SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

INFRA-2010-2.3.1 – First Implementation Phase of the European High Performance Computing (HPC) service PRACE

PRACE-1IP

PRACE First Implementation Project

Grant Agreement Number: RI-261557

D9.2.2
Final Software Evaluation Report

Final version

Version: 1.0
Author(s): Jose Carlos (BSC),
Guillaume Colin de Verdière (CEA),
Matthieu Hautreux (CEA),
Giannis Koutsou (CaSToRC)

Date: 23.07.2012

Project and Deliverable Information Sheet

<table>
<tr><td rowspan="6">PRACE Project</td><td colspan="2">Project Ref. №: RI-261557</td></tr>
<tr><td colspan="2">Project Title: PRACE First Implementation Project</td></tr>
<tr><td colspan="2">Project Web Site: http://www.prace-project.eu</td></tr>
<tr><td colspan="2">Deliverable ID: D9.2.2</td></tr>
<tr><td colspan="2">Deliverable Nature: < Report></td></tr>
<tr><td rowspan="2">Deliverable Level:
PU *</td><td>Contractual Date of Delivery:
31 / July / 2012</td></tr>
<tr><td>Actual Date of Delivery:
31 / July / 2012</td></tr>
<tr><td colspan="2">EC Project Officer: Leonardo Flores Añover</td></tr>
</table>

* - The dissemination level are indicated as follows: **PU** – Public, **PP** – Restricted to other participants (including the Commission Services), **RE** – Restricted to a group specified by the consortium (including the Commission Services). **CO** – Confidential, only for members of the consortium (including the Commission Services).

Document Control Sheet

<table>
<tr><td rowspan="5">Document</td><td colspan="2">Title: Final Software Evaluation Report</td></tr>
<tr><td colspan="2">ID: D9.2.2</td></tr>
<tr><td>Version: 1.0</td><td>Status: Final</td></tr>
<tr><td colspan="2">Available at: http://www.prace-project.eu</td></tr>
<tr><td colspan="2">Software Tool: Microsoft Word 2007</td></tr>
<tr><td>Authorship</td><td>File(s): D9.2.2.docx</td><td>Written by:</td><td>Jose Carlos (BSC),
Guillaume Colin de Verdière (CEA),
Matthieu Hautreux (CEA),
Giannis Koutsou (CaSToRC)</td></tr>
</table>

	Contributors:	Sadaf Alam (CSCS) Stephanos Androutsellis (GRNET) Axel Auweter (LRZ) Holger Brunst (TU Dresden) Carlo Cavazzoni (CINECA) Daniela Galetti (CINECA) Marcin Gebarowski (WCNS) Jose Gracia (HLRS) Erik Hagersten (Rogue Wave) Willy Homberg (FZJ) Radoslaw Januszewski (PSNC) Ivo Kabadshow (FZJ) Damian Kaliszan (PSNC) Jochen Kreutz (FZJ) Agnieszka Kwiecien (WCNS) Jesus Labarta (BSC) Can Ozturan (Bogazici University) Federico Paladin (CINECAf) Dirk Pleiter (FZJ) Roland Poppenreiter (JKU) Jeffrey Poznanovic (CSCS) Seren Soner (Bogazici University) Alexei Strelchenko (CaSToRC) Stephane Thiell (CEA) Mariusz Uchronski (WCNS) Andrea Vanni (CINECA) Torsten Wilde (LRZ)
	Reviewed by:	Florian Berberich, JUELICH Dietmar Erwin,, FZJ
	Approved by:	MB/TB

Document Status Sheet

Version	Date	Status	Comments
0.1	28/June/2012	Draft	
0.2	16/July/2012	Draft	Accommodated internal reviewer comments
1.0	24/July/2012	Final version	Accommodated further comments by reviewers and finalized

Document Keywords

Keywords:	PRACE, HPC, Research Infrastructure, Software Evaluation, Exascale, Supercomputing

Table of Contents

List of Figures

List of Tables

References and Applicable Documents

[1] NVIDIA GPUDirect, http://developer.nvidia.com/gpudirect
[2] OpenCL - The open standard for parallel programming of heterogeneous systems, http://www.khronos.org/opencl/
[3] PRACE Deliverable D6.6, *Report on petascale software libraries and programming models*.
[4] PRACE-1IP Deliverable D9.2.1 *First Report on Multi-Petascale to Exascale Software*.
[5] OpenCL 1.2 Specification, http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
[6] ViennaCL (Linear Algebra and Iterative Solvers) with support for NVIDIA and AMD/ATI GPUs, http://viennacl.sourceforge.net/
[7] Intel SDK for OpenCL Applications, http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/
[8] SNU-SAMSUNG OpenCL Framework, http://opencl.snu.ac.kr/

[9] IBM OpenCL Development Kit for Linux on Power http://www.alphaworks.ibm.com/tech/opencl
[10] Java Bindings to OpenCL (JOCL, enables applications running on the JVM to use OpenCL 1.1), http://jogamp.org/jocl/www/
[11] PyOpenCL (access to the OpenCL API from Python, supports OpenCL 1.1), http://mathema.tician.de/software/pyopencl
[12] PGI OpenCL Compiler for ARM, http://www.pgroup.com/products/pgcl.htm
[13] OpenCL 1.1 Specification, http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf
[14] Sean Rul, Hans Vandierendonck, Joris D'Haene and Koen De Bosschere, *An experimental study on performance portability of OpenCL kernels, Application Accelerators in High Performance Computing*, 2010 Symposium, Papers (2010).
[15] Purnomo, Budirijanto and Rubin, Norman and Houston, Michael, *ATI Stream Profiler: a tool to optimize an OpenCL kernel on ATI Radeon GPUs*, ACM SIGGRAPH 2010 Posters.
[16] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, H. Takizawa, H. Kobayashi, *Evaluating Performance and Portability of OpenCL Programs*, Science And Technology, p. 781-784, 2010.
[17] Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R. D., Kalé, L. and Schulten, K. (2005), *Scalable molecular dynamics with NAMD*. J. Comput. Chem., 26: 1781–1802. doi: 10.1002/jcc.20289.
[18] NAMD, http://www.ks.uiuc.edu/Research/namd/
[19] The Graph 500 list, http://www.graph500.org/
[20] LRZ, Energy Aware System Software, PRACE-1IP WP9, 2012
[21] http://www.nagios.org/about/overview/
[22] http://mathias-kettner.de/checkmk_livestatus.html
[23] D. Galetti and F. Paladin, *Design, development and improvement of Nagios system monitoring for large clusters*, White-paper to appear in http://prace-ri.eu/white-papers
[24] S. Soner, C. Özturan, Integer programming based heterogeneous CPU-GPU cluster scheduler for SLURM resource manager, 14th IEEE International Conference on High Performance Computing and Communications (HPCC-2012), June, 2012.
[25] Seren Soner, Can Özturan, Oğuz Tosun, Poster: Co-allocation based scheduling for parallel systems, Supercomputing 2011.
[26] S. Soner and C. Özturan, *Integer Programming Based Heterogeneous CPU-GPU Cluster Scheduler for SLURM Resource Manager*, White-paper to appear in http://prace-ri.eu/white-papers
[27] Altair, Scheduling Jobs onto NVIDIA Tesla GPU Computing Processors using PBS Professional,Resource Library, 2010
[28] J.Duato, F.D.Igual, R.Mayo, A. J.Peña, E.S.Quintana-Ortí, and F.Silla, An efficient implementation of GPU virtualization in high performance clusters, Euro-Par 2009 Workshops, 2009
[29] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, erformance of CUDA virtualized remote GPUs in high performance clusters, International Conference on Parallel Processing (ICPP), 2011
[30] http://lammps.sandia.gov/
[31] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, Dhabaleswar K. Panda, MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters,International Supercomputing Conference (ISC'11) , 2011
[32] Sreeram Potluri, Hao Wang, Devendar Bureddy, Ashish Kumar Singh, Carlos Rosales, Dhabaleswar K. Panda, Optimizing MPI Communication on Multi-GPU Systems using CUDA Inter-Process Communication,International Workshop on Accelerator and Hybrid Exascale Systems (AsHES) with IEEE International Parallel & Distributed

Processing Symposium (IPDPS'12), 2012
[33] http://www2.fz-juelich.de/jsc/jugene
[34] http://www.hlrs.de/systems/platforms/cray-xe6-hermit/
[35] http://www.fujitsu.com/global/about/tech/k/
[36] http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm
[37] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and J. Duato, Congestion Control in InfiniBand Networks,Proceedings of the 13th Symposium on High Performance Interconnects(HOTI'05),2005
[38] Barcelona Supercomputing Center. http://www.bsc.es/computer-sciences/performance-tools. Performance Tools. [En línea] [Citado el: 1 de June de 2012.]
[39] Temanejo- a debugger for task based parallel programming models. Steffen Brinkmann, Jose Gracia, Christoph Niethammer, and Rainer Keller. Ghent, Belgium : s.n., 2011. International Conference on Parallel Computing.
[40] The Scalasca performance toolset architecture. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. 6, s.l. : Concurrency and Computation: Practice and Experience, 2010, Vol. 22. pp. 702-719.
[41] Automatic Performance Analysis of Large Scale Simulations. S. Benedict, M. Gerndt, V. Petkov, M. Brehm, C. Guillen, and W. Hesse. Delft : Workshop on Productivity and Performance (PROPER 2009) in conjunction with Euro-Par 2009, 2009. LNCS 6043, pp. 199-207.
[42] Nagel, Wolfgang E. http://vampir.eu/. Vampir - Performance Optimization. [En línea] [Citado el: 1 de 6 de 2012.]
[43] Tallada, Marc Gonzàlez. http://personals.ac.upc.edu/marc/Marc_Gonzalez_Home_page/Welcome_PoTra.html
[44] Brief announcement: Lower bounds on communication for sparse Cholesky factorization of a model problem. Laura Grigori, Pierre-Yves David, James W. Demmel, and Sylvain Peyronnet. New York, NY, USA : s.n., 2010. Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA '10).
[45] BSC, Programming Models @. http://pm.bsc.es/ompss. [En línea] Barcelona Supercomputing Center. [Citado el: 1 de 6 de 2012.]
[46] Quantifying the potential task-based dataflow. Vladimir Subotic, Jose Carlos Sancho, Jesus Labarta, and Mateo Valero. Bordeaux, France : Euro-Par Conference, Lecture Notes in Computer Science (LNCS), 2011.

List of Acronyms and Abbreviations

ACML	AMD Core Math Library
AMD	Advanced Micro Devices
AMR	Adaptive Mesh Refinement
ANSI	American National Standards Institute
API	Application Programming Interface
APPML	AMD Accelerated Processing Math Libraries
APU	Accelerated Processing Unit
ARM	Advanced RISC Machines
BADW	Bayerische Akademie der Wissenschaften
BG/Q	BlueGene/Q
BLAS	Basic Linear Algebra Subprograms
BSC	Barcelona Supercomputing Center (Spain)
CaSToRC	Computation-based Science and Technology Research Center (Cyprus)

CCGRID	Cluster Computing and the Grid
CEA	Commissariat à l'Energie Atomique (represented in PRACE by GENCI, France)
CentOS	Community Enterprise Operating System
CINECA	Consorzio Interuniversitario, the largest Italian computing centre (Italy)
CINES	Centre Informatique National de l'Enseignement Supérieur (represented in PRACE by GENCI, France)
CooLMUC	Cooled Linux Cluster Munich
COP	Coefficient of Performance
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CSCS	The Swiss National Supercomputing Centre (represented in PRACE by ETHZ, Switzerland
CUDA	Compute Unified Device Architecture
DAS	Direct Attached Storage
DDR	Double Data Rate
DMA	Direct Memory Access
DNS	Domain Name Service
DP	Double Precision
DSP	Digital Signal Processing
DVFS	Dynamic Voltage Frequency Scaling
EDA	Electronic Design Automation
ESP	Effective System Performance
EtS	Estimated time to Solution
EU	European Union
FC	Fiber Channel
FFT	Fast Fourier Transform
FIFO	First In First Out
FP	Floating Point
FPGA	Field Programmable Gate Array
FZJ	Forschungszentrum Juelich
GbE	Gigabit Ethernet
GCC	GNU Compiler Collection
GDDR	Graphics Double Data Rate
GPFS	General Parallel File System
GPGPU	General Purpose Graphics Processing Unit
GPL	General Public License
GPU	Graphics Processing Unit
HDMI	High-definition Multimedia Interface
HLRS	High Performance Computing Center Stuttgart
HP	Hewlett Packard
HPC	High Performance Computing
HW	Hardware
IB	Infiniband
IBM	Formerly known as International Business Machines
ICHEC	Ireland's High Performance Computing Center
IDRIS	Institut du Développement et des Ressources en Informatique Scientifique (represented in PRACE by GENCI, France)
IEEE	Institute of Electrical and Electronic Engineers
IO (I/O)	Input/Output
IPC	Inter Process Communication

JKU	Johannes Kepler Universität
JSC	Juelich Supercomputing Center
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
LNET	Lightweight Network Library
LRZ	Leibniz Supercomputing Centre (Garching, Germany)
LU	Lower/Upper (decomposition)
MD	Molecular Dynamics
MIC	Many Integrated Core
MKL	Math Kernel Library
MPI	Message Passing Interface
MPP	Massively Parallel Processing
MS	Microsoft
NAMD	Not Another Molecular Dynamics program
NAND	Not AND gate
NASA	National Aerospace Agency
NP	Non-deterministic Polynomial
NTP	Network Time Protocol
NUMA	Non-Uniform Memory Access
NUMA-CIC	NUMAscale Cache-coherent Interconnect
NVVP	NVIDIA Virtual Profiler
OFED	Open Fabrics Enterprise Distribution
OpenCL	Open Compute Language
OpenMP	Open Multi-Processing
OpenSM	Open Subnet Manager
OS	Operating System
OTF	Open Trace Format
PCI	Peripheral Component Interconnect
PDU	Power Distribution Unit
PGAS	Partitioned Global Address Space
PGI	Portland Group Inc.
POSIX	Portable Operating System Interface
POTRA	Power Trace Analyzer
PRACE	Partnership for Advanced Computing in Europe
PSNC	Poznan Supercomputing and Networking Centre (Poland)
PUE	Power Usage Effectiveness
QDR	Quad Data Rate
QoS	Quality of Service
QPI	Quick Path Interconnect
QR	QR method or algorithm: a procedure in linear algebra to compute the eigenvalues and eigenvectors of a matrix
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
RDMA	Remote Data Memory Access
RISC	Reduce Instruction Set Computer
RNG	Random Number Generator
RPM	Revolution per Minute
RT	Run Time
RTC	Real Time Clock
SAN	Storage Area Network
SARA	Stichting Academisch Rekencentrum Amsterdam (Netherlands)

SAS	Serial Attached SCSI
SATA	Serial Advanced Technology Attachment (bus)
SDK	Software Development Kit
SGEMM	Single precision General Matrix Multiply, subroutine in the BLAS
SGI	Silicon Graphics, Inc.
SHMEM	Share Memory access library (Cray)
SIMD	Single Instruction Multiple Data
SLURM	Simple Linux Utility for Resource Management
SM	Streaming Multiprocessor, also Subnet Manager
SMP	Symmetric MultiProcessing
SNIC	Swedish National Infrastructure for Computing (Sweden)
SoC	System on Chip
SP	Single Precision, usually 32-bit floating point numbers
SPMV	Sparse Matrix Vector Multiplication
SRAM	Static Random Access Memory
SSD	Solid State Disk or Drive
SSU	Scalable Storage Unit
STFC	Science and Technology Facilities Council (represented in PRACE by EPSRC, United Kingdom)
TAU	Tuning and Analysis Utilities
TB	Tera (= 240 ~ 1012) Bytes (= 8 bits), also TByte
TCP	Transmission Control Protocol
TDP	Thermal Design Power
TFlop/s	Tera (= 1012) Floating-point operations (usually in 64-bit, i.e. DP) per second, also TF/s
Tier-0	Denotes the apex of a conceptual pyramid of HPC systems. In this context the Supercomputing Research Infrastructure would host the Tier-0 systems; national or topical HPC centres would constitute Tier-1
TUM	Technische Universität München
UiO	University of Oslo
UNICORE	Uniform Interface to Computing Resources. Grid software for seamless access to distributed resources.
UPC	Unified Parallel C
USB	Universal Serial Bus
UV	Ultra Violet (SGI)
UVAS	Unified Virtual Address Space
VHDL	VHSIC (Very-High Speed Integrated Circuit) Hardware Description Language
WCNS	Wroclaw Centre for Networking and Supercomputing
WP	Work Package

Executive Summary

This deliverable reports on the latest software developments in high performance computing, as identified by the PRACE-1IP, WP9 members. The developments reported on here are the result of experiments and measurements carried out by the project members on PRACE prototype computer architectures. The characteristics of these prototypes were selected in order to allow investigation into a number of key aspects relevant to high performance computing, namely interconnects, I/O, energy efficiency and accelerators. In this document a presentation of this work has been prepared under four major topics: programming languages, system software and software tools and a preliminary look on hardware. With a view towards Exascale computing, we will present our results and findings for each of these topics, based on which we will conclude with a set of recommendations.

1 Introduction

As the Exascale milestone approaches, academic and industrial institutions in the field of HPC have identified a number of issues that need to be overcome for the development of a practical and efficient Exaflop supercomputer. As an example, if one simply scales the power consumption of a Petaflop supercomputer by one thousand, it is clear that major breakthroughs in power efficiency and cooling are required to arrive at an architecture which would be viable at Exascale. Although power consumption is an obvious example of this need for innovation, other, perhaps more subtle, issues are identified, such as interconnect architectures, I/O systems and the need for dense massively-multithreaded devices to serve as coprocessors.

The methodology of WP9 of PRACE-1IP, which identified these topics early on in the Preparatory Phase of the project, is to develop and investigate each of these issues on dedicated hardware, through measurements and experiments on prototype architectures which have been designed to each address a subset of the outlined bottlenecks. The members of this WP enjoy access to a number of diverse architectures, such as GPU, FPGA and DSP clusters, clusters of directly water cooled nodes, ARM processor clusters, a system with cache-coherent shared-memory over hundreds of CPU cores and two prototypes with a novel parallel I/O design. This makes our consortium an ideal forum for identifying innovative solutions and proposing recommendations for next generation software and hardware designs.

The current document has precisely this goal.We focus on the status of software while giving a brief overview of the conclusions we have arrived regarding hardware design. Our presentation begins with an investigation in programming languages in Chapter 2, where we document the work carried out to assess developments in CUDA and OpenCL, some insight into pragma-based languages such as OpenACC and OmpSs and an assessment of the state of the PGAS version of C, namely UPC. Chapter 3 deals with system software, and will detail the achievements in issues related to operating systems and system management, resource management, data management and the MPI implementation and other communication libraries. In Chapter 4 we present the work done concerning various software tools, such as tools for the support of task-based and asynchronous programming, developments in intelligent collection and filtering of trace data, modelling of parallel applications and tools which assess the scalability of parallel software codes. An outlook towards hardware designs is presented in Chapter 5. Each chapter, or chapter section, concludes with a set of recommendations arrived to and justified by the results presented in this document. We conclude with a summary of our findings (Chapter 6). In the Annex of this, in section 7.1, we include a detailed description of the work carried out on Energy Aware System Software.

2 Programming Languages

In this chapter, we present the results of our investigation into a select set of programming languages, or programming models, which are targeting high performance computing systems.

Most of this chapter deals with developments in GPU programmability: first of all, the latest developments in mixing CUDA with MPI have been investigated, developments which mainly improve access to main memory. Secondly, we look into OpenCL where a number of kernels have been ported and compared with optimized C and CUDA. The third section is an initial investigation of the pragma-based OpenACC language which has been carried out with comparisons of performance and a look at ease of programmability.

The remaining two sections of this chapter deal with two programming languages: OmpSs and UPC. In both these sections, a port of the synthetic "Hydro" benchmark is carried out through which aspects such as scalability, performance and correctness, amongst others, are investigated.

For each language, a short description is given, with some code snippets when appropriate. After the presentation of our results we rate each language with a positive or negative mark (with the appropriate justification) in seven areas: scalability, performance, productivity, sustainability, correctness, portability and availability. Our conclusions are presented, with an attempt to give some specific recommendations concerning each programming language.

2.1 CUDA and MPI

2.1.1 *Description*

Nowadays Graphics Processing Units (GPUs) have been established as an excellent power-efficient solution for modern HPC systems and have already proved advantageous for certain classes of algorithms over traditional CPU clusters or massively parallel supercomputers. One of the most challenging issues here is to develop applications for multi-GPU systems since it is very often impossible to accommodate a problem on a single GPU on one hand and, on the other hand, one has to deal with a complex memory hierarchy of the underlying heterogeneous system.

Currently there are two APIs available for heterogeneous computing: the NVIDIA Compute Unified Device Architecture (CUDA) designed mainly for NVIDIA GPUs and OpenCL, an open source project hosted by the Khronos Group. While the later provides tools for developing cross-platform applications (also for multi-core x86 micro-architectures), the former API has a significant advantage due to recently introduced features that allow direct memory access (DMA) engines to access GPU data directly, moving memory from one GPU to another within the same node.

In particular, these features include GPUdirect technology [1] that enables peer-to-peer data transmission between peer-accessible-devices, i.e. those that are attached to the same IOH chip or different IOH chips if they are linked, e.g., via AMD's HyperTransport interconnect (currently, Intel's QPI and PCIe protocols are incompatible, so it's impossible to establish direct communication between GPUs that are connected to different Intel mainboard chipsets.) Another multi-GPU technology introduced in recent CUDA releases is CUDA IPC that allows different host (for example, MPI) processes to access the same GPU buffer if the processes operate on the same compute node.

The important underlying ingredient for both technologies is a Unified Virtual Address Space (UVAS), where each process within a node running on a 64bit OS can use a single address space for the host and all available CUDA devices of compute capability 2.0 and higher (e.g. for Fermi and Kepler GPUs). In particular, this address space is used for all allocations in host memory via the `cudaHostAlloc()`CUDA Run Time (RT) API function (or corresponding CUDA driver API function) and in any of the device memories via `cudaMalloc()` or `cudaMallocAsync()`RT API functions (or their driver API counterparts). Thus, any host pointer returned by `cudaHostAlloc()` can be used directly within CUDA kernels running on GPUs. An essential limitation of this GPUdirect technology is that it is restricted to a single host process: a virtual address from one process cannot be dereferenced in the address space of another. This limitation is diminished by a new family of CUDA IPC functions which provide the capability of exporting a memory handle to a GPU memory allocation from one process directly into the address space of another process within the same compute node. The API functions also provide an IPC mechanism for passing CUDA events between processes.

Below we detail some basic interface functions for both APIs (taken from the CUDA Toolkit reference manual, version 4.2).

1. GPUdirect:

`cudaDeviceCanAccess()` – enquiries peer-accessibility of a given device in the system for the current device.

`cudaDeviceEnablePeerAccess()` – enables registering memory on peer device for direct access from the current device.

`cudaMemcpyPeer()` – copies memory from device to another (peer-accessible) device. This function is asynchronous with respect to the host but serialized with respect to the all pending and future asynchronous work in the current device (`cudaMemcpyPeerAsync()` avoids this synchronization).

2. CUDA IPC:

`cudaIpcGetMemHandle()` – gets an interprocess memory handle for an existing device memory allocation.

`cudaIpcOpenMemHandle()` – opens an interprocess memory handle exported from another process and returns a device pointer usable in the local process.

`cudaIpcCloseMemHandle()` – unmaps memory returned by the previous API function. This does not affect the original allocation in the exporting process as well as imported mapping in other processes.

2.1.2 *Code Examples*

Example 1: Peer-to-peer communications between GPUs within the same process

Assume two GPUs on the system, with the ids marked by `gpuA` and `gpuB`:

```
int is_p2p_accessible = 0;
cudaSetDevice(gpuA);       //switch to device 'A'
cudaMalloc(&d_A, bytes);   //allocate memory on the device 'A'
cudaDeviceCanAccessPeer(&is_p2p_accessible, gpuB, gpuA);
if(is_p2p_accessible){
      cudaSetDevice(gpuB); //switch to device 'B'
      cudaDeviceEnablePeerAccess(gpuA, 0);
```

```
      kernelB<<<...>>>(..., d_A, …);
                          //kernelB has access to ptr d_A
      cudaDeviceDisablePeerAccess(gpuA);
}
```

or:

```
if(is_p2p_accessible){
      cudaSetDevice(gpuB);       //switch to device 'B'
      cudaMalloc(&d_B, bytes);   //allocate memory on the device 'B'
      cudaDeviceEnablePeerAccess(gpuA, 0);
      cudaMemcpyPeer(d_B, gpuB, d_A, gpuA, bytes);
                                 //copy d_A to d_B
      cudaDeviceDisablePeerAccess(gpuA);
}
```

Note, while in this example `kernelB` is executed on device 'B', it has access to memory allocated on device 'A' (if device 'A' is peer-accessible for 'B') via the PCIe bus. As was mentioned above, this requires the application to be run on a 64-bit OS and devices of compute capability 2.0 and higher.

Example 2: GPU – Aware MPI

Code example without MPI integration:

```
//Process A (sender):
cudaMalloc(&s_device_ptr, bytes);
void *s_host_buffer = malloc(bytes);
cudaMemcpy(s_host_buffer, s_device_ptr, bytes, cudaMemcpyDeviceToHost);
MPI_Send(s_host_buffer, bytes, MPI_CHAR, 1, 1, MPI_COMM_WORLD );
…
//Process B (receiver):
void *r_host_buffer = malloc(bytes);
cudaMalloc(&r_device_ptr, bytes);
MPI_Recv(r_host_buffer, bytes, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
cudaMemcpy(r_device_ptr, s_host_buffer, bytes, cudaMemcpyHostToDevice);
```

Code example with MPI integration (enable `MPI_Send/Recv` directly from/to GPU):

```
//Process A (sender):
cudaMalloc(&s_device, bytes);
//work with s_device buffer on GPU A
MPI_Send(s_device, bytes, MPI_CHAR, 1, 1, MPI_COMM_WORLD );
//Process B (receiver):
cudaMalloc(&r_device, bytes);
MPI_Recv(r_device, bytes, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
```

Note that the second pattern exploits the UVAS property exposed by the CUDA 4.0 (and later) toolkit. It is currently supported by the MVAPICH2 MPI library.

Example 3: CUDA IPC technology

Code example of peer-to-peer communication for two MPI processes within a compute node (assume that each process operates a separate GPU):

```
//Process A (sender):
cudaIpcMemHandle_t s_handle;
```

```
cudaMalloc(&s_dev, bytes);
cudaIpcGetMemHandle(&s_handle, s_dev);
MPI_Isend(s_handle, sizeof(cudaIpcMemHandle_t), ... );
…
//Process B (receiver):
cudaIpcMemHandle_t r_handle;
MPI_Irecv(r_handle, sizeof(cudaIpcMemHandle_t), ... );
cudaIpcOpenMemHandle(&r_dev, r_handle, cudaIpcMemLazyEnablePeerAccess);
kernelB<<<...>>>(...,r_dev, ...);
cudaIpcCloseMemHandle(r_handle);
```

The IPC functionality is restricted to devices with support of unified addressing on Linux operation systems. The `cudaIpcOpenMemHandle()` function in this example can attempt to enable peer access between two devices as if the user called `cudaDeviceEnablePeerAccess()`, as demonstrated in Example 1.

2.1.3 *Experiences and Results*

In order to test the CUDA UVAS based technologies we modified the STREAM benchmark suit that was available for single GPU bandwidth measurements. All test runs were performed on CaSToRC's prototype machine that comprises of 8 GPU nodes (2 NVIDIA M2070 GPUs per compute node) with Mellanox interconnect. Each node is equipped with a two-socket mainboard, with Nehalem Xeon CPUs operating at 2.7GHz. The software configuration used for the benchmark includes the CUDA 4.1toolkit (with NVIDIA driver version 285.xx.xx.xx), the GNU compiler v 4.4.5 and MVAPICH2 (1.8a2) for MPI.

The latest version of MVAPICH2 incorporates optimized support for GPU to GPU communications via the standard MPI interface. In particular, it includes support for point-to-point and collective operations, pipelined data transfer with automatically provided optimizations, GPUdirect (peer-to-peer) and CUDA IPC.

Taking the CUDA version of the STREAM benchmark as a template we implemented three types of tests: 1) for 'pure' intranode peer-to-peer communication within a single host process, 2) for intranode/internode MPI communications using CUDA IPC 3) standard MPI communications that served in our case as a reference implementation. (As in the standard STREAM benchmark we considered both double and float data types). All versions provided no difficulties in implementation. However the first two cases required much less programming efforts due to UVAS technology. In general, the advantage of the later is that the programmer does not need to keep account of which memory space a given pointer belongs to, and also does not need to explicitly indicate the direction of memory copies (e.g. device-to-host, host-to-device and device-to-device). The only consideration here is that the host memory must be page-locked, something which may potentially degrade overall system performance if too much memory is allocated. Due to its simplicity, it is quite straightforward to incorporate these technologies in existing MPI-CUDA applications.

In Figure 1 we show results obtained using our adapted STREAM benchmark. We see that in all three cases, and more importantly in the case of peer-to-peer communications and MPI communication, we obtain values of the sustained bandwidth as expected given the PCIe specification and QDR Infiniband respectively. This shows that the overheads involved in the NVIDIA stack which abstracts the memory hierarchy, e.g. translating addresses and setting up the buffers for the node-to-node communication, have minimal effect on the sustained bandwidth.

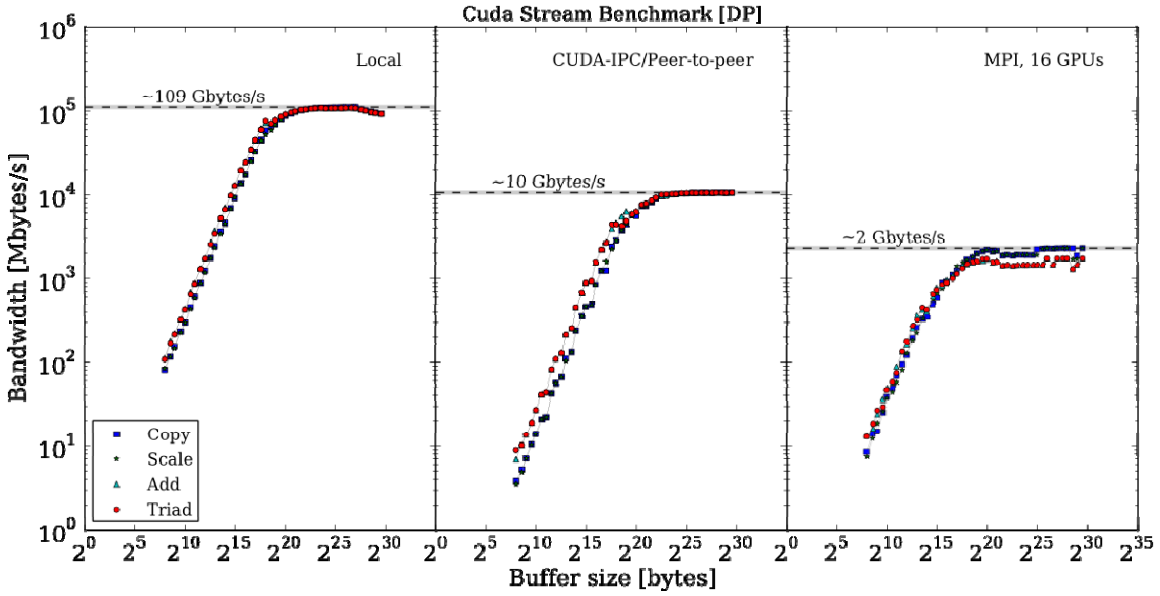


Figure 1: Results of our adapted STREAM benchmark. We show results for the memory bandwidth within the GPU (left), between GPUs on the same PCIe bus (centre) and for off-node GPUs, connected over a QDR Infiniband interconnect (right).

2.1.4 *Pros and Cons*

In Table 1 we have tabulated the Pros and Cons of using this hybrid model for programming systems of distributed GPUs.

	Pros	Cons
Scalability	The main objective of the considered technologies is to avoid unnecessary system memory copies that will considerably reduce interprocess communication time and, as a consequence, improve scalability of a Multi-GPU application.	
Performance	The technology will improve performance if communication is the main bottleneck.	Currently it is supported for intra-node data transfers, and even in this case is restricted to single IOH chip configurations in the case of Intel hardware.
Productivity	Code development is straightforward and similar to standard MPI communication patterns. The UVAS eliminates necessity to distinguish between host and device pointers which simplifies code structure and provides more flexibility for programming on heterogeneous systems.	It may require minor code redesign for existing CUDA applications, especially if they rely on old communication patterns, in which one creates an intermediate buffer on the host
Sustainability	CUDA has become the *de facto* programming environment for	

	Pros	Cons
	hybrid HPC systems with NVIDIA GPUs. In particular, taking into account that NVIDIA has already cooperated with interconnect vendors to incorporate their Multi-GPU solutions into the HPC market, the technologies considered here are expected to become standard approaches in the development of large-scale CUDA applications	
Correctness	The detailed multi-GPU technologies introduce no further complexities in code debugging	
Portability	Code may be run on any accelerator cluster with NVIDIA GPUs, even if there is no interconnect supporting these technologies	Currently, there is only one MPI implementation that supports GPUdirect with the CUDA API, and this is MVAPICH2
Availability	Both the NVIDIA CUDA toolkit and MVAPICH are freely available	

Table 1: A list of advantages and disadvantages in using hybrid CUDA+MPI programming for multiple GPUs, based on our experience in this work.

2.1.5 *Conclusions and recommendations*

At present, MPI + CUDA still remains the most straight forward and portable way of writing multi-GPU code for NVIDIA devices. Ease of programmability is set aside in favour of code performance; explicitly managing data transfers between devices means more control over where data buffers reside, which usually leads to better memory management and therefore more optimal code.

In this specific work we have taken a closer look at the latest developments NVIDIA has incorporated in its software, which primarily concern data transfers to and from the GPU, which is the main bottleneck in most multi-GPU codes. Developments such as the Uniform Virtual Address Space and direct communication over PCIe indicate that NVIDIA is looking into ways of making data accesses from the GPU both faster and less cumbersome for the developer. This fact, along with the established *de facto* status of MPI in parallel programming, encourages further use and maintenance of code-bases written in CUDA/MPI.

2.2 OpenCL

2.2.1 *Desription*

OpenCL (Open Computing Language) is an open, royalty-free standard for general-purpose parallel programming of heterogeneous systems, maintained by the Khronos Group [2]. It provides a framework for cross-platform computing on a range of modern processors, including CPUs, GPUs and APUs. The framework includes a programming language, based on C99, and an API. The language is used for writing specific functions (kernels) executed on OpenCL devices.

OpenCL supports both data-parallel programming and task-parallel programming. It is also interoperable with MPI and other standard libraries. An overview of the OpenCL architecture, its execution and memory model are described in [3].

Since the publication of the PRACE-PP deliverable D6.6 [3] and PRACE-1IP deliverable D9.2.1 [4] a new version of OpenCL has been released: the OpenCL 1.2 Specification (rev. 15, November 15, 2011) and OpenCL 1.2 Extensions Specification (rev. 15, November 15, 2011) [5]. OpenCL 1.2 retains backwards compatibility with previous 1.0 and 1.1 versions, but enhances functionalities and performance with new features. The most important for high performance computing are: custom devices and kernels, device partitioning and separate compilation and linking of objects.

OpenCL (v1.0, v1.1 or v1.2) supports a range of processors found in personal computers, servers and handheld/embedded devices, including NVIDIA GPUs, AMD Fusion APU series, AMD GPUs and CPUs, ARM GPU, Intel CPUs and IBM servers.

Libraries for scientific computing in OpenCL include: APPML (BLAS, FFT) for AMD GPUs (earlier delivered as part of ACML), CUBLAS (BLAS) and CUFFT (FFT) for NVIDIA GPUs, ViennaCL (Linear Algebra and Iterative Solvers) with support for NVIDIA and AMD/ATI GPUs [6], CMSoft OpenCL FFT and Linear Algebra.

Tools and development kits for OpenCL software development are available and evolving. The main programming environment is the AMD Accelerated Parallel Processing SDK, with full support for OpenCL 1.2. The SDK includes the AMD APP Profiler for performance analysis of the code, and the AMD APP KernelAnalyzer – a tool for static analysis of the performance of OpenCL C kernels. There is also the AMD CodeAnalyst for Windows with OpenCL support, which collects and analyses the OpenCL API execution performance from both CPUs and GPUs.

The gDEBugger may also support software developers in the process of debugging applications, including the OpenCL kernels [6]. It is available as a Microsoft Visual Studio plug-in on Windows and a standalone program on Linux. There are also other SDKs and frameworks available, delivered by Intel [7], SNU-SAMSUNG [8] and IBM [9]. The Java Bindings to OpenCL (JOCL) enable applications running on the JVM to use OpenCL [10], and PyOpenCL allows access to the OpenCL API from Python [11]. There is also a PGI OpenCL compiler for ARM CPUs which support OpenCL 1.1 [12].

2.2.2 *OpenCL Code Example*

An OpenCL kernel is expressed as a C-language routine (Figure 2).

```
__kernel void gpuReduction(__global int* inTab,
                           int nelems,
                           __global int* out) {

  uint loops = nelems/GROUP_SIZE;
  uint loop;
  if(nelems%GROUP_SIZE > 0) { loops++; }
  uint const id = get_local_id(0);
  uint const groupId = get_group_id(0);
  __local int sdata[GROUP_SIZE];
  if(id == 0) { out[0] = 0; }
  for(loop = 0; loop < loops; loop++) {
    sdata[id] = 0;
    uint i = loop*GROUP_SIZE + id;

    if(i < nelems) { sdata[id] = inTab[i]; }
    barrier(CLK_LOCAL_MEM_FENCE);

    for(unsigned int s=GROUP_SIZE/2; s>0; s>>=1) {
      if (id < s) {
        if(sdata[id + s] > sdata[id]) {
          sdata[id] = sdata[id + s];
        }
      }
      barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (id == 0) {
      if(sdata[id] > out[0]) {
        out[0] = sdata[id];
      }
    }
  }
}
```

declaration of kernel method

local memory array declaration

threads synchronization

parallel max reduction

Figure 2 Simple OpenCL kernel

2.2.3 *Experience & Results*

The work done within this task concentrated on an evaluation of OpenCL as a programming language and a development environment for efficient scientific computations. We also considered the porting effort and differences between OpenCL and CUDA, as the most mature and most widely used development platform for GPGPUs. The evaluation started by implementing several synthetic kernels in OpenCL (from the Euroben benchmark set), and after observing promising results, we continued with more realistic scientific applications, namely DL_POLY and NAMD. The early results regarding Euroben were reported in [4]. Experiences and results are discussed below for each application.

1. DL_POLY

DL_POLY is a molecular dynamics simulation application developed by the Science and Technology Facilities Council in Daresbury Laboratory, UK. A domain decomposition approach has been used for parallelizing DL_POLY using the MPI library. DL_POLY is highly efficient and scalable to thousands of CPU cores. A DL_POLY port for GPUs exists with NVIDIA CUDA, which has been developed by the Irish Centre for High-End Computing (ICHEC) in collaboration with Daresbury Laboratory. A “constraints shake” DL_POLY component has been ported to OpenCL by the Wroclaw Centre for Networking and Supercomputing (WCNS) in collaboration with ICHEC and Daresbury Laboratory.

DL_POLY's "constraints shake" component has been ported to OpenCL from existing CUDA code. During this porting work several OpenCL language properties and limitations have been identified. The two programming languages differ at most in handling data structures. In the CUDA version of the component the structures are widely used both in the host code and in the GPU kernels. According to the OpenCL 1.1 specification, section 6.8 [13], structures in this language cannot contain OpenCL objects (buffers, images etc.). Attempts to use structures with buffers (a direct port from the CUDA code) failed, as the data inside a buffer were not accessible from the kernel code. This problem has forced a change in the way of handling data structures and passing individual objects to kernels directly as arguments, one by one.

Another difference is the way one defines the number of threads in groups, with no straightforward transition. CUDA uses specific operators `<<<…>>>` in which a developer declares two `dim3` structures (one for the number of groups and one for the number of threads). For example, a declaration `<<<(1, 300, 1), 64>>>` tells the CUDA API to run 300 groups in the 2nd dimension, each with 64 threads.

In OpenCL a developer must declare how many threads as a whole will be used to execute the kernel. Initially these numbers were declared in the following way: number of threads in group as (1, 64, 1), and number of threads globally as (1, 300×64, 1). This solution gave 300 groups in the 2nd dimension, with 64 threads each. Unfortunately it turned out that, with the above settings, the threads in OpenCL were placed in the 2nd dimension, while in the original CUDA code threads were located in the 1st dimension. Correct size declarations in OpenCL are respectively: (64, 1, 1) and (64, 300, 1). That still gives 300 groups in the 2nd dimension with 64 threads placed in the 1st dimension.

We propose a simple algorithm for converting CUDA group sizes into OpenCL compatible ones:

CUDA groups and threads: `<<<(a, b, c), (i, j, k)>>>`

OpenCL group threads: `(i, j, k)`

OpenCL all threads: `(a*i, b*j, c*k)`

The differences presented above were the most significant ones encountered while porting the "`constraints_shake`" module. If a developer is familiar with both languages and knows about the differences between them, the code can be ported faster and with fewer mistakes.

It must be mentioned that a direct port will usually not use all the advantages of OpenCL and the resulting code needs further optimization in this respect.

OpenCL limitations which make a developer's life harder:

- no C++ like templates in the kernel code – this limitation results in increasing the number of lines of OpenCL code,
- no support for passing structures as OpenCL kernel arguments if the structure elements are pointers – this limitation means that each pointer field must be passed to the OpenCL kernel separately and the size of the OpenCL code grows.

The great improvement for developers is the AMD gDEBugger tool, which lets one debug OpenCL kernel code. Still, the tool was not always working properly with the rest of the OpenCL environment (SDK, drivers) and for a period of time was not available on Linux in the newest version. The debugger together with the new tools for code analysis (AMD APP Kernel Analyzer and AMD Code Analyst) is simplifying the development and performance tuning of OpenCL applications for AMD target architectures.

DL_POLY test runs have been executed on a local WCNS GPU machine – 2× AMD Radeon HD 6950 and on Tesla M2050 GPUs, at CEA. Test runs were performed for 2 MPI processes and 2 OpenMP threads on the TEST7 (Gramicidin A, molecules in Water) benchmark for double floating point precision. The test case is an example of a real scientific problem. Figure 3, Figure 4 and Figure 5 show the average duration time per invocation for the "`constraints_shake`" component initialization, particular kernel calls, and read/write from/to GPU operations.

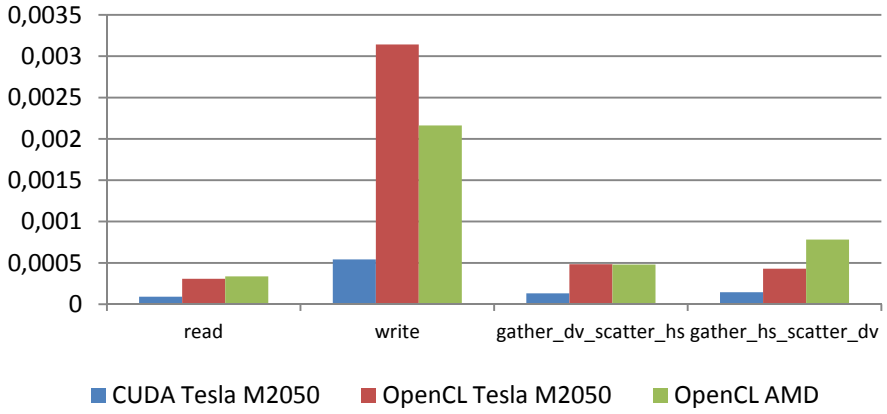


Figure 3. CUDA vs OpenCL for DL_POLY constraints shake component

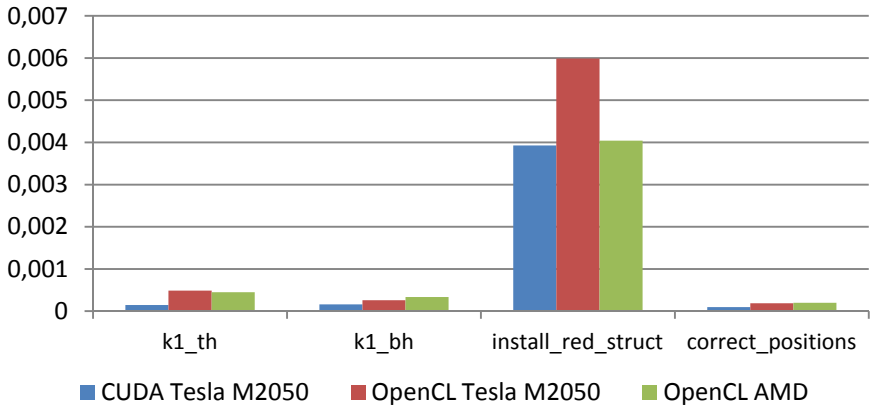


Figure 4. CUDA vs OpenCL for DL_POLY constraints shake component

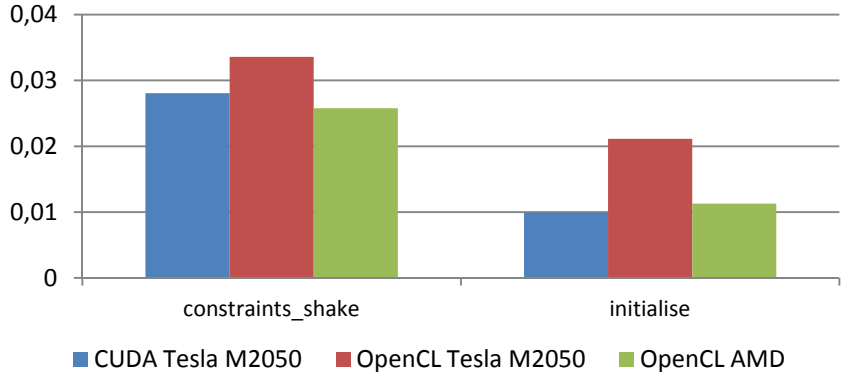


Figure 5. CUDA vs OpenCL for DL_POLY constraints shake component

Performance results show that the OpenCL implementation works slower than the CUDA version of almost all "`constraints_shake`" component algorithms. The biggest difference between average duration time per invocation for Tesla M2050 GPU is for the

`write` operations (OpenCL code is 5× slower than CUDA code) and the kernel `gather_dv_scatter_hs` (OpenCL code is 3× slower than CUDA code). For other kernels OpenCL calls are 2-3× slower than particular CUDA calls. OpenCL execution times on AMD GPUs are shorter than for Tesla M2050 GPUs for `initialization`, `write` and `install_red_struct` and `k1_th` kernels. For other calls, execution times on AMD GPUs are longer than for CUDA code run on Tesla M2050. The reason for this difference in execution times on different hardware is that the OpenCL code was written for NVIDIA GPUs, and so does not take into account the specificities of AMD GPUs.

Work on an OpenCL port of DL_POLY's algorithms is continuing within the PRACE-2IP project.

2. NAMD

NAMD is a parallel molecular dynamics code which performs simulations of large biomolecular systems [17]. The application is parallelized based on Charm++ parallel objects and scales to hundreds of processors on high-end parallel platforms and tens of processors on commodity clusters.

NAMD is partly ported to CUDA, and thus able to take advantage from resources equipped with NVIDIA GPUs. The application uses the GPU for nonbonded force evaluation. PSNC, with support from WCNS, started to work on the OpenCL port to enable the application to run on other GPU architectures. NAMD authors were contacted, and they are aware of these efforts. There was no closer collaboration established.

The main task was to implement an OpenCL version of the kernel performing the nonbonded force evaluation. The automated translators from CUDA to OpenCL were also tested (i.e. SWAN), for comparison purposes, with no positive results. The CUDA kernel is quite complex, includes macros, and the tools couldn't cope with this complexity.

Preliminary tests of the OpenCL and CUDA implementation were performed using the test case "apoa1", downloaded from the official NAMD website [18]. A comparison of the results generated by the OpenCL and CUDA kernels revealed differences and correctness problems in the OpenCL kernel. Further improvements have been introduced into the kernel, but the process is still ongoing due to difficulties with tools for debugging a certain kernel code encountered during the project timeframe.

Several environments have been tested, including the NVIDIA Nsight and AMD gDEBugger. On the GPU cluster at PSNC (available in PRACE as a Tier-1 machine) one node was 'isolated' with a Windows 7 OS on-board and MS Visual 10 with the NVIDIA Nsight Profiler installed, but the profiler supports only OpenCL API calls and no kernel debugging. The gDEBugger installed on the PRACE prototype at PSNC with AMD Brazos (E-350) was difficult to be used remotely due to problems with a GPU's visibility. Local usage works fine, but is not always possible, depending on the location and accessibility of the target architecture.

Work on the OpenCL port of NAMD is continuing within the PRACE-2IP project. The new tools from AMD are being evaluated and it will be seen if they address previous problems with the debugging and the analysis process.

3. Euroben

The Euroben benchmarks `mod2am/MxM`, `mod2as/SPMV` and `mod2f/FFT` were ported to OpenCL at WCNS and PSNC. We report benchmarks results on NVIDIA GTX480, AMD/ATI Radeon HD 6950 and AMD Brazos platform (Zacate E-350: CPU 1.6 GHz 2 cores, with AMD Radeon HD 6310 492 MHz). They are compared to sequential benchmarks

written in the C language, with results gathered on an x86_64 machine with Intel Core i7 CPU 3.20 GHz.

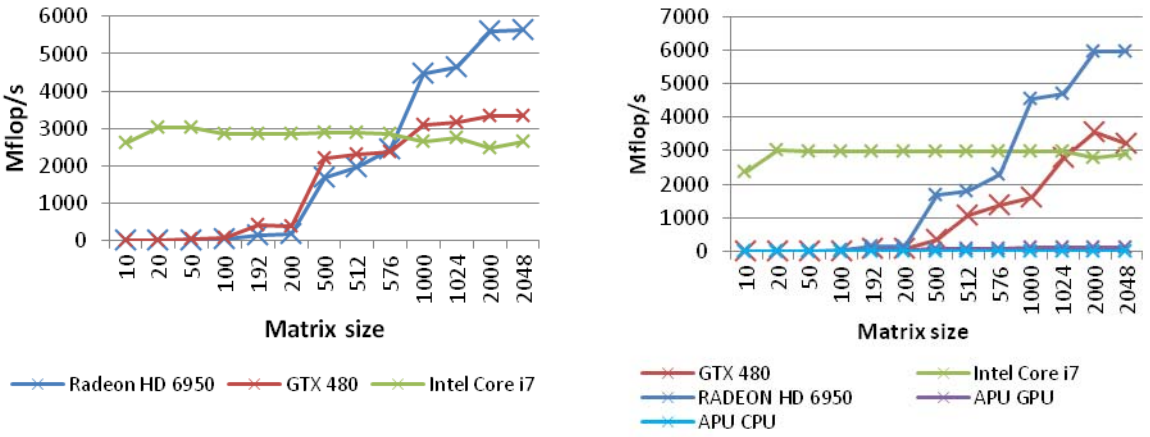


Figure 6 OpenCL `mod2am` results: DP (left), SP (right)

The AMD Brazos platform supports only single-precision floating-point arithmetic. Thus, for comparison purposes, the benchmarks have been run in both double- (DP) and single-precision (SP). Figure 6, Figure 7 and Figure 8 show results of all the benchmarks.

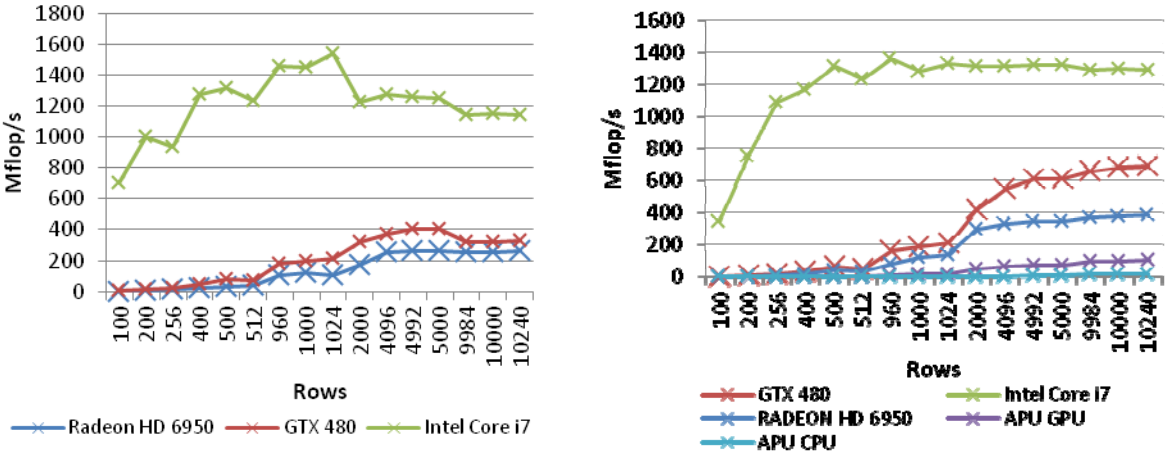


Figure 7 OpenCL `mod2as` results: DP (left), SP (right)

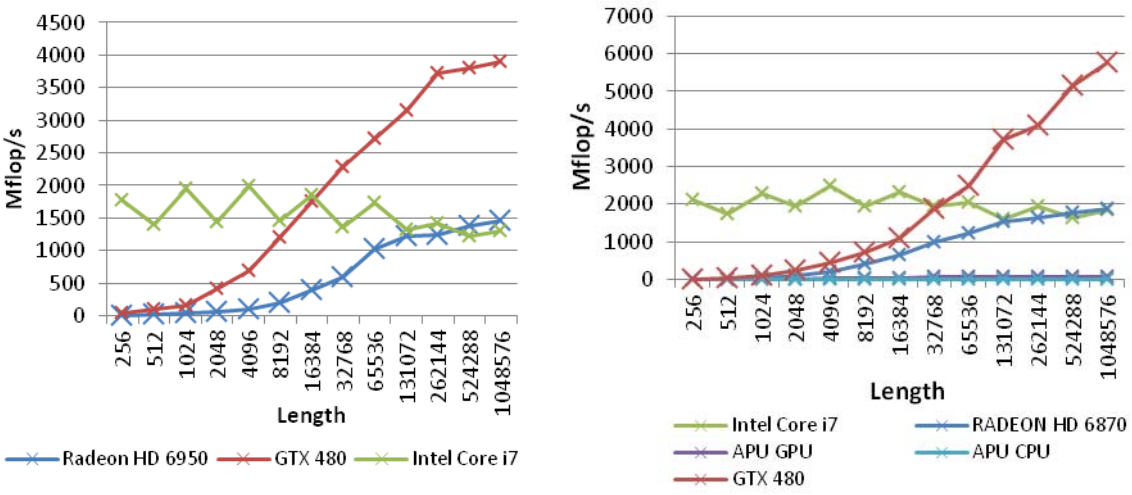


Figure 8 OpenCL `mod2f` results: DP (left), SP (right)

Results of the `mod2am` and `mod2f` benchmarks (Figure 6 and Figure 7), for both DP and SP, show that the OpenCL GPU version reaches better performance for bigger problems. The `mod2am` implementation gets better results on NVIDIA on smaller problems, probably because of higher clock frequency, while AMD is better for bigger problems.

In Figure 8, one can see that OpenCL implementations of `mod2as` have worse performance on all devices than the C CPU version. This is probably because of the high number of memory operations in this benchmark and the higher clock frequency of the CPU. The best OpenCL result was achieved by NVIDIA in both DP and SP tests.

GPUs achieved less Mflop/s in double-precision than in single-precision in every run. This loss of performance is due to increasing of the precision, but also inability to use the texture memory, as there is no image format for double values in OpenCL.

The preliminary results presented above gathered on the APU were, as expected, worse than on other platforms, due to the smaller clock frequency of both the CPU and the GPU. The APU is designed for low power consumption so the charts could change completely if the power consumption factor would be also considered.

In order to better test the AMD Brazos platform the `mod2am` benchmark was also prepared using an OpenCL BLAS implementation (APPML). We have tested BLAS 1.6 and 1.8 versions of the library. The porting proved to be very simple and required only few modifications in the code, namely changing:

`clGetDeviceIDs(CL_DEVICE_TYPE_GPU)` to
`clGetDeviceIDs(CL_DEVICE_TYPE_CPU).`

As a result we tested several different versions of the benchmark: the PRACE reference implementation, an optimized CPU implementation (using either ACML or MKL libraries) and the OpenCL BLAS implementation for both CPUs and GPUs (Figure 9).

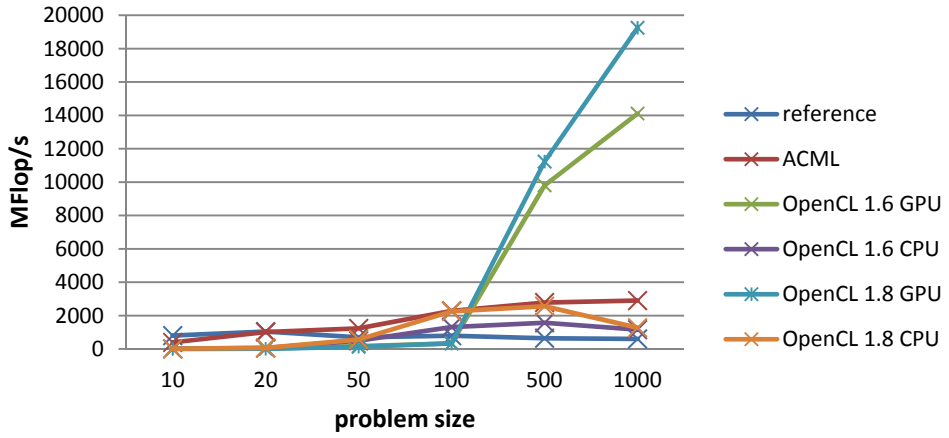


Figure 9: OpenCL `mod2am` results on APU (SP)

As expected, the results show very good performance of the optimized libraries compared to the basic implementations. For small problem sizes the performance of the OpenCL implementations seem to be inferior compared to the ACML version. One of the reasons is an overhead of calling the complex OpenCL stack versus just calling a simple routine. It can be seen that as the size of the problem grows, the OpenCL CPU version is able to achieve similar results to that of the specialized library, which shows that the OpenCL code can be efficient also for x86 CPU cores. The cause for the regression of the performance of the OpenCL version for the largest matrix size is currently under investigation.

The immense increase in performance of the GPU versions starting from a certain size can be explained by the logic of the APPML library – if the problem size is not large enough the library will use only a limited number of cores. That, in conjunction with the low GPU clock frequency, results in low performance for small cases and high performance when the parallelism of the GPU can be fully exploited.

2.2.4 *Pros & cons*

Table 2 lists our understanding of the pros and cons of using OpenCL, based on our experiences within this work.

	Pros	**Cons**
Scalability	Massive data parallel language. Scales extremely well and achieves very high performance on SMP systems.	
Performance	Has big potential for massive-parallelism. It is possible to achieve very high performance on inexpensive GPU hardware.	The code should be architecture-oriented. PCI bus data transfer is still a bottleneck.
Productivity	Developing code is quite easy for developers that are familiar to programming in C or CUDA. Tools exist for simplifying software development, and scientific libraries are evolving.	Obtaining very efficient kernel code may require more effort, experience and using device specific information. Tools for memory problem detection and debugging are not always sufficient for target architectures other than AMD.

	Pros	Cons
Sustainability	The Khronos Group consists of many industry-leading companies and institutions including AMD, IBM, Intel and NVIDIA.	-
Correctness	-	-
Portability	Excellent, as it may run on a number of architectures. The kernel code can be easily transferred from one architecture to another. The compiler is built into the runtime.	To reach optimal performance it may be required to adjust the code to a specific device.
Availability	Open and royalty-free standard. It is actively maintained and developed.	

Table 2: OpenCL pros and cons

2.2.5 *Recommendations*

OpenCL itself is a powerful tool offering great possibilities to HPC. The standard is evolving, and it may be seen that the Khronos Group takes into account the developer's community requests.

Developing OpenCL code still needs more effort than developing CUDA code. The programming environments and tools are becoming more mature, especially regarding debugging and performance analysis, but tools for different target architectures vary in both quantity and quality. OpenCL is a natural choice if an application is to be developed for AMD target architectures, since on this architecture one finds the most mature tool chain and support available (e.g. community forums).

OpenCL code can be run on different platforms – GPUs, APUs and multicore CPUs from different vendors. Although it is possible, the results presented above and previous research [14][15][16] show that OpenCL code still may require some optimization to achieve good performance on different hardware. Thus OpenCL is a natural choice in cases where portability is one of the most important factors of the application. Where, for instance, it is important that the application runs on the latest architectures, as long as they are OpenCL enabled. Subsequent adjustments can then be made if performance becomes critical.

In the case where one wishes to port a code from CUDA to OpenCL, the application model and algorithms involved need to be reconsidered. This is important due to several performance improvements available, when considering the hardware characteristics, and differences with other frameworks, like CUDA. Porting may require changes, in the data structures as well as in other aspects, to achieve good performance.

2.3 OpenACC

2.3.1 *Description*

OpenACC is an open standard for directive-based programming of heterogeneous computers. The porting process involves incrementally adding directives to existing CPU code. More specifically, OpenACC supports the host-device execution model, where the majority of computation occurs on the host (e.g. CPU) and the compute-intensive code regions are offloaded to the accelerator device (e.g. GPU). The host is in charge of coordinating both host and device execution. Directives are used for mapping the code's loop-level parallelism to the various levels of hardware concurrency on the accelerator.

Since OpenACC is for programming in the host-device execution model, it is important to note that the host and device often have separate memory spaces. Therefore, OpenACC includes the ability to manage data movement between the host and device. Directives are used to inform the compiler about data movement, and the compiler then generates runtime library calls to perform the data movement. Various directives allow the programmer to keep the data resident on the device across accelerator region launches. Additionally, in order to allow tuning for the accelerator's on-board memory hierarchy, OpenACC includes directives that provide cache hints to the compiler.

Hybrid OpenACC programming is currently supported with distributed memory-based approaches. Also, it is possible to use OpenMP alongside OpenACC. In addition, OpenACC provides a directive to make the address of device data available on the host – this feature allows mixing of OpenACC with other accelerator languages (e.g. NVIDIA's CUDA) and numerical libraries.

Currently, three compiler vendors support all or part of the OpenACC standard: CAPS, Cray and PGI. Since the standard is so new, programming tools such as debuggers and performance profilers are in the early stages of support. In terms of performance analysis tools, there are a number of vendors that appear to have at least some support (e.g. University of Oregon's TAU, Cray's perftools, CAPS Performance Analyzer, NVIDIA's NVVP, etc). Debugger support is claimed by Allinea's DDT product and Rogue Wave's TotalView product; however, we have had some challenges using these tools on non-trivial OpenACC codes.

Although OpenACC is currently only supported on NVIDIA-based systems, there are no constraints within the specification that limit hardware support for other vendors. Looking toward the future, OpenACC can be seen as a precursor to a possible upcoming OpenMP standard that supports accelerators. The creators of the OpenACC API (CAPS, Cray, NVIDIA, and PGI) are all members of the OpenMP Working Group on Accelerators. In order to address the complexities of heterogeneous systems, a new OpenMP standard would need to feature the following: coordination of host-device execution, explicit management of data movement, and enhanced control for mapping loop-level parallelism onto the appropriate levels of hardware.

2.3.2 *Code examples*

```
// Example #1:  Simple kernel execution with asynchronous launches
// Asynchronously issue work to the accelerator
#pragma acc parallel loop copy(a[0:n]) async(async_id)
for(i = 0; i < n; i++) {
      a[i] += value;
}
```

```
// Host does some work (e.g. counting) while waiting for acc region to
finish
while(!acc_async_test(async_id)) {
      counter++;
}

// Example #2:  OpenACC and CUDA interoperability
// Wrapper for launching the CUDA kernel
void smul_vector_cuda(real_t vin[], real_t vout[], int vsize) {
      smul_vector<<<NBLOCKS, NTHREADS>>>(vin, vout, vsize);
}
// Allocate and initialize the device data and then call the CUDA wrapper
void cuda_interop() {
      [...]
      #pragma acc data create(vin[0:VSIZE])  copyout(vout[0:VSIZE]) {
            // Initialize the data on the device
            #pragma acc parallel loop
            for (i = 0; i < VSIZE; i++) {
                  vin[i] = ((float)i);
                  vout[i] = 0.0f;
            }
            // Call the wrapper to launch the CUDA kernel
            #pragma acc host_data use_device(vin, vout)
            smul_vector_cuda(vin, vout, VSIZE);
      }
}
```

2.3.3 *Experience & results*

In order to investigate the programmability of OpenACC and the performance of the associated compilers, we ported the Hydro benchmark to use OpenACC. This provides an interesting comparison against the existing CUDA implementation of Hydro. This study mainly used the Cray and PGI compilers because these were already available on our systems at CSCS. In the future, we look forward to study the CAPS compiler's features and performance in greater depth.

As a first step, we ported the simplest existing Hydro implementation to OpenACC. This was straightforward; however, the implementation was not amenable to the GPU architecture because the amount of work per loop structure was too small. This issue caused a large amount of overhead due to the outer loop launching many kernels that each operated on a small amount of data. With this initial implementation, the trade-off was between the number of kernel launches and the amount of work per kernel. Performance was very poor compared to the CPU-only version.

Due to this problem, we focused on a slightly more complex version of Hydro that allowed a variable amount of work per loop structure – the initial "2D sweep" version is called `HydroC99_2Dmpi`. The first step for porting this code was to ensure the x86 code was properly vectorized with each compiler. Since this code is C-based and uses pointer arrays, we needed to ensure the compiler understood that there were no vectorization hindrances due to overlapping arrays – the C `restrict` keyword was used to accomplish this. Additionally, the various compilers had difficulty vectorizing some of the multi-dimensional array accesses; so, these were converted to 1-D arrays, which was time-consuming. With OpenACC, the goal

of the developer is to expose parallelism to the compiler, and similar loop-level and data optimizations are beneficial to both CPU and accelerator architectures.

After the appropriate unfolding to 1-D arrays was achieved, allowing the compilers to vectorize efficiently the array accesses, we focused on adding OpenACC accelerator region directives to each of the important loop structures in the main `hydro_godunov` solver. Since the current compilers support different accelerator region types (`parallel` vs `kernels`) two separate versions of the application were created. Both implementations also needed explicit `loop` directives at the various levels of loop hierarchy. Additionally, the PGI compiler required an `independent` clause on the `loop` directives in order to enable parallelization due to potential dependencies with array index calculations. However, the most challenging effort here was to ensure that the direction of data movement between each accelerator region was correct – this is specified with `copy()`, `copyin()` and `copyout()` clauses on each accelerator region. During this step, it was essential to monitor the correctness of the results in order to fix data movement bugs. At this early stage in the porting process, it is important to note that performance was far worse than simply running the x86 version. Essentially, each accelerator region generated a number of costly data transfers to and from the device, and this completely dominated the runtime of the application.

After all of the `hydro_godunov` solver's loops were running on the accelerator, the next step was to ensure the many arrays were resident on the device for as long as possible during the execution of the `hydro_godunov` solver. There were many arrays that needed to be added to an OpenACC data region that surrounded the `hydro_godunov` solver's outer loop. At this level in the code, the majority of the arrays could simply be allocated on the device using a `create()` clause, while the `uold` array was copied to and from the device using a `copy()` clause. Additionally, all of the `hydro_godunov` solver's accelerator regions were updated to use the `present()` clause, which tells the compiler and runtime that the specified arrays already exist on the device. Although this incremental step provided a huge performance increase over the non-resident data version, the performance was still quite slow due to costly work in the function `compute_deltat()` for each timestep.

For the next optimization step, we discovered that some of the loops were not being parallelized in an efficient manner. For the Cray implementation with `parallel` accelerator regions, `collapse` clauses had to be added to all of the triply nested loops because the outer loop had a small trip count, which caused the number of thread blocks to be small with lots of serial work. The `collapse` clause caused a compiler loop transformation that ensured a large number of thread blocks with many threads per block. For the PGI implementation with `kernels` accelerator regions, the `gang` and `vector` clauses were used to explicitly manage the loop scheduling, which provided a performance improvement.

Next, we looked at how to expand the number of loops that run on the accelerator since each timestep also calls `compute_deltat()`. Two more loop structures were accelerated and a more encompassing data region was utilized for `uold` in order to minimize data transfers of this array. One required code change was related to two reduction variables in `courantOnXY()`. The two variables were actually pointers to double data types, and this caused issues for the compilers, so local reduction variables were created and then the results were copied to each pointer's data. This was a minimal code modification. After these optimizations, the MPI+CUDA implementation was still running about 2.3x faster than the MPI+OpenACC implementation; this was due to the large data transfers that were occurring since the `uold` array needed to be copied multiple times within each timestep. However, the OpenACC accelerator regions themselves were running at competitive performance to the corresponding CUDA kernels.

At this point in the porting process, we realized that the only way to improve the data transfer behaviour was to also add the `make_boundary()` kernels and data transfers (called inside the `hydro_godunov` solver) to the accelerator code as well. Without this addition, the large data transfers of `uold` occurred multiple times at every call of `hydro_godunov`, which was highly inefficient. To do this, we began by adding accelerator regions to all of the major loop structures in `make_boundary`, as well as a nested data region for the local data. With some of the accelerator regions, we hit compiler errors related to trouble parallelizing loops, and we were therefore unable to finish porting the `make_boundary` section of code within the timeframe for this document. As a temporary workaround until the compiler errors are diagnosed, we used the OpenACC `host_data` directive to make a call to the pre-existing CUDA version of the `make_boundary` code. This feature tells the compiler to use the device address of the associated array when calling the CUDA `make_boundary` code. While the `make_boundary` code itself is not performance critical, this simple workaround provided the final step for keeping most of the data resident on the accelerator (minimizing data transfers) – since all intensive sections of code within the solver were now ported to the GPU. The Cray 8.1 compiler currently supports the `host_data` directive, while the PGI compiler is expected to support this feature in their upcoming 12.5 version.

The performance characteristics of the final MPI+OpenACC Hydro benchmark are shown in Figure 10 and Figure 11. We chose to compare performance to the existing MPI+CUDA version because we were interested in investigating whether the OpenACC compilers can produce well-optimized executables compared to hand-written CUDA. Additionally, multi-node runs were performed to verify that the hybrid parallelization approach was working efficiently. Tests were performed on a Cray XK6 system. It features 176 nodes, each one equipped with 16-core AMD Opteron CPU, 32 GB DDR3 memory and one NVIDIA Tesla X2090 GPU with 6 GB of GDDR5 memory. The Cray compiler v8.1 and CUDA 4.1 were used for these specific experiments.

Future work will focus on fixing the handful of `make_boundary`'s OpenACC accelerator regions, so that the whole application relies solely on OpenACC. We do not expect any major challenges here, only more time to diagnose the issues and possibly pass comments on to the compiler vendors. Additionally, we expect to gain access to the CAPS implementation of OpenACC and this compiler will be investigated further. Finally, much work needs to be completed in order to understand the specific performance-related strengths and weaknesses of each of the compilers.

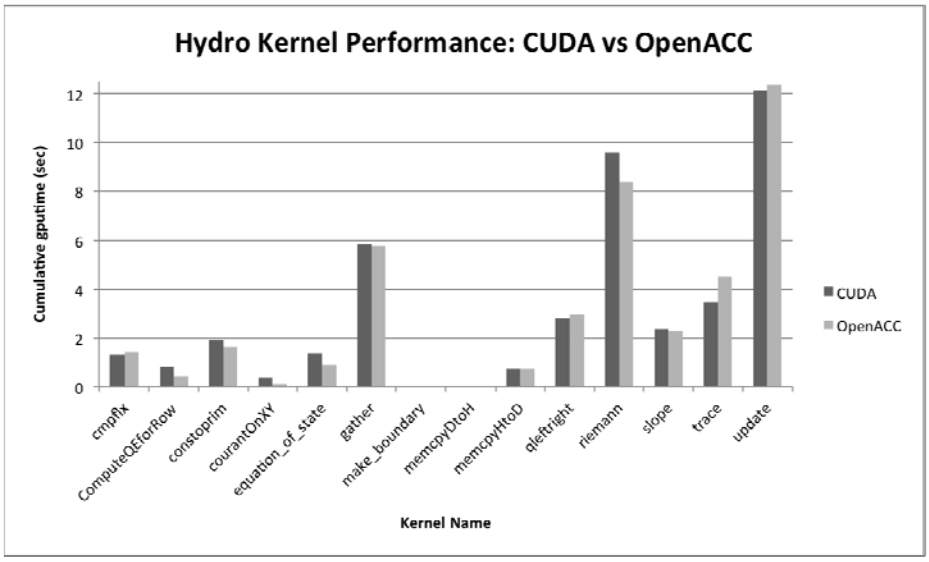


Figure 10: Competitive kernel performance. The chart above shows Hydro's kernel performance compared between the hand-written CUDA and directive-based OpenACC implementations. For all of Hydro's intensive code regions, OpenACC kernel performance is very competitive (sometimes faster, sometimes slower) to the associated hand-written CUDA kernels. As previously mentioned, note that `make_boundary` uses the same CUDA implementation for both versions; however, this function does not include any performance-critical loops. Comparisons were made using CUDA 4.1 and a Cray CCE 8.1 pre-release compiler, and the runs used a single Cray XK6 node (X2090 GPU) with the following Hydro parameters: `nx=ny=7500, nxystep=1500`.

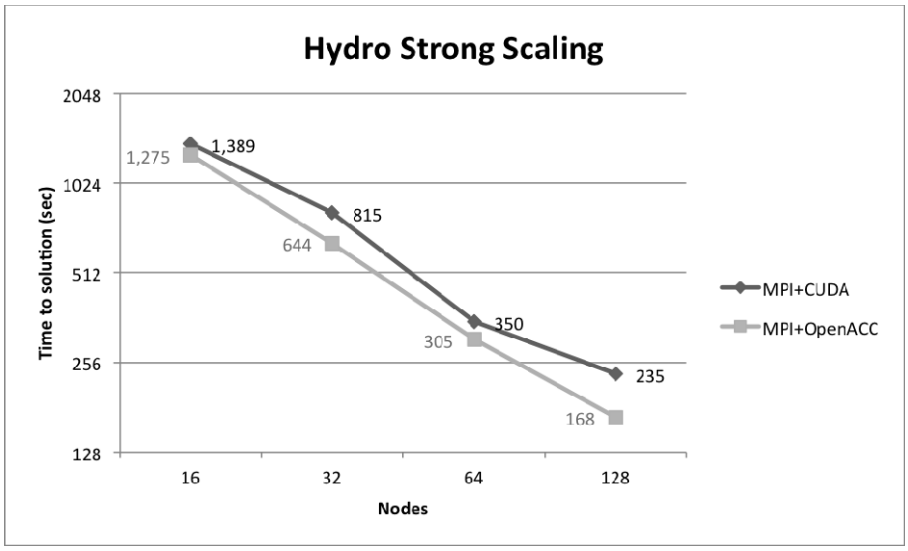


Figure 11: Competitive performance at scale. This chart gives the results of strong scaling experiments between the MPI+CUDA and MPI+OpenACC implementations of Hydro. As previously mentioned, the MPI+OpenACC implementation uses the CUDA `make_boundary` implementation via the OpenACC `host_data` directive as a temporary workaround; however, the vast majority of the GPU application runs with OpenACC. Each time-to-solution point was obtained from the average of five independent runs of 500 timesteps, `nx=ny=30000` and `nxystep=1500`. Like before, comparisons were made using CUDA 4.1 and a Cray CCE 8.1 pre-release compiler, and parallel runs were performed with a single MPI process per XK6 GPU node.

2.3.4 *Pros & Cons*

In Table 3, we list the pros and cons of using OpenACC as we experienced them during the porting of the Hydro code.

	Pros	Cons
Scalability	(Not directly applicable; OpenACC works with distributed memory approaches)	(Not directly applicable)
Performance	The reported Hydro kernel performance shows that OpenACC compilers are capable of competing with hand-written CUDA. The OpenACC standard allows for a significant amount of tuning via cache hints, loop-level parallelization hints and data movement control. However, if the kernel performance is lacking for a given application, the OpenACC standard allows for simple integration of external accelerator numerical libraries and/or low-level implementations (e.g. highly-tuned CUDA kernels). Other features such as 'async' allow for greater simultaneous work of the host and device (i.e. the host can do work or communication while waiting for the device to finish intensive calculations).	Similar to all accelerator-targeting programming languages, the ability to achieve accelerator data locality is crucial. The OpenACC standard provides mechanisms to accomplish this; however, the beta status of the compilers sometimes makes this challenging to achieve due to missing support for complicated data structures, unsupported portions of the standard, etc. Also, the current OpenACC standard does not directly address the opportunity for host-device work sharing for a given parallel loop. Finally, other lower-level approaches (e.g. CUDA or OpenCL) give the programmer greater flexibility when hand-tuning.
Productivity	The directives allow for incremental development of a given code, which can be very beneficial for productivity (e.g. starting with a scalar code, then porting it to OpenMP, and then OpenACC). This style of development focuses on exposing parallelism to the compiler, so it can also be beneficial for the same code base in terms of improving x86 vector performance.	The beta status of the compilers causes some productivity slow downs – each compiler has varying levels of support for each of the features. Some issues we have previously experienced include: compiler bugs that caused excessive data movement, lack of support for complicated data structures, challenges parallelizing certain types of loops, etc.
Sustainability	Concepts from OpenACC seem likely to be pushed into an upcoming OpenMP standard, which might promote further compiler and architecture support. Many specific OpenACC features seem applicable to future node architectures, for example: fine control over loop-level parallelism and hints for placement of data at various levels of the memory	With many upcoming architectural examples like fused CPU-GPU and heterogeneous x86 cores with automatic OS scheduling, it is not completely clear if a host-device style programming model will be required in the future.

	Pros	Cons
	hierarchy.	
Correctness	It is possible to incrementally develop for the accelerator using the same code base (i.e. starting with scalar code, then porting to OpenMP, then porting to OpenACC). As opposed to lower-level accelerator programming (e.g. with CUDA or OpenCL) where the code base must be changed, this incremental development approach improves correctness because results can be easily debugged at each development step.	Data movement can sometimes be a challenge for correctness; for example, a common pitfall is to mistakenly assume that values computed on the device have been transferred to the host, or vice versa. Additionally, current debuggers only provide beta support for OpenACC, so complex codes can sometimes be challenging to get working with debuggers. Also, not all compilers are generating sufficient debug information, and there are other general issues regarding accelerator kernel characteristics changing when the debug flag is enabled.
Portability	Currently, there are three different compilers that accept OpenACC directives. PGI and CAPS are available for many flavours of accelerator-based systems. Codes that are implemented with OpenACC can be run on CPU-only or heterogeneous accelerator systems by telling the compiler whether to accept the OpenACC directives or not. This type of single source code portability is one of the strongest benefits of OpenACC.	Currently only NVIDIA GPUs are supported as an accelerator target, although there is no reason why other accelerator targets cannot be supported. Cray's compiler is not portable to other non-Cray systems; however, the OpenACC code itself is portable. In terms of language portability, it is unclear whether OpenACC makes sense to use with object-oriented approaches (e.g. C++) – this is an area for further investigation.
Availability	PGI, Cray and CAPS all recently had early releases of compilers that support all or a portion of the standard. Having many available compilers directly equates to a better user experience – the user can swap compilers if functionality or performance is lacking for a given compiler on a given application.	Beta status of the compilers means that progress can be hindered by availability of upcoming releases that contain specific bug fixes. Additionally, there are currently no widely available open-source OpenACC compilers (probably because they require significant auto-vectorization capability)

Table 3: Advantages and disadvantages of using OpenACC, based on our experiences within this work.

2.3.5 *Recommendations*

Through the work detailed above we have arrived at a set of recommendations concerning the usefulness of OpenACC:

1. Encourage accelerator directive integration into the larger, more widespread OpenMP standard
2. Motivate tool developers (e.g. debuggers and performance analysis tools) to support the various OpenACC implementations
3. Create more application benchmarks and micro-benchmarks to investigate performance characteristics among the various OpenACC compilers
4. Create feature benchmarks to test newly implemented OpenACC features, such as: asynchronous data transfers, asynchronous kernel launches, and cache placement hints
5. Investigate whether these directives are useable with object-oriented languages (e.g. using the `declare` directive).

2.4 OmpSs

2.4.1 *Description*

OmpSs is based on the OpenMP programming model with some modifications to its execution and memory model in order to support asynchronous parallelism and heterogeneous devices such as GPUs. It is open source and available for download at: https://pm.bsc.es/ompss.

In particular, the OmpSs execution model is a thread-pool model instead of the traditional OpenMP fork-join model. The master thread starts the execution and other threads cooperate executing the work it creates (whether it is from worksharing or task constructs). Therefore, there is no need for a parallel region. Nesting of constructs allows other threads to become work generators as well. On the other hand, the OmpSs memory model assumes a non-homogeneous disjoint memory address space. As such, shared data may reside in memory locations that are not directly accessible from some of the computational resources. Therefore, all parallel code can only safely access private data, while for shared data it must specify how this is going to be used (see below). This assumption is true even for SMP machines as the implementation may reallocate shared data taking into account memory effects (e.g., NUMA). In order to support OmpSs, programs must be modified to reflect this runtime model. This process is referred to here as "taskification".

Furthermore, OmpSs allows annotating task constructs with three additional clauses: `Input`, `Output`, `Inout` and `Target`. `Input` specifies that the construct depends on some input data, and therefore, is not eligible for execution until any previous construct with an output clause over the same data is completed. `Output` specifies that the construct will generate some output data, and therefore, is not eligible for execution until any previous construct with an input or output clause over the same data is completed. *`Inout`* specifies a combination of input and output over the same data.

Finally, to support heterogeneity and data motion between address spaces a new construct is introduced: the `target` construct. It allows one to specify on which devices the construct should be targeting (e.g.,cell, gpu, smp, etc.) and also specifies that a set of shared data may need to be transferred to the device before the associated code will be executed. In addition, there is a specific construct called `implements` which specifies that the code is an alternate implementation of the target devices of the function name in this clause. This alternate clause can be used instead of the original one if the implementation considers it appropriate.

2.4.2 *Porting a scientific application to OmpSs*

The application "HYDRO" was chosen to evaluate the OmpSs programming language. HYDRO is a simplified version of the astrophysical code RAMSES, in that it lacks Adaptive

Mesh Refinement (AMR). It is effectively a 2D Computational Fluid Dynamics code of reasonable size (~1500 lines for the sequential F90 version). The space domain is rectangular two-dimensional, split into a regular Cartesian mesh. The code solves compressible Euler equations of hydrodynamics based on finite volume numerical methods, using a second order Godunov scheme. Also, a Riemann solver computes numerical flux at the interface of two neighbouring computational cells on a regular 2D mesh. These functions are in the `hydro_godunov.c` and are the best candidates in HYDRO to parallelize using OpenMP and OmpSs.

Initially, we took a C version of HYDRO in the PRACE_HYDRO_V1 package to port to OmpSs. In this version, we took a similar approach as in OpenMP, namely taskificating in a fine-grain approach the iteration loops. These loops proved to be too-fine grain in the end, however, leading to too much overhead.

A more recent version of HYDRO takes a more coarse-grain approach, something which facilitates better the porting to CUDA. The reason for this is because it is structured in a much cleaner way, where data structures are more visible and it is also easier to identify the computation blocks in order to facilitate the porting to CUDA. The name of this last version is HydroReference20120210. The results presented in this document are from this last version. In this version, we parallelized, using OmpSs, the main iteration of the Godunov scheme which is implemented in the routing `hydro_goduno()`.

```
 for (idimIndex = 0; idimIndex < 2; idimIndex++) {
    make_boundary();
    Allocate local variables
    for (j = Hmin; j < Hmax; j += Hstep) {
#pragma omp task concurrent (*uold)
{
      Allocate local work space for 1D sweeps
      gatherConservativeVars();
      constoprim();
      equation_of_state();
      slope ();
      trace();
      qleftright();
      riemann();
      cmpflx();
      updateConservativeVars();
      Deallocate local work space
}
    }
#pragma omp taskwait
}
```

Figure 12: Taskification with OmpSs of the `hydro_goduno()` routine.

Figure 12 shows the taskification with OmpSs of the main routing `hydro_goduno()`. The taskification consists on adding OmpSs pragmas inside the loop iterated by `j`, in order to taskify for each loop iteration in each of the two dimensions. A synchronization OmpSs pragma was added at the end to wait for the finalization of the previous launched tasks in the loop iterated by `j`. These changes are emphasized in bold in Figure 12. Also, we allocate and

deallocate the required local work space for each task every time that the task is created. As can be seen, the taskification granularity is selected to be a slice for each X and Y dimension. In each slice a gather operation is performed to copy the required data in a temporal buffer, then all the computation operations are performed in this buffer, and at the end, the results are placed in the original locations in memory.

A finer-grained taskification is tested which consists on taskificating every computational function in a slice. However, due to the data dependencies among these functions this did not achieve any parallelization of the computation. Figure 13 shows the functions and data structures used in these functions, and the data dependencies among them. As you can see, there are data dependencies among the functions that prevent parallelizing the computations in a slice.

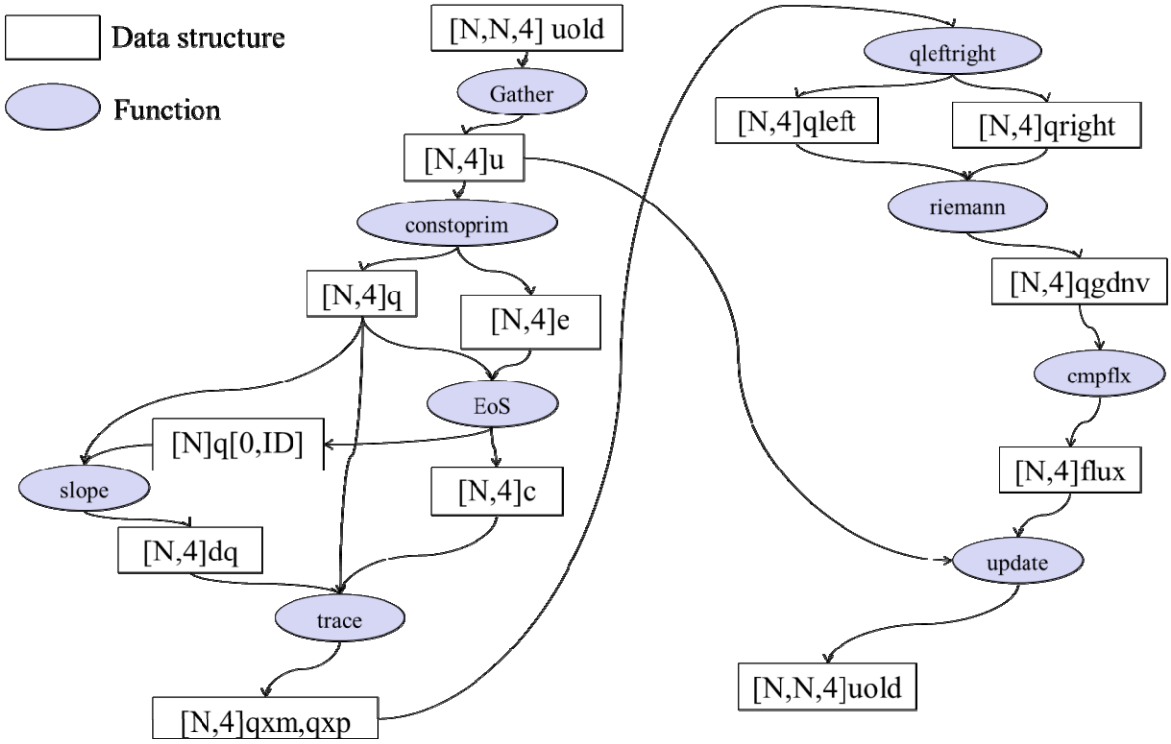


Figure 13: Functions and data dependencies in a slice of the `hydro_godunov` iteration

2.4.3 *Experimental results*

The selected application benchmark HYDRO is run with different configurations of numbers of nodes to evaluate the scalability of the code. The number of MPI processes used is 32, 64, 128, 256, 512, and 360 using two threads per MPI process.

The machine used is a cluster that comprises of 126 compute nodes. Every node has two Intel Xeon E5649 6-Core processors at 2,53 GHz running a Linux operating system with 24 GB of RAM, 12MB of cache memory and 250 GB local disk storage. They are also equipped with 2 NVIDIA M2090 cards each, with 512 CUDA Cores and 6GB of GDDR5 Memory. The interconnect network is non-blocking and is based on QDR InfiniBand with a bandwidth of 40Gbps.

The input deck used is the typical one for scalability studies called `input_sedov_10000x10000.nml` where the problem size is `nx=10000` and `ny=10000` and runs 100 iterations.

Figure 14 shows the runtime of the MPI version of HYDRO with OmpSs as a function of the number of MPI processes. Two threads per MPI process are used in all cases and 6 MPI processes were placed per node. As can be seen, HYDRO scales quite well with the number of MPI processes. The execution time is 36 seconds at 35 MPI processes and decreases down to 7 seconds on 320 MPI processes.

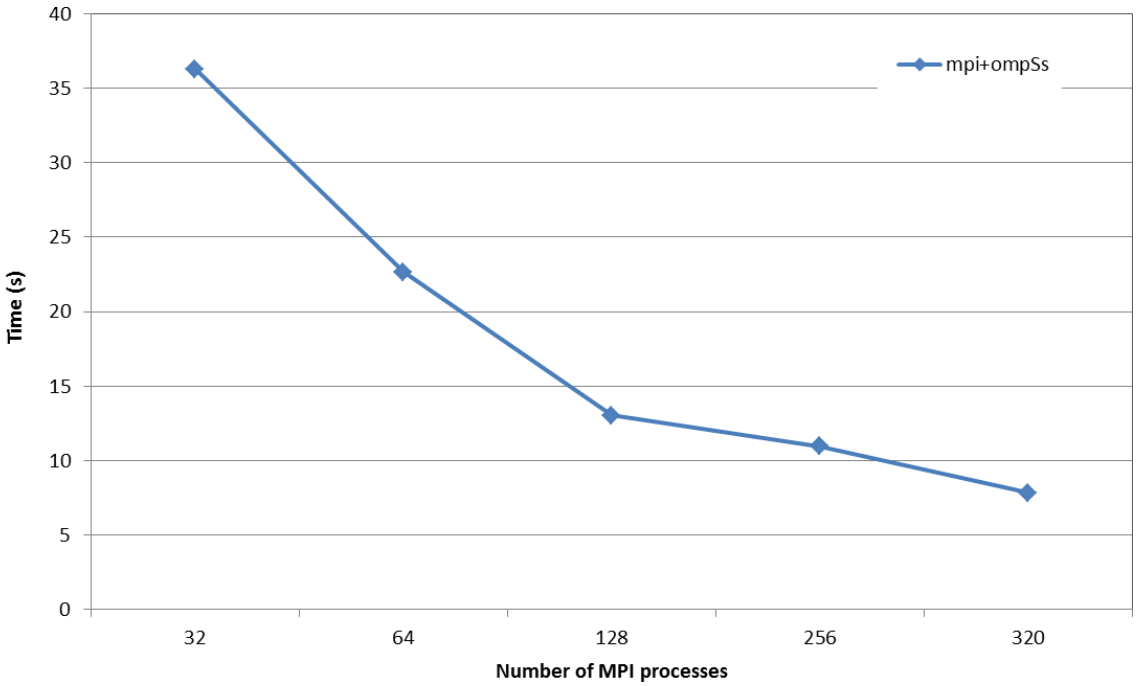


Figure 14: Runtime of HYDRO when scaling the number of MPI processes

2.4.4 *Pros and Cons*

Table 4 summarizes the pros and cons of the OmpSs programming language for various areas including scalability, performance, sustainability, portability, availability, and productivity. The major advantage of OmpSs is that its main objective is to extend the well-known standard OpenMP with new directives to support asynchronous parallelism and heterogeneous devices. This allows to achieve a higher communication-computation overlap so codes are able to scale better at large scale where communications are inevitably becoming the major bottleneck. Furthermore, OmpSs is open source and can be freely download.

Area	Pros	Cons
Scalability	Higher communication-computation overlap is achieved in order to hide the communication overheads occurring at large scale. It supports both shared and distributed memory systems.	Manual taskification of the communications is required, independently from the computations.
Performance	Allows for asynchronous parallelism. It also supports heterogeneous devices.	The overhead of task spawning and bookkeeping could degrade the performance if the computation granularity of the tasks is very small. Performance is limited to the capabilities of user in creating data dependency and extracting

Area	Pros	Cons
		parallelism
Sustainability	BSC is providing support	It is not yet a standard
Portability	Runs on standard x86 and x64 processors. The source code of the application remains the same.	Requires the Mercurium compiler and the Nanos++ runtime.
Availability	Open source developed actively by the BSC.	
Productivity	Very short, readable code. Easy to program and maintain. Clear and powerful concepts for parallel programming extended from the standard OpenMP.	Practically no tools are supported for assisting the porting and development of applications. Additionally, development time is affected to some extent due to unavailability of debuggers.
Correctness		Caution is required to make sure that data dependencies are correct.

Table 4: Pros and Cons of the OmpSs programming language.

2.4.5 *Recommendations*

When porting HYDRO to OmpSs we identified several recommendations for programmers in order to write a much cleaner and portable code when programming with a task-based parallel programming language. These are simplifying data indexing, pushing data allocation close to where it is needed, customizing blocking size, and using a top-down approach for taskification. These are briefly described below:

1. **Simplify data indexing:** The data structures such as arrays and matrices used in the computations are declared in a way that their size is explicit, without requiring to perform any index arithmetic. This enormously facilitates the programmer to identify data dependences among the computations and consequently propose an efficient taskification of the application's code.
2. **Pushing data allocation close to where it is needed:** We have found that the common practice today is to allocate the data required in tasks globally, i.e. outside of the task declaration. The advantage of doing this is that it minimizes the overhead of allocating data, because it is performed only once regardless of the number of tasks. However, this approach might be prone to generate data dependencies among tasks due to sharing the same data structure. These data dependencies are unfortunately sequentializing tasks and thus prevent the parallezation of the code. This problem can be easily solved by allocating the particular data required for each task close to where the task needs it. By doing so, data dependencies among tasks are minimized. The only drawback of doing this is that it may generate a higher overhead due to the allocating/de-allocating of data for every task.
3. **Customizing blocking size:** When taskifying code it is desirable to have the flexibility to adjust the computation granularity of the tasks. This can be achieved by customizing the block size when partitioning the space domain of the scientific applications. Finding an optimal computation granularity is critical to performance in order to balance the overhead of spawning multiple tasks with the performance improvement achieved by having a higher parallelism.

Identifying the optimal granularity is non-trivial, because it depends on many parameters such as the underlying machine. Thus an auto-tune tool may be interesting to use during the runtime of the application, to find the optimal block size.

4. **Top-down approach for taskification:** One interesting approach when taskifying an application is that the taskification process should be able to follow a top-down approach. With this approach programmers are able to go from a coarse-grained to finer-grained computations in order to have the flexibility to control the task granularity more efficiently. A coarse-grained approach would involve taskifying an application's routines, and a finer-grained approach might be based on taskifying loops. For the latter, it might be interesting to structure the computation in nested loops.

2.5 UPC

2.5.1 *Description*

In this Section we briefly introduce the concepts of UPC, its main features and constructs, and the most prominent compilers and tools currently available.

A very brief introduction to UPC

UPC is a Partitioned Global Address Space (PGAS) language. It is an extension of ANSI C that provides access to a shared, partitioned address space (any valid C program is also a valid UPC program). The variables stored in this shared address space can be directly used by any thread/processor, however each variable is associated with (or has affinity to) a specific, single thread. This allows programs to be more easily written with a view of the entire memory available to all processes, and at the same time to exploit data locality (affinity) for boosting performance. UPC programs follow the "Single Program / Multiple Data" execution model, and instantiate a number of threads that operate on either private data, or the shared data within the global address space. The number of threads can be declared during compilation or at the moment of running. Qualifier keywords are used to declare whether data is private or shared, as well as how arrays could be distributed among threads. Being derived from C, UPC supports pointers. UPC pointers can be declared as either shared or private, and they can point to either shared or private memory. Because UPC pointers have to keep track of more than simply the memory location they point to, their representation has considerable impact on code performance. UPC contains its own language primitives to provide dynamic allocation of shared variables, data movement between local and remote memories, collective movements between threads, etc. Parallelism is most often implemented by looping over all the data items, and having each thread or process only operate on the data that has affinity to them. UPC adds a special type of loop to accomplish this task.

Main features and constructs

We briefly review some of the key distinguishing features and language constructs of UPC:

1. **Identifying threads:** The number of threads available to the program is maintained by the `THREADS` variable, whereas each running thread is identified by the variable `MYTHREAD`.
2. **Synchronisation:** Synchronisation between different threads is achieved through a set of functions such as barriers, locks.

   The UPC construct `upc_forall` is an iteration statement similar to the C for loop, with one additional argument that determines which thread executes a given iteration, or in

other words to which thread this iteration has affinity. This argument can be either an integer (in which case the affinity matches the THREAD numbers), or a pointer to shared memory (in which case the affinity is that of the object pointed to). Finally some functions in UPC are collective, i.e. they are executed by one thread, but they affect all of the threads.

3. **Allocation and sharing of data:** In UPC, dynamic allocation of shared memory is achieved through the collective functions `upc_global_alloc`, `upc_all_alloc` (the latter is a collective version of the former), and `upc_alloc`, which allocates shared memory with affinity to the calling thread. The memory is allocated according to the declaration:

```
shared [blocksize] char[nblocks * blocksize]
```

The above declaration dictates the affinity of each block of the allocated array to a specific thread. Specifically, the first `blocksize` bytes have affinity to thread 0, the next `blocksize` bytes to thread 1, etc. `Upc_free` is used for freeing such allocated memory. Data is transferred between shared/non-shared memory with the functions `memget` and `memput`.

Main compilers and implementations

UPC compilers are compliant to a UPC specification that is not part of the ANSI C standard.

Prominent compiler implementations include the following[1]:

- HP UPC (commercial) [http://h30097.www3.hp.com/upc/]
- Cray UPC (commercial) [http://www.cray.com/Home.aspx]
- GCC UPC (free, developed by Intrepid) [http://www.gccupc.org]
- Berkeley UPC (free) [http://upc.lbl.gov/]
- Michigan Tech MuPC (free) [http://www.upc.mtu.edu/]
- IBM UPC Alpha Edition (commercial).

2.5.2 *Description of work done*

We now describe the main focus of our investigations and experiments on UPC.

Main topics of investigation

Our goal was to evaluate the feasibility and practicality of porting non-trivial code to UPC, to be executed over a large number of nodes in a HPC system. In particular, we were interested in the following aspects of the language:

- Ease of implementation and porting to UPC
- Adaptation to shared memory model, and issues with affinity
- Identification of issues relevant to synchronization between threads
- Performance, optimization and scalability

In our studies, it soon became apparent that there was a trade-off between the above aspects of the language, which warranted deeper investigation. More specifically:

- **Shared memory model allows simpler implementation:** The shared global address space model allows programs to be ported to UPC with potentially limited effort, since

[1] see also http://upc.wikinet.org/wiki/Compiler_Software

the developer can consider the entire shared memory to be available to all threads and processes.

- **Lack of affinity causes performance degradation:** On the other hand, we find that performance degradation is bound to occur if data locality and affinity issues are not taken into account in the implementation.
- **Enforcing affinity adds code complexity:** However in order to adapt one's implementation so that it will take into account data locality and affinity, more complex and intricate code needs to be written, to an extent undoing some of the benefits of the shared memory's model.
- **Code complexity makes synchronization more difficult:** Finding the optimal amount of thread synchronization required to ensure correct execution may not be trivial. The increased code complexity may lead to excessive use of synchronization constructs (barriers, locks, etc), possibly resulting in performance degradation or even race conditions.

In our experiment we came across all of the above issues, and we discuss them in more detail below.

Ported code and description of experimental setup

We initially focused on porting the Graph500 [19] benchmark code to UPC. We found the code's complexity to be prohibitive, and the documentation somewhat lacking, so after some issues with library functions that we had difficulty in porting we decided to shift our focus to the Hydro code, which was suggested within the scope of the project.

The Hydro code was suggested by the CEA/IDRIS teams. We ported it to UPC, performed various cycles of optimization, and ran extensive experiments focusing on performance, scalability, shared memory allocation and synchronization. Our results are discussed in Section 2.5.3.

Finally we also developed our own synthetic benchmarks targeting performance, affinity and cross-node memory access issue. Following are the key details of our experiments, platforms and tools:

- Compilers used:
  - Berkeley UPC
  - gcc-upc.
- Machines used:
  - First experiments on HLRS Nehalem cluster.
  - More experiments on HLRS Cray XE6 Hermit.
- Experimented with different:
  - Synchronisation schemes (barriers, locks)
  - Memory/pointer sharing and accessing approaches
  - Input data sizes
  - Numbers of threads, cores, nodes

In the following section we present and discuss our findings.

2.5.3 *Experimental outcomes and discussion*

In our experiments we tried to address and study various issues that impacted our outcomes, some of which have also been identified in other similar studies. We present and discuss them together with our experimental results.

Ease of implementation, coding and productivity

Porting the code to UPC required some effort in getting acquainted with the language's additional syntax and constructs, as well as issues of thread handling, shared memory allocation, etc.

A first running version was achieved with reasonable effort. However several further stages of improvements and optimizations were carried out those tried to address issues of synchronization, performance and scalability, memory affinity, handling of shared pointers as well as others and are described below. In order to completely address all of the above issues a considerable study and re-design of the entire algorithm would be required.

We were forced to address the fact that, although in principle we could consider all of the shared memory as available to all threads, there was no (obvious) way to ensure that each thread would only access the memory that was within its affinity, for maximum performance.

The code readability inevitably suffered as provisions were included for the above issues, resulting in a more complex software artefact.

According to our developers' diaries, the initial port took roughly 4 weeks, followed by various cycles of new optimised versions weekly.

Synchronisation issues

Synchronisation between threads in UPC is handled through locks and barriers. In a first implementation it is easy to "over-engineer" the problem, inserting more locks and barriers than is absolutely necessary, to ensure that the code will be executed in the required fashion. It was immediately apparent that this caused performance delays, and even risked to create race conditions.

Reaching the optimal amount of synchronization can be tricky, especially as the code becomes more complex due to other requirements.

We managed to improve the performance of a correctly running version of our code by about one order of magnitude by optimizing synchronization.

Performance and scalability

We ran experiments based on the Hydro code, as well as on synthetic benchmark code we created, which could be tuned to perform either completely in-affinity memory accesses, or mostly out-of-affinity memory accesses. The synthetic benchmark was used to study both issues of performance and memory affinity.

Figure 16 and Figure 17 refer to the synthetic benchmarks runs. The execution time scales linearly with the problem size and number of available cores for a given number of nodes. However we observe some clear performance issues which we attributed to shared memory affinity. In short our findings suggest that:

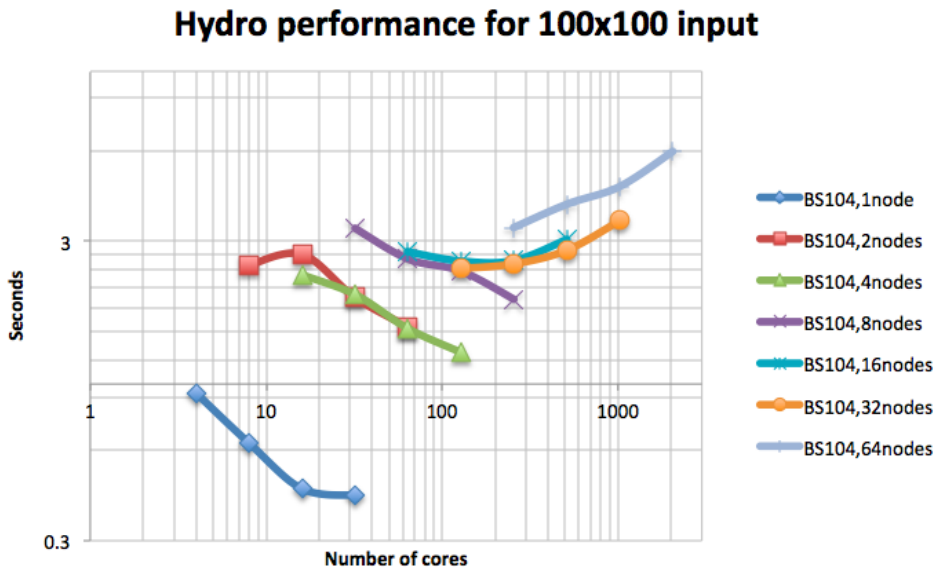
- Performance scales as expected across number of threads.
- A significant degradation occurs in memory accesses outside threads' affinity, especially when accessing memory across nodes.

Figure 15 refers to our runs with the ported Hydro code, which includes a considerable amount of out-of-affinity memory accesses.

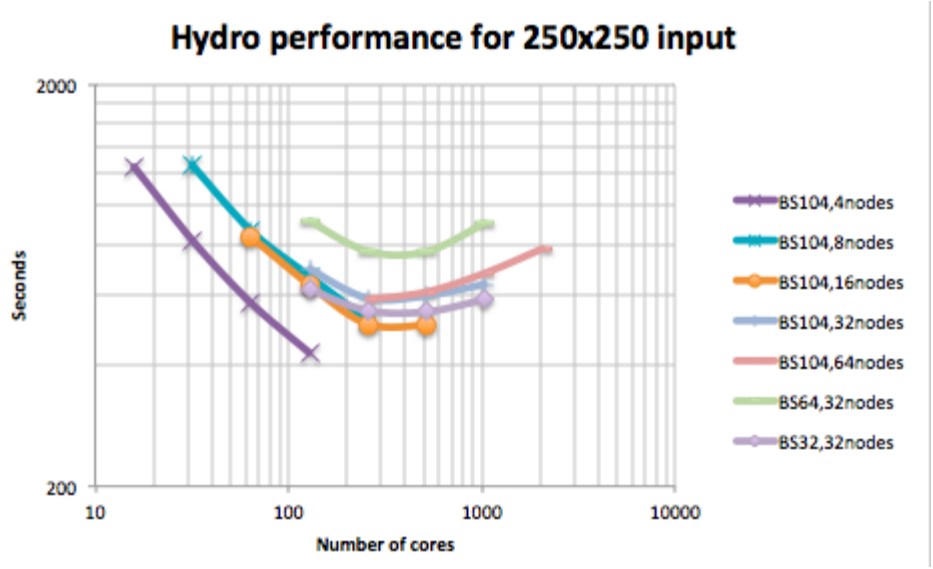
Overall, we have ran our experiments on configurations of up to 128 nodes and almost 4096 cores, we have exhausted the available input data sizes, and we have observed and described the performance and scalability patterns of this application.

We show the results of running three different input data sizes, namely 100x100, 250x250 and 1000x1000. We observe a similar performance pattern up to a certain number of cores (different for each input data size and `BLOCKSIZE` used), and then a turning point as performance begins to deteriorate. We believe that two effects come into play:

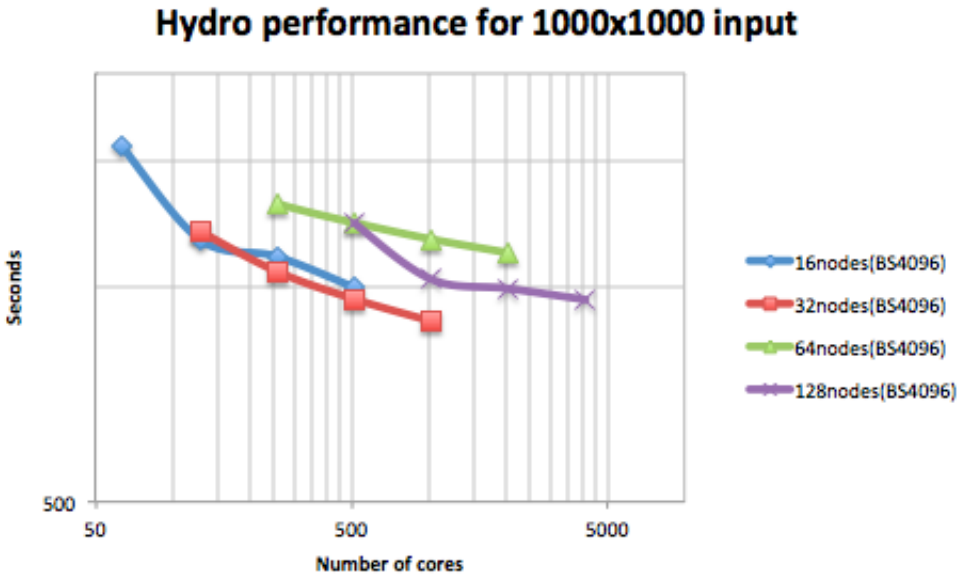
- As the number of cores increases, at a certain point the computation/communication ration becomes so small for each thread that the communication overhead dominates the thread time, and the computational gain is lost. That's where we observe the turning point in the curves. We identify these at roughly 200 cores for the 100x100 input, at 500 cores for the 250x250 input, and probably over 5000 cores for the 1000x1000 input. This suggests a limit for the number of cores that it is reasonable to deploy for a problem of such size.
- Another factor that we expected would affect our results is the fact that for a specific input size and `BLOCKSIZE`, there's a limit to the number of threads that can be utilized to solve the problem. More threads should simply not be assigned any task. In some of our runs, e.g. in the 1000x1000 data size, we expected this to show up in our graphs. Specifically we expected that adding more than 1000 cores in this case would not improve overall performance. However we were surprised to see that it does. One theory is that as there are more than one independent `upc_forall` loops, perhaps the additional cores are more efficiently allocated so that different cores are used in the different loops (if available).



(a)



(b)



(c)

Figure 15: Performance of UPC Hydro code implementation, as the number of cores and nodes is increased, for (a) 100x100 input data size, (b) 250x250 and (c) 1000x1000. Each line corresponds to a different number of nodes and/or block size.

Shared memory locality and affinity

In order to investigate further how performance is affected by issues of memory locality and affinity, we devised a simpler, synthetic benchmark that however handled shared memory in the same way that our main program (the Hydro code) did. We ran experiments in which we would select to have memory accesses either only within affinity, or mostly out of affinity. We ran these across different numbers of cores and nodes. The results can be seen in Figure 16.

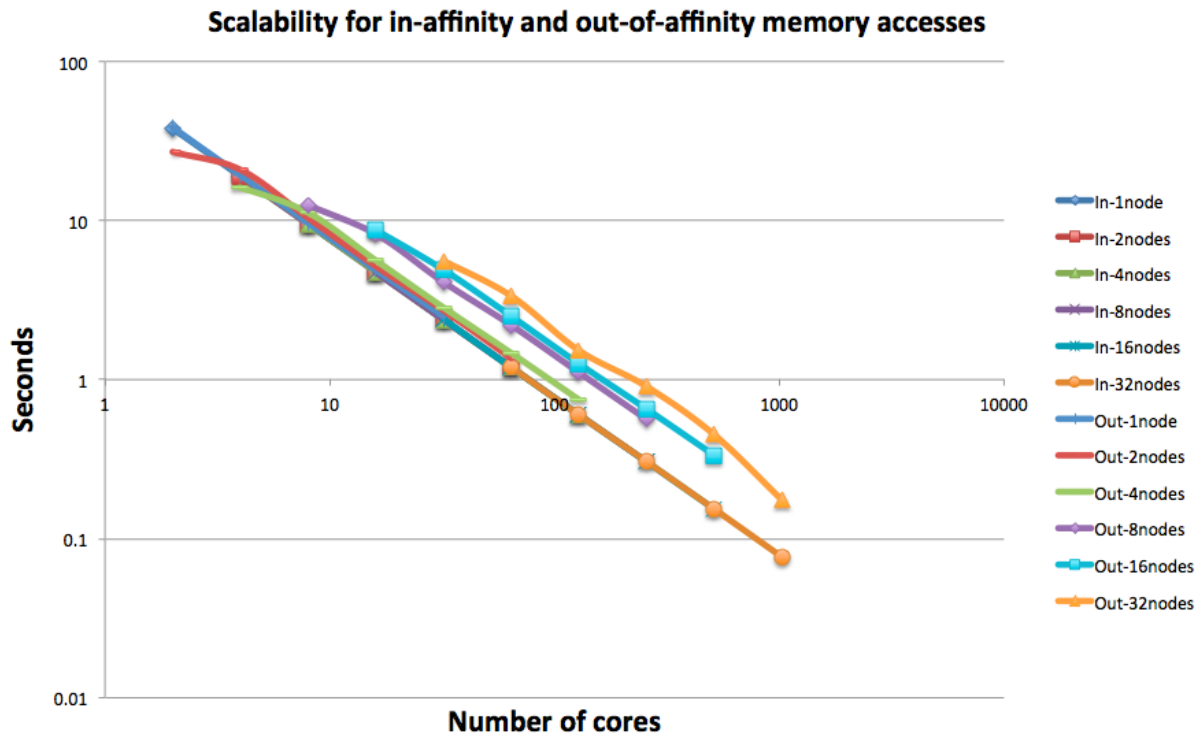


Figure 16: Scalability of code for in-affinity and out-of-affinity memory accesses

The figure (note that both axes are logarithmic) shows an almost perfect performance scalability as the number of cores/nodes increases. The lower-left group of curves are from the in-affinity runs. The upper-right group are froming the out-of-affinity runs. We see that there is a performance degradation there, which as expected increases as the number of nodes involved in the experiment also increases. Figure 17 shows how the number of nodes affects the performance degradation for out-of-affinity runs in the Hermit Cray XE6. We plot the ratio of average time to run with out-of-affinity accesses over average time to run with in-affinity accesses, against the number of nodes. Specifically we see that running on 32 nodes causes performance to degrade by roughly 300% (i.e. become 3 times slower).

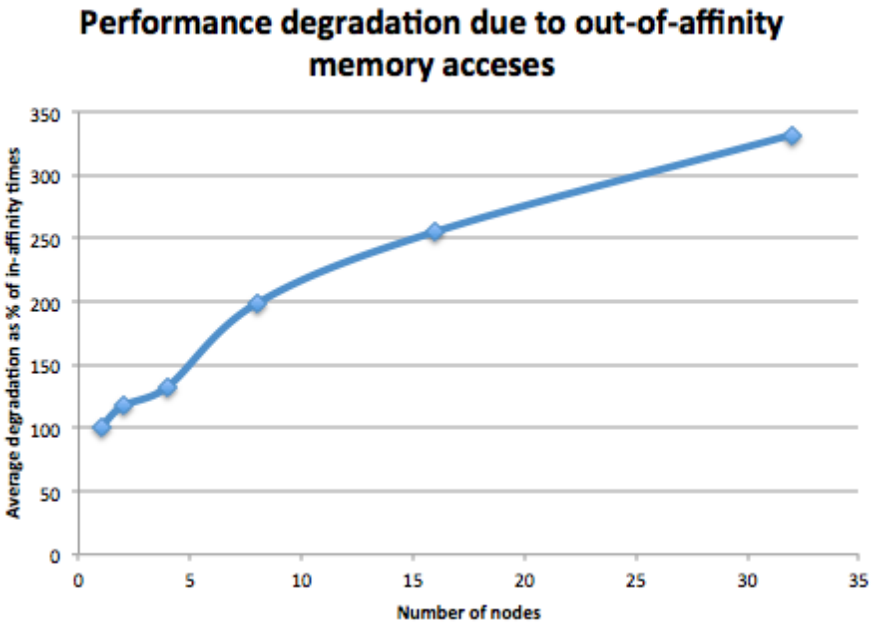


Figure 17: Performance degradation due to out-of-affinity memory accesses.

So to summarize, that for the given ration of in/out of affinity memory accesses scalability is almost perfect while increasing the number of threads and nodes (seen in Figure 16). This is an important characteristic of UPC, as it suggests that for a specific ratio of in/out affinity accesses and for specific sizes of problems, UPC can offer very good scalability. However in the case of the Hydro code performance suffers due to data localization/affinity issues, especially across nodes. As seen in Figure 15, the performance is not optimal, and as the number of nodes increases it is affected by out-of-affinity memory accesses as described above.

Unfortunately ensuring that the code will always perform memory accesses that are within affinity is not at all straightforward. An example of a data affinity issue we encountered with the Hydro code is briefly outlined in Figure 18, and discussed below.

This figure shows the main shared data structure used by the Hydro code (`uold`). `Uold` effectively consists of a 3D "cube" of data, (along axes `i,j,v`), but is stored as a 1D array. UPC only allows splitting the array in a series of contiguous blocks of fixed size, and giving affinity of each block to one thread in a periodic, round-robin fashion.

The algorithm will at some point want to assign blocks with a common value of `j` (such as the horizontal slice shown in the figure) to one thread, whereas at other points it will want to assign blocks with a common value of `i` to one thread. And on top of that, it will perform some additional boundary operations, working on neighbouring slices. However there is no way to arrange the data within the 1D array such that these blocks will always be contiguous. For example in the figure we see that the values corresponding to the horizontal slice are dispersed throughout the entire array. In order to achieve some (not complete) affinity, very specific conditions should apply, e.g.

```
(nx*ny) mod (blocksize*nthreads) = 0
```

But this is rarely the case, resulting in performance degradation when memory accesses out of the thread's affinity occur.

In contrast, a language such as MPI is not restricted by a linear, periodic, `BLOCKSIZE`-based array affinity model, like UPC. In MPI, the decomposition of a 2D or 3D problem (like the hydro problem) can be performed based on a set of tools, such as the `MPI_CART` set of functions. These create communicators based on Cartesian topologies, according to the programmer's specifications. A memory "grid" is thus created and stored (size, shape, map of processes to blocks, etc), and each process can then access this information by calling other `MPI_CART` functions. This makes it a lot more feasible to break down the problem into units that will be processed within each thread's affinity.

When we changed the code to avoid these out-of-affinity accesses (but then the resulting code didn't perform the correct operations), we saw that we were able to avoid the performance penalties.

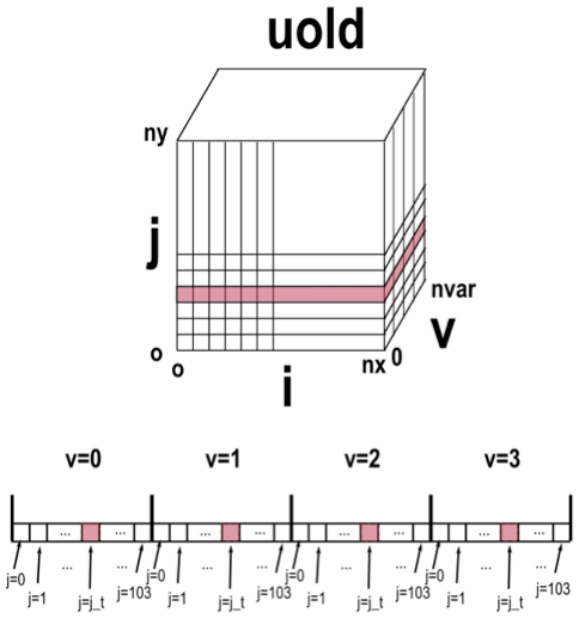


Figure 1.1.3

Figure 18: Schematic representation of the main shared data structure of the Hydro code

Shared pointer handling

We give an example of how incorrect handling of a shared pointer can cause dramatic performance degradation of at least 1-2 orders of magnitude. We allocate a shared global array as:

```
uold = (shared[BLOCKSIZE] double *)
       upc_global_alloc(numberofblocks, BLOCKSIZE*sizeof(double));
```

Then within each thread we access it, in two different ways:

- We directly access it with:

```
x = uold[i];
uold[j] = y;
```

- We define a local variable and set it to point to the global array:

```
shared[BLOCKSIZE] double *puold;
puold = Hv->uold;
```

and we access the shared memory as:

```
x = puold[i];
pould[j] = y;
```

The first version degraded performance by one to two orders of magnitude, due to the fact that the global shared pointer to the shared array belonged to one thread (thread 0), so all other threads would have to access thread 0 to read the pointer value at every access, whereas in the second version thread 0 only needed to be accessed once, to

copy its value to the local pointer. In Figure 19, the top group of curves show the effect on performance of the first version. The second version, however, may impose stricter synchronisation requirements.

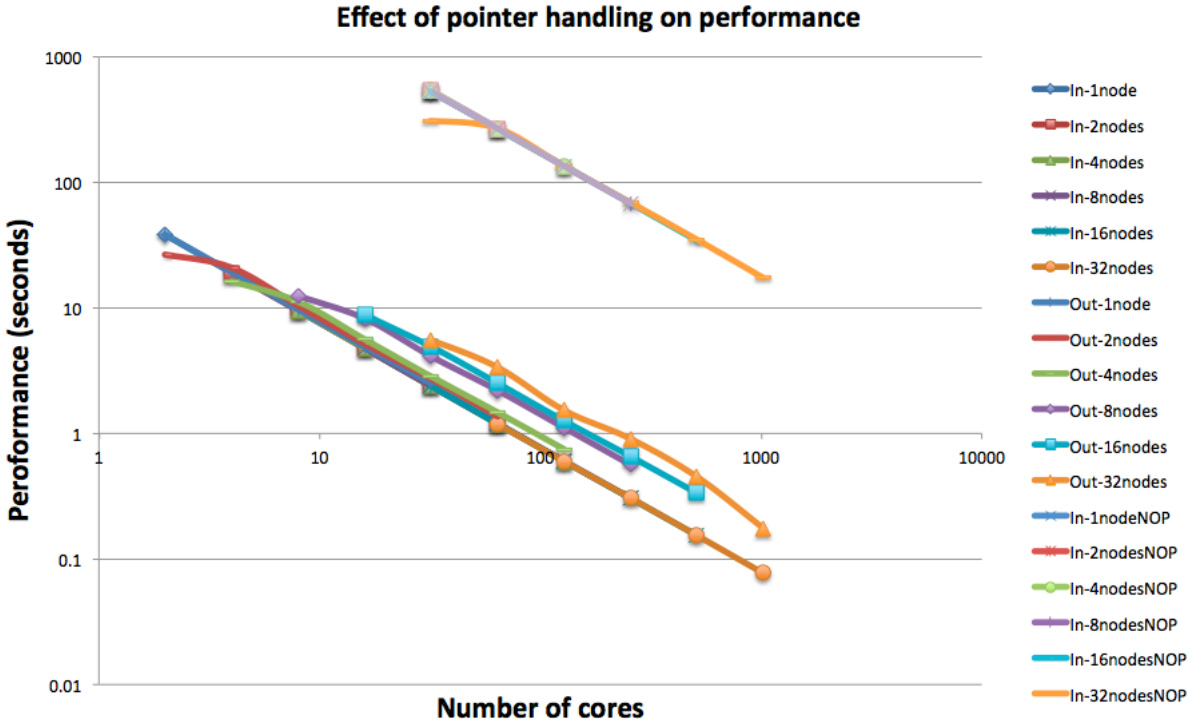


Figure 19: The effect of shared pointer handling on performance. The top set of lines shows the performance degradation when directly accessing the shared data structure, instead of using a local pointer for indirection.

2.5.4 *Pros and Cons*

We have rated UPC in seven aspects based on our experiences in this work. These ratings are shown in Table 5.

	Pros	Cons
Scalability	Good when thread has affinity to the memory it is accessing.	One may need to employ practices which require explicit knowledge of the memory model to obtain good scaling. In such cases, the benefit of using UPC over MPI is not obvious.
Performance	In the serial sections, UPC is identical to C, with the well known benefits in performance associated.	
Productivity	For some simple parallel applications, implementing an algorithm in UPC can be much faster than in MPI.	For complex data accesses, where threads need to access non-local data, the developer may require to explicitly take care of data accesses, leading to considerably more complex code.
Sustainability	The first version of UPC was released in 1999 and development of UPC	Although versions have been released continuously for almost 15

	Pros	Cons
	compilers is still ongoing.	years, UPC is still not considered a mature programming language for parallel architectures
Correctness	C debugging tools can be used for the serial versions of the code	
Portability	Porting from C to UPC can be trivial for kernels where threads operate on local data	For more complex kernels where threads operate on data local to other threads, one may require increased effort for porting as well as a detailed understanding of the application's details such as data paths.
Availability	A number of open-source and commercial implementations are available. The specification itself has reached a level of maturity to allow for a several conforming compilers.	

Table 5: Pros and cons table of UPC

2.5.5 *Conclusions and recommendations*

UPC is a language that, under certain conditions, can achieve excellent scalability and performance. The shared memory model allows ease of programming, provided that the algorithm to be implemented allows taking full advantage of its capabilities, without stumbling upon affinity and locality issues.

Being a relatively contained extension of the C programming language, it reduces the entry barrier for new programmers.

In terms of maturity, the tool support for UPC may not be sufficient yet, and it is also not clear how committed a community of users exists, however there are already various compiler implementations at relatively stable stages. Having said this, there are some cross-compiler portability issues that still need to be addressed.

The shared memory space model promises ease of implementation, however for top performance it must in essence be managed as if it were distributed. This may require considerable software engineering and programming skills, and can come at a cost to code readability. Perhaps a non-computer scientist porting their code to UPC would have some difficulties, compared to other models, e.g. OpenMP.

Regarding performance and scalability, our experimental results are not conclusive.

Additional issues with UPC include:

- `BLOCKSIZE` is required to be known at compile time. So a less flexible round-robin thread distribution approach is used, that has poorer performance.
- Array distributions in UPC are one-dimensional. A 2D-block distribution is not possible, as discussed in the previous examples.
- It lacks the programming tools that, for example, MPI or OpenMP offer, for distributed execution, communication and synchronization.

- Some performance variations were observed that we were not able to explain (they could be due to the runtime infrastructure).

In summary, UPC is a language definitely worth keeping an eye on, but some improvements would considerably increase its capabilities.

3 System software

3.1 Introduction

This section focuses on the various studies related to system software that were completed in the PRACE-1IP WP9 work package or are still ongoing on the associated prototypes. The chapter is separated into four sections corresponding to "Operating System and System Management", "Resources management", "Data management" and "MPI and Communication libraries".

For each subtopic, the motivation, description and results of the studies achieved by the WP9.2 task contributors are detailed. In order to provide technical insights for the deliverable, every prototype owner and task contributor was asked to provide their technical recommendations based on their own work. The associated sets of recommendations are presented for each subtopic. These recommendations provide the overview of the envisioned issues and solutions for upcoming system designs. They are outlined in four summary subsections, below each topic.

3.2 Operating System and System Management

3.2.1 *Energy Aware System Software (LRZ)*

Motivation

With energy efficiency being one of the major problems to overcome in the Exascale challenge, future HPC system software needs to become energy aware. Energy awareness at the system software level encompasses two aspects. First, power consumption of the system needs to be monitored in order to measure the energy required to run a given application (i.e. Energy-to-Solution). Second, the operational parameters influencing the power and performance characteristics of the system need to be tuned to improve Energy-to-Solution.

Description of the work

The CooLMUC prototype, at BADW-LRZ, uses smart power distribution units (PDUs) to monitor the power consumption of every compute node in the cluster. In addition, the power consumption of the cooling equipment is monitored using a digital three-phase current and voltage meter. An Energy-to-Solution system has been implemented on this prototype to first obtain the power readings of the components and second, calculate the Energy-to-Solution for a given application considering the power and estimated cooling consumptions of every allocated node.

Evaluations of the modification of the maximum frequency of the allocated cores, helped by a proper configuration of the SLURM resource manager, have been performed in order to assess the possible benefit of DVFS (Dynamic Voltage Frequency Scaling) for applications using the APEX MAP benchmark. This benchmark performs memory operations according to a selectable pattern. Thus, the study compares a randomly distributed access patterns to a stridden pattern, which is comparable to the STREAM benchmark.

Results

As expected, the study outlines that the power consumption is higher at higher processor frequencies. Since the benchmark workload is the same across the frequencies, higher processor frequencies also cause shorter application run times. Yet, in case of the random memory access pattern, the results show that the performance increase at higher frequencies cannot justify the increase in power, as opposed to the stridden case. Thus, Energy-to-Solution is best at the highest frequency in the stridden case while it is best at an intermediate frequency level for the random case. The study [20] confirms that adjusting system parameters can optimize the energy consumption of scientific applications. A more detailed description of the work carried out within this subsection is included in section 7.1 in the Annex of this document.

3.2.2 *Monitoring with Hierarchical Nagios (Hnagios) (CINECA)*

Motivation

Monitoring has always been an important part of the management system activities. With the increase of the number of server nodes of a computing cluster and the resulting raise of the probability of failures, monitoring is becoming even more important. Between the different monitoring solutions, Nagios is one of the most implemented open source solutions in the IT field. Some improvements are however required to make it a useful component for the proper management of High Performance systems.

Description of the work

The Nagios monitoring system [21] provides a central view of the system's status. Different dashboards provide at-a-glance access to monitoring information and views provide users with quick access to the information they find most useful. Alerts can be sent via email or mobile text messages, providing details, useful for starting the resolution process immediately. A basic configuration of a monitoring system with Nagios is composed of a central server which collects the results of active and/or passive checks on different hosts and related services. For each host of the system, a list of services to monitor can be defined. This works well for regular aggregation of hosts but for large systems with complex topologies, adapted views are required. Thus, for an efficient management of the entire system, a customized configuration and scripting of the Nagios installation is mandatory.

In order to manage the high number of alarms required to monitor all the nodes of the HPC clusters and their services, it was necessary to only use passive checks to distribute the execution of the tests on the monitoring targets and to spread the execution of the checks on the time frame in order to avoid overloading the Nagios server, thus improving its availability. Despite this design, the quantity of signals to handle on the server side is still very large and spotting critical situations and handling efficiently emergencies is very difficult. To overcome that issue, a hierarchical organization of the components only sending critical summaries with alarms and information organized in synthetic views is chosen. It allows system administrators to distinguish between critical and non-critical events.

For that purpose, Mathias Ketner's Nagios MK livestatus plug-in [22] is installed and configured to collect all the signals and organize and filter them before transmission to the server. It results in an architecture where each cluster has its standard Nagios server running on the cluster management node. This server is the peripheral Nagios instance that manages the nodes in sets of common types (compute, login, storage, etc.), summarizes cluster status,

and sends the result to a second level of the Nagios instance. This instance, the central Nagios, is the only component that sends alerts.

Results

These customized changes, combined together, allow one to distinguish at a glance between more and less critical status without loss of information. Indeed, the peripheral Nagios server can be used to find the details of a particular alarm. The consequence is an easier management of the priorities and organization of the repair service in a way suitable for the whole production of the computing system, modulating the time reaction for each event. All these improvements, designed and developed during the PRACE-1IP project, have already become an important part of the management of the High Performance Systems of CINECA. Such an architecture being very scalable and easily adaptable to different topologies or node hierarchies is a good candidate for a monitoring schema usable with Exascale dimension. More details concerning these improvements to HNagios can be found in Ref. [23].

3.2.3 *Technical recommendations*

A set of technical recommendations was compiled based on the results as presented in this section. These recommendations are shown in Table 6.

Task contributor/ Prototype	Lessons learned	Recommendations
LRZ - CoolMUC	The 1 minute read interval for some power sensors is not good enough for detailed power analysis of jobs. It is usable for Energy to Solution (EtS) measurements for jobs typically running longer than an hour.	Future systems might need a partition of nodes that are equipped with fine grain power measuring equipment for detailed power analysis.
LRZ – CoolMUC	Monitoring the energy consumption of all jobs running on the system (178 nodes) produces a lot of data over time even with a coarse grain solution of 1 minute (node power measurement, node load, CPU temperatures, system cooling power measurement, rack network equipment power measurement).	Future systems with >10000 of nodes need to have ways for defining the level of detail for power measurements required for each job. For example a simple 3 level approach can be envisioned. Level 1 would store power data at the resolution rate of the measuring equipment, level 2 would store data in specific intervals (e.g. every 5 min) and level 3 would just store the EtS for the job.
LRZ – CoolMUC	Although the Energy-to-Solution system implemented in the prototype through the resource manager Prologue and Epilogue scripts yields	Future Exascale-ready resource managers should include the ability to adjust the power related control knobs by default and further work

Task contributor/ Prototype	Lessons learned	Recommendations
	the desired settings to efficiently tune the power profile of an application, it requires manual work by both, the system administrator and the user.	should be done on automating the process of tuning these knobs for the best energy efficiency on the resource manager level.
LRZ – CoolMUC	The failure of the cluster management node resulted in a "zombie" system.	Single point of failures need to be removed from future systems. Every important part required for system management need to be redundant.
LRZ – CoolMUC	The separation from management, service and HPC interconnect network allowed for system management even though the HPC interconnect was experiencing some connectivity issues.	Future systems should provide different system networks for different duties and responsibilities
LRZ – CoolMUC	The redundant remote access to the cluster system management node via the service network and HPC network allowed for an easy upgrade of the HPC network stack.	Remote access to the system management node via all available system networks should be standard for future systems.
LRZ – CoolMUC	The separation from management, service and HPC interconnect network allowed for system management even though the HPC interconnect was experiencing some connectivity issues.	Future systems should provide different system networks for different duties and responsibilities
LRZ - CoolMUC	A pump failure in the infrastructure water loop running through the HPC system water heat exchanger was not detected because the temperature sensor on the infrastructure side before the heat exchanger showed correctly normal temperature but the HPC system water temperature increased till an emergency shutdown occurred.	It is important to not just monitor the system infrastructure but also to monitor the building infrastructure correctly. It is also important to think about all possible failure scenarios and to put appropriate sensors on the right places and integrate them with the system monitoring system. Cooling loops for instance need not just temperature sensors but flow sensors as well.

Task contributor/ Prototype	Lessons learned	Recommendations
CINECA	Nagios can be configured for monitor hybrid cluster easily	Configure Nagios for provide different dashboards in order to collect or provide different views for different goals. Configure Nagios in hierarchic way for clearer cluster views. Deploy Nagios in Central – Peripheral configuration for more flexible maintenance
CEA – Exascale I/O	Diskless boot of the embedded server modules is useful for system management. For example it eases embedded server module replacement.	We recommend diskless boot of embedded server, along with a resilient diskless boot service provider (e.g., high-availability of the administration nodes).

Table 6: Technical recommendations for operating systems and system management design. Where appropriate, the PRACE prototype utilized for the study is listed in the first column.

3.2.4 *Summary*

Software provisioning is one of the key aspects of system management. Diskless boot provides quick return to service in case of failure of hardware components. Indeed, it automates the process of associating the hardware with the particular software stack and configurations that make it work as expected. A fault-tolerant diskless provisioning system can thus greatly help to reduce the unavailability time of broken components. Network errors are common errors and the resiliency of a system can be enhanced using multiple networks shared by the nodes. A strong fault-tolerance of the services hosted by the management nodes over a robust multiple lanes network is mandatory. Every important part for system management needs to be redundant.

Efficient and scalable monitoring of both systems and facility equipment like power supply or cooling supply, is necessary to understand the failure scenario and its origins in order to take the right decisions to reduce the intervention time before returning to proper service of the systems.

Power management requires adapted sensors at every level of the systems and a correctly sized analysis back-end to handle the large amount of data to store and process. Big data tools and methodologies must be followed with great attention in this context.

Power information collection is the first step to improve power efficiency. Power efficiency requires additional primitives at both the operating system and the resource management levels to provide and leverage a fine grained control of the applications requirements. More work is required in state-of-the-art resource managers and operating systems to facilitate the

configuration of runtime and better adapt low level performances to application characteristics.

3.3 Resources management

3.3.1 *Integer programming based scheduler for heterogeneous systems in SLURM (Bogazici Univ.)*

Motivation

State-of-the art supercomputers are heterogeneous employing both CPU cores and GPUs. Yet, scheduling algorithms employed on these systems are based on the traditional CPU-only scheduling algorithms. Schedulers need to consider multiple criteria such as GPU awareness, interconnection topology and energy. Hence, the optimization problems to be solved by a scheduler become more complex and need more advanced general optimization tools. Integer programming (IP) techniques provide a general framework for solving complex optimization problems. Development of an IP based SLURM scheduler can be quite useful, since SLURM is used on at least 40% of the supercomputers in the Top 500 list.

Description of the work

A SLURM scheduler plug-in called IPSCHED employing CPLEX IP solver is developed. The scheduler takes windows of jobs and solves an assignment problem that matches jobs to CPU-GPU resources. The plug-in is available at http://code.google.com/p/slurm-ipsched/. The plug-in can also be used to implement custom schedulers by just changing the IP formulation. The performance of the plug-in has been tested by actual SLURM emulation that made it possible to realistically compare SLURM's original best fit scheduler with that of IPSCHED.

Results

Various tests have been carried out using the ESP synthetic workloads. Results show that parallel CPLEX could solve the resulting hard optimization problems involving thousands of variables in about 3 seconds, hence making it possible to use CPLEX in production environments. It is also observed that utilization of resources is increased when compared with the SLURM's own scheduler plug-in. Detailed discussion of implementation and results are provided in Refs. [24], [25] and in the PRACE white-paper in Ref. [26].

3.3.2 *Managing GPUs using PBSPro (CINECA)*

Motivation

The job schedulers play an important role in the optimization of the resources employment and exploitation. In a batch system configuration it is possible to choose among different sets of rules, either simple or complex, for scheduling jobs. CINECA tested on their Tier-1 hybrid CPU-GPGPU system a batch system that differs from the one installed on the prototype, in order to raise the possibility to find what will best fit with future Exascale systems.

Description of the work

PBS Professional is the professional version of the Portable Batch System (PBS), a workload management solution, originally developed to manage aerospace computing resources at NASA. The PBSPro version installed in CINECA (10.4) did not provide built-in support or integration with the vendor drivers and development and runtime libraries of the graphic card

(CUDA). The scheduler is capable of allocating GPUs as resources only with a proper custom configuration but does not bind jobs to a single specific GPU device. At the moment of the tests, PBSPro couldn't take advantage of the new features introduced in new versions of CUDA.

PBSWorks had provided in October 2010 a technical paper [27] where two different approaches to GPUs scheduling are proposed: a "Basic" one and an "Advanced" one. In the basic scheduling, GPUs are configured as single custom resources, while in the advanced configurations GPUs are PBSpro virtual nodes (vnodes).

As the Basic GPUs scheduling solution is more flexible, it better fits with the CINECA heterogeneous system load. Furthermore, it is easier to implement and configure for end users. This solution for GPU assignment allocates them as exclusive, enumerable and consumable resources. On the other hands the CUDA runtime libraries support GPU sharing across multiple threads, allowing, in principle, to have more mixed GPUs and CPUs jobs within the same node. This CUDA feature is not explicitly supported by PBS and, as a result, the exclusive allocation of GPU resources places some constraints on job placement on nodes. In order to avoid conflicts between different jobs and, at the same time, take advantage of the whole computational power of the system, CINECA is encouraging users to allocate full nodes and use both GPUs and CPUs in their applications.

Results

The PBSPro features for handling GPUs resources suffer from the recent usage of these cards in the HPC area. The main consequences are highlighted by missing features to share or split the resources of the cards. At the present time the GPU cards are allocated like an on/off switch. Altair already announced an Exascale road-map, so future improvements are expected.

3.3.3 *Integration of rCUDA with SLURM resource management (CSCS)*

Motivation

As the GPGPU accelerator based systems are increasingly targeted for scientific applications, the efficient use of these devices has become a critical consideration for HPC centers as well as application developers and end users. Remote CUDA (rCUDA) provides a virtualized access to GPU resources in a GPU cluster by eliminating the tight coupling of host CPU and GPU and allowing users to target as many or fewer resources per CPU required by an application [28][29]. At the same time, the HPC centers have deployment and operational flexibility to allowing them to serve diverse sets of workloads without compromising of wasted resources. We attempted to integrate rCUDA into a scalable resource management infrastructure called SLURM, which has been deployed to multiple, Petascale HPC sites. This integration would allow users to transparently request the required resources independent of underlying hardware features.

Description of the work

In a system employing rCUDA, CUDA applications seamlessly interact with virtual GPU devices representing those which are remotely accessible, offered by the rCUDA servers. However, the current scheduling scheme is not suitable for pools of resources decoupled from computing nodes. For this new approach to be integrated into SLURM, a different way of allocating resources is needed. As a resource can now be allocated independently from its physical location, global resource counters have to be used. As rCUDA enables GPU sharing among processes, we propose SLURM to allow two execution modes regarding remote GPU

allocation: exclusive and non-exclusive, selected by the system administrator. Non-exclusive mode will maximize resource utilization, at the expense of attaining lower performance, while exclusive mode will lead to maximum performance. Both modes still bring the previously mentioned advantages: a decoupled pool of resources introduce.

The described functionality still requires the development of a new Gres (aka generic resources) plug-in for SLURM, possibly based on the existing one. At this moment, for testing purposes, on the SLURM submission script rCUDA servers have to be explicitly started on the desired nodes prior to start the client tasks.

This work has been conducted in collaboration with Antonio Penya, University of Valencia, Spain.

Results

The computer used to setup the experiments was the cluster named "fuji" from CSCS. It has five compute nodes, each equipped with two Intel Xeon E5670 processors and 24 GB of main memory. Two of the nodes -"fuji1" and "fuji2"- are also equipped with two NVIDIA Tesla C2050 GPUs each, while the rest of the cluster nodes do not have any GPU installed, which fits the rCUDA target environment. The Operating Systems running in the cluster nodes are heterogeneous, being the GPGPU servers a Red Hat Enterprise Linux Server release 5.5 and a CentOS Linux release 6.0, and the GPGPU client nodes, Scientific Linux release 6.1. This does not pose any particular inconvenience for the rCUDA framework itself, as all the OSs are 64-bit versions, therefore matching the rCUDA requirement of all clients and servers executing the same bitwise architecture. The compute nodes are connected through a Gigabit Ethernet network and a QDR Infiniband interconnect. Mellanox OFED v1.5.3-3.0.0, CUDA 3.2 and 4.0 with NVIDIA Developer Drivers for Linux 270.41.19, and SLURM 2.4.0-pre1 are installed across the whole cluster. This experiment runs LAMMPS with the input "in.lj" employing the mvapich provided by Mellanox OFED employing the LAMMPS CUDA library and the rCUDA free version [30]. It spawns two MPI tasks on fuji1 and employs the GPUs located at fuji2. The rCUDA servers are supposed to be already running, although they can also be easily started and terminated within the SLURM script.

3.3.4 *Technical recommendations*

In Table 7 we list a set of recommendations based on our study on resource management software.

Task contributor / Prototype	Lessons learned	Recommendations
Bogazici University	Heterogeneous CPU-GPU supercomputers are being built. Yet, scheduling algorithms employed on these systems are based on the traditional CPU-only scheduling algorithms. In particular, best-fit type algorithms that only make decisions based on available CPU core counts may introduce problems by filling nodes with jobs that use only CPU-cores and hence may	Job schedulers need to look beyond the first job in the job queue not just for backfill-jobs but also for jobs that require GPU resources. Combinatorial optimization algorithms that take into consideration both CPU and GPU resources need to be designed.

Task contributor / Prototype	Lessons learned	Recommendations
	be preventing the use of GPUs by other jobs.	
Bogazici University	Integer programming techniques provide general frameworks for solving NP-hard combinatorial problems. Relaxations of integer programming formulations and solution of relaxed versions by linear programming can help us to develop heuristics for NP-hard problems.	Integer programming and linear programming techniques can be utilized in job scheduling to solve combinatorial optimization problems.
Bogazici University	Besides the GPU resources, schedulers for state-of-the-art supercomputers need to consider other criteria like the interconnection topology and energy. As a result, the optimization problems to be solved by the scheduler become more complex and need more advanced general optimization tools.	Industrial strength integer programming package like the parallel CPLEX can be used to solve complex scheduling optimization problems fast. For SLURM, IP-SCHED plug-in developed as part of WP9.2 and available at available at http://code.google.com/p/slurm-ipsched can be used as a template to develop other customized integer programming based plug-ins.
Bogazici University	Tests carried out with our IP-SCHED scheduler plug-in showed that commercial CPLEX can run in parallel and be able to solve integer programming problems with thousands of variables fast. Free `Lp_solve` package on the other hand is quite slow and can handle only small number of variables.	It is recommended that CPLEX be used in a scheduler plug-in and not the free lp_solve.
Bogazici University	To evaluate the performance of SLURM and in-house developed schedulers, workloads from the Parallel Workload Archive and ESP workloads are used.	Realistic heterogeneous CPU-GPU workloads need to be generated synthetically and/or CPU-GPU supercomputer sites should make available

Task contributor / Prototype	Lessons learned	Recommendations
	However, these are geared towards CPU-only architectures. Hence, for heterogeneous CPU-GPU systems, they need to be modified by introduction of CPU-GPU jobs.	anonymously their workloads to researchers.
Bogazici University	It is observed in various SLURM emulation tests that large sized jobs (jobs using large number of cores) are not delayed so as to have lower average waiting times. In the literature, it is suggested by Kraken supercomputer administrators that large jobs should be favoured. However, jobs using GPUs may be using smaller number of cores. Hence, favouring of small sized jobs may lead to lower average waiting times.	The issue of whether we should favor scheduling of large or small sized jobs needs to be further studied especially on heterogeneous CPU-GPU systems.
CSCS – Interconnect Vertualization	CSCS has been investigating rCUDA (remote CUDA) virtualization interface and its integration to the SLURM resource management system, which is a site-wide job management and accounting system, in collaboration with the University of Valencia. Currently, a custom job script is required as the GPU resources are not provided by the SLURM as independent resources and therefore cannot be scheduled without allocating the host CPU. Currently, the prototype is setup in exclusive mode, where a job has exclusive access to compute resources.	An additional SLURM plug-in is needed for independently allocating and binding CPU and GPU devices to support rCUDA client/server model. This study demonstrates how a solution devised for small-scale clusters can be extended to large-scale installations, where users typically use a job scheduling interface. There are still several challenges but the CSCS prototype has provided a platform to explore a solution in collaboration with international researchers and potentially SLURM developers
JUELICH – Novel	Flash memory cards add an	Develop new resource

Task contributor / Prototype	Lessons learned	Recommendations
Exascale I/O	internal storage resource to HPC systems which needs to be managed to facilitate job processing on I/O nodes, e.g., data staging between external and internal storage.	management concepts and their integration into the system management tools and batch queuing systems.

Table 7: Technical recommendations based on our study of resource management software. Where appropriate, the PRACE prototype utilized for the study is listed in the first column.

3.3.5 *Summary*

The recent evolutions of system architecture favour the usage of accelerators in addition or in complement of traditional CPUs. Resource managers now need to deal not only with the CPU and memory resources but also with accelerators, whether shipped on node or placed in dedicated farms. New storage devices enable to introduce new layers in data hierarchies and can also be viewed as new resources to manage. The support of all these new resources are too limited in current solutions and more work is required to fully benefit from the heterogeneous computing and storage capacities offered by modern and future systems.

Advanced scheduling of jobs on the available resources is still based on heuristics and best effort algorithms that are hard to implement in an efficient way with an increasing number of resource types and constraints. More deterministic approaches, trying to solve the associated NP-complete problem in the most efficient manner, are promising solutions that need to be moved from the research area to the industrial usage. They help to define a generic way to manage the scheduling problem applying a same strategy to multiple set of resource types. The integration of Integer programming and Linear programming solvers in state-of-the-art resource managers is an interesting path to follow. These methodologies rely on dedicated mathematics libraries. Currently, no open source solution is viable compared to proprietary one. Open-source alternatives should be enhanced for more independence on a midterm basis.

3.4 Data management

3.4.1 *JSC I/O prototype evaluations (CSCS)*

Motivation

Next generations of HPC systems have different requirements for network, computing, storage and applications etc. The requirements concerning the I/O subsystems are more critical due to the continuously growing gap between computing and I/O performance increase during the past. The goal of this project is to deploy a PCI-E SSD cards on x86 cluster followed by adding SSD cards to the new IBM Blue Gene series system called BG/Q.

Description of the work

This effort involved evaluation of hardware and low-level software. Specifically, the SSD cards have been successfully integrated and their correct functionality performance has been heavily tested using synthetic benchmarks.

Results

The target parallel file system, GPFS, drivers and benchmarks tuning are taking place during this period on the BG/Q to sustain the peak performance of the SSDs.

Important steps for the Future:

- Promoting SSDs integrations in HPC systems.
- Develop and implement concepts which improves the usability of SSDs integrated into HPC systems
- Proving the concepts of scaling SSDs in HPC and Parallel file systems.
- Running real application that includes IO benchmark to address the benefits.
- Include QoS in the file system

3.4.2 *GPFS and Lustre evaluations (CINECA)*

Motivation

Data are increasing their importance together with the performances of the supercomputers. The choice of parallel file system that best fit the load requirement and its planning, installation and configuration is fundamental in order to maximize performance and limit the impact of HPC infrastructure problems.

Description of the work

GPFS (General Parallel File System) is a full featured POSIX IBM product. It was tested on the CINECA tier 1 system, a Linux cluster where about 300 nodes are GPFS clients, with six disk servers with six fiber channel (FC4) links. The file system is about 90 TB with a block size of 4 MB (minimum fragmentation 128KB), with data and meta-data mixed on same disks. It is configured to relay on RDMA data transport on Infiniband QDR, single rail switch. The back-end Storage consists of 12 RAID6 arrays of disks (8+2 1TB 7.2Krpm SATA disks).

Lustre file system is a full featured POSIX Open Source product maintained and developed by Whamcloud. It was tested on a different hardware/storage: four client nodes and three servers with four fiber channel (FC8) links. The file system was about 40 TB with a block size of 1 MB. Data and meta-data weren't mixed by design. The data transport was LNET over TCP/IP (10 GbE). The back-end Storage consisted of RAID6 arrays of disks (10+2 2TB 10Krpm SATA disks, but meta-data on DAS).

Results

Scalability of disk arrays numbers is good in both file systems. The scalability issue of GPFS comes from the number of supported clients. Vice versa, the issue scalability of Lustre is due to the Meta data server and the centralized meta-data management, which is also a single point of failure. In addition, all GPFS server functions and meta-data can be replicated and disks can replaced without production interrupt, invoking simple management commands. GPFS is preferable from the point of view of availability and resiliency and for easier management.

3.4.3 *Technical recommendations*

We provide a set of technical recommendations on data management software in Table 8.

Task contributor / Prototype	Lessons learned	Recommendations
CINECA	CPUs based jobs are less I/O intensive than the GPUs ones.	Deploy parallel file system taking in account CPUs GPUs hosts or only CPUs hosts. Separation between data and meta-data. Deploy different parallel file system for different usages.
JUELICH – Novel Exascale I/O	Various challenges to integrate flash memory into massively parallel HPC architectures and to enable its efficient use remain to be addressed.	Further explore the design space and foster the integration of additional storage levels into parallel file systems and the development of I/O middleware.
JUELICH – Novel Exascal I/O	Significant efforts are needed to adapt applications for making efficient use of additional storage level implemented using flash memory.	Foster development of I/O interfaces which facilitate optimal use of the available resources. Provide support for application developers to adapt their applications.
CEA – Exascale I/O	The scalability of the system management provided with the Xyratex solution is limited. The use of SSUs ("Scalable Storage Units") is very convenient and scale in terms of hardware and Lustre performance, however, we encountered several scalability issues with "only" 9 SSUs. For example, the default configuration management tool used (puppet) doesn't scale.	Foster vendors to improve their system management software in order to scale for larger systems. Vendors are not always aware of this issue until they build a large system.
CEA – Exascale I/O	The integration of the Xyratex Lustre storage solution to our R&D compute center environment was not trivial. External services provisioning like NTP, DNS was supported, but more complex LDAP settings, specific Infiniband partition key (P_key) or Lustre Network (LNET) index were not	System management developments remain in order to facilitate the integration of such Lustre storage solution in existing production HPC environments. Such systems should remain open to allow on-site specific configuration to be supported, especially for large HPC compute

Task contributor / Prototype	Lessons learned	Recommendations
	configurable without bypassing Xyratex system management. The good point is, however, that the use of a standard Linux distribution based on Red Hat Enterprise Linux 6 with open source tools allowed us to do it without much pain.	centers.
CEA – Exascale I/O	Software RAID using Linux device-mapper can be used to build embedded high-end storage systems. Expected system performance were reached, however, some system failures were encountered that may be related to the RAID stack.	Don't ban the use of new RAID engine for disk arrays, like device-mapper based RAID. Still, more work and research have to be done in terms of software RAID resiliency.

Table 8: Data management software recommendations. Where appropriate, the PRACE prototype utilized for the study is listed in the first column.

3.4.4 *Summary*

Nodes and system architectures are constantly evolving. Accelerated nodes tend to produce and or access more data than traditional nodes. The balance between performances and IO bandwidth must be adapted to that aggregation. More bandwidth and capacity are required to store the information produced and consumed by next generation systems. Meta-data is an important criterion that can severely limit the efficiency of a file system.

An IO back-end must be scalable and adaptable in terms of provided services but also in terms of required services. Indeed, to be fully operational, a file system needs to be properly integrated in the data center. Administrators must be able to manage and configure the whole system efficiently at scale, including start/stop of the back-end as well as software upgrade. An open-source solution is more reactive. By enhancing it "on need", it is possible to adapt the existing components and cope with unexpected requirements or behaviour that are inherent to state-of-the-art HPC installations.

By reducing the usage of dedicated hardware, software RAID is promising even if not yet completely mature. More work is required to ensure the resiliency of software RAID stacks.

Data hierarchies are evolving too. Depending on the I/O workloads, it can be necessary to think about providing multiple sets of file systems and data back-ends, either local or remote, in order to provide the most adapted solutions to the various requirements of the applications. This can be done transparently or on demand, using a yet to define API, depending on the targeted efficiency.

3.5 MPI and Communication libraries

3.5.1 *MVAPICH2-GPU evaluation (CSCS)*

Motivation

MVAPICH2-GPU provides an optimized GPU-GPU communication interface for MPI communication for clusters with NVIDIA GPU devices and Infiniband interconnection network [31], [32]. This effort aims at investigating the feasibility of this interface for application development and characterizing performance of end-to-end GPU-to-GPU communication over the interconnection network to the hardware and tools developers. The MVAPICH2-GPU interface provides a direct mechanism to transfer a buffer from a host GPU to the destination GPU memory without explicitly copying data between host and GPU memories. In order to avoid memory copying from the GPU buffer to the network buffer, NVIDIA provides a feature called GPUDirect, which enables the GPU and the Infiniband driver to share their address spaces. Existing applications can gain 10-15% performance improvement by simply linking to the MVAPICH2 MPI library that has been CUDA enabled.

The final goal of this work is to completely bypass the host memory copying. With the latest announcement of GPUDirect-RDMA for the next generation GPU devices codenamed Kepler, we expect performance and scaling efficiencies to improve significantly as data from GPU devices can be copied over to the network buffer without intermediate copying into the host address space.

Description of the work

The CSCS prototype system, which is composed of dual-socket Intel Westmere nodes and two NVIDIA M2090 GPU devices and QDR Infiniband interconnect, has been used to deploy the MVAPICH2-GPU. The system has been setup with the latest stable release of CUDA and driver that support both the GPUDirect and the uniform memory address space capabilities. CUDA version 4.1 and the driver version 295 have been used for the experiments. MVAPICH2 version 1.8 has been installed on the system, which has been built with CUDA 4.1. In order to perform the performance characterization, the MPI micro-benchmarking code for latency and bandwidth experiments have been extended to include GPU bindings. This work has been done in close collaboration with Professor D. K. Panda and his group at the Ohio State University, USA.

Results

Data transfers between host to memory using the pinned GPU memory has typically highest bandwidth values, while the data transfers from GPU memory to the host is the slowest path. The MPI bandwidth over the Infiniband is typically lowest on a QDR platform. Hence, in an optimal case, the application should experience no less than the peak MPI bandwidth. The results with MVAPCIH2-GPU confirm that the overheads are negligible for different message sizes and communication patterns. Typically the performance is limited by transfer bandwidth between the host and the GPU devices. Furthermore, optimization with NVIDIA GPU to GPU peer-to-peer communication interface can also be exploited by MVAPICH2-GPU implementation and our experiments on the iDataPlex prototype, single node with 2 GPU devices, demonstrate this capability.

3.5.2 *Evaluation of Infiniband routing schemes (CSCS)*

Motivation

A number of high-end computing platforms, especially the ones with a few tens of thousands of nodes, have network topologies that are scalable both in terms of aggregate bandwidth and cost. This includes torus topologies of BlueGene and Cray XT and XE series systems, where the interconnect switching and routing schemes are co-designed together with the MPI implementation [33][34][35].

High-end systems based on the commodity Infiniband interconnect have traditional hierarchical topologies that have been constructed using switches. These include fat tree and Clos, which have been developed using switches with large number of ports. The cost of the switches does not scale linearly with the number of ports [36]. Thus, the main drawback is extensibility in a cost effective manner. We therefore attempt to design and evaluate performance of two-dimensional torus topology interconnect using low-port count & cost effective Infiniband switches.

Description of the work

The study has been performed in multiple phases. The first phase was designing and deployment of a prototype using multiple of 8-ports switches. The targeted platform was consisted of 32 IBM iDataPlex M3 nodes and two different interconnect partitions. Each IBM iDataPlex node was composed of two 6-cores Intel Westmere processors and two NVIDIA M2090 GPU devices. In addition, there was a 36-ports Infiniband QDR switch and 16, 8-ports QDR, unmanaged switches for the 2D torus topology. The design of the 2D torus topology using 8-ports switches is shown in figure below.

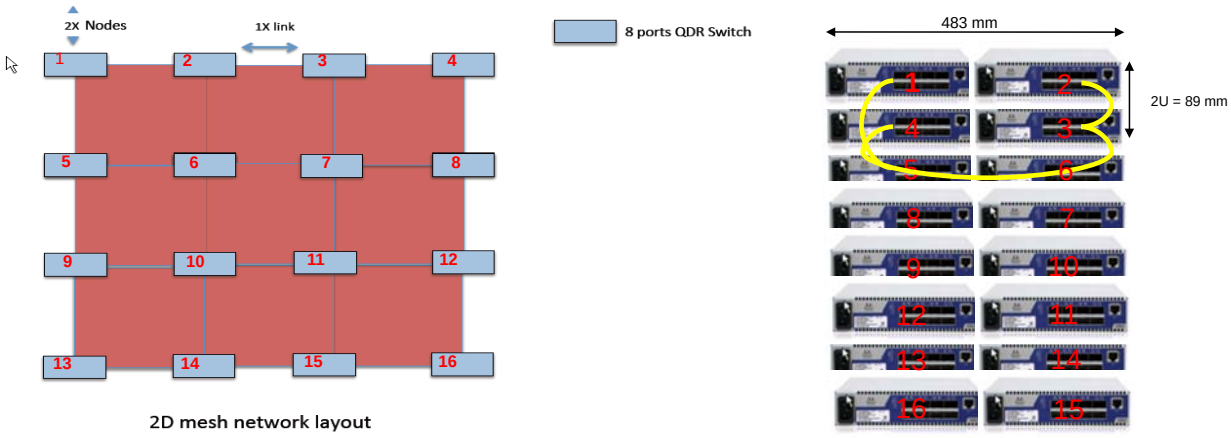


Figure 20: 2D torus topology & the 8-ports switch layout

Two partitions were designed to comparatively evaluate performance of a full crossbar 36-ports switch and with the 2D topology setup. The results are comparable since the node design and system software is identical on both partitions.

As a consequence, the second phase of the study was the deployment of the two partitions in an optimal manner. This includes an optimal routing scheme for the given topology and development of topology aware MPI.

The final phase was the evaluation of the two topologies using extensive micro-benchmarking. We deploy Infiniband monitoring and management tools to perform

measurements on the switches themselves while running micro-benchmarking applications on the node to validate performance results and to identify any network related issues, for example, network congestion [37].

Results

Both partitions have Linux version 2.6.32 and MVAPICH version 1.8 built with GCC 4.6 compiler. On the 36-ports partition, the standard minimum hop routing scheme was implemented while on the 2D torus partition torus-2QoS was implemented. In order to measure the improvements with the optimized routing scheme with the given topology, we compared results using a non-optimized scheme called LASH-DOR. The results showed a significant improvement in performance, by a factor of 2 or more, with the optimized routing scheme. The results were comparable with the 36-ports QDR switch while for some message sizes, the 2D topology yielded better performance for MPI collective communication operations as shown in Figure 21 using the Intel MPI benchmarks for the MPI_Allreduce operations. We observed similar behavior for the point to point benchmarks. All tests were performed for 128 MPI tasks.

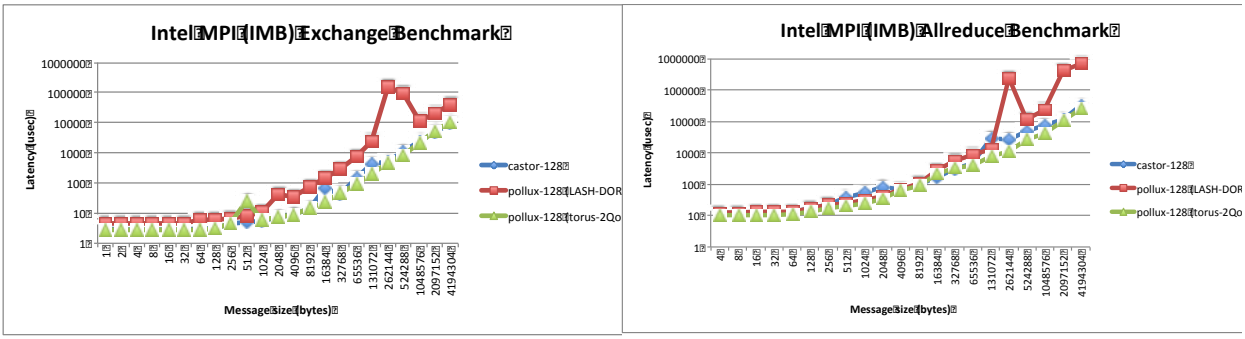


Figure 21: Impact of optimized routing schemes on the two test partitions: 36-ports QDR switch with the deualt routing (castor-128), 2D torus setup with LASH-DOR (pollux-128 (LASH-DOR)), and with Torus QoS (pollux-128 (torus-2QoS)).

We deployed the Unified Fabric Management (UFM) utility from Mellanox to observe network patterns on different nodes and switches. The results with the monitoring tool confimed the slowdown due to network congestion. A snapshot of the tools is shown in Figure 22.

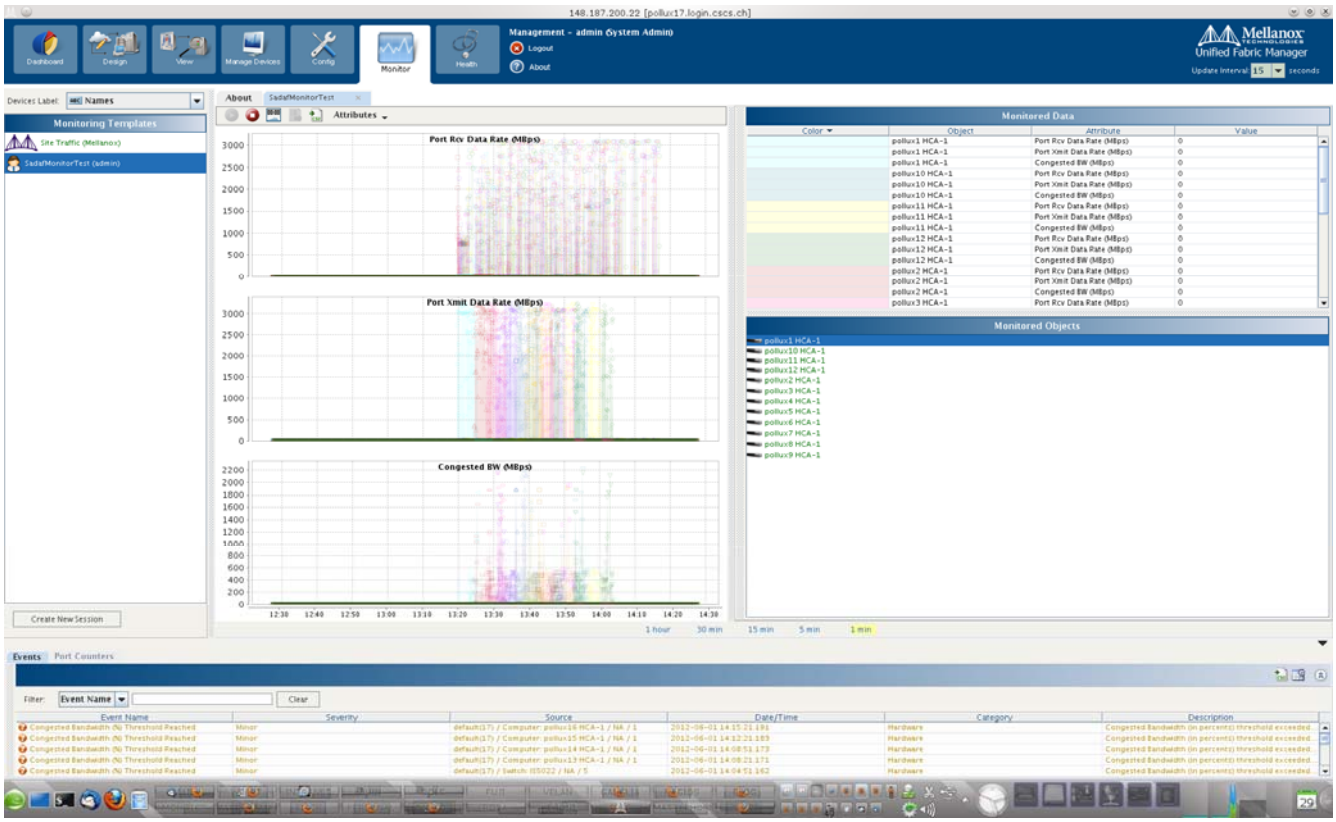


Figure 22: Monitoring of the network congestion using the UFM tool

On the 2D torus network, we also investigated the whether or not topology awareness plays a critical role. We collected message injection rates using the OSU MPI micro-benchmarks with regular and random node ordering. Results are shown in Figure 23. For small message sizes, communication between adjacent nodes yield better performance results. As the message sizes increase and the communication becomes bandwidth bound, the difference between regular and radom ordering diminishes.

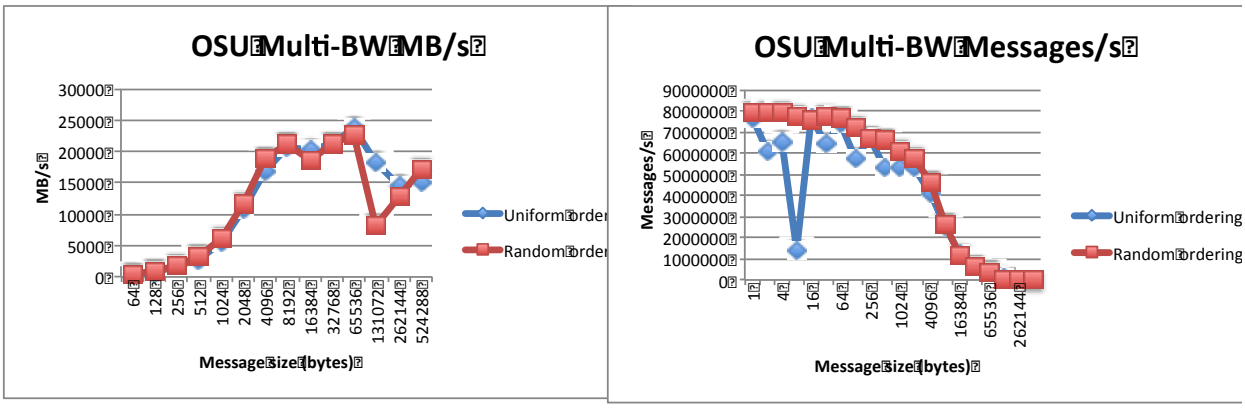


Figure 23: Impact of network traffic routing

3.5.3 *Technical recommendations*

We list technical recommendations for communication libraries in Table 9.

Task contributor / Prototype	Lessons learned	Recommendations
CSCS – Interconnect Virtualization	System engineers and administrators need to carefully evaluate IB routing schemes for unconventional IB topologies using unconventional switching components. We experimented with different topologies (2D mesh and torus) and routing schemes (LASH, DOR and Torus-2QoS) to identify system throughput bottlenecks. MPI library also needs to be topology aware.	As regression suite is needed for setting up direct-connect, high dimensional topologies to tune and troubleshoot functionality and performance issues. Currently manual intervention and expert knowledge is needed to accomplish this task. This could also be a co-design opportunity between process and interconnect vendors, system software and library developers to provide highly efficient I/O implementation to users.
CSCS – Interconnect Virtualization	MPI communication without explicit memory transfers is currently available in MVAPICH2-GPU MPI library implementation. Experiments show high throughput and high productivity for different communication patterns. This work is done in close collaboration with Prof. DK Panda group at the Ohio State University and the prototype provided a collaboration opportunity.	We need to collaborate with other MPI developers and probably with MPI forum to offer code and performance portable implementations. In order to ensure wider adoption together with code and performance portability, interoperability with other accelerator paradigms must be investigated.
CSCS – Interconnect Virtualization	Designing direct-connected topologies using InfiniBand 8-ports (unmanaged) switches requires an in-depth understanding of the cabling from processor and between switches. This is not scalable to large scale installations using open source management, trouble-shooting and diagnostics interfaces. Incorrect cabling from processors and between switches can severely impact communication	In order to introduce cost effective and scalable interconnect topologies using commodity components, especially unmanaged switches, further research and investment is needed both at the fabrics level and also for the management and troubleshooting tools such as OpenSM.

Task contributor / Prototype	Lessons learned	Recommendations
	throughput.	
CSCS – Interconnect Virtualization	Currently, PCIe peer-to-peer communication is not fully supported by many vendors. Hence, a true host bypass, which CSCS GPU virtualization prototype intended to evaluate, could only be implemented through a software layer to communicate between GPU and the network interface. This is also a performance critical design component for clusters based on PCIe accelerators.	As accelerators become central to the node design, direct communication channels must be provided by vendors and should be supported by Linux kernels to allow for direct memory transfers between the accelerator memories and the network interface, without an intermediate copying step to the host memory. Research and development is needed for the Linux kernel extensions in collaboration with vendors and could be a co-design opportunity for PRACE.

Table 9: Technical recommendations for communication libraries. Where appropriate, the PRACE prototype utilized for the study is listed in the first column.

3.5.4 *Summary*

Data hierarchies and system architecture are evolving to integrate the ever increasing number of components included in successive state-of-the-art supercomputers. The topologies involved at every level of the supercomputers have consequences on the performance of the whole systems. The mostly used interconnect topologies does not scale well in terms of prices and cabling.

It is necessary to find solutions to reduce the price of the interconnection network for larger machines while still providing sufficient behaviours in terms of bandwidth, latency, manageability and resiliency. Intra-node topologies are more and more complicated and require a perfect knowledge of the interaction to choose and tied the components that have to work together. The frontier between intra-node and inter-node topologies must be observed with great attention to maximize the usage of the external connections of the nodes.

Software stacks involved in the communication of parallel applications at every level must be mature enough to leverage all the primitives offered by the underlying hardware. The only way to achieve the highest performance is to use the most advanced features of the involved hardware.

3.6 Conclusions

The topics discussed in the previous sections outline several problems that can be mapped to five transverse key concepts that have already been identified by the WP9 community and expressed in the D.9.1.2 deliverable.

Scalability, addressed since many years up to the Petascale level, is one of the key issues and becomes more and more problematic not only on the computing side but also on the I/O and the management sides. Every part of a system must now be designed with that goal in mind. Vendors take that consideration into account more than ever and this is a good sign for the future of HPC.

Heterogeneous architecture as well as new Data and Memory hierarchies bring additional complexity that must be addressed at every level of the software stack, from the operating system to the communication libraries and primitives, taking into account the final goal of scheduling jobs in the most efficient manner on the set of interconnected resources. All the components glued together in modern supercomputers are not sufficiently managed in a coherent way and more work is required to leverage all the capabilities of the hardware as well as to select efficiently the most adapted resources for the different applications.

The efficiency of the next systems in term of energy consumption will be one of the most observed elements. Taking power and cooling consumptions into account in the system software stack will help to optimize the amount of power required to execute the various workload. We are still at the beginning of the power aware era in HPC. Sensors still need to be enhanced and better integrated in the system management tools. Power and cooling are far from being treated as the resources they are in a scheduling point of view.

Despite the fact that fault-tolerance is discussed and predicted as the next level challenge since many years now, it has always been tackled by improved reliability of the supercomputer's building block and optimized checkpoint/restart strategies. However, the scale required to achieve the ExaFlop/s, associated with the heterogeneity and the complexity of the building block of next systems could make this prediction come true. Resiliency of the systems must be guaranteed to reduce the mean time to interrupt of applications. Failures need to be treated as a possible and manageable input in application runtimes and no longer as the final output of computations. A lot of work is required in that area. 2IP-WP11 should bring useful insights in this field.

The next hurdles to clear are high. First steps have been done but the goal is still far. The different elements included in the system software stack must be co-designed to leverage every single feature. To have the best interconnect topology or routing schema may not suffice since the resource manager may only allocate rows of nodes. To have the most coupled topology and scheduling strategy is good, but if the application processes are spread in an incoherent manner or if the I/O bandwidth is too tight, it will waste cycles in pending communications. Information for adapted actions is a way to make things work efficiently. Systems have to be co-designed with applications. Applications will need to co-exist with system components, processing live information to adapt their behaviour to the different volatile characteristics.

4 Recommendations on HPC tools

This section identifies research on tools for High Performance Computing (HPC). These tools are particularly focused on performance analysis, debugging, optimizing, and monitoring that should be supported in order to enable applications to scale to large systems, in special, in order to reach Exascale performance.

The objective is to avoid "flying blind in the midst" of programming and optimizing codes for very large scale. The behaviour of applications on complex systems is often different from the programmer's assumptions or models in mind. In current practice, decisions to implement certain optimizations are based on not sufficiently detailed data and certainly on limited information of the expected gain that a given code restructuring will obtain. This obviously results in poor application performance especially at the large scale.

The HPC landscape is getting more complex and changing in many dimensions. Architectures are mostly based on multi-core processors with deep memory hierarchies. It is typical today to see at least three cache levels where the top level of cache is shared among all cores in the processor. Processor architectures are also showing a wide variety with different features and performance. Examples of this are the traditional Intel x86, IBM Power processors, the integrated approach with AMD Fusion – which combines CPU and GPUs on a chip – ARM processors to tackle the market of low power, and the MIC many core processors from Intel. This complex environment is making the optimization of applications more difficult, because what it is working for some architecture usually is not the case for another different one, and thus it requires specialized tools to effectively deal with this.

At the same time, there is a strong demand for more parallelism at the chip level and also at the system level mostly driven by the need to reach exascale. Applications should be able to exploit this huge amount of parallelism in order to harness the potential of exascale machines. However, most of today's applications do not properly scale at this level. For this reason, there is a strong need for performance tools to analyze and optimize these applications for large systems.

In addition, there are new user groups that are starting to use HPC systems that are requiring better tools to assist them in properly using these complex systems. These user groups are coming from emerging application areas such as Linguistics and Biology. These groups are experts in their field which is quite different from computer science. Because of that they have little knowledge in how to efficiently optimize applications for large scale. Therefore, they need tools for performance analysis, debugging, and optimizing their applications with a special emphasis on easy-to-use and automate code transformations. In particular, they are interested in tools as simple as click one button to get an instant advice and/or automatic translation to an optimized version of their code.

The last driver that we have identified for needing better tools for HPC is due to the recent moving trend of HPC to the industry and commercial sectors. One popular commercial sector is the financial services where HPC is being used to substantially improve risk management, trading analytics, and wealth management. In this environment providing a low time-to-market, higher productivity and reliability are key to success. For this purpose, better tools to are needed in order to support these critical commercial application requirements.

We have identified four areas that need support for advancing HPC tools in the future: Task-based/asynchronous, intelligence, models, and scalability. These are described in more detail in the following sections.

4.1 Task-based/asynchronous support

Task-based dataflow programming models such as OpenMP and OmpSs [38] are showing to be very powerful in exposing high level of parallelism in a wide range of scientific applications. It has been shown that they are very efficient for shared memory machines and allow exploiting the potential of multicore processors and accelerators such as GPUs. In addition, there is a hybrid dataflow programming model MPI/OpenMP and MPI/OmpSs that parallelizes computation on the distributed-memory nodes using MPI.

Task-based parallel programming languages require the programmer to partition the traditional sequential code into smaller tasks in order to take advantage of the existing dataflow parallelism inherent in the applications. The programmer indicates data dependencies between parts of the code which become tasks and then are scheduled by an execution framework at runtime. The major advantage is that it can "react" on internal irregularities, e.g. differences in execution time of each task, and external conditions, such as hybrid architectures. Furthermore, it has the potential of taking advantage of very distant parallelism – parallelism of code sections that are mutually far from each other.

Debugging refers to be able to seeing and controlling the execution of a program. Traditional debuggers for sequential programs such as gdb are not sufficient for dealing with the complexity of task-based parallel programming languages. Traditional debuggers work at a single thread and working at a line level in the code. They are good in checking serial correctness of the code. However, traditional debuggers are not enough to deal with the complexity of multiple threads running concurrently in the application. A task-based parallel program differs from debugging an otherwise parallelized program as the parallelization is determined by data dependencies instead of explicit scheduling of tasks and synchronization between them. For debugging task-based parallel programming languages it would be interesting to see which is the status of each task, which is the function that it is executing and in which thread is executing this task. For task dependencies it would be interesting to see which memory addresses cause some dependency with other tasks and which tasks depend on each other.

In addition, it is necessary to control the execution by stepping through the application task-wise. Debugger users should be able to block tasks and even prioritized tasks in order to see what the impact on performance and correctness of such as transformations are. Furthermore, it will be necessary to add/remove data dependencies among tasks.

Therefore, the development of new debugging tools capable of assisting the programmer with debugging task-based parallel programs is a crucial condition for effectively exploiting the possibilities of these models.

Another recommendation is about the lack of support of metrics and models developed specifically for task-based parallel programming languages. This models and metrics may be interesting to support in current tools in order to understand the behaviour of individual tasks that run in an application. In particular, it would provide insights in what are the critical tasks that are likely to have a biggest impact on the performance of the application. Also it could help answer the question of what would be the impact on the overall application performance and when to accelerate/optimize a particular task in the application. Answering these questions and measuring the impact of them is crucial for the optimization of the task-based parallel programming languages.

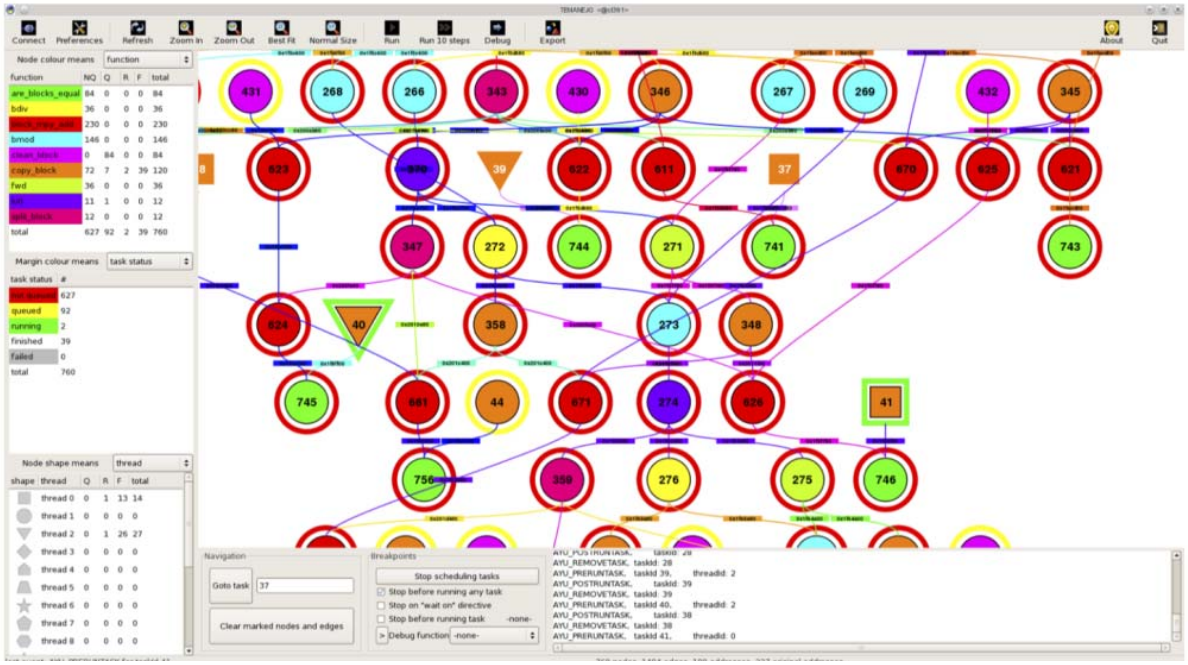


Figure 24: Example of data dependency graph among tasks in the Temanejo tool.

4.1.1 *Recommendation evidences on existing tools*

An illustrative example to show these recommendations is found in the Temanejo tool [39]. Temanejo is a debugger for task-based parallel programming languages. It is still under development. Figure 24 shows a screenshot of the task dependencies among tasks for some example code in Temanejo. Also, the memory addresses that actually are creating data dependencies can be seen. This tool provides full support for the OmpSs programming language [38], but not standard OpenMP. In addition, it only provides basic support for MPI by allowing only one thread per MPI process.

As can be seen, the visualization of the data dependency graph is a flat graph. It would be interesting to support hierarchical dependency graph visualization. This is critical for more complex applications where the number of tasks is huge. In this scenario, providing a hierarchical view would enormously facilitate the debugging of applications.

Moreover, it would be interesting to support some kind of analysis of the dependency graph in such a way that it could identify the critical path of the execution and the highest and lowest possible concurrency during the execution.

4.1.2 *European contributors*

In Europe there are two strong teams that are contributing to these different areas. They are briefly described below in Table 10.

Recommendation area	European institution/Country	Tool name	Description
Metric/models for task-based programming models	JÜLICH SUPERCOMPUTING CENTRE (JSC) / Germany	Scalasca [40]	A performance analysis toolset that has been specifically designed for use on

Recommendation area	European institution/Country	Tool name	Description
			large-scale systems supporting MPI and MPI/OpenMP
Debugging for task-based programming models	High Performance Computing Center Stuttgart (HLRS) / Germany	Temanejo [39]	A debugger for applications parallelized with the use of the OmpSs programming model

Table 10: European contributors for the recommendations for the support of task-based parallel programming languages.

4.2 Intelligence

The amount of information collected when tracing applications is growing at a fast pace. As applications run on ever larger parallel computers the larger is the amount of information collected. Also, applications are getting more complex involving the simulation of multiple physical models or multiple simultaneous physical phenomena. For example, combining chemical kinetics and fluid mechanics or combining finite elements with molecular dynamics. This is inevitable requiring the need of smarter tools to deal with this huge amount of information. Furthermore, there are new user groups that are starting to use HPC systems that are requiring better tools to assist them in properly using these complex systems.

There are two basic directions for future development on providing more intelligence to the existing tools. One direction is to address the scalability issue by deciding what information is relevant to capture from the traces obtained in the parallel execution. Blindly capturing everything from all the processes during the whole duration of the application is impossible. Some aspects to consider for reducing the size of the traces are:

1. If the application behaviour is very repetitive a few iterations would be enough, but the tool has to detect the period. Also, the tool would need to be able to detect whether this behaviour changes over time or not. In case it changes, it should obtain the new behaviour of the new period.
2. Improvement of sampling techniques by automatic detection of the optimal sampling frequency of applications. It is possible to extract this frequency using spectral analysis. It is shown that the optimal frequency is very useful to extract significant performance information very efficiently and accurately showing their internal iterative structure of the application without recording everything from the application.
3. Highlight interesting data. Not every MPI call has the same amount of information. i.e. many Iprobes are in sequence and therefore only the first, the last and the number of messages in the sequence are needed. Interestingly, it might still be useful only to select at random a portion of the records.
4. Only recording MPI calls that imply long delays. These calls have a high probability to negatively impact the execution time of an application.

The second direction is focused on increasing the insight of information that can be extracted from the raw data in the trace. Instead of depicting the huge number of information from the traced data which is nearly impossible, it will be more practical to find/show/map only the interesting data which is typically very small compared with the whole data trace.

There are several techniques that can be developed in this direction:

1. Spectral analysis. This technique is based on applying wavelet transform, Fourier transforms or other transformation to the trace in order to easily detect the most important frequencies of the application execution. These main frequencies are strongly related to the internal loops of the application' source code. Therefore, it automatically obtains the structure of the application in phases where these phases are clearly repetitive over time.
2. Clustering. The idea is to identify computation regions of similar behaviour. For example, areas that shows similar in terms of duration or hardware counters regardless if they are the same or a different routine. It might be that different routines in the application may have similar behaviour or one routine may show different behaviours overtime. These kinds of effects can be detectable with these kinds of techniques.
3. Automated detection of known problems. For example, scalability problems due to load imbalance can be automatically detected. Moreover, these techniques can identify the problem, localize in the code where it is happening, and finally provide some solutions in order to fix it.

4.2.1 *Recommendation evidences on existing tools*

To illustrate the clustering technique we will use the EXTRAE tracing tool. Figure 25 shows the clustering of the IPC metric for a 64-process run of the application SPECFEM3D. As you can see, it allows us to quickly identify two areas of the code with a quite different IPC value between them – one has a high IPC in between 0.6 and 0.8, and the other a very low IPC in between 0.1 and 0.3.

The interesting thing is that this IPC does not belong to different functions, but to the same function executing in a different processor. This is illustrated in Figure 26 where we can see the distribution over time of the different computation regions showed before in Figure 25. As you can see, there are only a couple of processors that shows this low value of the IPC. This finding would be very difficult to detect in large traces. Clustering allows you quickly pinpoint these performance problems.

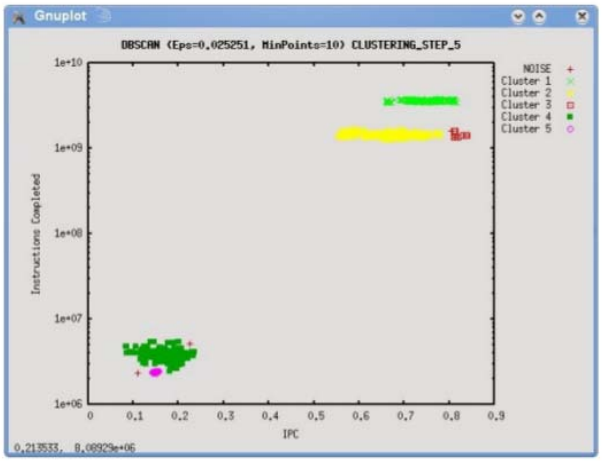


Figure 25: Clustering the IPC

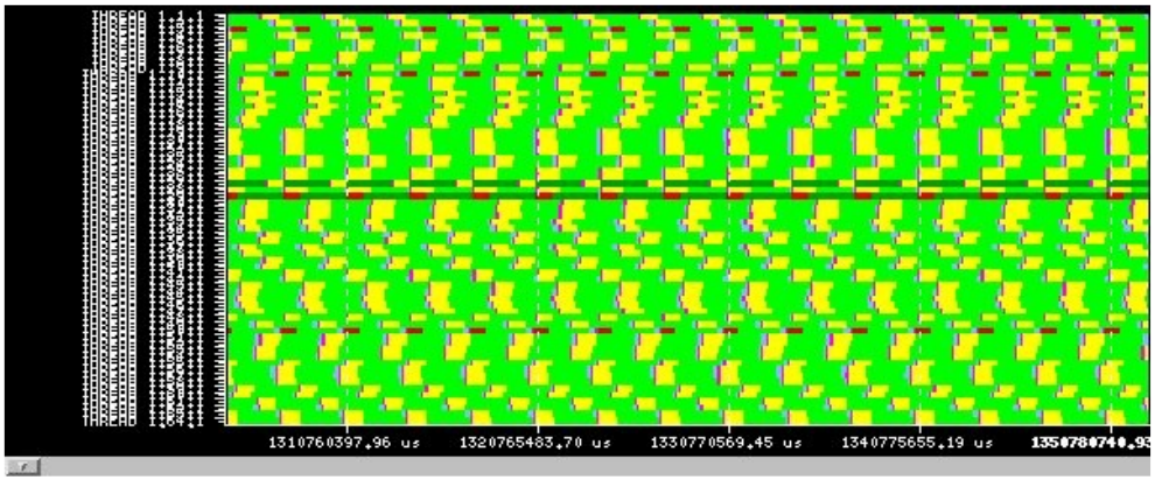


Figure 26: Cluster distribution over time

This clustering technique could be also applied to task-based parallel programming languages. In this context, it will be useful for instance to identified different performances of the same task on different processors and also characterize the behaviour of the different tasks trying to optimize the task with low performance.

Additionally, in Vampir we can find an interesting utility that is based on changing the opacity of the colours when we are visualizing the trace. Figure 27 shows a trace with different communication and computation records where it is visualized using a high opacity factor. As you can see, using a high opacity factor it easily highlights the records with red colour that corresponds to communication operations. This is a technique that follows the approach of find/show/map only the interesting data in the whole data trace, in this particular case the communication operations.

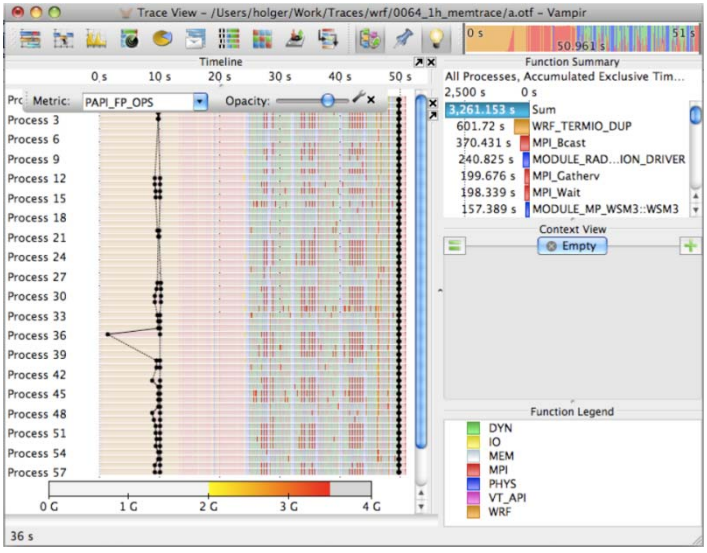


Figure 27: Opacity selector in Vampir.

4.2.2 *European contributors*

Table 11 summaries the European institutions that are developing tools with emphasis of providing more intelligence in order to better analyze the huge information in raw data.

Recommendation area	European institution/Country	Tool name	Description
Spectral	Barcelona Supercomputing Center (BSC) / Spain	Paraver [38]	Performance visualization and analysis tool based on traces
Clustering	Barcelona Supercomputing Center (BSC) / Spain	EXTRAE [38]	A tracing tool to extract computation and communication events from applications
Clustering	Technische Universität München (TUM) / Germany	Periscope [41]	A scalable automatic performance analysis tool
Highlight interesting data	Technische Universität Dresden/Germany	Vampir [42]	Vampir provides an easy to use analysis framework which enables developers to quickly display program behaviour at any level of detail

Table 11: European contributors for the recommendations on intelligence.

4.3 Models

As the cost of developing, deploying and maintaining high performance systems rises, it becomes more and more important to predict system performance in advance. This can be achieved by using analytical model of applications. These models will be used to analyze, predict, and calibrate performance for the systems of interest. They are becoming the overall predictors of how the whole system performs.

Concretely, a performance model analyzes both application and system characteristics. Application characteristics are defined uniquely for each application and include processor flow, data structures used, frequency of use and mapping onto the system, and their potential for resource contention. System characteristics include node configuration (processors per node, shared resources) and inter-processor communication (latency, bandwidth, topology). Many of these are measured for an existing system or need to be specified/simulated for a future system. A separate performance model is usually developed for each application of interest and thus the approach is application-centric.

Performance modelling can be applied to every stage of the design of a large scale system. Specifically, in an early stage, they are becoming very important to guide system development and procurement decisions of large scale parallel systems. Then as the hardware becomes available, they will validate their predictions with real-world tests. Moreover, in this later stage they can be also used to understand the complex interaction between applications, software environments and computer hardware in order to achieve higher performance by optimizing the system.

We can identify various main areas of further development on performance modelling:

1. Root cause. An automatic methodology able to show general but non-trivial performance trends. An analytical model which consists on several performance factors is used to underpin the root cause of application performance degradation. It tackles the problem raised when the temporal or spatial distance between cause and symptom of a performance problem constitutes a major difficulty in deriving helpful conclusions from performance data.
2. Methodological. It is related to the development of models to understand better the algorithms used in the applications. These models can be used to drive the development of new algorithms that are more efficient. There is some recent work that has shown a model for determining the lower bounds on the number of words moved between processes in a direct linear algebra algorithm (matrix multiply, LU, Cholesky, QR factorizations, and so on). This information can be used to find algorithms that attain these lower bounds, specifically on interesting classes of sparse matrices.
3. What-if models. Instead of building an accurate model of the application, they are focused on modelling a particular performance factor of the application. Using simulations tools they are able to predict the application's performance when this factor is improved by several times. Analyzing what-if scenarios are interesting to evaluate whether some code modification is worth to pursing or not.
4. Power models. Building power models based on hardware counters available on multicore architectures. It is based on actual power measurements. The counters are sampled at regular intervals and the activity in each component in the processor is linearly correlated with the actual power consumption. Stochastic methods are usually used to help on building these power models.

4.3.1 *Recommendation evidences on existing tools*

The root cause approach can be found in the Scalasca tool [40]. Figure 28 shows a 3D view of the processes from the Zeus MP/2 astrophysics code arranged in a sphere showing the computation time (shown on the left) and their associated communication wait states (shown on the right). As can be seen on the left graph, the processes in the outer region of the sphere are the ones waiting longer on communication operations (shown with the red colour). Looking at the root cause of this phenomenon in Scalasca it is displayed on the right graph the computation time of the processes. There is a strong correlation in between the computation times and the high wait delays. Processes in the inner region of the sphere topology carry more computation load than the outer region. Therefore, processes at the rim of the inner region delay those farther outside producing these longer wait times.

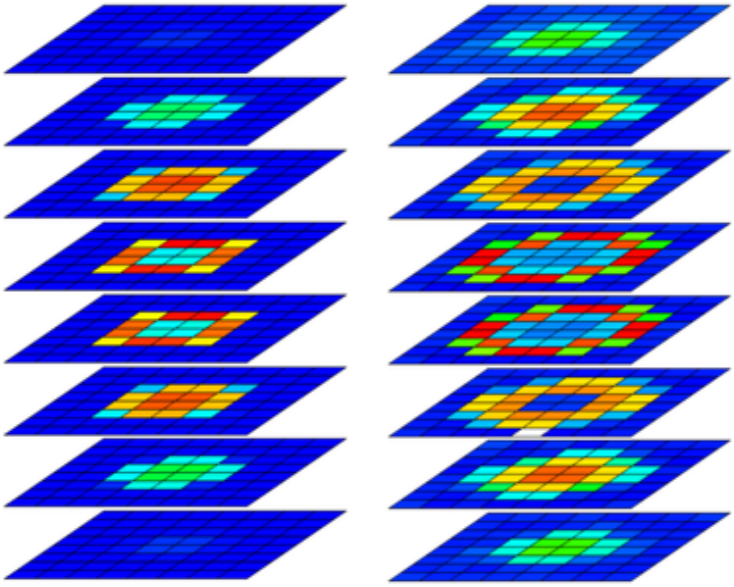


Figure 28: A 3D view of several processes arranged in a sphere showing the computation time (shown on the left) and their associated communication wait states (shown on the right).

Another illustrative example found in today's tools for the recommendation on What-if models is provided by the tool Mpisstrace. This is shown in the Figure 29 where a particular task out of the four tasks that the Cholesky application is composed of is sped up 2× in the different number of cores. As you can see, this tool is able to predict for each number of cores which is the most effective task to sped up. In particular, the `sgemm_tile` task is the one that delivers the highest improvement in performance for Cholesky when it might be sped up by a factor of 2× for the case of one core. Based on this information programmers could focus on optimizing the right task avoiding a waste of time optimizing other tasks that are not having a bigger performance impact.

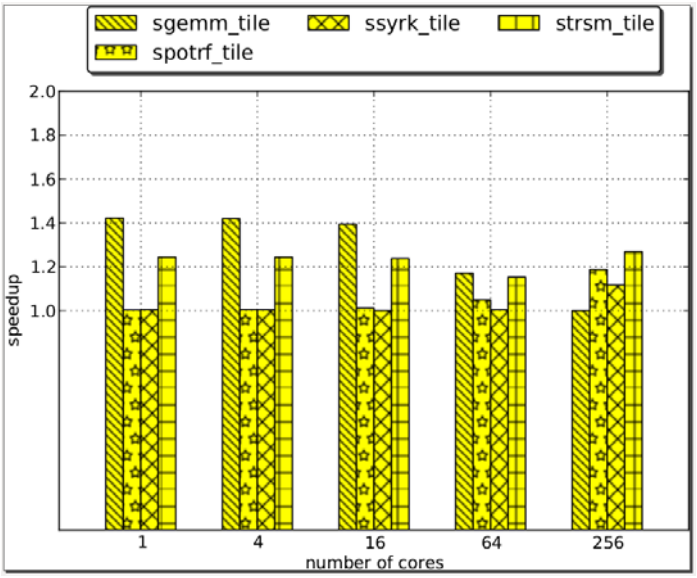


Figure 29: Speedup when one task is speeded up by 2× in the Cholesky application.

4.3.2 *European contributors*

In Europe there are two strong teams that are contributing to these different areas. They are briefly described below in Table 12.

Recommendation area	European institution/Country	Tool name	Tool description
Root cause	JÜLICH SUPERCOMPUTING CENTRE (JSC) / Germany	Scalasca [40]	A performance analysis toolset that has been specifically designed for use on large-scale systems supporting MPI and MPI/OpenMP
What-if models	Barcelona Supercomputing Center (BSC) / Spain	Mpisstrace [43]	A debugger for applications parallelized with the use of the OmpSs programming model
Power models	Barcelona Supercomputing Center (BSC) / Spain	POTRA [44]	A framework for Building Power Models For Next Generation Multicore Architectures

Table 12: European contributors for the recommendations on models.

4.4 Scalability

The most important aspect of any tool designed for exascale computing is scalability. Some current approaches like offline filtering and processing of traces will partially solve the problem in terms of analysing smaller regions. Storing large amounts of data in the order of terabytes (or even petabytes) (e.g. for the original/full trace) will be costly at exascale. Even if other approaches such as event based or data-size based tracing might be effective they are not completely beneficial because the traces get too big as well. We need a very fine detail of what is going on in the application and we are overwhelmed by huge amounts of data. Managing and storing efficiently this data is becoming critical for performance analysis.

There are some approaches that look promising to tackle the problem of scalability: Sampling techniques and intelligence based monitoring techniques can substantially increase the scalability of the tools as mentioned early in Section 4.2.

1. **In-memory or online processing:** It allows tools to identify the important events during the execution of the application, and hence reduces the overall data volume to store.
2. **Management data by re-structuring of traces:** There are several techniques such as partitioning traces into different files using OTF (Open Trace Format), segmenting traces and pre-computing general summaries.
3. **Parallelizing tools for visualizing large sets of trace data:** One approach in this direction allows the analysis of traces immediately after they run by using cores from the original CPU set.
4. **Automatic reduction/compression to manageable size:** Compression techniques can be applied to drastically reduce the size of the traces. Also Lossy compression techniques to further be applied to further reduce the size of traces.
5. **Multi-scale:** Many scientific applications are based on multi-scale modelling which calculates material properties or system behaviours on one level using information or models from different levels. On each level particular approaches are used for the description of a system. For example in physics there are level of quantum mechanical models (information about electrons is included), level of molecular dynamics models

(information about individual atoms is included), meso-scale or nano level (information about groups of atoms and molecules is included), level of continuum models and level of device models. In this context, performance tools might also follow this approach of multi-scaling by tracing and analyzing one level at a time rather than the whole program.

6. **Pixel-bar charts:** These charts allow for visualizing large amounts of multi-attribute data. The approach is a generalization of traditional bar charts and x-y diagrams, which avoids the problem of losing information by aggregation or over plotting large amounts of data points. The basic idea is to use the pixels within the bars to present the detailed information of the data records. Our so- called pixel bar charts retain the intuitiveness of traditional bar charts while allowing very large data sets to be visualized in an effective way.

4.4.1 *Recommendation evidences on existing tools*

Figure 30 illustrates using Vampir [42] a pixel bar chart for performance visualization of I/O events in a million-process run of some scientific application. As we can see, it is possible to easily have a global overview of size, duration, and bandwidth of the I/O operations for very large process counts (one million) overtime. Time corresponds to the X axis and processor is shown in the Y axis. For example, it is possible to see in the Size bar, that there are few processes among the million that send very large I/O files (red dots) and these ones are also causing a big delay as it is shown in the Duration bar (yellow and red dots).

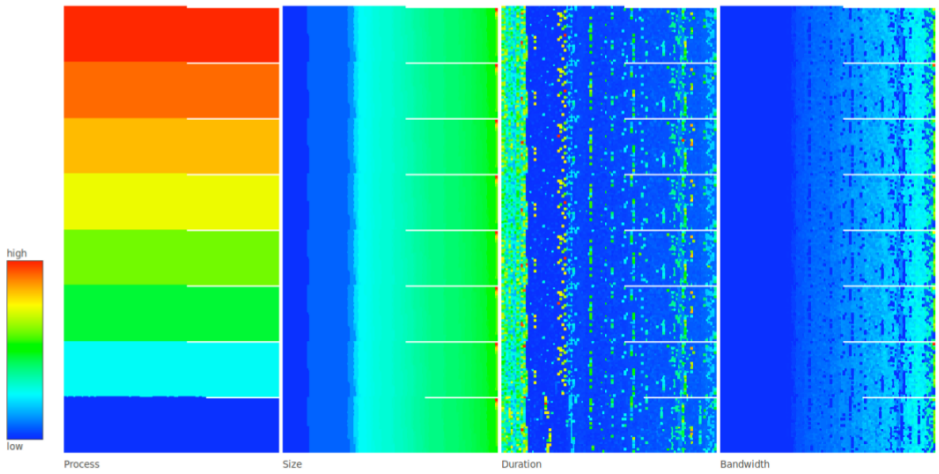


Figure 30: Pixel bar-charts for performance visualization of I/O events in a million-process run

Another interesting example to illustrate how existing tools are coping with scalability issue in analysing large data set is shown in Figure 31 and Figure 32. Figure 31 depicts the full trace from a 64-process run of the GROMACS application. It shows the various iterations and the communication operations performed during each iteration. The same behaviour of the application can also be observed in Figure 32 where it shows only 15% of the trace records from the full trace. This example demonstrates that there is no need to store the full trace because a small portion of it is enough to still see the full behaviour of the application. This is a technique that belongs to the smart approaches based on sampling techniques.

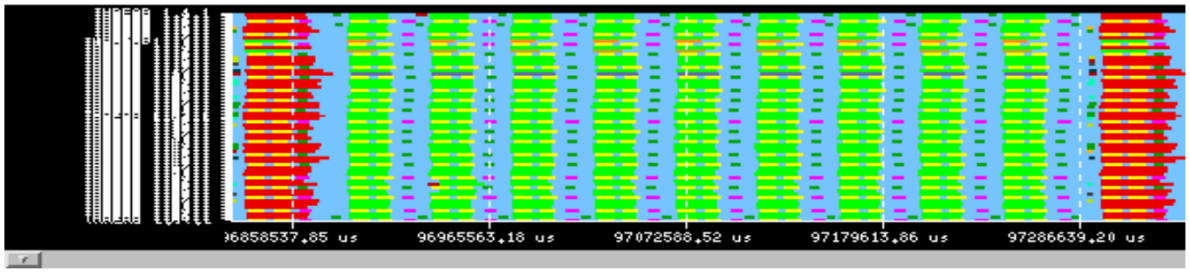


Figure 31: Original full 64-processes GROMACS trace

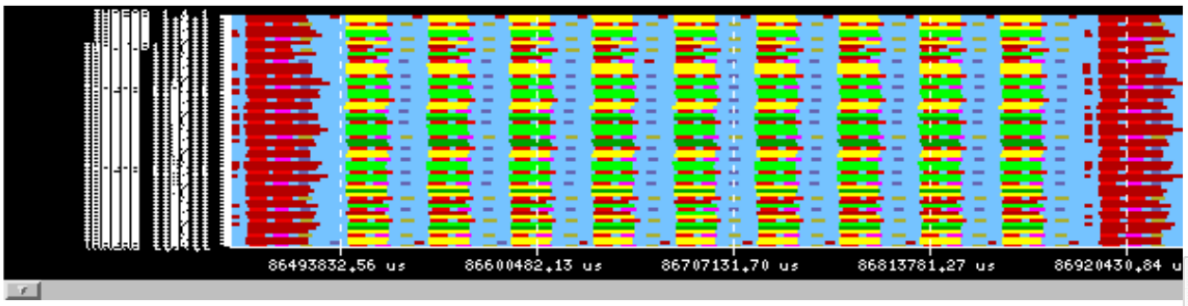


Figure 32: 15% of records from the 64-processes GROMACS trace

4.4.2 *European contributors*

In Europe there are two strong teams that are contributing to these different areas. They are briefly described below in Table 13.

Recommendation area	European institution/Country	Tool name	Tool description
Data management	JÜLICH SUPERCOMPUTING CENTRE (JSC) / Germany	Scalasca [40]	A performance analysis toolset that has been specifically designed for use on large-scale systems supporting MPI and MPI/OpenMP
Data management	Technische Universität Dresden/Germany	Vampir [42]	Vampir provides an easy to use analysis framework which enables developers to quickly display program behaviour at any level of detail
Online techniques	Barcelona Supercomputing Center (BSC) / Spain	EXTRAE [38]	A tracing tool to extract computation and communication events from applications

Table 13: European contributors for the recommendations on scalability.

4.5 Specific recommendations from PRACE prototypes

This section summarizes the specific technical recommendations from prototypes owners about the further development in HPC tools.

1. **Portability.** Portability of the software tools to support different hardware and programming models (e.g. NVIDIA/ATI, CUDA/OpenCL) will be very useful.
2. **GPU communication support**. Benchmarks measuring bandwidth and latency between GPUs are indispensable for the performance analysis of multi-GPU codes. Enable tools like the parallel ping-pong test program LinkTest to support GPU-GPU communication.
3. **Support for analysing communication-computation overlap**. Trace-based, timeline-style performance analysis tools can greatly enhance the ability of a programmer to identify opportunities for optimization of accelerator codes. The timeline-style view shows when host and device events occur; using this feature, the programmer can easily spot inefficiencies and look for opportunities for overlap. A particularly useful example of this technology is Vampir.
4. **Support for accelerator dataflow visualization**. Tools that help to visualize an accelerator application's host-device data flow characteristics would be useful. This is essential when optimizing for data locality in order to reduce costly host-device data transfers.
5. **Directive-based support**. Many performance analysis and debugging tools do not yet support directive-based accelerator programming models. This suggests that performance analysis and debugger tool vendors should support directive-based approaches.
6. **Proprietary tools needs more data analysis support**. Relying on basic NVIDIA tools turned out to be the only stable, efficient and highly scalable approach. Data analysis requires a bit of hand-coded post processing of textual output. Third party tools generally need deep testing prior adoption in a production/highly parallel/hybrid environment.
7. **Support for I/O performance counters**. Integrate monitoring and analysis of I/O performance counters into performance analysis tools. Application developers today have very limited options to retrieve performance data concerning the I/O sub-system. This becomes even more critical if complexity increases due to additional storage levels.
8. **Power analysis**. Future systems with more than 100,000 of nodes need to have ways for defining the level of detail for power measurements required for each job. For example a simple 3 level approach can be envisioned. Level 1 would store power data at the resolution rate of the measuring equipment, level 2 would store data in specific intervals (e.g. every 5 min) and level 3 would just store the Energy to Solution (energy consumed) for the job.
9. **Memory layout support.** Integrate/provide specialized performance counter (cf. hpmcount/IBM) in performance analysis tools. Provide extended compiler support to tailor memory layout onto the nodes (NUMA-aware compiler).

5 Hardware recommendations

Having looked at recent developments in programming languages for high performance computing, we turn our focus to hardware and the associated technologies being developed as we target Exascale systems. In this chapter we will present some of the results of the evaluation of the prototype hardware which we have carried out within this work package, and arrive at a number of recommendations this evaluation has led to.

5.1 Lessons learned and Recommendations

The prototypes evaluated under PRACE-1IP WP9 have proven to be an extremely valuable source of information. By experimenting on novel computer architectures, we have evaluated the potential of certain technologies to evolve into Exascale supercomputers. An important result of this work was to confirm current trends in academia and in industry and to provide quantitative results on such aspects as power efficiency and scalability, results which can be used to make explicit recommendations on which architectures have a potential to scale to Exascale.

The conclusion of this study is that there are a number of promising technologies, however no single architecture prevails. On the other hand, there are certain technologies that are optimal for certain classes of problems, depending on the specific work-load and computational needs (e.g., I/O intensive, memory intensive, floating point intensive, etc.). Our findings are presented the form of a set of tables, where the recommendations reflect the lessons learned from each prototype architecture.

5.1.1 *Accelerators*

Accelerators are currently subject to intense research. GPUs, DSPs or FPGAs demonstrate great potential in power efficiency (i.e. Watt per Flop ratio). They are seen as the most obvious path to the Exaflop at a reasonable power budget. The first section of this chapter therefore investigates accelerators, namely an FPGA (Table 14) cluster and three GPGPU clusters (Table 15, Table 16 and Table 17).

Lessons learned from the prototype	Recommendations
Our prototyping efforts of a matrix multiplication accelerator support the general wisdom that special-purpose accelerators are 2-3 orders of magnitudes more energy-efficient than today's general-purpose multicore processors.	To achieve the desired energy-efficiency for next-generation supercomputers, we recommend replacing assemblies of COTS general-purpose processors with low-power processors with attached accelerators, such as reconfigurable logic.
Our FPGA accelerator prototype shows that today's FPGA's are capable of delivering competitive double-precision floating-point performance with 2 orders of magnitude higher energy efficiency than general-purpose multicores. This assumes that the accelerator architecture achieves a high utilization of the special-purpose hardware of FPGA's, such as multiply-and-accumulate or SRAM arrays.	We recommend considering reconfigurable logic (FPGA's) as an energy-efficient accelerator platform for double-precision floating-point computations.
Accelerator design requires experienced	We recommend a concerted educational

Lessons learned from the prototype	Recommendations
designers with interdisciplinary skills in algorithms, computer architecture, hardware design, and EDA tools.	effort to focus on interdisciplinary engineering curricula on the skill set required for designers of future accelerator architectures.
Summary	
Our accelerator design and prototyping efforts corroborate the potential of accelerated computing for improving energy-efficiency by 2-3 orders of magnitude compared to general-purpose multicore processors. We have demonstrated competitive double-precision floating-point performance on reconfigurable logic (Xilinx FPGA's) with significant reductions in energy consumption. Thus, we believe that FPGA's have the potential to evolve into a competitive accelerator platform for future exascale supercomputer nodes.	

Table 14: Energy-to-solution prototype from from JKU.

The results of an investigation in GPUs for General Purpose computing (GPGPUs) are described in Table 15, Table 16 and Table 17. The prototype at CaSToRC shows that a substantial amount of work to adapt codes is required, while the prototype from CSCS emphasizes the necessary links between hardware and software at the system level. The CINECA prototype demonstrates the significance of handling hardware failures, which have a probability of occurrence proportional to the size of the machine.

Lessons learned from the prototype	Recommendations
Performance and energy efficiency of modern HPC clusters can be noticeably improved by the use of GPUs as accelerators.	Do further investigations on GPU computing to enlarge the field of application for hybrid clusters.
GPU communication in hybrid clusters is a crucial point and still poses a challenge.	Continue work on improvement of GPU communications, both intra and inter node, to mitigate possible bottlenecks
Approaches like GPUDirect and collaborations between vendors as with NVIDIA and Mellanox for improvement of GPU inter-node communications looks promising.	Implement a standard protocol that can be used to improve communications between different PCI devices.
Summary	
GPU communication in hybrid clusters still poses a challenge but approaches such as GPUDirect and collaborations between different vendors for improvement of GPU inter-node communications appear promising.	

Table 15: Interconnect Virtualisation prototype from CaSToRC.

Lessons learned from the prototype	Recommendations
Designing directly-connected topologies using InfiniBand 8-port (unmanaged) switches requires an in-depth understanding of the cabling from processor and between switches. This is not scalable to large scale installations using open source management, trouble-shooting and diagnostics interfaces.	In order to introduce cost effective and scalable interconnect topologies using commodity components, especially unmanaged switches, further research and investment is needed both at the fabrics level and also for the management and troubleshooting tools such as OpenSM.

Lessons learned from the prototype	Recommendations
Incorrect cabling from processors and between switches can severely impact communication throughput.	
Currently, PCIe peer-to-peer communication is not fully supported by many vendors. Hence, a true host bypass, which CSCS GPU virtualization prototype intended to evaluate, could only be implemented through a software layer to communicate between the GPU and the network interface. This is also a performance critical design component for clusters based on PCIe accelerators.	As accelerators become central to the node design, direct communication channels must be provided by vendors and should be supported by Linux kernels to allow for direct memory transfers between the accelerator memories and the network interface, without an intermediate copying step to the host memory. Research and development is needed for the Linux kernel extensions in collaboration with vendors and this could be a co-design opportunity for PRACE.
GPU to GPU direct memory transfers over PCIe is supported by NVIDIA drivers. Currently, this is limited to single IO hubs, therefore, we cannot develop a multi-GPU server for GPU virtualization using commodity components.	There could be different solutions: (1) a PCIe bus can be introduced that is extensible to multiple ports without a significant performance loss; (2) there could be PCIe chipsets that could provide extensible peer-to-peer interfaces; (3) a GPU/accelerator could be a standalone unit (self-hosted) and could initiate communications to other GPU devices and CPUs in the cluster.
Summary	
As accelerators become central to the node design, direct communication channels must be provided by vendors and should be supported by Linux kernels to allow for direct memory transfers between the accelerator memories and the network interface, without an intermediate copying step to the host memory.	

Table 16: Interconnect Virtualisation prototype from CSCS.

Lessons learned	Recommendations
Hybrid CPUs + GPUs architecture is affected by more HW failures and damages compared to uniform clusters. This is related to the large amount of HW components: more available devices increases the rate of failure events.	It is mandatory to provide a semi-automatic monitoring system which will alert the system administrators for possible failures.
CPU or GPU HW component failures could put off line an entire CPU-GPU host and vice versa. This kind of event may affect all jobs running on the host even if these jobs do not belong to the same user. Again, host failures may affect CPU and GPU jobs running on the same server.	A predictive failure tool could help prevent user job failures by, e.g. draining the host in advance.
Direct and indirect power consumption of	In order to optimize power consumption,

Lessons learned	Recommendations
hybrid CPU and GPU clusters is less than the combined consumption of 2 separate clusters. On the other hand, idle GPUs or CPUs still consume power.	fully hybrid CPU-GPU clusters may not be the optimal solution. During the planning stage it may be import to evaluate the needs of the users, and thus deploy clusters equipped with a non-uniform ratio of CPUs and GPUs.
Summary	
On a hybrid cluster it is crucial to deploy a tool for predicting and/or preventing hardware failures.	

Table 17: Interconnect Virtualisation prototype from CINECA.

5.1.2 *I/O*

As machines get larger, the possible simulation sizes increase accordingly, thus increasing requirements on storage and I/O bandwidth. Designing an efficient I/O subsystem for Exascale computing has evolved into a challenge itself. The prototype from FZJ casts light on the coming massive usage of flash technologies in the HPC world, detailed in Table 18.

Lessons learned from the prototype	Recommendations
NAND flash memory card technology becomes more mature and provides a real opportunity to mitigate the performance gap between volatile and non-volatile memory/storage access.	Promote the integration of flash memory card technologies into future HPC architectures.
Various challenges to integrate flash memory into a massively parallel HPC architecture and to enable its efficient use remain to be addressed.	Further explore the design space and foster the integration of additional storage levels into parallel file systems and the development of I/O middleware.
Summary	
NAND flash memory is the only technology which in the near future allows to significantly mitigate the I/O performance bottleneck. The integration of these technologies into massively parallel HPC systems will be an important step towards the design of an Exascale I/O subsystem.	

Table 18: Novel MPP Exascale system I/O prototype from FZJ.

The second I/O prototype is installed at CEA (in collaboration with BSC, CINES, Daresbury, FZJ and HLRS), and focuses on the evolution of the storage system (Table 19). The goal is to study how to efficiently store the Petabytes of data that will be produced by an Exascale machine.

Lessons learned from the prototype	Recommendations
The reduced overall hardware components involved in Xyratex's Lustre storage solution (ClusterStor 3000), with embedded server modules, allow for a faster deployment than with a solution based on I/O nodes plus standard DAS or SAN disk arrays. It also	Foster the use of embedded servers in storage systems for Exascale I/O.

Lessons learned from the prototype	Recommendations
shows good performance per spindle and eases the whole system maintenance.	
Software RAID using Linux device-mappers can be used to build embedded high-end storage systems. Expected system performance was reached, however, some system failures were encountered that may be related to the RAID stack.	Do not discourage using new RAID engines for disk arrays, like device-mapper based RAID. On the other hand, more work and research have to be invested in terms of software RAID resiliency.
Summary	
The use of embedded server modules in a high-performance Lustre storage system has demonstrated competitive performance, energy consumption and manageability. Resiliency of Linux software RAID needs to be investigated and improved in the long run.	

Table 19: The Exascale I/O prototype from CEA et al.

5.1.3 *Energy efficiency*

A general awareness on energy preservation, the ever increasing price of energy and the impact of HPC on the global environment have encouraged us to design new computing centres (infrastructure) as well as new types of nodes (hardware architecture). The Energy-to-Solution prototype at LRZ (Table 20) provides insights on energy efficiency as we scale towards Exaflops.Table 21, which presents the results of the BSC prototype, shows how high density nodes can be made of low power, commodity components such as those available in the embedded world or in the phone/tablet market.

Lessons learned from the prototype	Recommendations
Increased leakage currents at higher water temperatures and relative lower efficiency of current adsorption machines reduce the benefits of hot water reuse through adsorption refrigeration.	In order to benefit from reusing hot water for cooling, components with low leakage currents should be used and the efficiency of adsorption refrigeration machines should be improved.
A water cooled systems and compressors that produce cool air using the system water cooling loop work well to create a room neutral rack system. But the compressor power consumption at higher water temperatures negate possible benefits of adsorption when compared with free cooling.	Cool all components with water so that the compressors can be removed from the system. This should allow future HPC centres to take advantage of possible hot water recycling via adsorption.
A pump failure in the infrastructure water loop running through the HPC system water heat exchanger was not detected because the temperature sensor on the infrastructure side before the heat exchanger showed correctly normal temperature but the HPC system water temperature increased till an emergency shutdown occurred.	It is important to not just monitor the system infrastructure but also to monitor the building infrastructure correctly. It is also important to think about all possible failure scenarios and to put appropriate sensors in the right places and integrate them with the system monitoring system. Cooling loops for instance need not just temperature sensors but flow sensors as well.

Lessons learned from the prototype	Recommendations
Disassembling and moving a water cooled rack system is currently not easy because all water pipes are soldered.	Having a connector based system for the water pipes between racks would make assembly, disassembly and extension much easier.
The 1 minute read interval for some power sensors is not good enough for detailed power analysis of jobs. It is usable for Energy to Solution (EtS) measurements for jobs typically running longer than an hour.	Future systems might need a partition of nodes that are equipped with fine grain power measuring equipment for detailed power analysis.
The water cooled prototype in combination with free cooling shows a PUE of less than 1.2. This is much better than traditional air cooled systems (> 1.4).	Water cooling in combination with free cooling should become the standard cooling technology for future generation HPC systems.
Summary	
Future systems should use direct water cooling of all components. Energy conservation and recovery options should be considered including adsorption, free cooling and hot water reuse.	

Table 20: Energy-to-solution prototype from LRZ.

Lessons learned from the prototype	Recommendations
Per SoC compute density is not enough to dominate total node and blade power consumption.	Increase compute density by replacing SoC with one which has more cores (e.g. Tegra 3).
Compute node (and SoC as well) has unnecessary components which consume power but are not used for computation.	The node that we are using is designed for embedded development, and has many components that are not needed in HPC. One solution is to remove all unnecessary electronics like the RTC (Real Time Clock), USBs, HDMIs etc. If possible, consult SoC maker for stripped down version of SoC.
Blade power supply has huge intrinsic losses.	Size and/or design power supply properly so that the expected consumption is on the point of best power supply efficiency.
Summary	
It is important to increase compute density, such that the power consumption of the processor is a significant portion of the power consumption of the node	

Table 21: Energy-to-solution prototype from BSC.

5.1.4 *Interconnects*

Not all algorithms can be easily decomposed into MPI tasks. For such codes, a finer grain parallelism offers the possibility to increase performance. Large NUMA nodes have a significant advantage for such classes of codes. The prototype of UiO demonstrates this potential, as is described in Table 22.

Lessons learned from the prototype	Recommendations
Shared memory architectures promote the use of parallel computers for "not yet parallel" application codes via a) a simple global memory model b) simple and fast "one-sided" memory access (copy) c) small latencies compared to distributed memory systems. The number of synchronization points and buffer memory in the FMM can be reduced compared to distributed memory systems through direct access.	Further explore and adapt the NUMA design to increase the number of cores per node.
Summary	
NUMA architectures bridge the gap between non-parallel applications and HPC codes via a simple memory model. NUMA nodes, as part of a larger distributed system, will be an important step to designing Exascale machines. NUMA nodes allow dynamic and effective workload balancing for non-static workload problems (e.g. MD).	

Table 22: NUMA-CIC prototype from UiO.

5.2 Summary

We provide the following summary of the recommendations in this chapter, concerning the path towards Exascale supercomputers:

- **Acceleration:** Accelerated nodes will most likely form a significant component of an Exascale system, however further experiments are required to identify whether this acceleration will be based on GPU, FPGA, DSP or other emerging architectures. An important effort must be made for improving communication as well as developing tools or methods to track (and possibly correct) hardware failures.
- **Compactness:** The density of nodes must increase yet with no significant penalty on power consumption. The impact of ARM based architectures must be studied as soon as they will be more widely available.
- **Maintain a partition of "fat" nodes:** As some workloads cannot be scaled to highly parallel nodes, a proportion of large NUMA nodes, with a high core count, should be present in an Exascale machine, to accommodate all classes of applications.
- **Efficient I/O subsystem.** Performing I/O at scale should be taken into account when designing an Exascale machine. Flash based technology has been shown to be a path to fast, scalable and resilient nodes. Embedded server modules will help build a manageable and efficient storage system without scarifying energy efficiency.
- **Energy efficiency:** Node design (and accordingly, the computer centre architecture) must evolve towards a PUE of less than 1.2. Watercooling is the most promising technology as of today.

6 Conclusions

So far, international efforts have highlighted that a number of solutions in power efficiency, interconnect scaling and I/O, among others, are required to meet the Exascale milestone. The current document reports on results and outcomes of an investigation into a rich set of novel computer architectures, with the intention of contributing to this global effort in reaching the level of innovation required for Exascale computing. The work detailed here has looked into programming languages designed for HPC, system software, tools for analyzing and monitoring parallel applications as well as certain aspects of hardware design. Here we summarize with an overview of the conclusions and recommendations presented in this document.

For the case of programming languages, we have looked more closely into accelerators, and more precisely GPUs. We conclude that certain pragma-based languages, such as OpenACC and OmpSs show promising results, however are still rather immature and not suitable for production-level software. When investigating OpenCL and CUDA, we find that although the former has reached a certain level of maturity, positive developments made by NVIDIA in their hardware prove difficult to accommodate in OpenCL. This means that developers of GPU applications are more likely to persist on using CUDA for their production level codes. In terms of multi-GPU programming, we found that the combination of MPI+CUDA allows access to the latest technological innovations in GPU/CPU decoupling.

The work carried out in system software has highlighted the importance of co-design in the Exascale era. More precisely, regarding fault tolerance, we have identified that in current systems this is achieved by combining all the more reliable hardware with efficient checkpoint and restart mechanisms. For Exascale, however, this may prove unrealistic. It is therefore crucial that system software be made resilient to hardware faults. Another important outcome of this work has been the identification of the need for more hardware-aware system software, namely system software which can adapt according to, e.g. power consumption and which is aware of the locality of the compute nodes and the interconnect topology.

This work has also identified a number of requirements for software development tools, i.e. applications that assist the development of software, for next generation supercomputer architectures. Firstly, it is crucial that tools be developed for accelerated architectures, and namely tools which can trace the movement of data between GPUs and between GPU and main memory. Additionally, we found that tools which can report on I/O performance and which register I/O counters are lacking or insufficient, something especially important given that I/O is expected to be a major bottleneck in need of significant innovation on the path towards Exaflop. However, more generally, we have identified a need for innovation in the reporting and filtering of traces and debugging information. Major steps are required in the presentation of hardware counters and program flows from millions of cores for these to be useful to the programmer.

Finally, we looked at certain aspects of hardware and their suitability within Exascale. Apart from the need for more dense computational units, such as accelerators, we found that there is much room for improvement in terms of I/O and energy efficiency. Flash based storage devices have proven to be most effective in this area, while we have had positive results with the warm water cooled prototype.

This document has reported on the status of the research achievements of a pan-European collaboration of computer centers in the area of HPC innovation. Driven by the particular interests of each individual center, and simultaneously by the common European goal of maintaining competitiveness as we approach the Exascal milestone, our intention is to compose a valuable resource, partly served by this deliverable, for the relevant HPC

stakeholders in Europe. We believe that the result of this ongoing collaboration has been successful in identifying and evaluating promising computer architectures as part of Europe's efforts towards Exascale computing. The results presented here will guide further investigations into novel computer architectures, which is an ongoing activity in PRACE.

7 Annex

7.1 Energy Aware System Software

In the next pages, a write-up detailing the results on “Energy Aware System Software” follows, as outlined in Subsection 3.2.1 of the main text

ENERGY AWARE SYSTEM SOFTWARE

With energy efficiency being one of the major problems to overcome in the Exascale challenge, we predict that future HPC system software needs to become energy aware. Energy awareness at the system software level encompasses two aspects. First, the power consumption of the system needs to be monitored in order to measure the energy required to run a given application (i.e. Energy-to-Solution). Second, the operational parameters influencing the power and performance characteristics of the system need to be tuned to improve the Energy-to-Solution.

POWER MONITORING AND ENERGY-TO-SOLUTION

In the CooLMUC prototype at BADW-LRZ, we have implemented an Energy-to-Solution system that is sketched in Figure1. CooLMUC uses smart power distribution units (PDUs) to monitor the power consumption of every compute node in the cluster. In addition, the power consumption of the cooling equipment is monitored using a digital three-phase current and voltage meter. The obtained power readings are forwarded via Ethernet to a virtual machine where the values are stored in a database. In addition, the database also holds the resource manager's accounting data.

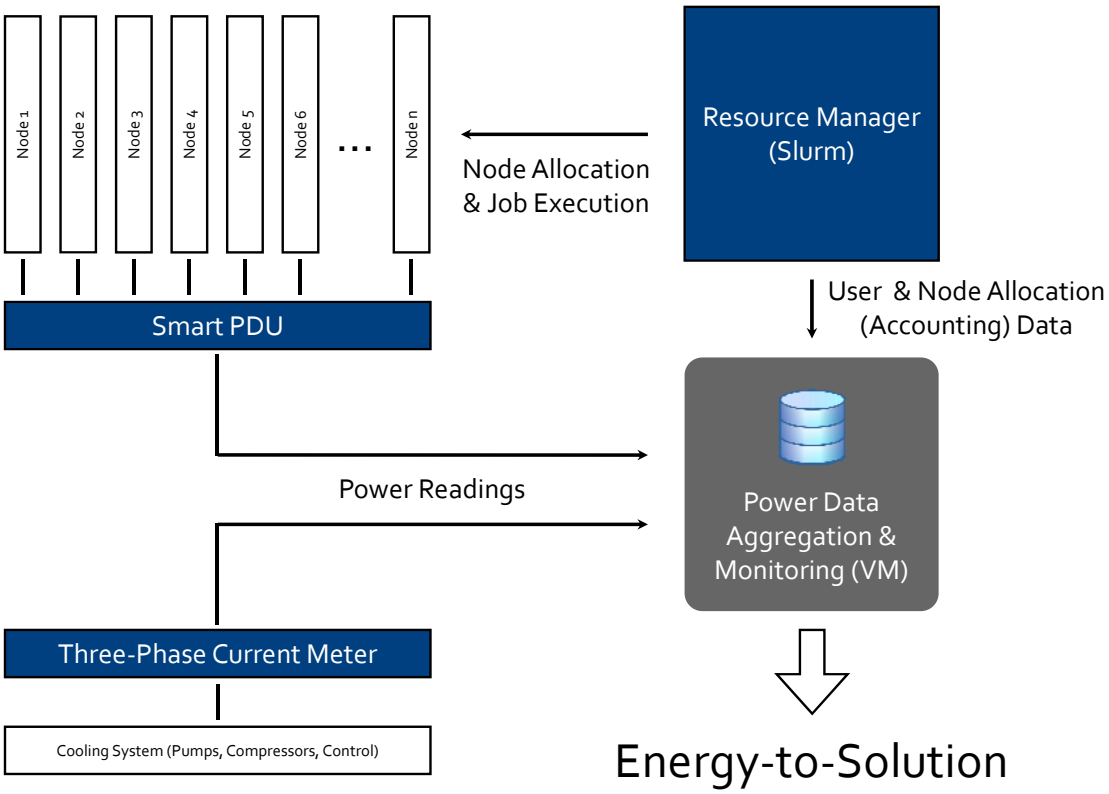


Figure1. Energy-to-Solution setup of the CooLMUC PRACE Prototype

To calculate the Energy-to-Solution for a given application, the accounting data is used to determine when and on which of the compute nodes the application had been run. Then, the power consumption of these nodes during the runtime of the application can be obtained from the database as a series of power readings over time. Since the resource manager ensures that only one application can use a node at a given time, the entire power consumption of the node can be attributed to the application.

In order to include the power for the cooling system into the Energy-to-Solution metric, we assume that all energy consumed by the compute nodes has to be cooled due to the principle of energy conservation. Yet, in our prototype setup, we can only measure the power consumption of the entire cooling system at once. Thus, we attribute the total power consumption of the cooling system to the individual nodes according to their own power consumption as measured in the smart PDUs.

By integrating the power readings of all nodes on which an application was run and their respective fraction of the cooling power, we obtain the Energy-to-Solution which is reported in kWh back to the user.

TUNING THE ENERGY-TO-SOLUTION USING SYSTEM CONFIGURATION PARAMETERS

Several control knobs exist to tune the performance and power characteristics of HPC systems, such as:

- Power gating (shutting down of unused parts of a chip, such as entire CPU cores)
- Dynamic voltage and frequency scaling (including technologies like Intel TurboMode)
- Enabling or disabling of HyperThreading
- Mapping configuration of threads to cores (pinning)

Making use of these knobs can help to fine-tune the system for improved energy efficiency. However, it is important to note that each HPC application has a characteristic profile. Therefore it is necessary to find the best settings to optimize the Energy-to-Solution for every application individually.

We have chosen to investigate the impact of dynamic voltage and frequency scaling on the Energy-to-Solution. For this, we have extended the Slurm resource manager with the ability to set fixed CPU frequencies for given applications through the existing Prologue/Epilogue mechanisms. In theory, applications that are memory or interconnect network bound can be run at lower CPU clock rates without sacrificing performance. Since lower clock rates result in lower power being consumed the Energy-to-Solution will improve whenever the lower clock rates do not significantly improve the application runtimes.

For our analysis, we have selected the APEX MAP benchmark which performs memory operations according to a selectable pattern. We compare a randomly distributed access pattern to a strided pattern which is comparable to the STREAM benchmark. The runs are monitored using our Energy-to-Solution setup described in the previous section.

Figure2 and Figure3 show the power profiles of the benchmark runs for the random case and the strided case, respectively. As expected, the power consumption is higher at higher processor frequencies. Since the benchmark workload is the same across the frequencies, higher processor frequencies also cause shorter application runtimes. Yet, in case of the random memory access pattern (Figure2), we observe that the performance increase at higher frequencies cannot justify the increase in power, opposite to the strided case(Figure3). Thus, the Energy-to-Solution is best at the highest frequency in the strided case and it is best at 1400 MHz for the random case (Figure4).

Hence, we have shown that adjusting system parameters can optimize the energy consumption of scientific applications.

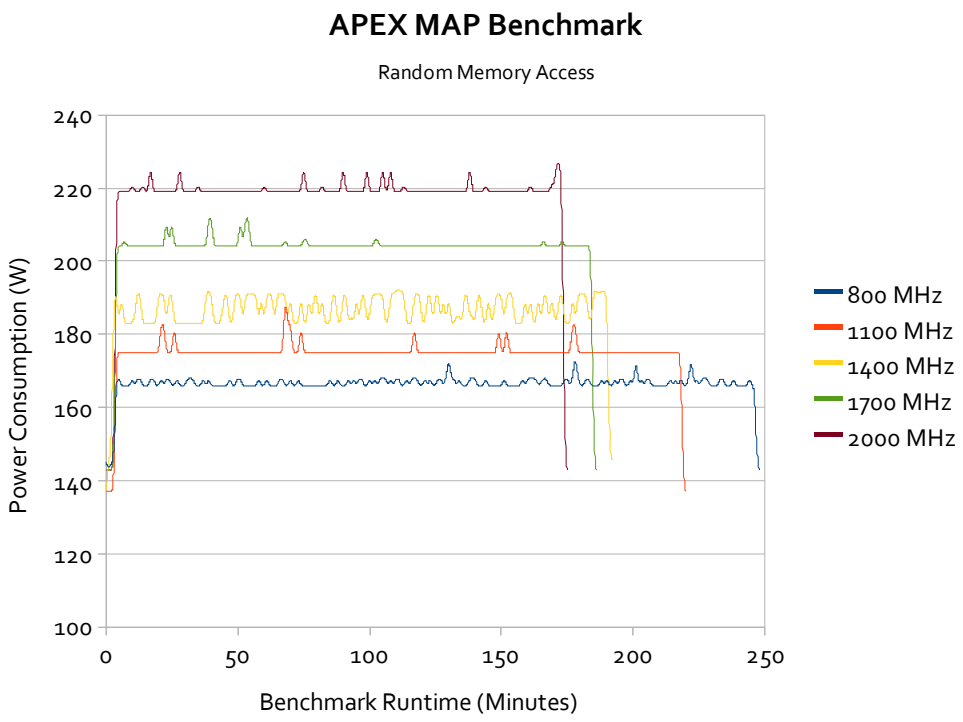


Figure2. Power profiles of the APEX MAP benchmark on CooLMUC (Random Memory Access) at different CPU frequencies

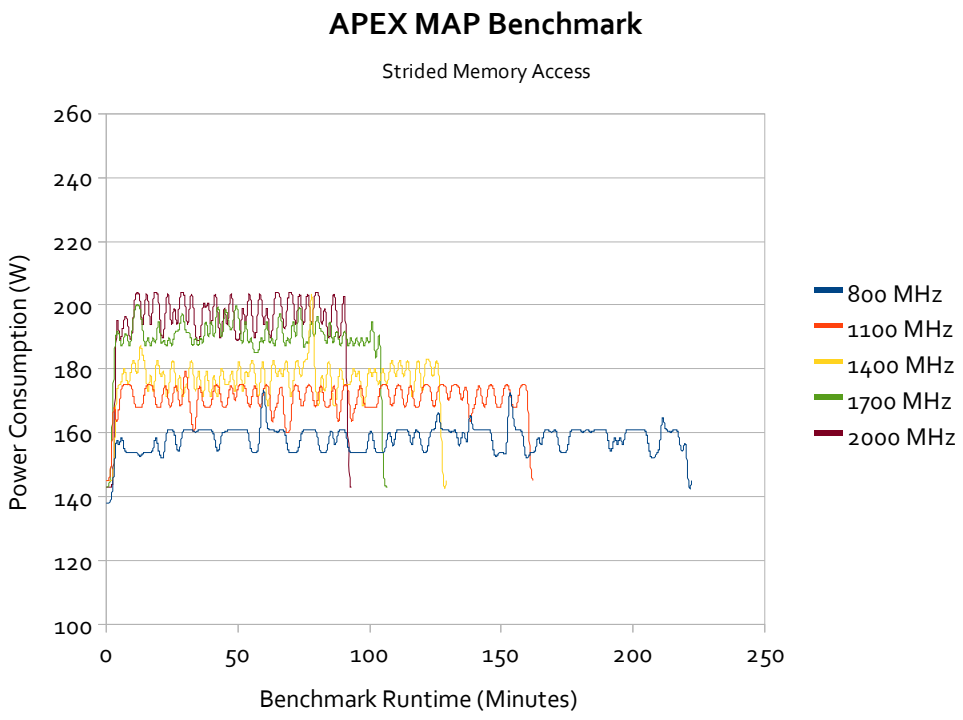


Figure3. Power profiles of the APEX MAP benchmark on CooLMUC (Strided Memory Access) at differernt CPU frequencies

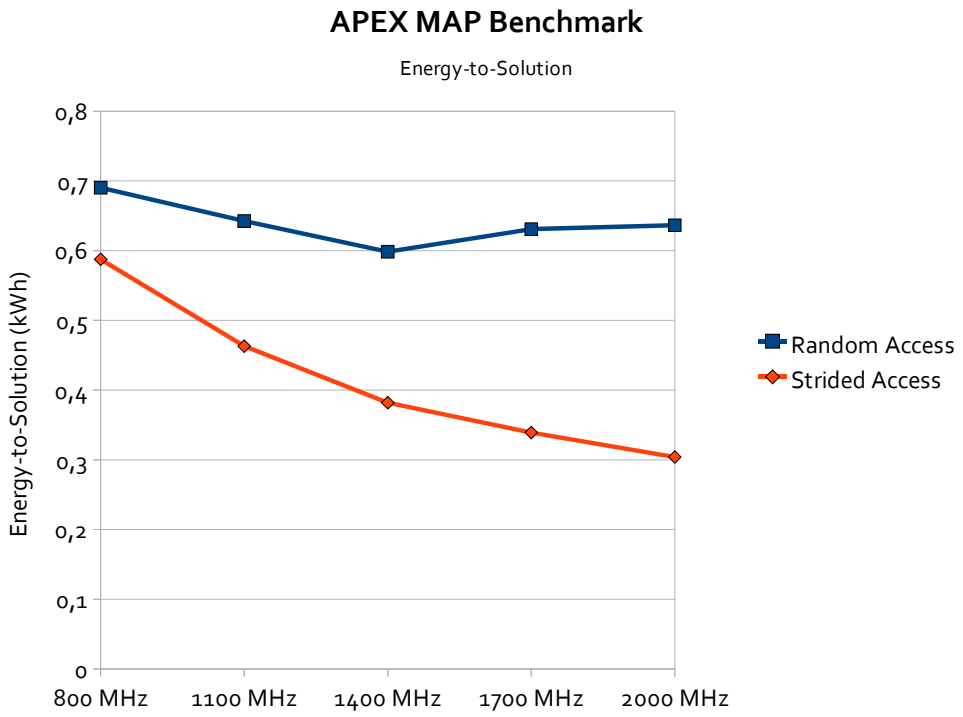


Figure4. Energy-to-Solution of APEX MAP at different frequency levels

Although our implementation through Prologue and Epilogue scripts yields the desired settings, it requires manual work by both, the system administrator and the user. Future Exascale-ready resource managers should include the ability to adjust the above mentioned control knobs by default and further work should be done on automating the process of tuning the knobs for the best energy efficiency on the resource manager level.