D3.1

Runtime systems (OmpSs, OpenStream) and communication libraries (GPI, MPI): Analysis of the hardware system characteristics and design of a preliminary software implementation

Workpackage:	WP3	Enablement of Software Compute Node	
Author(s):	Valeria Bartsch, Carsten Lojewski		FHG
	Antoniu Pop		UOM
	Vishal Mehta, Paul Carpenter		BSC
	Andrea Bartolini		ETHZ
Authorized by	Paul Carpenter		BSC
Reviewer	Loic Cudennec		CEA
Reviewer	Luca Benini		ETHZ
Reviewer	David Bull		ARM
Dissemination Level	Public (PU)		

Date	Author	Comments	Version	Status
2016-08-04	Paul Carpenter	Template draft to contributors	V0.0	Draft
2016-10-26	Paul Carpenter	First complete draft	V0.9	Draft
2016-10-31	Paul Carpenter	Incorporated internal review feedback for submission to European Commission	V1.0	Final
2017-03-14	Andrea Bartolini	Fixed typo in Formula (3e)	V1.1	Final

Executive Summary

In this deliverable, we describe the runtime systems (OmpSs and OpenStream) and communication libraries (GPI and MPI) being developed in the ExaNoDe project. 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 it having to be ported to the specific Unimem APIs defined in D3.6 [2].

OmpSs and OpenStream are extensions of OpenMP with new directives for offloading tasks. OmpSs uses directionality clauses on tasks and address-based tracking of data dependencies in the runtime system, and it supports heterogeneous devices such as GPUs and FPGAs. OpenStream has explicit dependencies in the source program marked using streams. Together, OmpSs and OpenStream explore two different trade-offs relating to performance and overheads vs. ease of programming. Both programming environments are being extended to leverage the Unimem architecture, with specific optimizations in the compiler (OpenStream) and runtime system (OmpSs and OpenStream).

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. As such, GPI is an appropriate communication library to benefit from and evaluate the Unimem architecture. MPI is the standard message-passing API supported by all serious HPC systems and employed by the vast majority of scientific applications. Despite its importance, the development of a Unimem-optimized MPI library has proceeded slowly in the first year of the ExaNoDe project. For this reason, we have transferred this activity from CEA to BSC in the project amendment.

Finally, this deliverable describes other runtime support, specifically regarding thermal and power management and runtime libraries for performance-critical primitives. These technologies will be made available and potentially integrated into the optimized implementations of GPI, OmpSs, OpenStream and MPI.

In summary, this deliverable provides the design of a preliminary software implementation for each of the runtime systems and libraries. Since the precise hardware characteristics of the final prototype are not yet known, preliminary design has proceeded based on the general characteristics of the Unimem architecture. Work is ongoing, and will be described further in D3.2, "*Runtime systems (OmpSs, OpenStream) and communication libraries (GPI, MPI): Advanced implementation customized for ExaNoDe architecture, interconnect and operating system,"* to be issued in M24 of the project.

Table of Contents

Table of Figures

Table of Tables

List of abbreviations

Term	Definition
API	Application Programmer Interface
BW (MPI)	Busy Waiting MPI
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
ISP	*i*-th Step Problem
MCTP	(Fraunhofer's) Multicore Thread Package
MPI	Message Passing Interface
MPSD / MPMD	Multiple Program Single/Multiple Data
NUMA	Non-Uniform Memory Access
OS	Operating System
OTC	Optimal Thermal Controller
PGAS	Partitioned Global Address Space
PoC	Proof of Concept (prototype)
RDMA	Remote DMA (Direct Memory Access)
RTM	Reverse Time Migration
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

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] and ExaNeSt [4]. 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 [2].

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.

The runtime systems and communication libraries are summarised in Table 1. OmpSs is a task-based programming model that extends OpenMP with new directives for asynchronous parallelism and heterogeneous devices such as GPUs and FPGAs. The OmpSs environment is built using the Mercurium source-to-source compiler and Nanos++ runtime system. Nanos++ supports SMPs, GPUs, FPGAs and clusters. In ExaNoDe, the cluster implementation of Nanos++ is being leveraged as the basis for efficient runtime support for offloading tasks across nodes on the Unimem architecture, with automatic management of data transfers and data locality. 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. This FPGA support will be leveraged and evaluated on the ExaNoDe Proof of Concept (PoC).

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. Whereas OmpSs uses directionality clauses on tasks and address-based tracking of data dependencies in the runtime system, 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.

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. As such, GPI is an appropriate communication library to benefit from and evaluate the Unimem architecture.

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. Specifically, the scientific applications in ExaNeSt will require an efficient implementation of MPI. As described in Section 3.2, the development of MPI has proceeded slowly in the first year of the ExaNoDe project. For this reason, we have responded by transferring this activity from CEA to BSC in the project amendment.

Finally, this deliverable describes other runtime support, specifically regarding thermal and power management and runtime libraries for performance-critical primitives. 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 [1]), 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
Use of heterogeneity	Explicit	Explicit	Automatic (runtime system)	Explicit?
Base language(s)	C, C++, FORTRAN	C, FORTRAN	C, FORTRAN, CUDA	C
Started year	1991	2006	2002 (GridSs, OpenMP influence)	2011
Licence	TBD	GPL	GPL	GPL
Anticipated method to exploit Unimem	Zero-copy message passing	Memory segments and one-sided communications using Unimem	Cluster implementation using Unimem hardware (with single-copy data transfers)	Unimem: one-sided communication with zero or one copy, depending on dynamic task placement decisions
WP2 Mini-app	All	GPI test suite, separate stencil kernel	MiniFE	HydroC, MiniFE and NEST

2 Runtime systems

2.1 OmpSs

This section was contributed by BSC.

2.1.1 Introduction to OmpSs

The OmpSs programming model aims to provide productive support for developing parallel applications on heterogeneous systems, with a focus on ease of use and performance. OmpSs is a combination of principles from OpenMP and StarSs [13]. It extends OpenMP with directives for task-based asynchronous parallelism supporting address-based data dependencies. It also has support for heterogeneous architectures, including multiple address spaces and multiple versions of the same task to target different types of device (e.g. processor cores, GPUs, FPGAs and/or Intel Xeon Phis).

The OmpSs environment, shown in Figure 1, is based on the Mercurium source-to-source compiler and Nanos++ runtime system. Mercurium translates the OmpSs directives (pragmas) into calls to the Nanos++ runtime API that create new tasks, wait on data, etc. The translated program is then compiled with the native C/C++/Fortran and CUDA/OpenCL compilers, and linked with the appropriate runtime libraries, including the Nanos++ runtime system. At run time, Nanos++ tracks dependencies to build a dynamic dependency graph, and it schedules ready tasks onto the available devices, taking into account data locality, and automatically performing data transfers into and out of software-managed data caches.

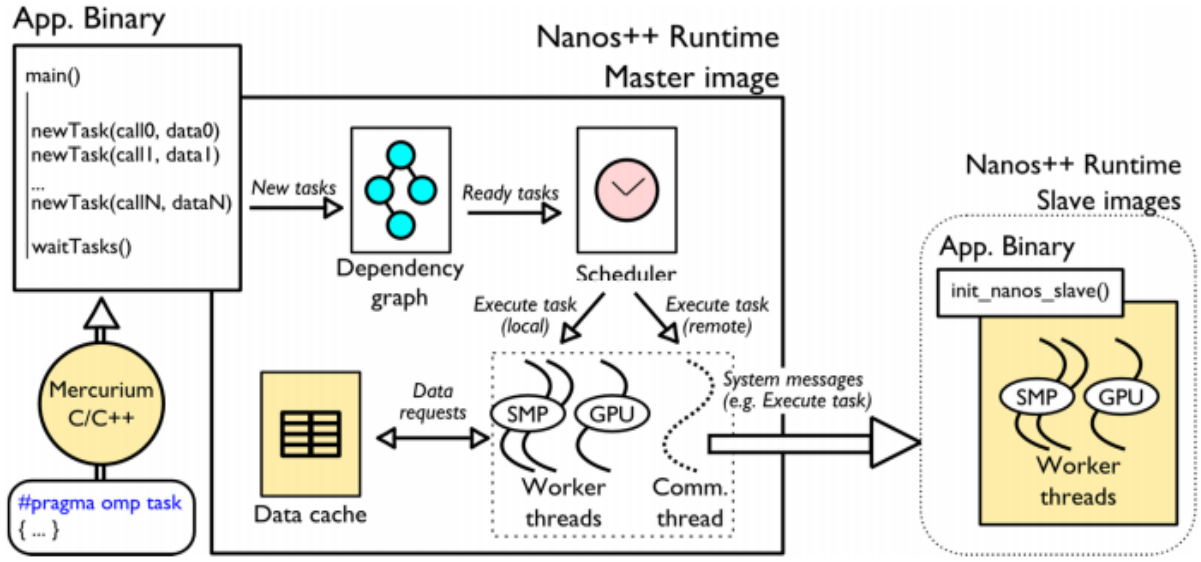


Figure 1: OmpSs toolflow: Mercurium source-to-source compiler and Nanos++ runtime

2.1.2 OmpSs on distributed memory clusters

OmpSs supports distributed memory clusters using the cluster version of Nanos++, which provides the shared memory abstraction to the user. Nanos++ for Clusters is implemented using a PGAS (Partitioned Global Address Space) API known as GASNet (Global Address Space Networking). GASNet is a language-independent low-level networking layer intended for implementing parallel global address space SPMD languages.

The GASNet design is partitioned into two layers to simplify porting without sacrificing performance: the lower level is a narrow but general interface called the GASNet Core API, whose design is heavily influenced by Active Messages, and which is implemented directly

on top of the particular network architecture. The upper level is a wider and more expressive interface called the GASNet Extended API, which provides high-level operations such as one-sided puts and gets and various collective operations.

The cluster version of Nanos++ uses the GASNet Core and Extended APIs as a basis for implementing its shared memory abstraction, which allows users to execute tasks across parallel machines. The OmpSs scheduler is configured at runtime with the resource information. In the case of clusters, OmpSs also has to manage the user data across multiple nodes, which is accomplished using the GASNet Core API. OmpSs uses active messaging in order to send tasks and data to different nodes. The task dependencies are managed by the master node, which ensures correctness of the application execution. Figure 2 shows an overview of layers in OmpSs Cluster version.

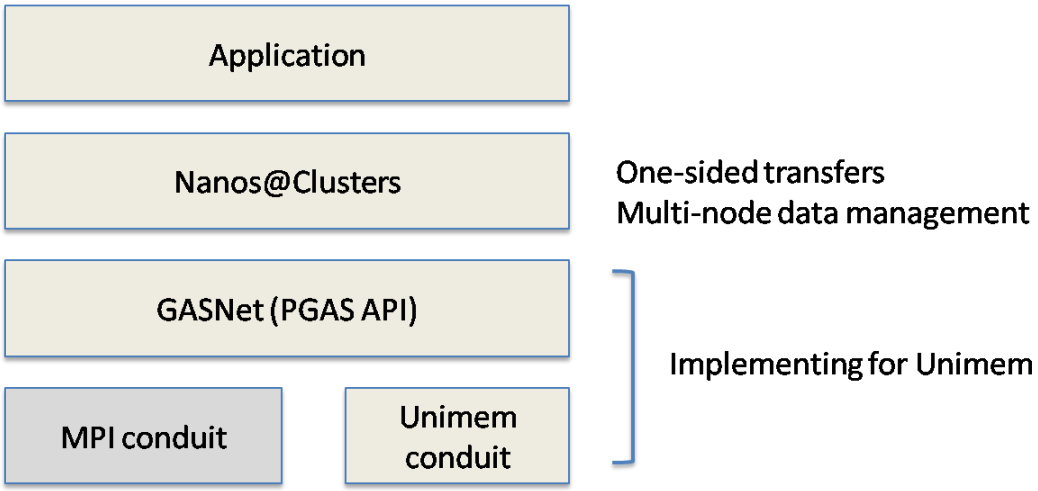


Figure 2: Unimem support layers in OmpSs runtime

The Nanos++ runtime system uses calls to the Extended API functions of GASNet to implement the bulk of the communication work (thereby ensuring optimal performance across platforms). The runtime also uses the core active message interface to implement non-trivial language-specific or compiler-specific communication operations that would be inappropriate in a language-independent API (e.g. implementing distributed language-level locks, distributed garbage collection, collective memory allocation, etc.). The Active Message features of the GASNet Core API provide a powerful extensibility mechanism which allows OmpSs runtime to implement a wide variety of specialized communication operations in context of data, runtime objects, user arguments etc.

2.1.3 Design of a preliminary software implementation

BSC has ported GASNet to the Unimem APIs and tested it using the software emulator from FORTH. Since the clusters version of Nanos++ uses mainly the GASNet Core API, this work has concentrated on implementing the Core API using a combination of MPI and Unimem RDMA.

The design of GASNet is indicated in Figure 3. Several implementations of the GASNet Core API are available optimized for different network configurations, including general-purpose UDP, general-purpose MPI, MPI over Gemini interconnect, InfiniBand interconnect, and shared memory. GASNet also includes a reference network-independent implementation of the Extended API that uses only the Core API, which means that a basic network architecture implementation requires only an implementation of the Core API. It is possible, however, for optimized network implementations to bypass the reference Extended API functions, by

providing optimized functions that improve performance using specific hardware support (e.g. special network support for puts/gets or hardware-assisted broadcast).

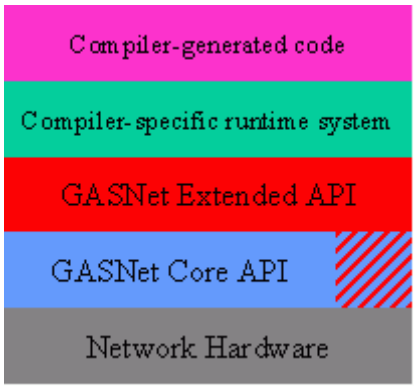


Figure 3: Layers in GASNet communication library

The GASNet conduit for Unimem allows the OmpSs runtime to transparently use Unimem features, including RDMA between nodes. In large systems, it can revert to using MPI whenever Unimem features are not available. Since the current implementation of Nanos++ for Clusters uses only the Active Messaging support in the GASNet Core API, we have focussed so far on Active Messages, and other basic functionality (e.g. initialization functions). GASNet consists of three kinds of Active Messages i.e. short, medium and long. They also have their corresponding reply functions having similar syntax. A short description is given below:

Short Active Message
These messages carry only a few integer arguments (up to gasnet_AMMaxArgs())
handler prototype:

```
void handler(gasnet_token_t token,
        gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

Medium Active Message
In addition to integer arguments, these messages can can carry an opaque data payload (up to gasnet_AMMaxMedium() bytes in length), that will be made available to the handler when it is run on the remote node.
handler prototype:

```
void handler(gasnet_token_t token,
        void *buf, size_t nbytes,
        gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

Long Active Message
In addition to integer arguments, these messages can carry an opaque data payload (up to gasnet_AMMaxLong() bytes in length) which is destined for a particular predetermined address in the segment of the remote node (often implemented using RDMA hardware assistance)
handler prototype:

```
void handler(gasnet_token_t token,
        void *buf, size_t nbytes,
        gasnet_handlerarg_t arg0, ... gasnet_handlerarg_t argM-1);
```

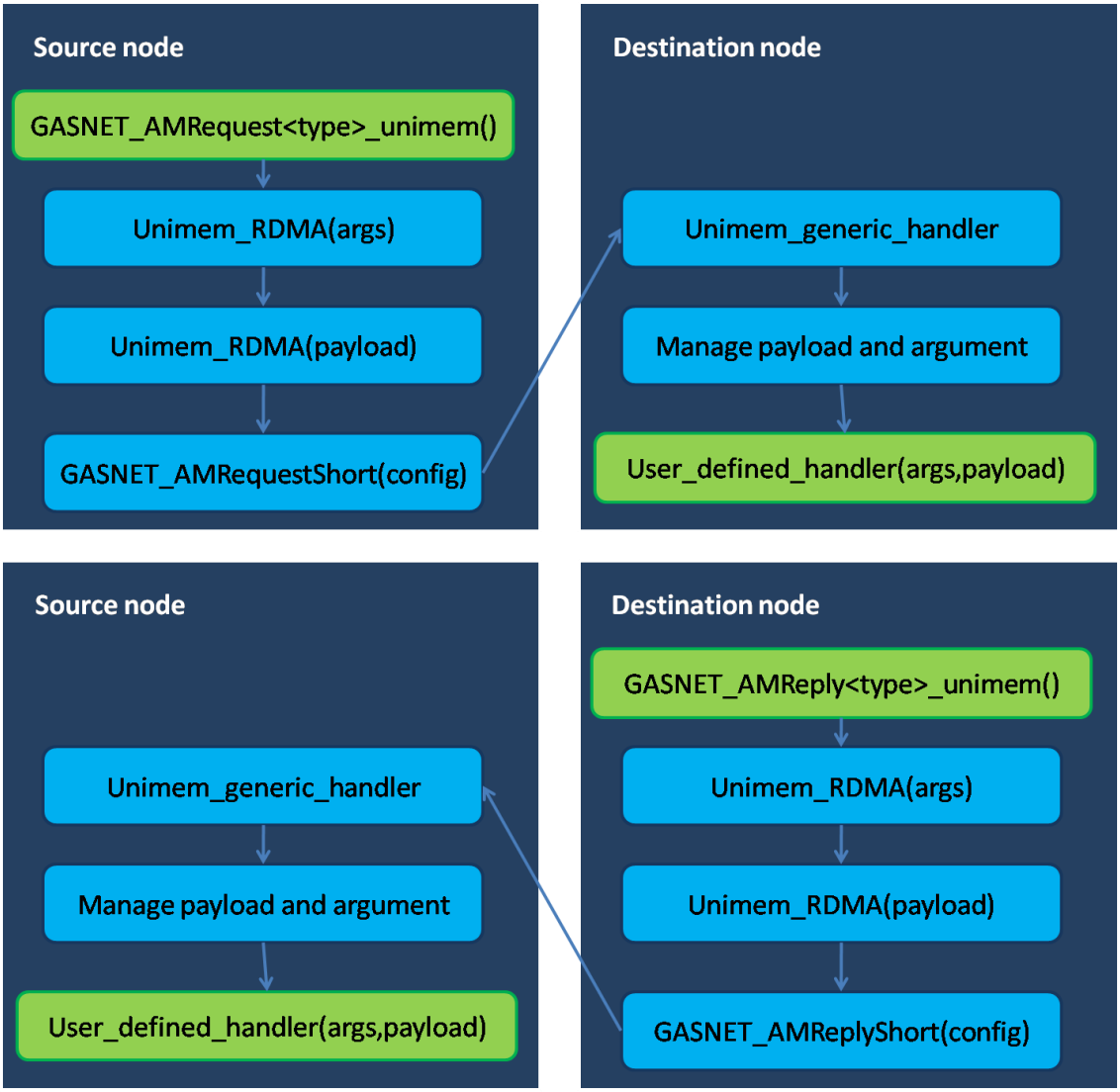


Figure 4: Active Message support in Unimem

Active messages involve transfers of the arguments and payload to the destination node and execution of the handler in the destination node. This is done in clusters using the underlying MPI conduit. When Unimem is available between two nodes, the GASNet active message calls are translated into GASNet short messages, while the arguments and data payload are managed using Unimem RDMA calls.

Figure 4 shows an implementation diagram for Active Messages for Unimem. Each call made to Active Message in the Unimem conduit gets converted into Unimem RDMA and short messages. When the size of the RDMA is less than the system page size, which is 4 KB in most systems, a memory copy is used for transferring the arguments over Unimem. The Unimem driver takes care of the memory transfers. This approach is used in case of transferring the arguments. In the case of data payload, we use RDMA approach owing to performance benefits.

2.1.4 Suitable ExaNode Mini-apps

For validation of Nanos++ on the Unimem architecture, we will use the MiniFE mini-app from ExaNoDe WP2. MiniFE is a finite element mini-application that implements computational kernels representative of implicit finite-element applications. It assembles a sparse linear-system from the steady-state conduction equation on a brick-shaped problem

domain of linear 8-node hex elements. It then solves the linear-system using a simple un-preconditioned conjugate-gradient algorithm.

The application is also scalable using MPI as well as OpenMP. As a general approach we will be using OmpSs tasks in porting MiniFE to ExaNode architecture.

2.2 OpenStream

This section was contributed by UOM.

2.2.1 Introduction to OpenStream

OpenStream [14] is a task-parallel, data-flow programming model implemented as an extension to OpenMP, designed for efficient and scalable data-driven execution. Arbitrary dependence patterns can be used 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).* As shown in Figure 5, each task computes on data available in its input buffers and writes 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. The privatization of data communicated between tasks is key to enabling the runtime to fully control the locality of memory allocation and of task placement, as well as providing transparent RDMA memory transfers between tasks.

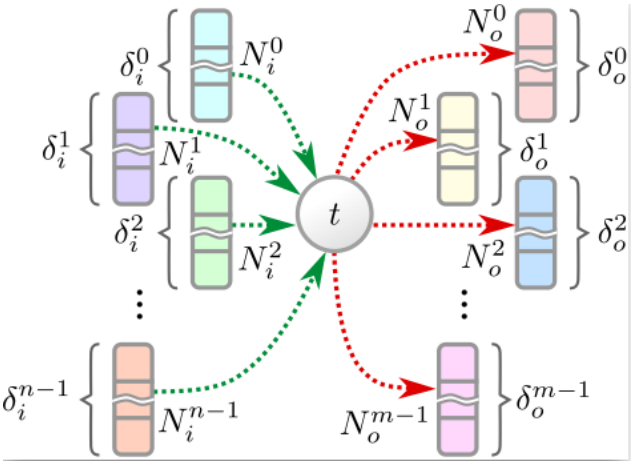


Figure 5: OpenStream task communication through private buffers

Each task is associated with a set of incoming data dependences and a set of outgoing data dependences, as illustrated by Figure 5. Each data dependence is associated with a contiguous region of memory, the input buffer and output buffer, for incoming and outgoing dependences, respectively. The addresses of these buffers are collected in the task's frame, akin to the activation frame storing a function's arguments and local variables in the call stack. While the frame is unique and allocated at the task's creation time, its input and output buffers may be placed on different NUMA nodes and allocated later in the life cycle of the task, but no later than the beginning of the execution of the task, reading from input buffers and writing into output buffers. Buffer placement and allocation time have a direct influence on locality and task-data affinity.

Finally, OpenStream also 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.

2.2.3 Design of preliminary software implementation

As the prototype boards have only become available with both RDMA and Atomics in M12, we have focused our initial work on a functional implementation on top of an emulation library for RDMA provided by FORTH and a functional emulation layer of inter-node Atomics that we have implemented at UOM (based on the API designed by FORTH for the Juno prototype) and made available to all WP3 partners. This initial implementation is now functional on the emulation platform, although as discussed in Section 2.2.2, the runtime is not yet optimized. Part of the difficulty for optimizing the implementation is that the specific tradeoffs of the ExaNoDe architecture cannot be captured easily with either emulation layers or Juno boards prototype. A preliminary optimization work will be conducted until we have access to more concrete hardware.

This first multi-node implementation was possible thanks to the collaboration between FORTH and UOM on the API and the emulation framework for RDMA and atomics.

2.2.4 Suitable ExaNode Mini-apps

OpenStream currently provides a front-end for C, but can be used with C++ with the appropriate wrappers. For this reason, we will target the HydroC, MiniFE and NEST Mini-apps identified in D2.1. HydroC will be the primary objective once we have validated the OpenStream implementation using micro-benchmarks from the OpenStream suite. Although no in-depth evaluation of the Mini-apps has been done for OpenStream, the programming model is fully general and should be suitable for all C-based applications.

3 Communication libraries

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 [4], 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.

Traditionally GPI is used in visualization and seismic imaging applications. In the past years GPI has been ported to several scientific applications within publicly funded projects. With the objective of running on current Petascale systems, the Exascale-ready GPI prototype has been run at the Extreme Scale-out Workshop in 2015 at the Leibnitz Rechenzentrum (LRZ) in Munich, Germany. The results were presented at the XXXL workshop at the ISC15 conference and the PARCO 2015 conference. Two applications have been developed using GPI: Reverse Time Migration (RTM), a seismic imaging technique and the ECED filter for noise removal from images. Strong scaling with over 90%/70% of parallel efficiency for the ECED filter/RTM application has been shown on the Haswell extension of the *SuperMUC* cluster on 84,000/43,000 cores.

Inside nodes, parallelism is handled using threads. The GASPI API is thread-safe and allows each thread to post requests and wait for notifications. Any threading approach (POSIX threads, MCTP, OpenMP) is supported since it is orthogonal to the GASPI communication model.

GPI supports interoperability with MPI in order to allow for incremental porting of applications. GPI supports this interoperability in a so-called mixed-mode, in which the MPI and GASPI interfaces can be mixed.

An interface allowing interoperability concerning the memory management has recently been established in the GASPI standard. GASPI handles memory spaces in so-called segments, which are accessible from every thread of every GASPI process. The GASPI standard has been extended to allow the user to provide an already existing memory buffer as the memory space of a GASPI segment. This new function will allow future applications to communicate data from memory that is not allocated by the GASPI runtime system but provided to it (e.g. by MPI).

3.1.2 Exploiting Unimem in GPI

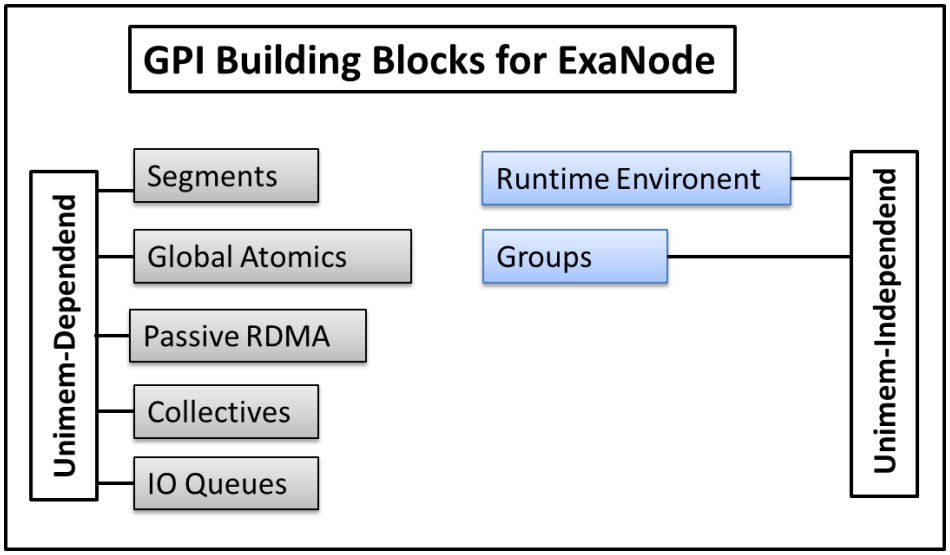


Figure 6: GPI Building blocks for ExaNoDe architecture support

In the context of the ExaNoDe hardware, GPI will be implemented using the RDMA primitives provided by the hardware and the interconnect. In the current design phase, the GPI Runtime reflects the interface description as given in [9]. To simplify the porting of the GPI Runtime, the GPI Kernel has been subdivided into Unimem dependent and Unimem independent modules, as shown in Figure 6.

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

Currently both modules are in the process of being ported to the aarch64 architecture.

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 [9]. This emulation library allows early tests on a standard x86_64 SMP system without the need to have real prototype hardware available on site. In the next step each module will be ported separately to the ExaNoDe prototype system and linked against the Unimem system libraries.

3.1.3 Design of preliminary software implementation

In close collaboration with FORTH, Fraunhofer has developed a first RDMA User level API for Unimem suitable for all runtime systems. This first API will be extended as soon as further requirements are needed to implement certain functionalities. After evaluating some of the core components of the API on the ExaNoDe prototype a first mapping of the Unimem dependent GPI modules to the user level RDMA API was found. The following chapters describe a possible first design of GPI on top of that RDMA API.

3.1.3.1 GPI Segments

Currently the ExaNoDe API of GPI supports a single RDMA capable segment with a maximum size of 256MB. GPI will relax this restriction in a sense that an application can allocate multiple GPI Segments within the 256MB. Furthermore GPI will also allocate some internal segments for collective operations.

3.1.3.2 GPI IO-Queues

GPI applications trigger one-sided RDMA reads and writes by placing communication tokens into 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 user-level API currently returns a unique communication identifier for each communication call without using communication queues. To implement different IO-Queues on top of that functionality, a map will be assigned over all IO-Queues that combines communication identifiers and queue ids. To allow concurrent threads to access the IO-Queues at the same time, a global spinlock is used to protect all involved API calls.

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 Sections 3.1.3.1 and 3.1.3.2.

3.1.3.4 GPI Global Atomics

Global Atomics can be built in a first instance by using directly the interface of the user-level RDMA API. However, all current available atomics are blocking and cannot be overlapped with computations. To be also scalable at large-scale a more efficient interface for global atomics has to be found and implemented as part of the user level API in a future version.

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. A first implementation can be built on top of the Unimem mailbox approach, where a process/thread can use system calls like `select`, `poll` or `epoll` to get signalled when an event occurs on the corresponding mailbox file descriptor.

A couple of programs have been developed to test this first design of each individual module on the ExaNoDe prototype or by using the emulation library.

3.1.4 Suitable ExaNode Mini-apps

Beside the GPI test suite a stencil kernel application will be implemented to show off the strength of overlapped and offloaded data communication on ExaNoDe/Unimem.

3.2 Message Passing Interface (MPI)

This section was contributed by BSC.

3.2.1 State of MPI port

In the original Description of Action (DoA), the optimized port was to have been designed and implemented by CEA. Unfortunately CEA have been unable to dedicate sufficient resources to this activity, so at M12 the project took the decision to transfer this activity to BSC in the amendment, with a realistic (increased) budget from 9 to 24 PMs. With the new plan, a preliminary implementation is expected in M24, with the fully optimized implementation delivered by M36. This is therefore the same schedule as in the original DoA,

but due to the change in partner there has been negligible progress as of M12 to report in this deliverable.

The implementation of MPI is crucial to the viability of ExaNoDe + ExaNeSt + ECOSCALE, and is especially important to the application partners in WP2 of ExaNeSt. For this reason, the requirements for the optimized MPI port were discussed in detail in the ExaNode + ExaNeSt + ECOSCALE Workshop, 20 September 2016 in Trieste. The conclusions were as follows.

3.2.2 MPI implementation: OpenMPI vs. MPICH

BSC has expertise in MPICH and contacts with the development team at Argonne, so it proposes to optimize the MPICH implementation of MPI. This would be the generic MPICH codebase rather than an optimized implementation such as MVAMPICH. The version of MPI was discussed in detail in the Trieste Workshop, and although the application partners of ExaNeSt are currently using OpenMPI, they are willing to evaluate MPICH.

3.2.3 Version of MPI specification

The application codes in ExaNeSt are generally MPI-2 or MPI-1.2, with certain cosmology codes moving to MPI-3, due to the better one-sided puts and gets for RDMA, sparse collectives and non-blocking collectives. There is a considerable diversity among codes in terms of their use of point-to-point vs. collective operations and blocking vs. non-blocking. For this reason, ongoing collaboration is required between the ExaNoDe MPI implementation effort and ExaNeSt. It will certainly not be sufficient to develop a "basic" implementation with optimized implementations of only the most basic MPI-1 calls.

3.2.4 Other technical issues

The partners in ExaNeSt have agreed to collect performance data from their applications in order to help with performance tuning, such as eager vs. rendezvous threshold, fan-in / fan-out factor and choice of fast paths in the implementation.

3.3 Suitable ExaNoDe Mini-apps

All ExaNoDe mini-apps considered in D2.2 *(Report on the ExaNoDe mini-applications)* [1] 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 Other runtime support

This section describes the advances in thermal and power management and runtime libraries for performance-critical primitives (e.g. scalable synchronization primitives and work-stealing load balancing). The intention is that these technologies will be made available and potentially integrated into the optimized ports of GPI, OmpSs, OpenStream and MPI.

4.1 Power and thermal control

This section was contributed by ETHZ.

In this section we discuss an extension of the runtime of the programming models for thermal and power control. This deliverable reports an assessment of the thermal effects visible in high performance computing processors, the available power control knobs, and their interactions with a programming model. We choose MPI for our analysis. We then propose a thermal controller formulation which is suitable for accounting workload properties as well as workload unbalance. In future works we will evaluate the achievable gain and the design trade-offs.

4.1.1 Introduction

Chiplet integration coupled with the high computational power and heterogeneous resources which characterize the ExaNoDe architecture will challenge conventional thermal and power management strategies. Variations of the application load will translate into highly variable power consumption and generate temperature hotspots within the severely thermally constrained, vertically stacked Chiplet environment. A holistic thermal and power management approach is therefore crucial. The approach should combine the fast reaction times, which are typical of hardware control loops, with the global view from software management.

4.1.2 Design of preliminary software implementation

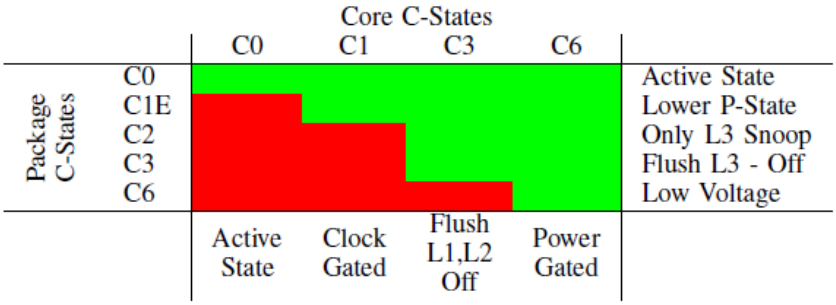
Within this task ETHZ is exploring and designing software-based distributed power and thermal controllers embedded in the runtime libraries which will adapt the operating conditions according to application demand and silicon thermal evolution, taking into account variability and heterogeneity in both workloads and hardware. The controllers will leverage hardware sensors and be aware of the existing hardware control loops to maximize precision and minimize reaction time.

We use as a proxy to evaluate these policies a state-of-the-art processor used in several supercomputer installations.

4.1.2.1 Power and Thermal Management

The power management states of computing elements are divided into sub-groups. The P-States include dynamic voltage and frequency (DVFS) operating points that target reduction of active power. C-States instead target the reduction of idle power. Both P-States and C-States are numbered from 0 to *n*, where a higher number means greater savings in power. However, in the case of C-States this means also longer transitions in and out of the state. In recent Intel architectures P-States can be selected independently for each core [10]. C-States are instead defined independently for cores and the “uncore” (package) region. The P-States are handled by the Linux OS by means of software frequency governors. C-States are triggered from CPU’s firmware, but the OS can provide hints on the appropriate C-State to the hardware through OS idle governors. Figure 7 shows the different architecture impacts and dependencies for the different cores and package C-States.

We will use this analysis in Task 3.3 to evaluate the design trade-offs as well as to quantify the achievable energy-saving gains.



Green: valid configuration for Core and Package C-States
Red: invalid configuration

Figure 7: Core and Package C-States and matrix showing valid and invalid combinations

We used a high performance computing node equipped with two Intel Haswell E5-2630 v3 CPUs, each with eight cores and 2.4 GHz nominal clock speed and 130 W Thermal Design Power (TDP) [11]. The node runs the SMP CentOS Linux distribution version 7.0.

4.1.2.2 Thermal Model

To understand the thermal properties of the node we have executed three main stress tests, on which we have:

(i) Kept the system in idle and measured the power and each core temperature after ten minutes,

(ii) Executed Robert Redelmeier's cpuburn power virus,[1] in sequence on each core of each socket in the node, leaving the remaining cores idle. We maintained a constant workload for ten minutes and measured the power consumption and temperature. This test was used to extract the maximum gradient between cores.

(iii) Executed the power virus for ten minutes on all the cores e simultaneously and we measured the temperature and the power consumption.

In all previous tests, the temperature and power values were measured by periodically reading the machine specific registers, with Turbo mode disabled. Results of our analysis are reported in Table 2.

Table 2: Thermal Model

Condition	Value
Average temperature - Idle cores	15.93 °C
Average temperature - Active cores	33.39 °C
Gradient - Idle cores	4.47 °C
Gradient - Active cores	4.79 °C
Gradient - Active core vs. idle cores	8.05 °C
Time to reach steady state	120 s

4.1.2.3 Power Model

In addition to the previous tests we have re-executed the power virus in different configurations (number of cores executing the power virus, turbo enabled/disabled; different

[1] cpuburn power virus by Robert Redelmeier: it takes advantage of the superscalar architecture to maximize the CPU power consumption

frequencies) while limiting C-States.[2] We maintained each configuration for ten minutes and measured the power consumed by each CPU.

Table 3: Power Model active power

	P-State (all in C-State C0)		
	Turbo	**Max freq.**	**Min freq.**
Puncore	17.13 W	17.13 W	12.76 W
Pcore	6.38 W	5.47 W	3.16 W

Table 4: Power Model idle power

	C-State		
	C1	**C2**	**C3**
Puncore	12.76 W	11.90 W	11.84 W
Pcore	1.32 W	0.38 W	0.00 W

Table 3 and Table 4 report the power split between core and uncore regions. Table 3 shows them for different P-States (Turbo, Max frequency and Min frequency), while in active C-State C0. Table 4 shows the power consumption in different C-States. The power consumption of the uncore (in C0 mode) is the same for Turbo and Max Frequency because the uncore component is not affected by the Turbo frequencies. Scaling down the frequency (to Min freq.), however, reduces the uncore power. In contrast, the core power consumption scales proportionally with the frequency. Idle power can be seen in Table 4, from which we see that C1 further reduces the power consumption by 58% compared with C0 C-State at minimum frequency, and that C3 significantly cuts the idle power by the 71% for the core but only marginally for the uncore. C6 instead zeroes the core power but reduces the uncore power by just 1%. Clearly these results show that to reduce the energy consumption, the parallel programming runtime (i.e. MPI) should prefer C1 and C3 states to save energy during idle intervals (communication phases) instead of P-States (DVFS).

4.1.3 MPI runtime and workload characterization

We use as a workload Quantum ESPRESSO [12] to match a real life workload scientific application to evaluate power and thermal management opportunities in Galileo. Quantum ESPRESSO is a software suite for molecular dynamics. Its main computational kernels include dense parallel linear algebra and 3D parallel FFT, which are both relevant for many HPC applications. In detail, we use a Car–Parrinello (CP) simulation, which prepares an initial configuration of a thermally disordered crystal of chemical elements by randomly displacing the atoms from their ideal crystalline positions. This simulation consists of a number of computation kernels that have to be executed in the correct order.

We use Intel MPI Library 5.1 for communication and the Intel ICC/IFORT 16.0 compiler. The Intel MPI runtime tries to achieve maximum performance by adopting by default a busy-waiting (BW MPI) policy that forces CPU polling on the network fabric controller during communication phases and synchronization points. Alternately programmers can specify an interrupt-based (IB MPI) communication. With IB communication, HPC applications release the processor during communication phases, giving control to the OS, which triggers an idle state (C-State). To isolate the impact of deep C-States from the different communication mechanisms we also run the interrupt-based communication while limiting the deepest C-State to C1 (IB MPI - C1 limited).

[2] C-States in Intel architectures can be limited through dedicated machine-specific registers

Table 5: Quantum ESPRESSO - Energy

	BW MPI	IB MPI	IB MPI (C1 limited)
Execution Time	261.52 s	283.48 s	283.48 s
Average Power	68.19 W	62.91 W	63.22 W
Energy	17.90 J	17.83 J	18.04 J
% time in C1	0.00%	9.74%	15.80%
% time in C2	0.00%	0.66%	0.00%
% time in C3	0.00%	5.40%	0.00%
Total idleness	0.00%	15.80%	15.80%

Table 5 shows the results of Quantum ESPRESSO using the different MPI runtime configurations. For each runtime configuration, we measure the execution time, average power consumption and total energy for both sockets. For also calculate the idleness, which represents the percentage of idle time, and we use hardware counters to measure the percentage of time in each C-State: C1, C2 and C3. For Busy Waiting (BW) MPI, we can see that, since it uses busy-waiting communication, 0% of its total time is spent in C1, C2 and C3, and the total idleness is 0%. In contrast, Interrupt-Based (IB) MPI has a significant percentage of idleness, which includes time in deep C-States. Due to the hardware policy, which triggers transitions to C-States using time-outs, the specific communication phase can finish before the deepest C-State is reached (C6). For this reason, Quantum ESPRESSO consumes 8% lower power with IB MPI compared with BW MPI. Using IB MPI does not, however, cause a significant energy reduction, due to a large increase in execution time. This could be caused by (i) the C-State transition time, as well as by (ii) additional communication latencies related to IB MPI communication. The IB MPI (C1 limited) configuration uses only the C1 idle state, which is characterized by a negligible transition time. When comparing its execution time with IB MPI we notice that there is no difference. This demonstrates that the slowdown in IB MPI in comparison with BW MPI is due to the IB MPI implementation rather than C-State transition times.

This is the first important insight of our work: the current IB MPI implementation has to be improved in order to exploit power-saving opportunities offered by the increased agility in C-state management of modern Intel CPUs.

4.1.4 HPC Optimal Thermal Control

In this section we present a mathematical Integer Linear Programming (ILP) formulation, namely the Optimal Thermal Controller (OTC), which matches all the requirements of high-performance computing systems and proactive thermal control: (i) limiting the future temperature of all cores below a critical threshold by selecting the best operating point; (ii) maximizing application performance (frequency of all the cores); (iii) reducing power consumption during communication phases; (iv) providing knobs to match the computation to communication ratio with the clock frequency.

The OTC operates at the node level and is composed of two main components: the thermal-aware task mapper and controller and an energy-aware MPI wrapper. The thermal-aware task mapper and controller (TMC) is triggered: (a) after the job scheduler has deployed the parallel application on the reserved portion of the HPC machine for the job execution, and (b) periodically with period T_s. At scheduling point (a) the TMC specifies the task-to-core mapping, which will be maintained until the application completes. Clearly, if a critical task is mapped to a thermally inefficient core this will more likely cause a severe degradation of the final application performance. To abstract this requirement, we use a per-task priority level. Higher priority means higher impact of task performance on the final application. The root MPI task (0) is always the most critical as it acts as collector/gatherer. At scheduling point

(b), the TMC selects the optimal frequency to be applied to the different cores for the following interval in order to maintain the future temperature of all cores below a maximum limited value. The energy-aware MPI wrapper (EAW) is event-driven and it acts as a bridge between the MPI synchronization primitives and the core's frequency selection. This programming model interface is reactive and reduces the operating point (lower frequency/deeper C-State) when the MPI runtime is busy waiting. When the execution flow returns to the application code, the frequency/operating point is restored to the one selected by the Thermal Controller.

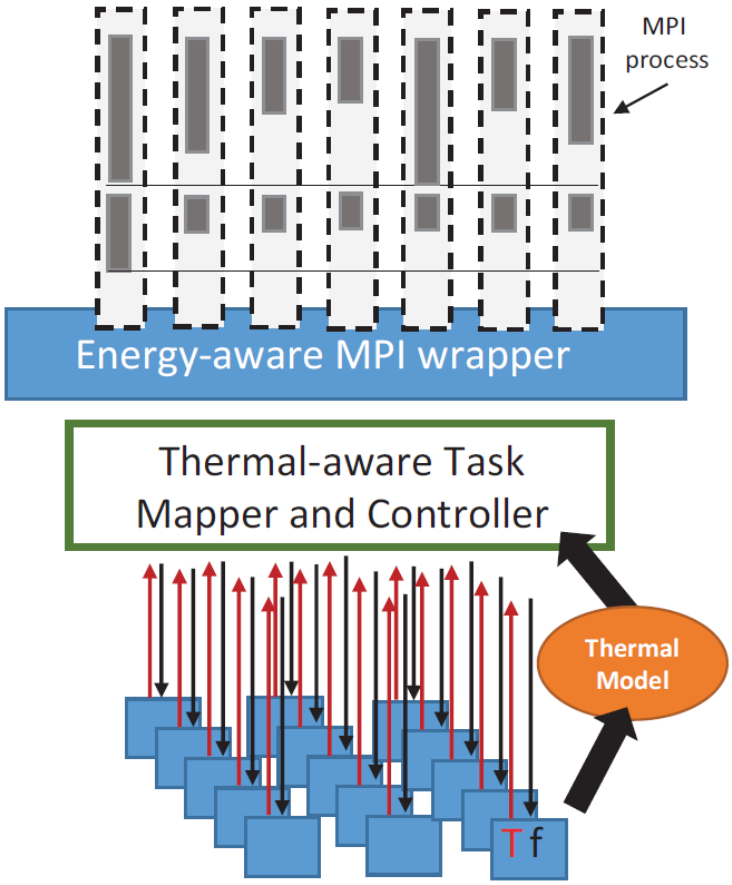


Figure 8: Operation of Thermal-aware Task Mapper and Controller

4.1.5 The First Step Problem (FSP)

This optimization problem is solved during the initialization of the application. Its purpose is to allocate the application tasks on the available cores and select for each of them the maximum frequency that meets the thermal constraint T_{max} in the prediction interval (PI_{FSP}). The prediction interval (i.e. the time horizon) plays an important role, indeed if it is too short, the Thermal Controller (TMC) cannot predict the impact of a task allocation on long term core temperature as its effect is hidden by the thermal capacitance, making the problem trivial. On the contrary, if the predicted interval is too long the TMC cannot take advantage of the thermal capacitance for sustaining short time power burst.

In addition, not all tasks have the same priority. This is matched in the optimization model which maximizes the frequency of the highest priority task penalizing the frequencies of other ones, the optimization problem considers K tasks to be assigned to N cores, where the number of tasks is less than or equal to the number of cores; i.e. $K < N$. Each core can be configured with a frequency in a set of M level of frequencies. The Object Function (O.F.) maximizes the sum of frequencies of all active cores γ_{jf} weighted by the priority δ_i of the task assigned on that core. To model the problem, we use two sets of binary decision variables:

$$x_{jf}^{i} = \begin{cases} 1 & \text{if core } j(j = 1, \ldots, N) \text{ works at frequency} \\ & f(f = 1, \ldots, M) \text{ executing job } i(i = 1, \ldots, K) \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

$$y_j = \begin{cases} 1 & \text{if core } j(j = 1, \ldots, N) \text{ is working,} \\ 0 & \text{otherwise, i.e., if it is idle.} \end{cases} \quad (2)$$

We can formulate the following ILP model with three constraints to model the assignments and the thermal bounds:

$$O.F. = max \sum_{i=1}^{K} \sum_{f=1}^{M} \sum_{j=1}^{N} \delta_i \gamma_{jf} x_{jf}^{i} \quad (3a)$$

$$\sum_{j=1}^{N} \sum_{f=1}^{M} x_{jf}^{i} = 1 \quad (3b)$$

$$(i = 1, \ldots, K) \quad (3c)$$

$$\sum_{j=1}^{N} \sum_{f=1}^{M} x_{jf}^{i} + y_j = 1 \quad (3d)$$

$$(i = 1, \ldots, K) \quad (3e)$$

$$\sum_{j=1}^{N} GS_{jl} \left(\vec{p}_j y_j + \sum_{i=1}^{K} \sum_{f=1}^{M} p_{jf} x_{jf}^{i} \right) + T_l^0 + T^a \leq T_{MAX} \quad (3f)$$

$$(l = 1, \ldots, N) \quad (3g)$$

The constraint (3b) specifies that each of the K tasks (i=0…K) must be assigned to exactly one core, j, that works at a single frequency, f. The constraint (3d) is needed to determine the y decision variables which represent the idle cores. These variables are used in the constraint (*3f*) in case there are less jobs than cores i.e., $K \leq N$. Finally, the constraints (3f) guarantee that the temperature of each core does not exceed T_{max} during the next predicted interval (PI_{FSP}). In the last constraint, (3f) GS is a gain matrix with dimension $N \times N$. This matrix is used to calculate the increment of temperature of all the cores when a core is subjected to a constant power input for PI_{FSP} seconds. $T^0{}_l$ represents the dependency of the future temperature (PI_{FSP}) on the current core temperatures. These values can be derived from a state-space thermal model, in future activities we will evaluate strategies to self-learn the thermal model from the compute core. T_a is the ambient temperature. When the number of jobs is less than the number of cores, the decision variable y_i is used in conjunction with the vector of idle powers ($\vec{p}$) to add the idle power components.

4.1.6 The *i*-th Step Problem (ISP)

After the tasks have been assigned to the cores in the FSP, the TMC has to solve periodically and at a finer time scale, the assignment problem of frequencies to cores. The ISP has the same objective function as FSP as well as the same thermal model formulation. However, the prediction interval for the ISP (PI_{ISP}) can in general be different from that of the FSP.

In contrast to the previous case, the model considers only active cores (T) because the thermal constraints cannot be broken by an idle core. This reduces the overall complexity. Since in this model, tasks have been already allocated by FPS, the tasks and cores do not need separate variables, thus a priority is referred to a core.

$$x_{rf} = \begin{cases} 1 & \text{if core } r(r = 1, \ldots, T) \text{ works at frequency} \\ & f(f = 1, \ldots, M), \\ 0 & \text{otherwise.} \end{cases}$$

The ISP model requires fewer constraints than FSP due the lower number of variables.

$$O.F. = max \sum_{a \in A} \sum_{f=1}^{M} \delta_a \gamma_{af} x_{af} \quad (4a)$$

$$\sum_{f=1}^{M} x_{af} = 1 \quad (4b)$$

$$(\forall a \in A) \quad (4c)$$

$$\sum_{a \in A} \sum_{f=1}^{M} GS_{la} p_{af} x_{af} + \sum_{i \in I} GS_{li} \vec{p}_i + T_l^0 + T^a \leq T_{MAX} \quad (4d)$$

$$(\forall l \in A) \quad (4e)$$

The constraint (4b) bounds each core to a selected frequency. The constraint (4d) guarantees the thermal limits imposed on the model. The set $A=a_i$ contains the index of the active cores and the set $I=i_i$ contains the index of Idle cores directly defined from the solution of *FSP*. The set $A \cap I$ must be empty. In general, the ISP problem is computationally simpler than the FSP problem due to the lower number of decision variables and constraints.

In future work, we plan to evaluate the trade-off between the overhead and the control granularity, as well as strategies to self-learn the thermal model.

4.2 Parallel runtime support

This section was contributed by UOM.

4.2.1 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.

4.2.2 Design of preliminary software implementation

In the first year of the project, UOM has ported the current state-of-the-art implementation [16] 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.

5 Concluding Remarks

This deliverable has described the runtime systems (OmpSs and OpenStream) and communication libraries (GPI and MPI) being developed in the ExaNoDe project. Each runtime system and library is described in detail together with the design of a preliminary software implementation to take advantage of the unique characteristics of the ExaNoDe architecture. Work is ongoing, and will be documented further in D3.2, which will be delivered in M24.

OmpSs and OpenStream are distinct task-based programming models that extend OpenMP with new directives. GPI is an open-source communication library that implements the GASPI standard PGAS API. MPI is the standard message-passing API supported by all serious HPC systems and employed by the vast majority of scientific applications. These runtime systems and libraries will provide standard and portable programming interfaces, so that an application can run efficiently on the ExaNoDe architecture.

6 References and Applicable Documents

[1] Dirk Pleiter et al, D2.1 Report on the ExaNoDe mini-applications, ExaNoDe project deliverable D2.1, 2016.

[2] Nikolaos D. Kallimanis et al, D3.6 Design of the ExaNoDe Firmware, ExaNoDe project deliverable D3.6, 2016.

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

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

[5] AXIOM project, EU H2020 project 645496. http://www.axiom-project.eu.

[6] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, Judit Planas: Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. Parallel Processing Letters **21**(2): 173—193 (2011)

[7] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé, Jesús Labarta. Productive Programming of GPU Clusters with OmpSs. 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), pp.557—568, 2012.

[8] Simmedinger Christian, Mirco Rahn and Daniel Gruenewald. The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures. Sustained Simulation Performance 2014. Springer international Publishing, 2015. 17-32.

[9] UNIMEM Mechanisms on the Euroserver Discrete Prototype Gen2 (64-bit). Document version 1.2.1, July 27, 2016.

[10] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the Intel Haswell processor. In Parallel and Distributed Processing

[11] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation Intel core processor. IEEE Micro, **34**(2): 6–20, 2014.

[12] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L. Chiarotti, Matteo Cococcioni, Ismaila Dabo, et al. Quantum ESPRESSO: a modular and open-source software project for quantum simulations of materials. Journal of physics: Condensed matter, **21**(39):395–502, 2009.

[13] J. M. Pérez, Rosa M. Badia and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September–1 October 2008, Tsukuba, Japan (2008), pp. 142–151.

[14] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, Nathalie Drach: Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. PACT 2016: 125-137.

[15] Andrey Rodchenko, Andy Nisbet, Antoniu Pop, Mikel Luján: Effective Barrier Synchronization on Intel Xeon Phi Coprocessor. Euro-Par 2015: 588-600.

[16] Nhat Minh Lê, Antoniu Pop, Albert Cohen, Francesco Zappa Nardelli: Correct and efficient work-stealing for weak memory models. PPOPP 2013: 69-80.