**: is used to restrict the power limit only to the cores of the CPU.
4. **PP1/Graphic Domain**: is used to power limit only the graphic component of the CPU. It is available only for client architectures due Intel server architectures do not implement graphic component into the package.

In the experimental result section, we focus our exploration on the package domain of RAPL controller because core and graphic domains are not available on our Intel architecture.
DRAM domain is left for future exploration works. We also tried to modify the time windows of package domain (which can be set in a range of 1ms to 46ms in our target system) to see its impact on application performance.
Our results show that this parameter does not lead to noticeable changes in the results obtained. For this reason, we report results only for the default 10ms time window configuration.

4.4 Architecture Target

In this work, we take as architecture target a high-performance computing infrastructure, which is a Tier-1 HPC system based on an IBM NeXtScale cluster. Each node of the system is equipped with 2 Intel Haswell E5-2630 v3 CPUs, with 8 cores with 2.4 GHz nominal clock speed and 85W Thermal Design Power (TDP, [29]). We selected this target system as it contains all the power management features (RAPL, per core DVFS, c-states) of future HPC computing nodes and can be used as a reference for future ARM systems.
Quantum ESPRESSO (QE) [30] is an integrated suite of computer codes for electronic-structure calculations and materials modelling at the nanoscale. It is an open source package for research in molecule dynamics simulations and it is freely available to researchers around the world under the terms of the GNU General Public License. Quantum ESPRESSO is commonly used in high-end supercomputers. QE main computational kernels include dense parallel Linear Algebra (LA) and 3D parallel Fast Fourier Transform (FFT). Moreover, most of application workload is based on LA and FFT mathematical kernels which makes our exploration work relevant for many HPC codes. In our tests, we use a Car–Parrinello (CP) simulation, which prepares an initial configuration of a thermally disordered crystal of chemical element by randomly displacing the atoms from their ideal crystalline positions. This simulation consists of a number of tests that must be executed in the correct order.

4.5 Monitoring Runtime

We developed a monitor runtime to extract system information synchronously with the application flow. The runtime is a simple wrapper of the MPI library where every MPI function of each process has been enclosed by an epilogue and a prologue function. We used the MPI standard profiling interface (PMPI), which allow us to intercept all the MPI library functions without modify the application source code. The runtime is integrated in the application at linked time. Hence, this Runtime can extract information distinguishing application and MPI phases as shows in Figure 15. The monitor runtime uses low-level instructions to access the Performance Monitoring Unit (PMU) with low overhead. We programmed per-core PMU registers to monitor frequency, CPI, and scalar/vector instructions retired. The monitor runtime can intercept a very high number of MPI calls of the application.

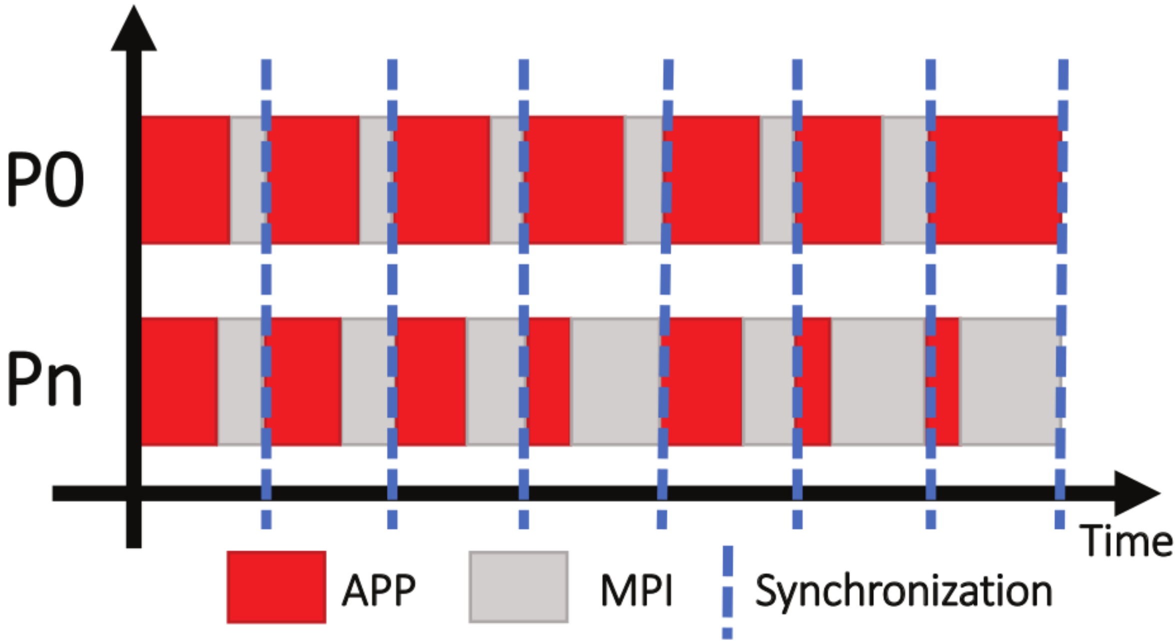

Figure 15: Monitor runtime

4.6 Methodology

We run QE–CP with a configuration of 16 MPI processes with a one-to-one bind to each core of our HPC node. We start by comparing different configurations of power capping in our test environment. Initially, we split the power budget in an equal manner on both sockets, we set a power consumption limit of 48 W on each socket, for a global power envelope of 96 W. This test shows that the core's frequencies on different sockets are heterogeneous, suggesting that the two sockets have different inherent power efficiency. To have the same frequency among all the cores, the tested computing node needs of 11.3% higher power on socket 0. As a consequence of this result, we run a set of benchmarks fixing the same frequency for all the cores while monitoring the power consumption of each socket. We use this per-socket power budget as power constraint to obtain the same frequency among all the cores. We execute again the tests using RAPL to impose these per-socket power caps and leave RAPL decides the actual frequency.

	Power		Frequency			Execution Time		
	DVFS	RAPL	DVFS	RAPL	DVFS vs RAPL	DVFS	RAPL	DVFS vs RAPL
1.5 GHz	$95.56W$	$94.81W$	$1499MHz$	$1766MHz$	-15.11%	$311.43sec$	$328.16sec$	5.10%
1.8 GHz	$111.86W$	$110.63W$	$1797MHz$	$2144MHz$	-16.22%	$274.11sec$	$274.42sec$	0.11%
2.1 GHz	$122.87W$	$120.71W$	$2094MHz$	$2323MHz$	-9.86%	$247.60sec$	$254.59sec$	2.75%
2.4 GHz	$134.44W$	$131.32W$	$2392MHz$	$2476MHz$	-3.37%	$231.19sec$	$239.65sec$	3.53%

Table 2: Quantum ESPRESSO - Power Capping

Table 1 shows the results of our set of experiments using different levels of power caps.
In the first column, there are reported the target frequencies used to extract the power limits specified in the second column. Second and third columns show the sum of power consumption of both sockets using DVFS and RAPL mechanisms for power capping. We can see that the power consumption is the same, so the power cap is respected and the tests are comparable. In the frequency columns are reported the average frequencies for the entire application and among all the cores. These columns show that RAPL has an average frequency of 11.1% higher than DVFS but, if we look at the execution time (reported in next columns), DVFS has a lower execution time, in average 2.9% faster than RAPL.
In the next sections, we will explore why DVFS power cap has a lower execution time respect to RAPL which, in contrast, has a higher average frequency.

4.7 System Analysis

Figure 16 shows a time window of the system-aware monitoring tool for both the power capping mechanisms while QE–CP iterates on the same computational kernel. The test reports the case of a power constraint relative to 1.5 GHz for DVFS and RAPL power cappers. So, the results are comparable directly.

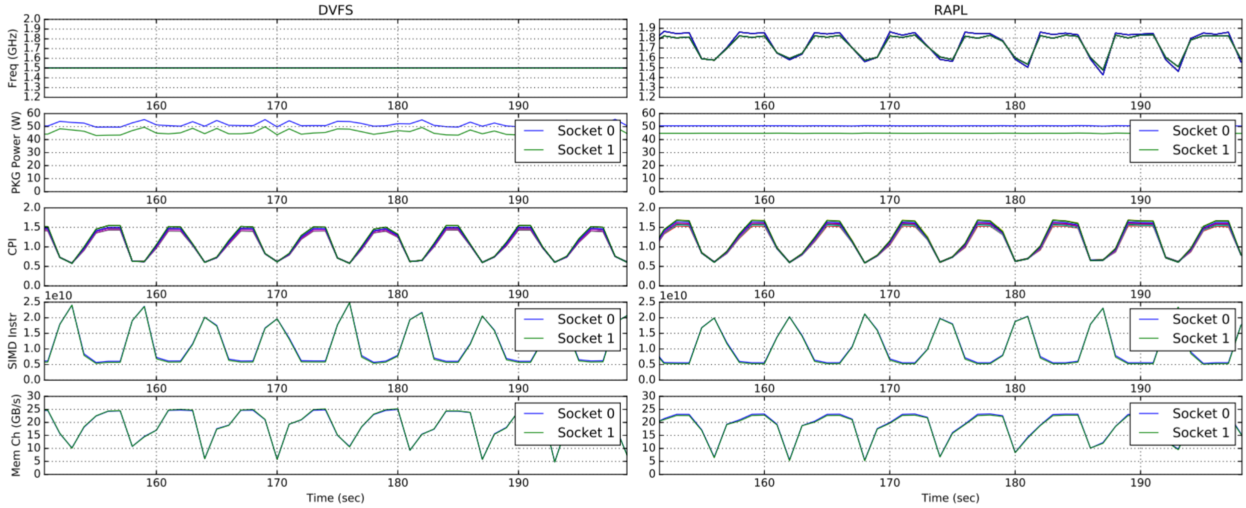

Figure 16: Comparison of DVFS and RAPL (Time window of 50 seconds)

First, we can check the correct behaviors of power capping logic by looking at the core's frequencies and package power consumption (first two top plots). In the DVFS plot on the left part of Figure 16, core's frequencies are fixed at 1.5 GHz while package power consumption floats around the average value as effect of the different application phases. In contrast, RAPL (on the right) maintains constant the power consumption for both the sockets while core's frequencies changes following the current application phase. Table 1 reports a similar average power consumption for both the two cases, thus the power cappers are working as expected. Both benchmarks show a lower CPI when the memory bandwidth is low and SIMD instructions retired are high. In these phases, RAPL has lower frequency than the DVFS case as effect of the higher power demand of SIMD instructions. On the other hand, RAPL assigns higher frequencies than DVFS when CPI is high and this happens when the application is moving data from/to memory as proved by the high memory traffic/bandwidth reported by the "Mem Ch [GB/s]" plot. In these phases, the number of SIMD instructions retired are lower and, as already pointed out and shown in the RAPL plot, the core's frequencies selected by RAPL increases above average due the higher power budget. However, increasing core's frequencies when the application is memory bound does not reflect in a consequent performance gain due the higher CPI and sub-linear dependency of application speed-up with frequency in these phases.

Hence, DVFS power capper is more efficient of RAPL (shorter execution time) for two reasons: i) DVFS executes with higher instruction per seconds when application has high SIMD instructions density. ii) RAPL instead reduces the core's frequency in the same phase to avoid excessive power consumption. On the contrary, RAPL increases the frequency during memory bound phases obtaining a similar average power as the DVFS case.

4.8 Application Analysis

In this section, we monitor the system using the application-aware monitoring runtime.
This runtime is able to recognize application phases marked by global synchronization points as depicted inFigure 16. In the Figure 17, Figure 18 and Figure 19 are reported the average values of performance counters of RAPL power capper divided into frequency operational intervals.
In Figure 17 is depicted the amount of time for computation (APP) and for communication (MPI) phases. Figure 18 shows the time gain for the DVFS power capper respect to RAPL power capper integrated on a given RAPL frequency range. Values are in seconds. Negative values mean seconds of application execution time reduction saved by DVFS with respect to RAPL power capper. Figure 19 shows the values for CPI and SIMD instructions retired for the same phases that RAPL executes at a given frequency.

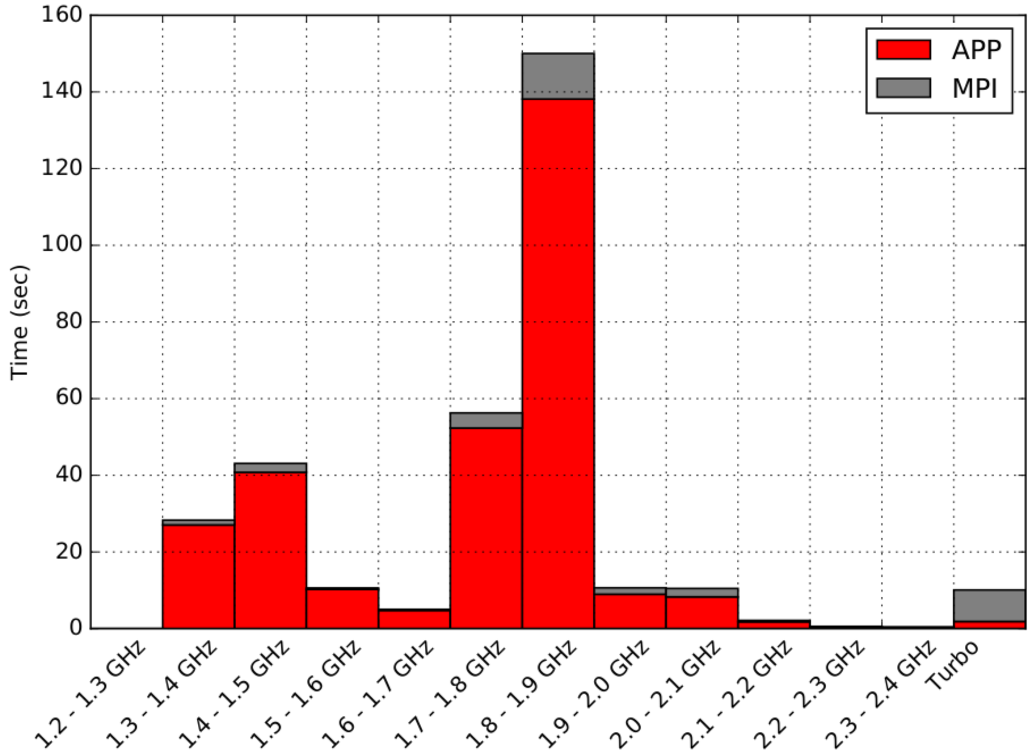

Figure 17: Sum of MPI and application time grouped by interval frequencies

To explain the behavior that characterizes DVFS and RAPL power cappers, we need to look at all the three plots together. Starting fromFigure 17, we can recognize that in the frequency intervals 1.3 to 1.5 GHz and 1.7 to 1.9 GHz, the DVFS power capper obtains its highest speed up. The first gaining interval is justified by the high number of SIMD instructions retired and by the lower CPI with respect to other application phases. Indeed fromFigure 18, we can notice that most of the SIMD instructions are executed with these lower frequencies by RAPL. In the interval 1.7 to 1.9 GHz, the CPI is higher and the SIMD instructions retired are not negligible. From Figure 19, we can recognize that most the application time is spent in this frequency range. With a lower SIMD instructions density respect to the 1.3 to 1.5 GHz interval. Hence, high CPI, low density of SIMD instructions and high frequency suggest memory bound phases as shown by previous section. Interesting these phases runs at higher frequency than the DVFS but leads to a performance penalty. This suggests side effects of high frequency in terms of memory contentions.

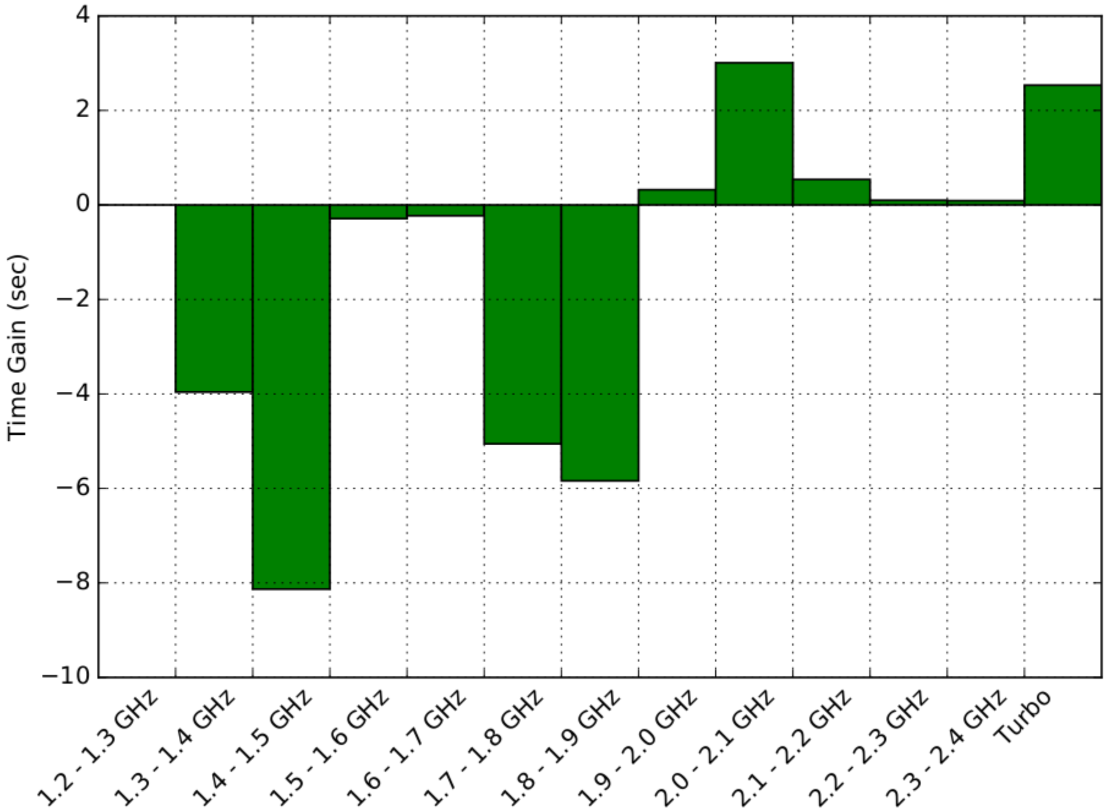

Figure 18: Time gain of DVFS w.r.t RAPL grouped by interval frequencies

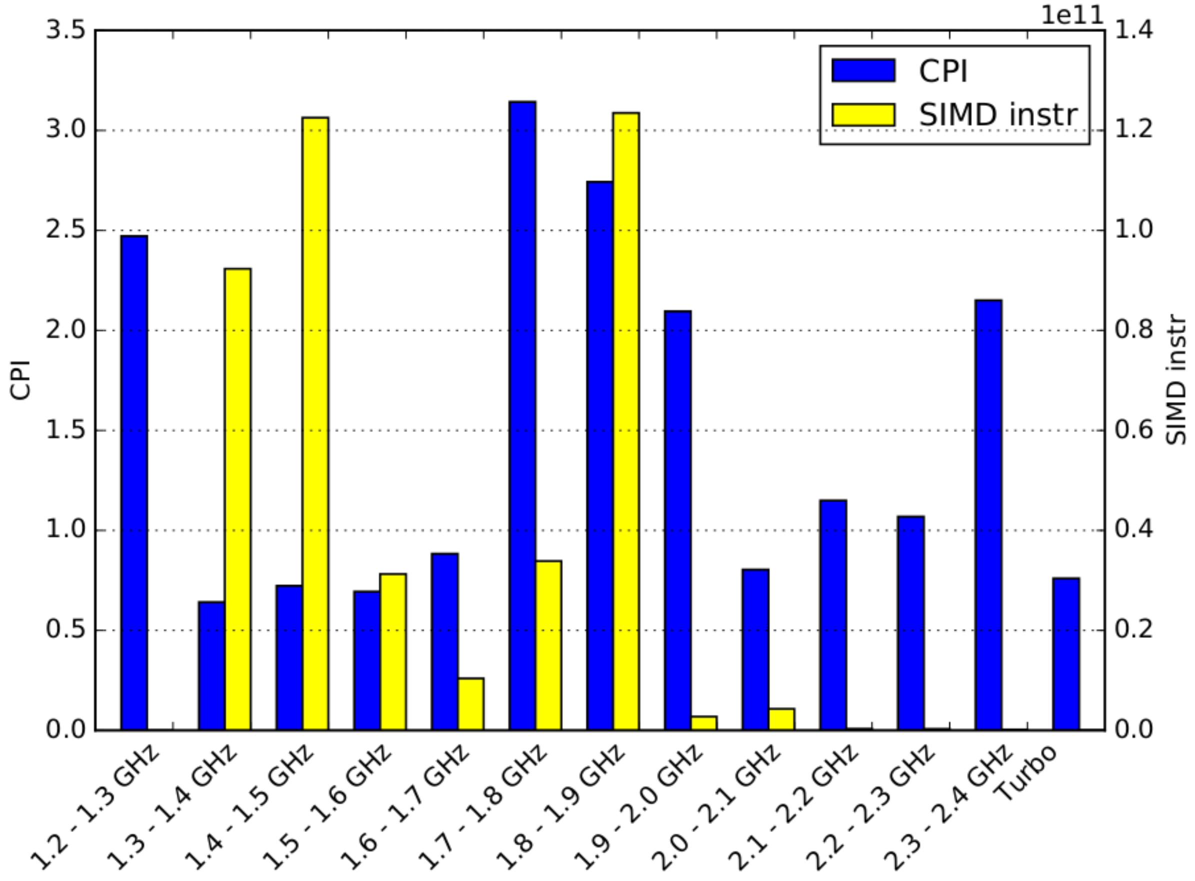

Figure 19: Average CPI and number of AVX instructions retired on different interval frequencies

In the interval 2.0 to 2.1 GHz, RAPL has a performance gain with respect to DVFS. This behavior is explained by the CPI and the number of SIMD instructions retired during this phase. In this interval, RAPL has a low CPI and does not perform SIMD instructions, so this phase scales its execution with the frequency. RAPL can dynamically manage the available power budget made available by the low SIMD instructions and can increase the frequency. This leads to a consequent performance increment.

In the turbo frequency interval, RAPL performs better than DVFS as it is a MPI reduction phase where only the root process is active. During the reduction, all the processes except the root MPI remain in a barrier to wait the termination of the root. This is explained by the high MPI runtime time (Figure 18) present at this frequency interval. Hence, RAPL can use the power budget released by the processes in barriers to speed up the root process leading to a performance gain.

4.9 Impact on the power and thermal runtime support

This analysis shows that DVFS based power capping is competitive w.r.t. hardware based mechanisms which suffer from being application agnostic and designed for short-time windows power cap (ms/s).

5 Concluding Remarks

This deliverable has described the runtime systems (OmpSs and OpenStream) and communication libraries (GPI and MPI) being integrated with the ExaNoDe hardware. Each runtime system and library is described with a focus on the design of the ExaNoDe software implementation and preliminary results. Work is ongoing, and the final versions will be in D3.3, which will be delivered in M36.

The strategies for the runtime systems and communication libraries are summarized:

- The OmpSs Communication layer (Nanos6 runtime system) decouples network transfers from other components. This allows to add a communication layer in a modular way. For the ExaNoDe project MPI has been chosen as underlying communication layer which will explicitly interface with UNIMEM.
- OpenStream directly couples to UNIMEM. The runtime fully controls the locality and memory allocation and data placement. Its data flow execution model with input and output buffers matches the UNIMEM memory model. OpenStream uses OpenCL to exploit FPGAs and integrates the EcoSCALE high-level synthesis toolchain.
- Dynamic load balancing has been implemented as a dynamic load balancing library on top of UNIMEM. It relies on remote atomic operations provided by UNIMEM for which an emulation has been developed which is integrated with the FORTH RDMA emulation library.
- GPI has identified UNIMEM dependent building blocks which are directly coupled to UNIMEM. An emulation library of the interconnect has been used to test the functionality on a standard x86_64 SMP system without the need to have real prototype hardware available on site. This has been an important step in a rapid integration of UNIMEM because the remote hardware platform has been too unstable for integration work and tests. All dependent modules have been implemented on top of the emulation framework and early tests were successful and the modules are being ported to the socket layer of UNIMEM.
- For the work on MPI the MPICH implementation is being used. Current work has focussed on providing full communication semantics on low-level network interface OFI. An OFI port is being built on top of the UNIMEM API connecting MPI to the UNIMEM model.

Limitations of UNIMEM have been discovered when customizing the implementations to UNIMEM and are under discussion with FORTH. We summarise the main points:

- Until July 2017, the prototypes had a bug when using more than one buffer allocation registered with the communication hardware, and no memory registration API was not available. There was also an issue that the UNIMEM API incorrectly specified that only one buffer could be registered at a time. This prevented registering user-provided memory buffers preventing in turn a low-latency zero-copy approach. The limitation is already resolved in the newest UNIMEM software and needs to be tested.
- A parallel startup mechanism like mpi_run or gpi_run is still not available yet and standard tools and scripts cannot be used on UNIMEM. In addition, environment settings/variables and command line arguments must be communicated to the remote node and setup correctly before a process inside a parallel topology can start. We are in the process of defining an interface that fulfils the requirements for GPI and MPI to start up remote processes.

At the moment measurements of simple latency and throughput times show that the UNIMEM implementation has to be improved to be competitive.

All runtime systems and communication APIs will support FPGA accelerators at the end of the project. Whereas OmpSs will leverage results from the AXIOM project building on HLS, OpenStream will leverage results from the EcoSCALE project building on the HLS support of the EcoScale toolchain. GPI will setup a small test system consisting of Xilinx Ultrascale+ FPGAs and ARM 64-bit in one package to test their FPGA support.

Finally, this deliverable describes other runtime support, specifically regarding thermal and power management:

- The ExaNode hardware does not provide hardware mechanisms to control power consumption, so the power and thermal control in the scope of the project will directly control the frequency of cores to optimise the power reduction while minimizing the application performance loss. With the MPI profiling tool of Quantum ESPRESSO as a reference application a DVFS (Dynamic Voltage Frequency Scaling) based power capping approach has been tested. The main result is that the DVFS-based power-cap is efficient and has shown competitive results with respect to hardware based power and thermal control mechanisms.

These technologies will be made available and potentially integrated into the optimized implementations of GPI, OmpSs, OpenStream and MPI.

6 Future Work

In year 3 of the ExaNoDe project the focus will be on discussions of further improvements of the UNIMEM library with FORTH, testing on real prototypes and the implementation of the mini applications. Specifically the following work is planned:

- OmpSs: Currently OmpSs uses MPI as underlying communication layer. The integration of UNIMEM will be done in the MPI layer. In future work (either in ExaNode or in the EuroEXA project) the data transfer messages can be eliminated on platforms implementing the UNIMEM architecture, while maintaining software compatibility with traditional distributed memory clusters.
- OpenStream: The next step is integrate the OpenStream environment with the EcoSCALE high-level synthesis toolchain to take advantage of the Xilinx Ultrascale+ FPGAs that will be available on the ExaNoDe system. The design of the current implementation was chosen to maximise the flexibility of the OpenStream framework so that FPGAs can be integrated in the OpenStream resource model and scheduler.
- GPI: We are building a small test-system consisting of Xilinx Ultrascale+ FPGAs and ARM 64bit cores in one package. This platform will be used to implement a GPI-Interface that is able to offload compute kernels to the FPGA and to monitor the external program execution. We will also evaluate different development environments for these FPGA kernels to be able to select the best workflow that integrates optimal into the GPI Build-Environment.
- MPI: Further optimized capabilities of the MPI Standard are going to be implemented during the coming months on top of a new version of the UNIMEM RDMA API, zDMA, along with the use of the Mailbox API, with the target of providing a complete and high-performance OFI provider on top native UNIMEM only, covering the entire MPI 3.1 Standard.
- Power and thermal control: In the ExaNoDe architecture no hardware feature to control the power consumption is present and as consequence we constrain the power consumption by directly controlling the frequency of the cores. It has been shown that DVFS based power capping is competitive w.r.t. hardware based mechanisms which suffer from being application agnostic and designed for short-time windows power cap (ms/s). In PY3 these mechanisms will be ported on the ExaNoDe hardware.

7 References and Applicable Documents

[1] N. D. Kallimanis, “D3.6: Design of the ExaNoDe Firmware,” 2016.

[2] ExaNoDe, “D3.7: Operating System Support for ExaNoDe,” 2017.

[3] “EUROSERVER project, EU FP7 project 610456,” [Online]. Available: http://www.euroserver-project.eu.

[4] “ExaNeSt project, EU H2020 project 671553,” [Online]. Available: http://www.exanest.eu.

[5] “EuroEXA project, project id 754337,” [Online]. Available: http://euroexa.eu.

[6] “AXIOM project, EU H2020 project 645496,” [Online]. Available: http://www.axiom-project.eu.

[7] “EcoScale Project, H2020 ICT project 671632,” [Online]. Available: http://www.ecoscale.eu.

[8] D. Pleiter, “D2.1: Report on the ExaNoDe mini-applications,” 2016.

[9] J. M. Perez, R. M. Badia and J. Labarta, “A dependency-aware task-based programming environment for multi-core architectures,” in *IEEE International Conference on Cluster Computing ICCC*, 2008.

[10] J. M. Perez, V. Beltran, J. Labarta and E. Ayguade, “Improving the Integration of Task Nesting and Dependencies in OpenMP,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[11] “INTERTWinE Project, project id: 671602,” [Online]. Available: http://www.intertwine-project.eu/.

[12] A. Pop and A. Cohen, “OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs,” in *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[13] A. Drebes, A. Pop, A. Heydemann, A. Cohen and N. Drach, “Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management,” in *IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.

[14] N. M. Le, A. Pop, A. Cohen and F. Zappa Nardelli, “Correct and efficient work-stealing for weak memory models,” in *ACM SIGPLAN Notices*, 2013.

[15] A. Rodchenko, A. Nisbet, A. Pop and M. Lujan, “Effective barrier synchronization on INTEL Xeon Phi Coprocessor,” in *European Conference on Parallel Processing*, 2015.

[16] C. Simmendinger, M. Rahn and D. Gruenewald, “The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures,” in *Sustained Simulation Performance*, 2015.

[17] “UNIMEM Mechanisms on the Euroserver Discrete Prototype Gen2 (64-bit),” 2016.

[18] ExaNoDe, “D2.2 Report on ExaNoDe mini-applications”.

[19] A. Borghesi, A. Bartolini, A. Lombardi, M. Milano and L. Benini, “Predictive Modeling for Job Power Consumption in HPC Systems,” in *International Conference on High Performance Computing*, 2016.

[20] A. Borghesi, C. Conficoni, M. Lombardi and A. Bartolini, “MS3: a Mediterranean-Style Job Scheduler for Supercomputers - do less when it's too hot!,” in *IEEE International Conference on High Performance Computing & Simulation (HPCS)*, 2015.

[21] A. Sirbu and O. Babaoglu, “Predicting system-level power for a hybrid supercomputer,” in *IEEE International Conference on High Performance Computing & Simulation (HPCS)*, 2016.

[22] J. Eastep, S. Sylvester, C. Cantalupo, F. Ardanaz, B. Geltz, A. Al-Rawi, F. Keceli and K. Livingstone, “Global extendible open power manager: a vehicle for HPC community collaboration toward co-designed energy management solutions,” in *Supercomputing PMBS*, 2016.

[23] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz and B. R. de Supinski, “A run-time system for power-constrained HPC applications,” in *IEEE Conference on High Performance Computing, Networking, Storage and Analysis*, 2015.

[24] O. Sarood, A. Langer, A. Gupta and L. Kale, “Maximising throughput of overprovisioned HPC data centers under a strict power budget,” in *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[25] H. Zhang and H. Hoffman, “A Quantitative Evaluation of the RAPL Power Control System,” in *Feedback Computing*, 2015.

[26] E. J. Hogbin, “ACPI: Advanced Configuration and Power Interface,” 2015.

[27] T. Org., “Top 500 Supercomputer Sites,” 2017. [Online]. Available: http://www.top500.org.

[28] V. Pallipadi and A. Starikovskiy, “The on-demand govenor,” in *Linux Symposium*, 2006.

[29] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty and S. Jourdan, “Haswell: he fourth generation INTEL core processor,” *IEEE Micro*, vol. 34, no. 2, pp. 6-20, 2014.

[30] P. Giannozzi, G. L. Chiarotti, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni and I. Dabo, “Quantum ESPRESSO: a modular and open-source software project for quantum simulations of materials,” *Journal of Physics: Condensed Matter*, vol. 21, no. 39, 2009.