SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

INFRA-2007-2.2.2.1 - Preparatory phase for 'Computer and Data Treatment' research infrastructures in the 2006 ESFRI Roadmap

PRACE

Partnership for Advanced Computing in Europe

Grant Agreement Number: RI-211528

D6.6
Report on petascale software libraries and programming models

FINAL

Version: 1.0
Author(s): Giovanni Erbacci (CINECA)
Carlo Cavazzoni (CINECA)
Filippo Spiga (CINECA)
Iris Christadler (LRZ)
Date: 26/10/2009

Project and Deliverable Information Sheet

PRACE Project	Project Ref. №: **RI-211528**	
	Project Title: **Partnership for Advanced Computing in Europe**	
	Project Web Site: http://www.prace-project.eu/	
	Deliverable ID: D6.6	
	Deliverable Nature: Report	
	Deliverable Level: PU	Contractual Date of Delivery: 31 / 10 / 2009
		Actual Date of Delivery: 30 / 10 / 2009
	EC Project Officer: Maria Ramalho-Natario	

* - 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

Document	Title: Report on petascale software libraries and programming models	
	ID: D6.6	
	Version: 1.0	Status: Final
	Available at: http://www.prace-project.eu/	
	Software Tool: Microsoft Word 2003	
	File(s): D6.6.doc	
Authorship	Written by:	Giovanni Erbacci (CINECA), Carlo Cavazzoni (CINECA), Filippo Spiga (CINECA), Iris Christadler (LRZ)
	Contributors:	Reinhold Bader (LRZ), Mark Bull (EPCC), Guillaume Colin de Verdiere (GENCI), Maciej Cytowski (PSNC), Montse Farreras (BSC), Maciej Filocha (PSNC), Georgios Goumas (GRNET), Jose Gracia (HLRS), Hans Hacker (BAdW-LRZ), Pierre-Francois Lavalleee (IDRIS), Olli-Pekka Letho (CSC), Walter Lioen (SARA), James Perry (EPSRC), Sami Saarinen (CSC), Ramnath Sai Sagar (BSC), Tim Stitt (CSCS), Aad Van der Steen (NCF), Volker Weinberg (BADW-LRZ)
	Reviewed by:	Thomas Eickermann (FZJ), Stefan Wesner (HLRS)
	Approved by:	Technical Board

Document Status Sheet

Version	Date	Status	Comments
0.1	26/March/2009	Draft	Initial Draft
0.2	27/July/2009	Draft	Chapters 3-5 improved
0.3	15/September/2009	Draft	Chapter 6 and 8 improved
0.4	5/October/2009	Draft	Included Chapter 7 and Appendixes
0.5	12/October/2009	Final Draft	For PRACE QA
1.0	26/October/2009	Final version	For EC

Document Keywords and Abstract

Keywords:	PRACE, HPC, Research Infrastructure, Petascale, Software libraries, Parallel Programming models and languages.
Abstract:	This deliverable identifies and analyses the programming models and the software libraries required by petascaling applications in the PRACE implementation phase. The work starts from an analysis of the applications mainly identified in D6.1 and D6.2.2, covering a broad range of scientific areas, and representative of the European HPC usage, to assess the state of the art with particular attention to programming languages, programming models and scientific libraries. Starting from this basis an accurate survey and analysis of the new upcoming programming models and languages suitable for petascale applications is presented, trying to identify gaps and opportunities provided in terms of performance and efficiency and benefit for parallelism. Furthermore, three numerical kernels, typical of the most important computational applications, have been selected, and coded, using twelve of the different programming languages and paradigms under investigation. This coding experience and the consequent porting and assessment on the prototypes selected by WP8, greatly contributed to better complete the analysis objective of this Deliverable.

Table of Contents

List of Figures

List of Tables

References and Applicable Documents

[1] PRACE Deliverable D 6.1, *Identification and categorization of applications and initial benchmark suite* (June 2008)
[2] PRACE Deliverable D 6.2.2, *Final report of Application requirements* (September 2008)
[3] Asanovic et. al., *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183 (2006)
[4] Top 500 Supercomputer Sites, http://www.top500.org
[5] The Message Passing Interface standard, http://www.mcs.anl.gov/research/projects/mpi/
[6] MPI 2.2, https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MpiTwoTwoWikiPage
[7] MPI 3.0, http://meetings.mpi-forum.org/MPI_3.0_main_page.php
[8] D. Bonachea, *GASNet Specification*, U.C. Berkeley Tech Report CSD-02-1207 (2002)
[9] D. Bonachea and J. Jeong, *GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages*, CS258 Parallel Computer Architecture Project (2002)
[10] W. Carlson et. al., *UPC: Distributed Shared Memory Programming,* Book of Wiley Inter-Science (2005)
[11] IBM UPC Compiler, http://www.alphaworks.ibm.com/tech/upccompiler
[12] GCC UPC, http://www.intrepid.com/upc.html
[13] Berkeley UPC, http://upc.lbl.gov
[14] Michigan Tech MuPC, http://www.upc.mtu.edu
[15] R. W. Numrich, *F--, A parallel extension to Cray Fortran*, Scientific Programming 6 (1997)
[16] R. W. Numrich and J. Reid, *Co-Array Fortran for Parallel Programming*, ACM SIGPLAN Fortran Forum 17 (1998)
[17] G95, http://www.g95.org
[18] J. Mellor-Crumney et. al., *A critique of Co-Array Features in Fortran 2008* (http://www.j3-Fortran.org/doc/meeting/183/08-126.pdf)
[19] K. Yelick et. al., *Titanium: A High-Performance Java Dialect*, In ACM (1998)
[20] P. N. Hilfinger, *Titanium Language Reference Manual*, technical report of University of California at Berkeley (2001)
[21] HPJava, http://www.hpjava.org
[22] Titanium, http://titanium.cs.berkeley.edu
[23] Chapel Language Specification v0.78, http://chapel.cray.com/spec-0.750.pdf (2008)
[24] B.L. Chamberlain et. al., *Parallel Programmability and the Chapel Language*, International Journal of High Performance Computing Applications 21 (2007)
[25] J. Dongarra et. al., *DARPA's HPCS Program: History, Models, Tools, Languages*, Advances in Computers (2008)
[26] P. Charles et. al., *X10: an object-oriented approach to non-uniform cluster computing*, OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (2005)
[27] X10 language, http://x10-lang.org
[28] Fortress project, http://research.sun.com/projects/plrg/
[29] Fortress project, http://projectfortress.sun.com/Projects/Community/
[30] Intel STM compiler, http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/
[31] TL2, http://stamp.stanford.edu/
[32] RSTM, http://www.cs.rochester.edu/research/synchronization/rstm/
[33] OpenTM Transactional API, http://opentm.stanford.edu/

[34] M. Wagner, *Grafikprozessoren als Hardwarebeschleuniger*, Master Thesis @ TU-Dresden (2008)
[35] CELL, http://www.ibm.com/developerworks/power/cell
[36] NVIDIA CUDA, http://www.NVIDIA.com/object/cuda_home.htm
[37] T. Halfhill, *Parallel Processing with CUDA*, Microprocessor Journal (2008)
[38] Portland Group, http://www.pgroup.com/resources/accel.htm
[39] Khronos Group, http://www.khronos.org/opencl
[40] OpenCL 1.0 Specification, http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pdf (updated May 16, 2009).
[41] R. Dolbeau et. al., *HMPP: A Hybrid Multi-core Parallel Programming Environment*, In the proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (2007).
[42] CAPS enterprise, http://www.caps-entreprise.com
[43] RapidMind, http://www.rapidmind.com
[44] Sh project, http://libsh.org
[45] M. McCool et. al., *Performance Evaluation of GPUs Using the RapidMind Development Platform,* SC06
[46] P. Bellens et. al., *CellSs: a Programming Model for the Cell BE Architecture*, ACM/IEEE Conference on Supercomputing (2006)
[47] J. P. Perez, *CellSs: making it easier to program the cell broadband engine processor*, IBM Journal of Research and Development 51 (2007)
[48] ClearSpeed, http://www.clearspeed.com
[49] ClearSpeed CSX700, http://www.clearspeed.com/products/documents/CSX700_Product_Brief.pdf
[50] S. Squires et. al., *Software Productivity Research in High Performance Computing*, in "CTWatch Quaterly – High Productivity Computing Systems and the Path Towards Usable Petascale Computing Part A: User Productivity Challenges"
[51] IDRIS *babel* and *vargas* hardware, http://www.idris.fr/eng/Resources/index-babel.html and http://www.idris.fr/eng/Resources/index-vargas.html
[52] DEISA benchmark, http://www.deisa.eu/science/benchmarking
[53] IOR benchmark, http://sourceforge.net/projects/ior-sio/
[54] PRACE Deliverable D 6.3.1, *Report on available Performance Analysis and Benchmark Tools, Representative Benchmark.*
[55] K. Kennedy et. al., *Defining and Measuring the Productivity of Programming Languages*, International Journal of High Performance Computing Applications 18 (2004)
[56] CLOC, http://cloc.sourceforge.net
[57] S. H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley Professional (2005)
[58] S. P. VanderWiel, *Complexity and Performance in Parallel Programming Languages*, International Workshop on High-Level Programming Models and Supportive Environments (1997)

List of Acronyms and Abbreviations

ACML	AMD Core Math Library
ALF	Accelerated Library Framework
AHTP	Advanced HPC Technology Platform. To be created in this project as permanent groups to identify and work on future technologies for multi-Petaflop/s systems
AMPI	Adaptive MPI
ANSI	American National Standards Institute
APGAS	Asynchronous Partitioned Global Address Space
API	Application Programming Interface
BLACS	Basic Linear Algebra Communication Subprograms
BLAS	Basic Linear Algebra Subprograms
BSCW	Basic Support for Cooperative Work, a collaborative workspace software package
CAF	Coarray Fortran
CFD	Computational Fluid Dynamics
CUDA	Compute Unified Device Architecture
DARPA	Defense Advanced Research Projects Agency, an agency of the U.S. Department of Defence responsible for the development of new technology for use by the military.
DEISA	Distributed European Infrastructure for Supercomputing Applications, a EU project by leading national HPC centres
DMA	Direct Memory Access
DP	Double Precision
ECC	Error Correcting Code
EIB	Element Interconnect Bus
ESSL	Engineering and Scientific Subroutine Library
Flop	Floating-point operation (1 Mflop/s = 10^6 Flop/s)
FFT	Fast Fourier Transform
FPGA	Field-Programmable Gate Array
General Partner	All PRACE partners, excluding the Principal Partners
GPFS	General Parallel File System (IBM)
GPGPU	General-Purpose Graphic Processing Unit
GPU	Graphics Processing Unit
HCE	Harwest Compiling Environment
HLL	High Level Library
HPC	High Performance Computing
HPCS	High Productivity Computing Systems, a DARPA project
HTM	Hardware Transactional Memory
HWA	Hardware Accelerator

IEEE	Institute of Electrical and Electronics Engineers
IEEE754	IEEE Standard for Floating-Point Arithmetic
ISC	International Supercomputing Conference; European equivalent to the US based Supercomputing conference. Held annually in Germany
LAPACK	Linear Algebra PACKage
LEF	LINPACK equivalent flops. This is a measure of how many "cycles" are used for an application, scientific area or other usage measurement where it is spread across different systems. It is essentially utilisation percentage multiplied by the R_{max} of a system and allows summation across systems
LCQD	Lattice Quantum Chromo-Dynamics
MASSV	Mathematics Acceleration Subsystem for Vectors
MD	Molecular Dynamics
MIMD	Multiple Instruction Multiple Data
MKL	Math Kernel Library
MoU	Memorandum of Understanding
MPI	Message Passing Interface
MPMD	Multiple Program Multiple Data
MTAP	Multi-Threaded Array Processors; computing element on ClearSpeed accelerator cards
NDA	Non-Disclosure Agreement
NetCDF	Network Common Data Form
OpenMP	Open Multi-Processing. An API for shared-memory parallel programming
PDE	Partial Differential Equation
PPE	Power Processing Element
PRACE	Partnership for Advanced Computing in Europe; project acronym
PP	Principal Partner. The five PRACE partners that have expressed interest in hosting a future Petaflop/s system
PVM	Parallel Virtual Machine
QCD	Quantum Chromo-Dynamics
RIDS	Reference Input Data Set
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SP	Single Precision
SPE	Synergistic Processing Element
SPMD	Single Process Multiple Data or Single Program Multiple Data
SSE	Streaming SIMD Extensions
SSE2	Streaming SIMD Extensions 2
STM	Software Transactional Memory
TCC	Transactional Coherence and Consistency

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
TM	Transactional Memory
UPC	Unified Parallel C
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VMX	Vector Multimedia eXtension
WP	Work Package
WP5	PRACE Work Package 5, *Deployment of prototype systems*
WP6	PRACE Work Package 6, *Software enabling for Petaflop/s systems*
WP7	PRACE Work Package 7, *Petaflop/s Systems for 2009/2010*
WP8	PRACE Work Package 8, *Future Petaflop/s computer technologies beyond 2010*

Executive Summary

The objective of WP6 Task 6 *"Software libraries and programming models"* is to understand and evaluate soundness, suitability and efficiency of new programming models, languages and libraries for petascaling applications in the PRACE implementation phase.

The work starts with an analysis of the applications mainly selected in D6.1 and D6.2.2, to better identify the state of the art in the field of programming languages, parallelization schema, programming models and software libraries, adopted by applications on present architectures. From this basis, an accurate survey and analysis of the new upcoming programming models and languages suitable for petascale applications is presented, trying to identify gaps and opportunities provided in terms of performance and efficiency and benefits for parallelism. The survey analyses the current standards for parallel programming and their evolution, the PGAS languages (UPC, CAF, Titanium), the next generation programming languages and models (mainly the languages introduced as a consequence of the DARPA HPCS project: Chapel, X10, Fortress) and then the languages, paradigms and environments for hardware accelerators (Cell programming, CUDA, OpenCL, CAPS HMPP, RapidMind, CellSs and Cn). The assessment of libraries for the petascale systems is presented, ranging from the basic libraries (for basic numerical computations) to I/O libraries, to communication libraries, to special purpose libraries (in charge for tasks not directly related to numerical kernels). Furthermore, to allow a comparative evaluation of new programming paradigms and languages and their soundness for petascaling, a specific experimental activity has been carried out: Three main numerical kernels, have been selected, and ported to twelve of the main programming languages and paradigms under investigation. This work, although qualitative, allowed to assess (for the first time, in a so ample manner) the maturity, the effectiveness, the soundness, and the programming effort of the different new languages and paradigms analyzed. The main findings of the present work are briefly summarised below:

MPI is still the *de facto* standard for HPC, but fundamental limitations are critical for hyper scalability. A more promising approach, to effectively scale up the applications, seems the hybrid programming model, based on coupling message-passing and multi-threading.

Different players are proposing new parallel languages, but none of them appears to be able to reach a production level maturity in the short term, and have a strong impact on the computational community. The DARPA languages still provide very poor performance and the work on them should be better focalised and unified, to be appealing for the HPC community. The PGAS languages could be more promising mainly if a standardisation process will be effective soon.

Hardware accelerators appear to have a big potentiality. Specific languages and programming environments have been defined and experimented in different applicative contexts. At present time it is difficult to identify common directions and guidelines for languages to efficiently exploit accelerators. The general impression is that all these languages are still "*hardware driven*" and not stable in their evolution. It is highly desirable that some standard will emerge soon in this context; the language on which most people pins their hopes is probably OpenCL. Another important issue in the direction of portability, productivity and performance seems to be represented by tools like RapidMind and the CAPS HMPP, but it is still too early to see what will be the real impact of this kind of tools. Probably a good direction to enhance the programming models for accelerators could be the availability of compilers with specific APIs to allow a semi-automatic code generation.

In summary, the Deliverable presents an analysis of the maturity, suitability, applicability of novel programming models, languages and libraries for Petascaling. The information is valuable also for other WPs, such as WP5, WP7 and WP8.

1 Introduction

The Partnership for Advanced Computing in Europe (PRACE) has the overall objective to prepare for the creation of a persistent pan-European HPC service. PRACE is divided into a number of inter-linked work packages, and WP6 focuses on the software for petascale systems.

The primary goal of PRACE WP6 "*Software enabling for Petaflop/s systems*" is to identify and understand the software libraries, programming paradigms and languages, tools, benchmarks and skills required by users to ensure that their applications can use a petaflop/s system productively and efficiently. WP6 is the largest of the technical PRACE work packages and involves all of the PRACE partners.

Task 6.6 is responsible for identifying the software libraries and programming models to implement petascaling applications. This is based on the analysis performed in D6.1 ([1]) and D6.2.2 ([2]) on a list of applications, covering a broad range of scientific areas and representative of the workload of today's European HPC systems. The outcome of these two deliverables is used to identify parallelization schemes, software libraries and programming models of these applications on present architectures. Key aspects of this work are the analysis of the current software libraries, their evolution to cope with petascale systems, new programming models and new tools to let current applications scale on petascale machines.

To better evaluate new programming paradigms and languages and their soundness for petascaling, three numerical kernels, typical of the most important computational applications, will be coded, using the main programming languages and paradigms investigated. This coding experience and the consequent porting and assessment on the prototypes selected by WP8, will greatly contribute to better complete the analysis, and evaluate the maturity, the effectiveness, the soundness, but also the programming effort of the different new languages and paradigms analysed. Furthermore, this experience will be useful to better evaluate if applications may find a smooth path toward petascaling, taking advantage of the evolution of the current paradigms, or if a significant re-coding and use of new libraries and new paradigms, or even new language extensions, or new frameworks to exploit the functionality of new accelerators, may represent a more promising approach.

1.1 Objectives

The objective of this work is to analyze in detail the role of new programming languages, programming models, frameworks for accelerators, as well as software libraries, in order to enhance applications to greatly benefit of petascaling systems.

In particular, the main objectives of the deliverable are:

- Identifying the programming models and the software libraries used today in PRACE applications;
- Analyze the evolution of the present programming models and new programming models and languages, and discuss their impact for petascale systems;
- Evaluate the potential of new languages and frameworks on prototype systems (based on hardware accelerators) to speed up applications, presenting some experiments on relevant numerical kernels;
- Evaluate the importance of the software libraries and the potential for optimizations on petascale systems.

1.2 Methodology

First the applications and software libraries already identified and analyzed in previous deliverables are reviewed to extract information about the programming models and software libraries currently used in PRACE applications. Then, the evolution of the main programming models, new languages and frameworks is analyzed to evaluate their pros and cons for applications in the context of petascale systems.

The evaluation of each programming model and language was assigned to the personal of the PRACE project with most experience or with a direct interest on that particular model, but more contributions from other partners are also taken into account. At the end of this phase, three numerical kernels have been selected and implemented using the main programming languages and models investigated, allowing a direct evaluation of these languages on the WP8 prototypes.

Finally software libraries have been analyzed. We have selected the main libraries used in PRACE applications and grouped by category and tagged. After this initial classification, each group has been analyzed by itself, underlying the strengths and weaknesses in a petascale context.

1.3 Structure of the Report

In Chapter 2 the applications identified by PRACE as representatives of the system load in PRACE centres are classified on the base of their parallel programming paradigms. Moreover, the most common software libraries used by the computational applications are classified on the basis of the characteristics relevant for petascale systems.

Chapter 3 analyses the current standards for parallel programming and their evolution (MPI, OpenMP, hybrid programming).

Chapter 4 presents the main features of the PGAS programming model, focusing on the specific implementation in the languages UPC, CAF and Titanium. The next generation programming languages and models, mainly the languages introduced as a consequence of the DARPA HPCS project (Chapel, X10, Fortress) are analysed in Chapter 5.

Chapter 6 introduces the languages, paradigms and environments for hardware accelerators: Cell programming, CUDA, OpenCL, CAPS HMPP, RapidMind, CellSs and Cn.

Chapter 7 focuses on a comparative analysis of the main programming languages and environments introduced in the previous chapters. This activity is accomplished by gathering information during the porting of three numerical kernels to selected languages and benchmarking them on WP8 prototypes.

Finally, Chapter 8 presents an analysis of the scientific libraries and their evolution for the petascale systems, ranging from the basic libraries to I/O libraries, from communication libraries to special purpose libraries.

Some conclusions and final remarks are drawn in Chapter 9. Five specific annexes complete the work.

2 Analysis of current HPC applications

The evolution of programming languages, models and software libraries for petascale applications has been addressed starting with the analysis of the programming models and libraries adopted by the current state of the art HPC applications. Deliverable D6.1 and its successor Deliverable D6.2.2 have identified and analysed a list of major applications used by the European scientific community involved in key areas of computational sciences.

In this chapter data previously collected and presented in D6.1 and D6.2.2 about codes is further analyzed to extract information about the programming models and software libraries currently used in PRACE sites and PRACE applications. In particular, applications are classified based on their parallelization paradigm and other characteristics of interest to understand how to scale the applications in petascale systems.

Furthermore, from these deliverables, a list of the scientific libraries used in the main PRACE applications analysed, has been obtained. These libraries are reviewed and classified, to evaluate the real impact of these libraries when used in applications scaling to petascale systems.

There are a huge number of parallel programming languages and models available in literature, many of them have never reached practical relevance for HPC or have the potential to achieve it. Therefore, the analysis covered in this chapter will be also useful to better identify the subset of programming languages and models whose evaluation should be regularly evaluated in light of future petascale systems.

2.1 Current programming models in PRACE application list

In deliverable D6.1 and then in D6.2.2 PRACE partners have compiled a list of applications covering all basic numerical algorithms and representatives of the system load in present HPC systems at PRACE partner sites. The applications in the list are used by all WP6 tasks to analyze application requirements for petascale system, to build a benchmark suite, to evaluate optimization strategies and in general to understand the challenges related to making applications suitable for a petascale system.

The selected applications address almost all fields of computational sciences, covering ten scientific areas and adopting the main algorithmic “dwarves”.

The scientific areas are:

- Astronomy and cosmology
- Computational chemistry
- Computational engineering
- Computational fluid dynamics
- Condensed matter physics
- Earth and climate science
- Life science
- Particle physics
- Plasma physics
- Other

The dwarves are those algorithm types which constitute classes where membership in a class is defined by similarity in computation and data movement and was first described from Lawrence Berkeley National Laboratory [3]. A dwarf represents a category of kernels that

share both computational and data structures. The dwarves adopted by the selected applications are:

- *Dense linear algebra* – data is stored in dense matrices or vectors and access is often via unit-level strides. Typical algorithm would be Cholesky decomposition for symmetric systems or Gaussian elimination for non-symmetric systems.
- *Sparse linear algebra* – data is stored in compressed format as it largely consists of zeros and is therefore accessed via an index-based load. Typical algorithm would be Conjugate Gradient or any of the Krylov methods.
- *Spectral methods* – data is in frequency domain and requires a transform to convert to spatial/temporal domain. They are typified by, but not restricted to, FFT.
- *Particle methods* – data consists of discrete particle bodies that interact with each other and/or the "environment".
- *Structured grids* – Represented by a regular grid. Points on the grid are conceptually updated together via equations linking them to other grids. There is high spatial locality. Updates may be in place or between 2 versions of the grid.
- *Unstructured grid* – data is stored in terms of the locality and connectivity to other data. Points on grid are conceptually updated together, but updates require multiple levels of redirection.
- *Map-reduce methods* – embarrassingly parallel problems, such as Monte Carlo methods, where calculations are independent of each other.

Identifying the dwarves used in each application is important when analysing the programming model and the parallel paradigm used by the application itself. In facts, the analysis done for an application can be easily extended to a set of different applications characterized by the same dwarves in terms of algorithmic kernels.

In the following we focus on the analysis of the programming model of the applications introduced, with particular attention to the parallelisation paradigm and the analysis of the software libraries adopted by these applications. Table 1 reports the PRACE applications initially selected in D6.2.2 plus some others that were selected later for the PRACE Application Benchmark Suite and for which we already had information. We underline that the final list of applications from the benchmark list has been updated over the time and, actually, some of the applications reported in Table 1 have been discarded while some new applications have been added.

For each application included in Table 1, the main programming model used and the different paradigms introduced are reported. Moreover, for each application it is indicated if the application itself has been benefiting from accelerators or not.

Application	Paradigm	MPI	OpenMP	Other	Accelerator
NAMD	Message driven	yes		charm++	CUDA
CPMD	Message Passing	yes	yes		CUDA (experimental)
VASP	Message Passing	yes			
QCD	Message Passing	yes			
GADGET	Message Passing	yes			Experimental (CUDA)
Code_Saturne	Message Passing	yes			
TORB	Message Passing	yes			
NEMO	Message Passing	yes	yes	autotasking	
ECHAM5	Message Passing	yes	yes		
CP2K	Message Passing	yes			CUDA
GROMACS	Message Passing	yes			CUDA
NS3D	Message Passing	yes		autotasking	
QuantumESPRESSO	Message Passing	yes	yes		
AVBP	Message Passing	yes			
HELIUM	Message Passing	yes			
GPAW	Message Passing	yes			
ALYA	Message Passing	yes	yes		Cell
BSIT	Message Passing	yes	yes		Cell
SIESTA	Message Passing	yes			
PEPC	Message Passing	yes			

Table 1: Classification of PRACE Applications

It can be seen from Table 1 that all the PRACE applications, except one, have been developed using the message-passing parallelization paradigm and the MPI communication library. Only NAMD is written using a message-driven paradigm. Some applications mix the message-passing paradigm with a multi-thread paradigm for the parallelization inside shared memory node. Multi-threading is implemented using OpenMP and in two cases the *autotasking*

implementation is adopted. On the current versions of PRACE codes the support for accelerators is not so common. For CPMD there is an experimental porting on Cell, but this is not jet included in a stable version. For GROMACS exists a special plug-in to use accelerators. NAMD includes the support for GPU in the stable version with a rich set of functionality. For CP2K there is some ongoing activity to exploit accelerators, but so far no production version is available.

From this analysis it turns out that the applications in the list, which represent the load of European Tier-1 systems, are quite similar from the point of view of the parallelization paradigm adopted. This does not necessarily mean that they will arouse similar problems/challenges to enable them for petascale system. This will depend mostly on the communication schema implemented by each application. There could be applications that may already scale well on petascale systems others that will require significant reengineering, or even to change the parallelization paradigm, in fact it is well understood that in many cases applications implemented using the message-passing paradigm fail to scale to high numbers of processors, due to communications overheads and task synchronization. In the next chapters possible alternative parallelization paradigms, useful to remove these bottlenecks, are analyzed in details. Moreover, large room for performance improvement could come from speeding applications with hardware accelerators.

2.2 Current libraries in PRACE application list

Most HPC applications rely on the use of external software libraries that implement standard operations in an efficient and robust way. This is true also for PRACE codes, in fact almost all codes require an external library and in most cases the library is a key component for the code performance, communication, data distribution and scalability. Then it is of fundamental importance to understand how these libraries will behave on petascale systems in order to enable the application itself.

Table 3 summarizes the most common software library either used in PRACE applications list or widely used at PRACE sites. With this table we try to identify the main characteristics of the libraries, relevant to port applications to petascale systems. To improve the analysis, the libraries are tagged and categorized. The set of library categories identified here will then be analyzed in detail in the next chapters. Table 2 contains the list of tags associated to each library category. Since many libraries do not fit in a single category, one or more tags are assigned to each library, which means that the library can fit in all the categories associated to the tags.

Category	Tag
numeric basic library	B
numeric high level	HL
communication	C
I/O	IO
special purpose	SP
architecture specific	AS
Low level	LL
non numeric	NN

Table 2: Library categories

Apart from the category, the libraries have been further analyzed to check if they adopt a parallel approach or not. In case a parallel version exists, a further analysis investigates if a multi-threaded version has been implemented and if the library has been ported to hardware accelerators or if a replacement version exists for accelerators. The result of this analysis is reported in Table 3 below.

Library	Classification	Version	Is Parallel?	Native Multi-threading	Source available
libm	LL	-			Y
szlib	LL	-			Y
FFTW2	B	2.1.5	Y	Y	Y
FFTW3	B	3.2.2		Y	Y
BLAS	B	-	PBLAS	Y	Y
ESSL	AS, B + HL	4.4	PESSL	Y	
ACML	AS, B	4.3.0		Y	
MKL	AS, B + HL	10.2	Y	Y	
NAG	B + HL	-	Y	Y	
LAPACK	HL	3.2.1	ScaLAPACK		Y
ScaLAPACK	HL	1.8.0	Y		Y
ATLAS	B	3.9.16		Y	Y
GOTO	B	-		Y	Y
METIS	HL	4.0.1	parMETIS		Y
GSL	HL	1.13			Y
WSMP	HL	9.9.10	Y	Y	
NUMPY	HL	1.3.0			Y
PETSc	HL	3.0.0	Y		Y
ARPACK	HL	-	PARPACK		Y
SPRNG	HL	4.0	Y		Y
BLACS	C	1.1	Y		Y
CHARM++	C	6.1.2	Y		Y
MPI[1]	C	2.1	Y		Y
CGNS	IO	2.5	Y		Y
HDF5	IO	1.8.3 or 1.6.9	Parallel HDF5		Y
NetCDF	IO	4.0.1	pNETCDF		Y
LibXML	NN, SP	-			Y
Tk/Tix	NN, SP	-			Y
TCL	NN, SP	-			Y

Table 3: Classification of most common libraries in PRACE codes

As appears from the Table above, a consistent portion of the scientific libraries analysed offer a parallel version and some others have implemented native multi-threading, but further testing activity should be addressed to check the suitability of the parallel versions to scale up for petascale applications.

[1] We refer to the MPI standard specification

3 MPI, OpenMP, mixed MPI-OpenMP: current standards and their evolution.

This chapter covers the current *de-facto* standards of parallel programming for applications able to run over the largest HPC systems in Europe. For each section, a short description of their main features is given. Also, where possible, we give a look to their proximal future evolution.

3.1 MPI

Message Passing Interface (MPI, [5]) is the current *de-facto* standard for programming distributed memory architectures, and is used in the vast majority of current HPC applications. Deliverable D6.1 surveyed 69 of the most heavily used applications running on PRACE centre machines. Of these, 67 were designed for distributed memory systems, and all of them were programmed using MPI. MPI consists of a subroutine library with interfaces in Fortran, C and C++.

3.1.1 *Current standard (MPI 2.2)*

The first version of the MPI standard (version 1.0) was published in 1994. A significant set of extensions to the library (version 2.0) followed in 1997, published as a separate document. Version 2.2, published in September 2009, essentially combines versions 1.0 and 2.0, with corrections, into a single document. Version 2.2 contains also minor updates and corrections to Version 2.1. These minor changes may include a few additional routines, and the addition of functionality to existing routines. The status of the discussions about Version 2.2 is public, and can be viewed at [6]. The full history of document versions is rather convoluted: a good summary is given in the introduction to Version 2.2.

3.1.2 *Overview of support in current implementations*

The majority of MPI implementations support MPI 2.x, with the exception of the dynamic process creation routines (Chapter 10 of Version 2.2), which may not be supported on HPC systems with very minimal operating systems running on the compute nodes.

3.1.3 *Future: MPI 3.0*

The MPI Forum is now focussing its efforts on the new version of the standard. MPI Version 3.0 will contain significant extensions to Version 2.x. This is at an early stage in the standardisation process, but some of the major areas being considered are as follows:

1. definition of an ABI (Application Binary Interface) to support linkage compatibility between different MPI implementations on a platform;
2. enhancements to collective operations, including
    a. persistent collective operations to allow optimisation where the same collective operations are repeated many times;
    b. new collective operations, including MPI_Reduce_scatter_block;
    c. sparse collectives, where the participating tasks form a subset of the tasks in a communicator;
    d. non-blocking versions of collective operations;
3. fault tolerance;
4. improved Fortran bindings;

5. extensions to generalised requests;
6. improved support for tools (especially debuggers);
7. improved support for hybrid programming, including the possibility of treating threads as MPI processes.

Again, the status of the discussions about Version 3.0 is public, and can be viewed by following the links from [7].

Of the topics under discussion, it seems likely that items 2c, 2d and 7 above are the most likely to have immediate applicability for mainstream HPC applications.

3.2 OpenMP

OpenMP is the current *de-facto* standard for programming shared memory architectures. The OpenMP API consists of compiler directives, runtime library routines and environment variables. Since no large-scale HPC systems have pure shared memory architecture, pure OpenMP programs cannot be run at large scale on the current generation of supercomputers. However, with the advent of SMP clusters, and more recently, of multi-core processors, OpenMP is of increasing interest in mixed-mode (or hybrid) MPI+OpenMP implementations of applications: see Section 3.3.

3.2.1 *Current standard (OpenMP 3.0)*

The current version of OpenMP is Version 3.0, released in 2008. The major changes from Version 2.5 are as follows:

- The addition of support for task parallelism via the *task* and *taskwait* constructs. A task construct identifies a block of computation that can be executed by any thread in the current team at some point in the future. Tasks can be arbitrarily nested: i.e. one task may contain child tasks, grandchild tasks, etc. When a barrier is encountered, all generated tasks must complete before the threads exit the barrier. The taskwait directive is used to ensure that all immediate child tasks of the current task have completed. OpenMP tasks are a very convenient mechanism for exploiting parallelism in loops with an unknown iteration count (e.g. while loops), and recursive parallelism.
- Better support for nested parallelism including per-thread internal control variables (allowing, for example "`omp_set_num_threads()`" to be called inside parallel regions), and new runtime library routines to determine depth of nesting, IDs of parent/grandparent etc. threads, and team sizes of parent/grandparent etc. teams.
- Additional support for loop parallelism, including the ability for OpenMP loop directives to parallelise perfect loop nests, a new schedule kind (auto) which allows the runtime to choose the best schedule for the loop, and library routines to get and set the schedule kind at runtime.
- Portable control of thread behaviour, including environment variables to set thread stack sizes, and to hint to the runtime how to treat threads which are waiting idle at locks or barriers. There are also library routines and environment variables that control the maximum depth of nesting and the total number of running threads.

For a full list of changes between OpenMP Versions 2.5 and 3.0, see Appendix F in the OpenMP Version 3.0 API specification.

3.2.2 *Overview of support in current implementations*

Almost all current implementations of OpenMP support at least Version 2.5. The majority of compilers also now support OpenMP 3.0 including:

- Intel 11.0: Linux (x86), Windows (x86) and MacOS (x86)
- Sun Studio Express 11/08: Linux (x86) and Solaris (SPARC + x86)
- PGI 8.0: Linux (x86) and Windows (x86)
- IBM XLC 10.1: Linux (POWER) and AIX (POWER)
- GCC 4.4

3.2.3 *Future*

Work on future versions (3.1 and 4.0) of the OpenMP API standard is now in progress. Version 3.1 is likely to contain fairly minor changes to the standard, mostly consisting of fixing errors and inconsistencies in Version 3.0, and a small number of new features, possibly including user-defined reduction operators and support for additional atomic operations. For Version 4.0, some of the issues been considered include:

- development of an error model;
- interoperability between OpenMP implementations, and with other threading models;
- mechanisms to specify thread to core mappings;
- support for accelerator devices;
- enhancements to OpenMP tasking, including task reductions and dependencies between tasks.

There are a number of commercial and research efforts underway to allow OpenMP programs to execute on heterogeneous multi-core systems (such as the Cell BE), and accelerators such as GPGPUs. If these come to fruition, OpenMP may prove an attractive way of portably programming such devices, in contrast to the increasing number of low-level and device specific programming models currently available. Extensions to OpenMP may be required to realise this efficiently.

3.3 Mixed mode (MPI+OpenMP)

With current ubiquity of multi-core architectures, mixed mode MPI+OpenMP is increasing in importance as a programming model. Many MPI programs do not scale to very large numbers of MPI tasks. This can be for a variety of reasons, including:

- the existence of replicated data between MPI tasks, causing the application to run out of memory;
- the overhead of communicating and storing data associated with halo elements (or other fake boundary conditions) when the domain size per task becomes small
- load imbalance;
- the overhead of calling MPI to communicate between MPI tasks on the same shared memory node;
- MPI implementations that do not take advantage of the shared memory within nodes to implement efficient collective operations.

In some circumstances, using OpenMP to exploit parallelism with a shared memory node (or within a multi-core processor) can alleviate some of these difficulties and enable the

application to scale better at high core counts. This may, however, come at some expense in terms of the complexity and maintainability of the code.

Currently only a relatively small number of applications use the mixed mode programming model: in the survey of applications reported in Deliverable D6.1, only 10 of the 69 applications used this model.

3.3.1 *Current status*

There is no separate standard that defines the behaviour of mixed mode programs. The OpenMP standard says nothing about its interaction with MPI; no does it attempts to define thread safety (which is a surprising difficult concept to specify!). As far as OpenMP is concerned, MPI is treated like any other library.

MPI, on the other hand, acknowledges the existence of threads: see Section 12.4 of the MPI 2.1 standard. It defines four levels of thread support:

- MPI_THREAD_SINGLE: only one thread will execute.
- MPI_THREAD_FUNNELED: the process may be multi-threaded, but the application must ensure that only the main (or master) thread makes MPI.
- MPI_THREAD_SERIALIZED: the process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").
- MPI_THREAD_MULTIPLE: multiple threads may call MPI, with no restrictions.

A given implementation supports one of the above levels: this can be queried using MPI_INIT_THREAD or MPI_QUERY_THREAD.

3.3.2 *Overview of support in current implementations*

Most current MPI implementations support at least MPI_THREAD_FUNNELED, and most current mixed mode applications do not assume a higher level of support.

Some MPI implementations (e.g. MPICH2 for Linux, and IBM MPI libraries for POWER5, POWER6 and BlueGene/P) provide support for MPI_THREAD_MULTIPLE. However, some of these currently suffer from bugs, or from very poor performance characteristics. It is therefore not realistic at the time of writing this document to implement an application that requires MPI_THREAD_MULTIPLE and expect good performance and portability across different systems.

3.3.3 *Future*

It is anticipated that the level (and quality) of thread support in MPI implementations will increase as multi-core architecture dominates the HPC market. The MPI Forum is considering extensions to MPI which would elevate the status of threads in MPI to be similar to processes. Discussions about this are at an early stage.

One extension to OpenMP which has been proposed is the concept of *subteams*. A common pattern in mixed mode programming is for one thread to perform MPI communication while the remaining threads share out some computation. Currently, in OpenMP, the *work-sharing* directives (e.g. DO/for, sections, single) apply to all the threads in the team, and so cannot be used to implement this pattern. The addition of subteams would address this shortcoming.

3.4 Final remarks

MPI and OpenMP are arguably the de facto community standards for distributed-memory programming and shared-memory programming respectively. MPI and OpenMP put together distributed-memory (provided by MPI) and shared-memory programming model (provided by OpenMP). While these technologies have served the parallel programming community admirably over the last couple of decades, recent advances in massively parallel multi-core architectures have revealed their fundamental limitations in the productive design of high performance codes.

Table 4 lists some of the main advantages and disadvantages experienced by users of the MPI and OpenMP models.

	MPI	OpenMP
Advantages	• Portable • Near optimal performance on most architectures • Strong foundation for higher level technologies • Broad support among community and vendors	• Supports finer-grained parallelism • Can be mixed with other programming models relatively easily • Supports incremental parallelisation • Broad support among community and vendors
Disadvantages	• Only supports parallelism at the 'cooperating executable' level • Requires too much information about "how" data is too be transferred rather than simply "what" is to transferred • Memory management overhead • Obfuscates algorithms with many low-level details	• Shared-memory bugs can be difficult to locate • No precise control over locality and processor affinity

Table 4: Advantages/Disadvantages exhibited by MPI and OpenMP models

4 Partitioned Global Address Space (PGAS) Programming Languages

As the High-Performance Computing (HPC) community begins to usher in the era of petascale computing, it is becoming ever more apparent that traditional programming models are failing to provide a satisfactory environment for the development of highly scalable codes. With petascale computing architectures exhibiting increasingly complex layers of parallelism and communication in the drive for performance, some inadequacies of existing programming models, such as those represented by MPI and OpenMP, are being exposed. In addition to performance, the community is also demanding *productivity* gains during the software development process, a trait traditional programming languages and models fail to address satisfactorily. For these reasons, it is inevitable that new parallel programming paradigms are being proposed to address these performance/productivity challenges.

The PGAS programming model is gaining much momentum as a novel approach to improving programmability and productivity of large-scale computing architectures. In this chapter we review the current state-of-the-art in PGAS language design and implementation with a particular focus on the PGAS languages UPC, Coarray Fortran and Titanium. Furthermore, we consider the next-generation Asynchronous PGAS (APGAS) programming model with specific focus on the Chapel and X10 programming languages in the next chapter.

The PGAS Programming Model

A new parallel programming paradigm that is gaining widespread interest within the HPC community is the Partitioned Global Address Space (PGAS) model. In this approach the best features of MPI and OpenMP are combined to provide an explicitly parallel, shared-memory like programming model whereby the programmer is exposed to the dual concepts of a *global address space* and a *locally partitioned address space*.

In this approach distributed memory semantics is integrated into the language definition, leveraging the languages' type system as well as the compilers' capacity to optimize data transfer concurrently with serial code. In the PGAS programming model the global address concept allows a thread or process to directly read or write data allocated by another thread or process. Figure 1 depicts the process and memory models underlying the message-passing, shared-memory and PGAS programming paradigms.

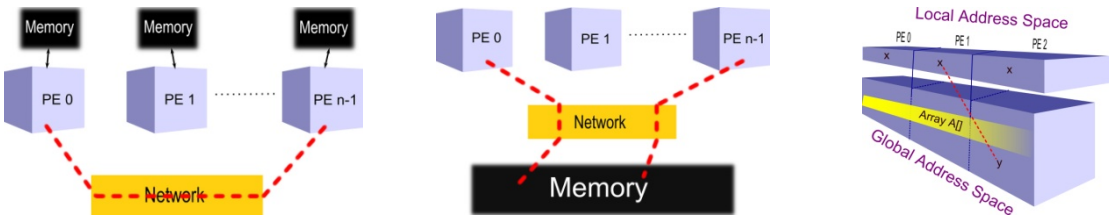

Figure 1: Message Passing, Shared Memory and PGAS models

Furthermore, memory is logically partitioned as *local* or *remote* depending on its locality to a given process or thread. This allows the programmer (and compiler) to expose the inherent NUMA-ness of current multi-core architectures or (in the case of a distributed memory system) of the various possible distances between local and remote compute nodes, which is unavailable in traditional shared memory programming models such as OpenMP, and for which explicit programming is required in MPI.

The PGAS programming model provides two major characteristics, namely *data partitioning* and the distinction between local and remote data that is used to minimize the communication overhead and get better performance. Performance of a PGAS-based algorithm depends on

the access patterns to the global address space. A number of processes can work independently in a Single Program Multiple Data (SPMD) model. Processes communicate through the global memory and can access sharable data while a private object may be accessed only by its own thread.

PGAS languages are designed such that implementations must generate *one-sided* communications, in which one process reads or writes to the addresses associated with a remote data entity. One possible middleware, which might be used by PGAS implementations, is GASNet [8][9] that provides a language-independent, low-level networking layer that offers network-independent, high-performance communication primitives.

Summarizing, the main features inherent in the PGAS programming model are:

- Executes typically in a SPMD fashion.
- Supports both shared and distributed memory architectures.
- Raises the level of abstraction compared to MPI.
- Typically exploits faster one-sided communication compared with MPI two-sided communications.
- Global address space supports construction of complex sharable data structures.
- High level constructs (e.g., distributed multidimensional arrays) simplify programming.
- Compilers can optimize parallel constructs.
- Portability.

In general the PGAS programming model has been implemented in traditional base languages (such as Fortran, C and Java) via language extensions. Examples of these approaches include Coarray Fortran (CAF), Unified Parallel C (UPC) and Titanium respectively. In the following sections each of these complementary implementations will be reviewed.

4.1 Unified Parallel C (UPC)

UPC[10] is a member of the PGAS language family but is also an explicit parallel extension of ANSI C. The development of UPC has been based on ideas introduced in other earlier languages that proposed parallel extensions to standard ISO C99: Split-C, AC, and Parallel C Pre-processor (PCP). UPC is an attempt to use the most suitable features of these languages and preserve the base concept and the main philosophy of the C language (e.g., a concise and efficient syntax). Any legal C program is also a legal UPC program.

4.1.1 *The Execution and Memory Model*

The execution model of UPC is comprised of a number of threads working independently in an SPMD fashion. The number of threads available to the program can be accessed via the THREADS variable. Each thread is identified by the variable MYTHREAD.

The number of threads can be specified at compile time by the user as a compiler option (*Static Threads* mode) or the compiled code may be run with varying numbers of threads (*Dynamic Threads* mode). In the first compilation mode, the compiler is able to generate more efficient code, but model size is fixed; it is necessary to recompile whenever the number of threads or the model size changes. The second compilation mode prevents the use of certain parallel constructs but allows to adjust the number of THREADS at each program run without the burden of recompiling the code. This is done via an environment variable or an additional variable to the execution command.

The memory associated with a thread is further divided into private and shared address spaces. Normal C variables and objects are allocated in the private memory space for each thread. Shared variables are allocated only once. Their particular location in memory is determined by how they are declared. Shared variables may not be declared automatic, i.e., may not occur in a function definition, unless they are declared *static*.

Shared arrays in UPC are distributed in a cyclic fashion across the threads, usually by a round robin strategy. For example, if arrays are thought of as intrinsically one-dimensional, the default distribution of an array is to place the 0-th element on thread 0, the 1-st element on thread 1, up to the (N-1)-th element on thread N-1, then the N-th element is placed on thread 0 and the process begins again.

It is possible to use a fixed block size as an additional specifier when declaring a shared global array in static mode. This leads to block-wise round robin allocation, where more than one element is cycled across the available threads, thus potentially improving the data locality. Unfortunately the block size currently cannot be dynamic i.e., it needs to be known at compile time. This imposes (unnecessary) constraints for the dynamic execution mode and can be circumvented only by the use of a helper data structure and the non-collective upc_alloc(), as shown in the kernel benchmarks.

Since different kind of variables can be involved during computation, UPC allows expressing different levels of memory consistency, namely strict or relaxed, which may be set by region of code by #pragma directives, or per variable by use of declaration qualifiers. Under relaxed mode, accesses may be reordered for speed, so care must be taken to prevent race conditions between threads, or parts of data being overwritten by simultaneous write accesses. Conversely, strict mode ensures that all accesses are absolutely ordered, and that each write is finished before the next may begin. Although relaxed mode accesses may be reordered, they may not move past a strict access. UPC provides locks that can be applied to sections of code to prevent simultaneous execution by more than one thread, and barriers provide explicit global synchronisation points. Additionally, the statement upc_fence exists as a null strict access, ensuring all accesses are complete. It should be noted that relaxed is the default behavior.

Since UPC is a dialect of C it supports pointers. There are four distinct possibilities for a pointer in UPC since they 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 the performance of the code.

In parallel programming the “owner computes” rule is very common. In general this fine grain parallelism is implemented by looping over all the data items and only those items owned by a thread or process are operated on by that thread or process. UPC adds a special type of loop to accomplish this task: upc_forall. Finally, UPC contains its own language primitives to provide dynamic allocation of shared variables, data movement between local and remote memories (upc_memget & upc_memput), collective movements between threads, scatters and gathers, computational collectives like reductions, thread-data affinity and advanced memory consistency.

4.1.2 *Current Implementations*

UPC v1.0 was published in 2001; last standard is v1.2 published in May 2005. UPC has become popular because it has achieved a higher degree of uptake and has been adopted by several influential manufacturers, e.g., CRAY.

Currently there are both free and commercial UPC compilers available, for a broad spectrum of architectures. In detail

- HP UPC (commercial): released April 2007, it supports all HP/Compaq branded platforms, including Linux, Tru64 and HPUX platforms.
- Cray UPC (commercial): built-in to the Cray C compiler on the Cray X1, X1E and future Cray vector-family platforms. Currently also available for Cray XT architectures under the Cray Compiler Environment.
- IBM UPC Alpha Edition (commercial) [11]: UPC compiler for IBM BlueGene and PowerPC SMP's running AIX or Linux. It requires IBM XL C compiler to run.
- GCC UPC (free) [12]: developed by Intrepid; it supports a lot of architectures (from x86 to Cray) and it uses the Berkeley runtime.
- Berkeley UPC (free) [13]: fully portable source-to-source compiler implementation. It supports a wide range of compilers and systems and uses GASNet as communication middleware.
- Michigan Tech MuPC (free) [14]: MPI-based reference implementation for Linux and Tru64.

4.2 Coarray Fortran (in the context of the Fortran 2008 Draft Standard)

Coarray Fortran, formerly known as F--, is an extension of Fortran 95/2003 for parallel processing created by Robert Numrich and John Reid. The coarray concept was designed as the minimal change needed to convert Fortran into a powerful and efficient parallel language.

First published in [12] and implemented in the Cray compiler targeted for parallel vector systems, a complete and full description of coarray syntax was published in 1998. The ISO Fortran Committee decided in May 2005 to include coarrays in the next revision of the Fortran standard ([16]).

4.2.1 *The Execution and Memory Model*

The coarray approach to parallelism falls into the SPMD category. An execution instance of the program is replicated – typically based on user-provided settings in the run time library – and the generated *images* then execute asynchronously and independently. The number of images is fixed and each image has its own index, retrievable at run-time.

Each image works on its local and sharable data and a sharable "object" has the same name on each image. An image moves remote data to local data through, and only through, explicit coarray syntax. Declaring an object with a *codimension* (using additional square brackets, or the CODIMENSION attribute) allow the contents of the object on one image to be accessed by any other image, while traditional Fortran objects are only locally accessible within each image. The semantics of information transfer across images is that of one-sided communication, enabling an optimized implementation to efficiently exploit the interconnect hardware, and allowing the programmer to overlap computation and communication where appropriate.

Coarray Fortran in the context of Fortran 2008 is presently the only object-oriented HPC language with integrated parallelism, although some restrictions apply to using object-oriented features in the context of parallelism. This is because efficiency of execution and relative ease of implementation were placed at a higher priority than providing the highest level of abstraction (as for example the Chapel language does). Allocatable coarrays are required to be allocated in a symmetric manner, thereby requiring no transfer of descriptor

information for coarray operations. Allocation as well as de-allocation of such entities therefore implies synchronization across all images. A coarray entity of derived type with a pointer or allocatable component will in contrast typically allocate image-dependent sizes for the dynamic component; the downside is that an additional latency overhead is incurred for transferring the content across images, and that the programmer must take greater care in handling the details of such transfers correctly.

A number of new intrinsics is defined which enable the programmer to control execution on a per-image basis, and to manage image indices as well as *coindices*. For efficiency as well as to limit the number of additional intrinsics, synchronization control is performed using statements like sync images, sync all, critical, etc. It is the responsibility of the programmer to insert synchronization statements in a manner that prevents race conditions or deadlocks, while maintaining good scalability. Between synchronization points, the Fortran processor can perform code optimization as if the implementation were serial.

4.2.2 *Current Implementations*

Coarray syntax has been a supported feature of Cray Fortran since release 3.1 (via the "–Z" compiler option on Cray T3E and Cray X1). In this particular case, during compile time the user has the possibility to specify the number of images.

Coarray Fortran is also being implemented for Cray XT platforms under the Cray Compiler Environment. The current release, however, suffers from serious compiler bugs as well as stability and performance issues. For example, it is possible to generate executables, which crashe compute nodes in a way such theyneed to be rebooted. Also, in numerous situations the compiler optimizer imposes pattern matching, which may generate incorrect code, and switching it off may produce very slow one-sided communication patterns with slowdowns in the order of 10-100X.

The Los Alamos Computer Science Institute (LACSI) at Rice University is working on an open-source, portable, re-targetable, high-quality CoArray Fortran compiler suitable for use with production codes. Their compiler uses the Open64/SL Fortran 90 source-to-source infrastructure to translate Coarray Fortran into Fortran 90 plus calls to ARMCI or GASNet, two multi-platform libraries for one-sided communication. It is available for (i.e. has been tested on) SGI Origin 2000 and Altix 3000, Compaq Alphaserver SC (Quadrics switch), Linux Itanium 2 with Myrinet 2000 or Quadrics II switches, and Linux Pentium with Ethernet. However, this provides only partial support for coarrays since the type system is not even fully integrated on the Fortran 90 level.

Andy Vaught's G95 compiler ([17]) partially supports coarrays on the Fortran 95 level, and is presently the most complete implementation from the functionality point of view; however concurrent optimization of data transfer is not done, and data transfer itself is still very inefficient, so the implementation can presently only be used for very weakly coupled algorithms. To use the coarray facilities with more than 5 images requires purchase of a license (at very moderate cost).

There is also a translator from CAF to OpenMP. This only provides a subset of the language, but most programs require very little adjustment to bring them into line with this subset. The subset has been chosen to make it possible to implement coarrays as ordinary arrays of higher rank, ideal in a shared memory environment. Once the code has been translated, an OpenMP capable compiler is needed, but these are widely available for a range of platforms from small workstations through commodity clusters to supercomputers.

4.3 Titanium

Titanium [19] is a SPMD Partitioned Global Address Space (PGAS) programming model, in affinity with UPC and CAF, which uses Java as its base language. Titanium is being developed at Lawrence Berkeley National Labs (LBNL) as an explicitly parallel dialect of Java for high-performance scientific computing.

Titanium adds the following extensions to the Java base language to support a PGAS programming model. The acceptance of Titanium in the HPC community seems low: Java is not a traditional HPC language and currently very few Java codes are being used at European HPC sides. For the sake of completeness, Titanium has been included and described in more detail in the language part of this document. However, it has been excluded from the comparison of the languages performed through porting the three Euroben kernels.

4.3.1 *The Execution and Memory model*

The standard Java language is ill-suited for use on distributed-memory machines because it adopts a dynamic task-parallel model and assumes a flat memory hierarchy. Titanium refers to the different instances of a program as processes and each one has its own local and global addressable memory. In Titanium the default declaration, unlike in CAF and UPC, is for data to be global (shared). The local qualifier overrides this.

Titanium supports the construction of distributed array data structures in the Partitioned Global Address Space, in which each process creates its share of the total array. Since distributed data structures are explicitly built from local pieces rather than declared as distributed types, Titanium is sometimes referred to as a “Local-View" language. To create shared data structures each process builds its own local array piece and then the individual pieces are exchanged (for objects, pointers are just exchanged). The exchange operation is a primitive all-to-all communicator in Titanium.

Titanium provides the built-in methods to query the environment for the number of processes and the index within that set of the executing processes. Because many scientific applications are written in a bulk-synchronous style, Titanium also provides a global barrier-synchronization construct as well as a set of collective communication operations to perform broadcasts, reductions, and scans ([20]).

Because Java has no real support for multi-dimensional array, Titanium adds a powerful abstraction, which provides the same kinds of sub-array operations available in Fortran 90. Titanium arrays are indexed by integer tuples known as *points* and built on sets of points called *domains*. Points and domains are first-class entities in Titanium so they can be stored in data structures, specified as literals and passed as values to methods and manipulated using their own set of operations. The true power of Titanium arrays stems from array operators. Array operators can be used to create alternative views of an array's data without an implied copy of the data. Titanium's domain calculus operators support sub-arrays both syntactically and from a performance standpoint. The tedious business of index calculations and array offsets has been migrated from the application code to the compiler and runtime system. In particular Titanium provides a rich set of array operations (like Translations, Restrictions and Slices) and a wide range of domain operators (like Intersections, Unions and Differences).

Like UPC, Titanium differentiates between *global* and *local* references. A *local* pointer must refer to an object within the same process partition, while a *global* pointer may refer to an object in either a remote or local partition. Pointers in Titanium are global by default, but may be designated *local* using the local type qualifier. Well-tuned Titanium applications perform

most of their critical path computation on data that is either local or has been pre-fetched into local memory. This avoids fine-grained communication costs that can limit scaling on distributed-memory systems with high interconnect latencies.

4.3.2 *Current Implementation*

Titanium adds new features to Java but it is not a true extension of Java language. Since the Titanium compiler is implemented as a source-to-source translator to C then any library offering a C-compatible interface is potentially callable from Titanium (this also includes many libraries written in other languages such as C++ or Fortran). Since Titanium has no JVM, there is no need for a complicated calling convention (such as the Java JNI interface) to preserve memory safety. To perform cross-language integration, programmers simply declare methods using the *native* keyword and then supply implementations written in C.

The LBNL Titanium compiler translates Titanium code into C code, which is then compiled with a vendor-provided C compiler and linked with the Titanium runtime system. For shared-memory systems, the pthreads library is used to manage to thread management, while the GASNet communication system is required for distributed-memory systems. Unlike HPJava [21], C is chosen as the compilation target instead of Java byte-code in order to maximize portability, since several high-end supercomputers such as the Cray X1 and the IBM Blue Gene lack Java compilers and possibly OS support required for a fully operational JVM.

The latest release of the Titanium compiler can be obtained here [22].

5 Next Generation Programming Languages and Models

One of the main goals of the U.S. DARPA HPCS project [25] is to raise HPC user productivity. As part of this project three novel parallel programming languages were investigated for development, with the aim of improving programmer productivity on next-generation computing architectures. The three languages are:

1. Chapel
2. X10
3. Fortress

These languages implement an Asynchronous Partitioned Global Address Space (APGAS) programming model; a term originally coined within the X10 language project. The APGAS model extends the PGAS programming model by providing a richer execution framework than the SPMD style generally used by the traditional PGAS languages. While CAF, UPC and Titanium follow the traditional SPMD mode; Chapel, X10 and Fortress choose a more flexible model and allow more general forms of control flow among concurrent tasks or activities.

The last section coves a new non-conventional parallel programming model called Transactional Memory, explicitly oriented for multi-threading systems. It puts the focus on the concept of *transaction* that ensures correctness, consistency and easy programmability.

5.1 Chapel

Chapel [24] is a new parallel language being developed by Cray Inc. as part of Cray's entry in DARPA's HPCS program. The main goals of Chapel are to:

- Improve programmer productivity.
- Improve the programmability of parallel computers.
- Match or improve upon the performance of current programming models.
- Provide better portability than current programming models.
- Improve robustness of parallel codes.

5.1.1 *The Chapel Programming Model*

Chapel supports a multi-threaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism.

Chapel is a global-view parallel language that supports a block-imperative programming style and it is designed around a multi-threaded execution model in which parallelism is not described using a processor- or task-based model, but in terms of independent computations implemented using threads. Rather than giving the programmer access to threads via low-level fork/join mechanisms and naming, Chapel provides higher-level abstractions for parallelism using anonymous threads that are implemented by the compiler and runtime system ([23]).

At its core, the Chapel programming model is governed by the concept of *locales.* We use the term locale to refer to the unit of a parallel architecture that is capable of performing computation and has uniform access to the machine's memory. For example, each node and its associated local memory would be considered a locale. Locale properties include:

- threads within a locale have uniform access to local memory;

- memory within other locales is accessible, but at a price;
- locales are defined for a given architecture by a Chapel compiler e.g., a multi-core processor or SMP node could be a locale.

Chapel supports a locale type and provides every program with a built-in array of locales to represent the portion of the machine on which the program is executing.

By exploiting locales, Chapel programmers can reason about the complex computing hierarchy of an architecture and select and initiate tasks on a given locale within their code (using the Chapel *on* construct). These operations extend those that are typically provided in traditional PGAS programming models, which is referred to as asynchronous PGAS programming.

Along with initiating tasks on different locales (as well as initiating within the current local using the *begin* statement), Chapel programmers can distribute data arrays across locales using built-in, as well as, custom-designed *domain types*. Chapel has three main classes of domains: *arithmetic*, *indefinite* and *opaque*. Each one has its own characteristics. Also the data distributions can follow traditional approaches (e.g. block cyclic, recursive bisection or graph partitioning) or a user can implement its own strategy. Data-parallel operations can subsequently be invoked both within and across locales on these types to achieve maximum parallel performance where possible.

To synchronise access to shared resources when reading and writing to memory, Chapel provides *sync types* for implementing *critical sections* and reducing dangerous race conditions. Chapel also supports atomic sections to indicate that a group of statements should be executed as though they occurred atomically from the point of view of any other thread. Atomic sections have been of recent interest to the software transactional memory community due to their intuitiveness and ability to replace explicit lock management in user code. By specifying intent rather than mechanism, atomic sections result in a higher-level abstraction for users than locks.

5.1.2 *Chapel Developments*

In addition to providing an APGAS parallel programming model the base language of Chapel was selected to contain the best features from most modern mainstream programming languages e.g. OOP classes and inheritance, iterators and generic/template programming.

It is hoped these collections of modern programming features and powerful parallelism constructs, along with an abstract view of the computing architecture using locales, will provide a rich environment for productively developing high-performance codes on next-generation computing architectures.

A Cray prototype of a Chapel compiler was released on April 16th, 2009. It can be downloaded and used under the terms of the BSD license. A production Chapel compiler is envisaged around the 2011 and 2012 timeframe.

5.2 X10

The X10 language is developed by IBM at the T.J. Watson Research Centre as part of the PERCS project within the DARPA HPCS initiative. As referred in the article [26], X10 is designed to balance four major goals:

- *Safety*: the programming model is intended to be safe in the interests of productivity.

- *Analyzability*: X10 programs are intended to be analyzable by other programs (like compiler or analysis tools).
- *Scalability*: a program is scalable if the addition of computational resources leads to an increase in performance.
- *Flexibility*: scalable applications will need to exploit multiple form of parallelism.

5.2.1 The X10 Programming Model

X10 is an "extended subset" of the Java programming language, strongly resembling it in most aspects, but featuring additional support for arrays and concurrency. Like Chapel, X10 has strong support for global-view domains and distributed arrays. X10 is based on a threaded model for spawning tasks.

In a similar way to Chapel's *locales* abstraction, X10 introduces the concept of a *place.* Places encapsulate an affinity between activities and data. All data that is writeable is assumed to be partitioned across *places*, thereby resulting in a Partitioned Global Address Space (PGAS). Activities may hold a reference to remote objects i.e., objects located at places different from the current activity's place. If an activity attempts to operate on a remote object, then it does so by logically creating a new activity at the remote place to perform the operation (Asynchronous PGAS model). X10 programmers can also explicitly initiate activities on remote places using the `async` and `future` expressions.

X10 has global address spaces. A scalar object in X10 is allocated completely at a single place. In contrast, the elements of an array may be distributed across multiple places. X10 supports *Globally Asynchronous Locally Synchronous* (GALS) semantics for reads/writes to mutable locations. Say that a mutable variable is local for an activity if it is located in the same place as the activity; otherwise it is remote. Within a place, activities operate on memory in a sequentially consistent fashion, that is, the implementation ensures that each activity reads and writes a location in one indivisible step, without interference with any other activity. Shared access to memory resources can be synchronised using the *atomic* expression. Atomic blocks are conceptually executed in a single step while other activities are suspended.

X10 has true multi-dimensional arrays (like Fortran) that can be distributed among activities. A multi-dimensional array is associated with a set of index points called its *region*. X10 also provides *distributions* that map points in a region to a *place (*distributions can be blocked, cyclic or a combination of both*)*. Data parallel operations can be applied to distributed points and parallel iterators can be defined.

5.2.2 *X10 Developments*

The current version of the X10 compiler and Common PGAS Runtime is v1.7.6 [27]. The tools can be downloaded for many different platforms.

5.3 Fortress

The overall goal of the Fortress Project [28] (by Sun Microsystems) is to create a modern programming language whereby nearly all the language semantics are defined at the library level, and not at the compiler level. This organisation aids the rapid growth of the language as new language ideas unfold.

Library-level definitions include data structures and types, through to operators and their precedence that can be easily swapped or replaced. An interesting feature of Fortress is that most language objects are implicitly parallel by default (e.g. the for loop). Fortress also

supports mathematical notation (through Unicode) for symbols and operators along with a hierarchical representation of target architecture's structure. Units of measurement are included in the type system e.g. meters, seconds, miles etc. Transactions and locality specification are also highly featured.

5.3.1 *The Fortress Programming Model*

Fortress differs from both Chapel and X10 languages in several respects. Fortress syntax emulates mathematical notation as closely as possible. It has a novel type system to better integrate functional and object-oriented programming, it allows for specification of data distribution through the use of "distribution" data structures, it supports static checking of physical units and dimensions, it supports embedding of domain-specific language syntax in programs, and it includes a component system to facilitate the process of compiling, linking, and deploying programs.

The basic building blocks of Fortress code are objects and traits. Objects define fields and methods, whereas traits declare sets of methods. Traits can declare abstract (i.e. header only) or concrete (i.e. header and definition) methods.

Fortress uses a threaded model for implementing concurrency and parallelism. In general threads can be *explicitly* or *implicitly* spawned. Each thread can be on one of five states depending on its status. Fortress has many constructs that lead to implicit thread creation:

- Tuple expressions.
- "do" blocks.
- Method invocations, function calls.
- For loops, comprehensions, sums, generated expressions, big operators.
- Extremum expressions.

Implicit threading in Fortress is executed in fork-join style; all threads created together, and all must complete before the expression completes. If any threads end abruptly, the group as a whole will also end abruptly. Furthermore, the programmer cannot interact with implicit threads in any way as the compiler generates them. The Fortress compiler may also interleave the threads any way it likes. Explicit threads are created using the *spawn* and *also do* expressions. In this mode the programmer can interact with the thread explicitly; spawn returns an instance of thread that can be controlled with: wait, ready, stop.

It is important to point out that threads evaluate expressions by taking steps. In fact the execution of a fortress program consists of evaluating the body expression of the run function and the initial-value expressions of all top-level variables and singleton object fields in parallel. Communication with the outside world is accomplished through input and output actions.

Fortress also maps a tree-like abstraction of the architecture hierarchy as a *region* (similar in concept to Chapel's locales). All threads, objects, array elements have an associated region. Leaves of the tree mapping are typically low-level processing elements (e.g. core in CPU). The root of the tree maps out a more high-level and abstract distribution (e.g. resources spread across entire cluster). Data distribution operations can be performed across regions whereby vectors, arrays and matrices are distributed across the machine. Fortress will usually determine the optimal distribution of elements; it is possible that each element of a data structure could be on a different region.

Fortress provides at least two mechanisms for coordinating access to shared variables in parallel operations: atomic operations and reductions. Atomic operations use Fortress's

transactional memory system to ensure that one or more operations occur atomically, but with minimal locking and contention. Atomic operations prevent data races, but do not by themselves ensure determinism or serial semantics. Reductions are constructs that perform associative operations in parallel and preserve serial semantics.

Asynchronicity is also provided in Fortress whereby programmers can explicitly spawn tasks/threads on a specific region.

5.3.2 *Fortress Developments*

Fortress did not make it through to Phase III of the HPCS initiative. In January 2007 the Fortress project was made available to the open source community and on 1 April 2008, the first version of the Fortress specification with a compliant implementation (Fortress 1.0) was released. Currently there is only a functional Fortress Interpreter available for download for many platforms ([29]).

5.4 Transactional Memory

Transactional Memory (TM) is a novel programming model for multi-core architectures that allows for concurrency control over multiple threads. Its main goals are to increase *code productivity* by providing a simple programming model to the programmer, and achieve higher performance through the concept called *optimistic parallelism.*

5.4.1 *Basic concepts*

A memory transaction is a sequence of memory operations that either executes completely or has no effect. According to the programming model supporting the concept of transactions, the programmer is able to include parts of his code within a transaction, indicating in this way that within this section there exist accesses to memory locations that may be performed by other threads as well. The TM system records the transactions of threads, and if two or more threads perform conflicting memory accesses (e.g. one thread reads variable *a*, a second thread writes variable *a*) then it decides how to handle this conflict. The common case is to allow one of the threads to commit its transaction and restart the transaction of the other conflicting thread.

The above simple description includes two important concepts. The first one is programming simplicity. The programmer does not needs to care about the detailed memory access pattern of each of the created thread. The programmer just needs to discover and pinpoint possible crosscutting memory-access paths made by different threads and include each of these paths within a transaction. The alternative approach would be to apply locking, either coarse-grained or fine-grained. Coarse-grained locking essentially locks large parts or all of the data structure accessed. It is easy to implement but leaves limited chances of high performance since threads in this case are practically serialized. On the other hand, fine-grained locking locks data elements, which can be too complicated and error-prone, without ensuring high performance, since frequent locks are quite expensive in modern multi-core processors. The locking performance cost needed to ensure semantic correctness of programs reveals the second benefit of TM. In fact in several multi-threaded programs, the possibility of two threads accessing the same memory location is rather small. In this case guarding this memory location with a lock does not seem a good idea. In the concept of TM the two threads will be allowed to access the memory locations without synchronization, and only if they perform conflicting accesses will the system pay the overhead of resolving the conflict and restarting

one of the threads. This optimistic parallelism is expected to provide higher performance in programs with irregular memory accesses to dynamic data structures.

Transactions are *atomic*, i.e. their effect (computations, memory accesses, etc) is visible to the system as a whole if the transaction commits, while no change to the system occurs if the transaction aborts. Moreover, a transaction runs in *isolation* that means that any stores within the transaction are not visible until the transaction commits. In order to implement TM, the system needs to book keep memory references done by a transaction, and compare these references with those of other outstanding transactions when they try to commit. Thus, the key mechanisms of a TM system are *data versioning* and *conflict detection.* These mechanisms can be implemented in software, hardware or in hybrid fission. The data versioning mechanism needs to keep different versions of data at the execution of various transactions. If a data store is written immediately to the target memory location, the approach is called *eager versioning,* while if the system handles copies of the data stored; the approach is called *lazy versioning.* As far as conflict detection is concerned, there exist two approaches as well. The system may either choose to check for conflicts during the execution of a transaction, as reads and writes occur (*pessimistic conflict detection*) or before the transaction attempts to commit (*optimistic conflict detection*). All techniques have advantages and disadvantages, and their efficiency depends on the memory access patterns of each individual application.

5.4.2 *Current implementations*

At the moment, most TM systems have been implemented in software. Software Transactional Memory (STM) systems extend programming languages or software libraries to incorporate the assets of TM and require no hardware modifications. However, it is a common view that although these systems constitute an important starting point for TM systems, they will need hardware support to alleviate very frequent STM operations from the significant runtime overheads.

Intel's STM Compiler Prototype Edition 3 implements the support for STM ([30]) that allows the programmer to use STM language extensions in an "OpenMP like" way. Sun has presented Transactional Locking 2 ([31]), which integrates STM implementation with reasonable performance, providing a programming environment similar to that attained using global locks, but with an improvement in performance. TinySTM is the European effort for STM based on fine-grained locking at byte level. The Rochester STM (RSTM, [32]) is a C++ library for multi-threaded, non-blocking transaction-based code. The OpenTM Transactional API ([33]) is a high-level API that extends OpenMP with memory transactions.

In Hardware Transactional Memory (HTM) the hardware manages data versions and tracks conflicts transparently as the software performs regular read and write operations. In order to implement HTM, one needs to modify the cache hierarchy and coherence protocols. Although HTM is expected to provide better performance, it is not problem-free. At first, the modifications that need to be done in hardware are neither straightforward, nor cost-effective. In addition, hardware implementations are not able to provide infinite resources, thus long transactions with many memory-access operations will not be supported entirely in hardware.

Rock is a multi-threading, multi-core, SPARC-family microprocessor currently in development at Sun Microsystems. Sun has confirmed that Rock will be the first processor to implement HTM. Stanford's Transactional Coherence and Consistency (TCC) has two prototypes: ATLAS and PLUTO. ATLAS is the first implementation of the TCC architecture. It uses PowerPC 405 processors with a modified data-cache that implements version management and conflict detection for transactional execution. The current design is mapped

on a single BEE-2 board, which allows for up to 8 processors. PLUTO is a TCC prototype based on the M32R embedded architecture. M32R is designed by Renesas Technology, which is a fully synthesizable core coded by Verilog-HDL and can be programmed to FPGA directly. It provides an efficient platform for supporting architecture development. PLUTO uses 2 M32R processors each with a modified data cache that implements version management and conflict detection for transactional execution. The two symmetric CPU cores are connected via an internal on chip bus. The bus arbiter supports TCC protocol such as commit, PhaseID controller. The new architecture has Exception/Interrupt for handling TCC overflow and violation. Finally, Log-TM is a hardware TM system designed and proposed by the University of Wisconsin based on the concept of per-thread logs that decouples version management from L1 cache tags and arrays. With LogTM, a transactional thread saves the old value of a block in a per-thread log and writes the new value in place using eager version management.

Overall, Transactional Memory seems a very promising novel parallel programming model incorporating concepts that well adapt to the architecture and behaviour of modern multi-core architectures. It is expected to provide meaningful performance improvements in applications that suffer from excessive synchronization, especially in the cases where this synchronization is substantially used to preserve the semantics of the original algorithm but the actual cases that two threads conflict for some resource are extremely rare. This is the case, for example, of the concurrent access to irregular data structures (e.g. databases, lists, trees, priority queues) where standard programming paradigms either lock the whole structure essentially serializing the access (coarse-grain locking), or lock small parts of it (fine-grain locking) suffering from the killing overheads of synchronization. TM is still under research and experimental assessment that has indeed verified its high productivity and performance benefits (speedup and scalability) for several classes of applications. However, at the moment, only STM systems and simulations of real HTM systems have been used. The general conclusion of the TM community is that only with hardware support has this programming paradigm chances to provide its performance benefits. It remains to be seen whether the silicon industry is eager to adopt and support this model, by providing the essential extensions to the current micro-architectures.

6 Languages, Paradigms and Environments for Hardware Accelerators

Green-IT is a recent trend in HPC. To ensure that the demand for higher peak-performance can be satisfied without a huge increase in the power envelope, special-purpose hardware has to be used. Computational hardware accelerators ensure that no extra energy or die area is wasted for unnecessary transistors or sub-optimal processor layouts. Their layout especially fits to certain (scientific) algorithms.

The necessity for hardware accelerators is obvious. But even if they have been available for several years, their biggest drawback, the programmability of these devices, has prevented a widely use.More and more success stories of early adopters of these technologies are spreading at conferences and in the HPC community, but very often, the reported codes are only test implementations of simplified mathematical kernels. Only very few scientific domains are actually using hardware accelerators for production runs, mostly those that have been used to adopting their codes to new platforms at the assembly code level. However, more and more accelerated systems are available throughout Europe and several more will be installed in the following years.

To make the power of accelerators available for the majority of HPC customers in Europe, programmability of special-purpose hardware accelerators needs to increase dramatically. A few attempts to accomplish this are coming from the hardware vendors themselves; some of them are targeting OpenMP compilers or at least OpenMP-like compiler directives to harness the massive parallelism of their devices (e.g., IBM for Cell). Other attempts are coming from software and traditional compiler companies that have developed language add-ons to simplify writing portable codes, e.g. RapidMind, CAPS HMPP or the new PGI compiler with GPU support. Another attempt comes from the Khronos Group, a standardisation organization that released the OpenCL standard in late 2008. OpenCL was meant to be a common interface to all different kinds of accelerators but has so far only been adopted by the typical GPU vendors AMD/ATI and NVIDIA. A standardized interface like OpenCL is one of the necessary steps to a wide adoption of hardware accelerators. Intel's announcement of the "Larrabee" GPU that consists of standard CPU cores instead of the highly tailored GPU cores of previous devices, received much attention from the HPC community. It is hoped that the regular Intel software tool chain can be used to program these devices for higher programmability, flexibility and performance, which is extremely necessary in environments where many legacy codes, consisting of hundreds of thousands of lines of code, are being used. In HPC software has usually outlived the hardware.

The available accelerator hardware within the PRACE prototypes are:

- Cell processors (especially the PowerXCell 8i double precision version that is being used in Roadrunner - the world's first Teraflop/s system);
- ClearSpeed e710 accelerator boards and ClearSpeed CATS units;
- NVIDIA GPUs and their high-performance GPGPU Tesla boards;
- FPGAs (Virtex-4 from Xilinx);
- Intel Larrabee GPUs.

Additional to the hardware, the access to the CAPS HMPP compiler and RapidMind licenses have been available through WP8 prototypes. Please see "Annex B: Hardware Overview" for more details on the systems used for development and benchmarking. Intermediate results on

the prototypes are available in the (PRACE-restricted) Deliverable D8.3.1 and final prototype evaluation reports will be included in the public Deliverable D8.3.2.

One of the goals of Task 6.6 is to find useful ways to offer the extreme computing power of hardware accelerators to a broader community. It is therefore necessary to compare not only the different hardware itself but also the ease-of-use from a programmer perspective. The remainder of this Chapter will give an overview of all available accelerator languages and paradigms. We will firstly review what programming languages (Table 5) and libraries are available for certain accelerators followed by detailed introductions to the languages.

	NVIDIA GPUs	AMD/ATI GPUs	IBM Cell	ClearSpeed CSX 700	X86 Multi-core
RapidMind					
CAPS HMPP					pthreads, SSE
CUDA					
Brook					
OpenCL				targeted 2010	
Cn					
PGI Compiler		through OpenCL			
IBM Cell SDK					
StarSs					

Table 5: Availability of programming languages for x86 multi-core and hardware accelerators

Assessment of the libraries

A short overview of available libraries for accelerators can be found in Table 6 below.

	NVIDIA GPUs	AMD/ATI	IBM Cell SDK	ClearSpeed
BLAS	CUBLAS	ACML	libblas	CSXL (only Level 3)
LAPACK	-	ACML	liblapack	CSXL (subset only)
FFT	CUFFT	ACML	libfft/libfft3d	CSFFT (1D, 2D, real, complex)
RNG	-	ACML	libmc_rand	RNG library

Table 6: Vendor implementations of mathematical libraries for hardware accelerators

Libraries can be used to measure acceleration of basic mathematical functions. However, as long as the transfer of data to the accelerator is the main bottleneck, the use of these libraries is restricted to very few codes that are able to automatically reuse the data on the accelerator. Using accelerators only by calling mathematical library functions does not deliver a significant performance impact for most real world applications.

Further studies ([34]) indicate that performance comparisons of two different libraries (in this case ACML for an AMD/ATI FireStream GPU vs. CUBLAS for NVIDIA card) yield only insight in the maturity of the libraries and does not give reliable figures on neither the underlying hardware nor on the programming model used for the specific accelerator.

6.1 Cell programming

The CELL processor consists of one PPE (Power PC Processor Element) and 8 SPEs (Synergistic Processor Elements). The PPE runs the OS and controls the SPEs. The SPEs are optimized for running compute-intensive applications and access data on the PPE or other SPEs via the EIB (Element Interconnect Bus) using DMA transfers.

Beyond assembly language level, the most powerful approach is building applications with IBM's Cell Software Development Kit (SDK). Since this is a non-portable dual-source approach and the creation of the threads that run on the SPEs and all DMA data transfers must be explicitly initiated, code overhead and developing efforts are quite huge. The most important library in the SDK is the SPE runtime management library (libspe), which provides functions for the management of the SPEs by the PPE. Both the IBM XL C/C++ & Fortran Compiler for Multi-core Acceleration and GCC support the SDK. Using the SDK at least 2 different source files have to be written: the SPE source file which consists of the code that will run on the SPEs and the PPE source file which consists at least of routines to create a context and a thread for each SPE.

For SIMD programming of 128 bits wide vectors the following libraries are provided:

- SIMD programming on the PPE:
  - VMX (AltiVec) library that provides functions for general-purpose vector processing and vector mathematics;
  - SIMD Math library that provides advanced mathematical functions on top of the AltiVec library, operating on vectors;
  - MASSV library that provides advanced mathematical functions, operating on arrays.
- SIMD programming on the SPE:
  - SPE intrinsic that provide the SPE's basic math and logical operations;
  - SIMD Math library that provides advanced mathematical functions, operating on vectors;
  - MASSV library that provides advanced mathematical functions, operating on arrays.

To simplify the development process the Accelerated Library Framework (ALF) was introduced with IBM SDK 3.0 ([35]). This processing environment handles many of the low-level details of SPE management. ALF supports the MPMD programming model and provides an abstract structured framework for offloading computationally intensive work to accelerators. This toolset was designed to manage applications on many different kinds of host-accelerator based systems. The ALF runtime library handles the task management and DMA transfers, while the programmer only has to focus on the implementation of the computational kernel and the data partitioning.

The recent version of the IBM XL C/C++ compiler can compile and link both SPE and PPE code segments with a single compiler invocation without the need of having 2 different source files. OpenMP pragmas mark regions of the PPE code which should be run in parallel on the SPEs. This writing easily portable programs with low development efforts.

The IBM SDK offers PPE implementations for all standard BLAS routines, but only the following BLAS routines have been optimized to use the SPEs:

- Level 1: S/DSCAL, S/DCOPY, IS/DAMAX, S/DAXPY, S/DDOT, DASUM, DNRM2, DROT
- Level 2: S/DGEMV, S/DTRMV, S/DTRSV, DGBMV, S/DGER, DSYMV, DTBMV, DSYR

- Level 3: S/DGEMM, S/DSYRK, S/DTRSM, S/DTRMM, S/DSYMM, S/DSYR2K

Concerning LAPACK, PPE implementation for all C/S/D/Z LAPACK routines exists, but only the following routines have been optimized to use the SPEs: DGETRF, DGETRS, DGETRI, DGEQRF, DGELQF, DPOTRF, DPOTRS, DBDSQR, DSTEQR, DGESVD, DGESDD. 1D+2D+3D single precision and 2D-3D double precision FFT are available for the PPE and only one set of SPE APIs for 1D single precision FFT is provided.

6.2 CUDA (by NVIDIA)

CUDA [36][37], the Compute Unified Device Architecture, is a parallel computing architecture developed by NVIDIA that enables scientific programming for NVIDIA GPUs. It includes the CUDA Instruction Set Architecture (ISA) and the parallel compute engine in the GPU.

6.2.1 *Programming Model*

The programming model of CUDA differs significantly from standard single- and multi-thread CPU model. To a CUDA programmer, the computing system consists of a host that is a traditional Central Processing Unit (CPU) and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units. The CUDA language is strictly related to the architecture of the modern NVIDIA GPU's, which consists of many Multiprocessors, each containing 8 streaming processors. The CUDA devices accelerate the execution of these applications by harvesting a large amount of data parallelism. Figure 2 clearly shows the differences between CPUs and GPUs.

Figure 2: Differences between CPUs and GPUs

The programming model behind CUDA is focused on the idea of *kernel*. A kernel is a function callable from the host and executed on the CUDA device - simultaneously by many threads in parallel. The host calls kernels use data resident on the NVIDIA card. The device code is written using ANSI C extended with keywords for labelling data-parallel functions, called kernels, and their associated data structures. The NVIDIA compiler, NVCC, internally divides the device code from the host code; the device code is compiled by a normal C++ compiler (like GCC on Linux, Microsoft compiler on Windows), the host code is first transformed to PTX code (an intermediate language) and then to GPU's assembler; at the end of the compiling process, the two parts are linked together to only one executable which contains both CPU and GPU code. In situations where there is no device available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU.

The best way of obtaining good performance with such architecture is to have a programming model based on SIMD (Single Instruction Multiple Data), because the many streaming processors (up to 240 with the current models) can execute the same code on different data. When a kernel is invoked, or launched, it means that it is executed as a grid of parallel lightweight threads. The creation of threads on the GPU is hardware accelerated, so the overhead is negligible. So the best way of obtaining the maximum performance is to load the chip with hundreds and (even better) thousands of small threads that can be executed in parallel on the Streaming Processors. Typically a small thread corresponds to the code inside "for" loops in regular C code that can be vastly executed in parallel on a GPU instead of sequentially.

A kernel launch requires the explicit specification of an execution configuration to define the number of concurrent threads. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism. Each of the threads that execute a kernel is given a unique *threadID* that is accessible within the kernel through the built-in variable. Threads are logically organized in *blocks* and blocks are organized inside a *grid*. This two-level hierarchy is illustrated in Figure 3. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or field. Threads within a block can cooperate among themselves by sharing data through some shared memory (discussed in the next paragraph) and synchronizing their execution to coordinate memory accesses.

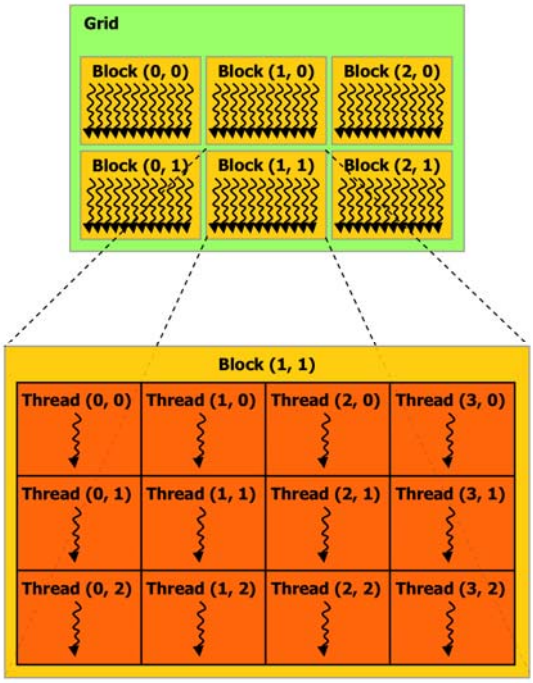

Figure 3: CUDA two-level hierarchy of thread organization

6.2.2 *Memory Model*

In CUDA, host and devices have separate memory spaces. This reflects the fact that devices are typically hardware cards that come with their own Dynamic Random Access Memory (DRAM).

In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer the required data from the host memory to the allocated device memory. Similarly, after device execution, the programmer needs to transfer the result data back to the host and free up the device memory allocated that is no longer needed. The CUDA runtime system provides APIs to perform these activities for use by the programmer.

Figure 4 shows an overview of the CUDA device memory model. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global device memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

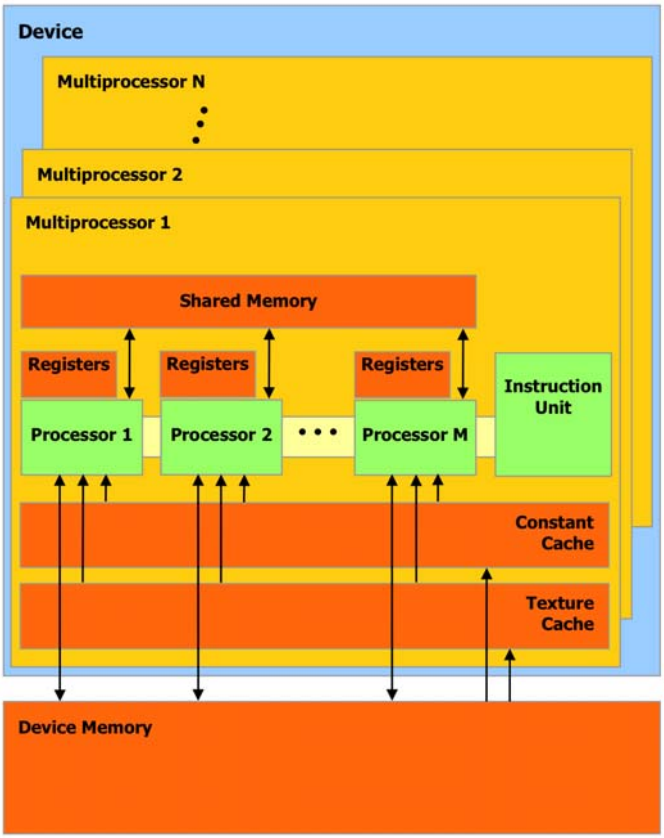

Figure 4: CUDA memory device model

It is really important to point out that different kinds of memory reflect different capability in term of access and latency. For example the shared memory can be as fast as a register when there are no bank conflicts or when reading from the same address or global memory is potentially more than 100x slower than shared memory and it is accessible from either the host or device.

6.2.3 *Current support and libraries availability*

CUDA is currently implemented as an extension to C and it supports a range of computational interfaces including OpenCL and DirectX Compute. Third party wrappers are also available for python, Fortran, Java, .NET and MATLAB. The CUDA toolkit is available for Linux, Windows and OSX.

CUDA makes available the extreme parallelism on current graphic cards but puts most design-decisions on the programmer. Memory transfers between host processor and GPU have to be orchestrated by the programmer as well as the slicing of the data to threads, thread blocks and grids of thread blocks. Furthermore it is necessary to optimize memory access to gain acceptable performance, which puts another burden on the programmer. Translating some C code to CUDA may not be difficult, depending on the type of algorithm, but low level knowledge of the hardware is required to optimize the code and squeeze every bit of

performance out of the GPU. Another consideration that has to be made on the current NVIDIA GPU's is that double precision performance is quite low compared to single precision floating point performance, this is due to the fact that every Multiprocessor has 8 single precision units, and 1 double precision unit. This performance gap can be seen on the Tesla product line technical specifications.

CUDA is not only used in HPC, it is mainly used for graphics programming and has a broad developer community, with many code snippets, examples and tutorials on the web. A few very promising code porting examples have been undertaken by HPC communities. The latest CUDA Toolkit, number 2.3 released July 2009, includes a new and improved profiler and debugger for the GPU, as well as complete BLAS (CUBLAS) and FFT libraries (CUFFT). CUBLAS and CUFFT support real and complex computation both in single and double precision.

It is not easy to understand or predict the future evolution. NVIDIA collaborates with a lot of partners, starting from PGI to independent companies like CAPS HMPP (see next Section 6.4), around the world to improve its toolkit and to spread the utilization of its proprietary architecture for HPC purposes. Recently Portland Group has released a new version (9.0) of its compilers and development tools. The update includes support for the PGI Accelerator programming model on x64-based Linux systems utilizing NVIDIA GPUs. The PGI Accelerator Fortran and C compilers are designed to automatically analyze a program's structure and data as well as to split portions of the application between a multi-core x64 CPU and a GPU. OpenMP-like compiler directives inside the code control the portions split between each device. More details can be found on the official website ([38]).

6.3 OpenCL

OpenCL (Open Computing Language, [39]) is a framework for writing programs that execute across platforms consisting of CPUs, GPUs, DSPs and CELL processors. OpenCL is being created by the Khronos Group with the participation of many industry-leading companies following the idea to create royalty-free standard for cross-platform heterogeneous parallel programming.

All the relevant information reported have been extracted from OpenCL technical specification. Version 1.0 has been released by the Khronos Group [40] on December 8, 2008.

6.3.1 *OpenCL Architecture*

The OpenCL architecture model is represented in Figure 5. The model consists of a host connected to one or more OpenCL devices. An *OpenCL device* is composed by one or more *compute units* (CUs) which are further divided into one or more *processing elements* (PEs). Computations on a device occur within the processing elements. Although OpenCL has been designed as a standard for many different accelerators, the dominance of GPU vendors is evident.

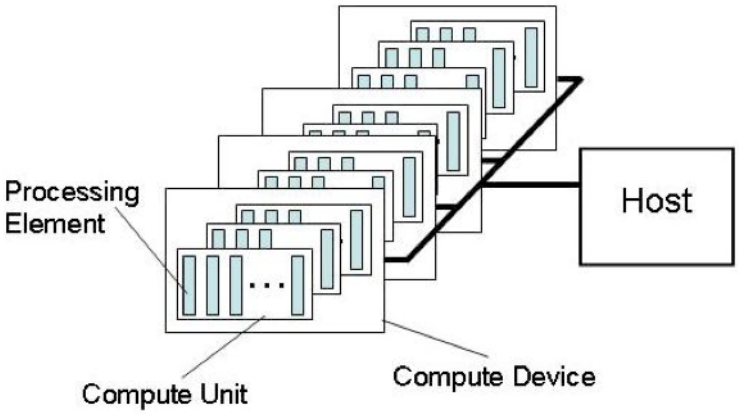

Figure 5: OpenCL Architecture Model

The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units (execute in lockstep with a single stream of instructions) or as SPMD units (each PE maintains its own program counter).

6.3.2 *Execution Model*

Execution of an OpenCL program occurs in two parts: *kernels* that execute on one or more OpenCL devices and a *host* program that executes on the host. OpenCL allows calling not only OpenCL kernels but also hardware specific kernels (called *native* kernels), e.g. written in CUDA, and hardware specific enhancements for certain architectures which undermines the portability. The host program defines the context for the kernels and manages their execution through *queues*. The core of the OpenCL execution model is defined by how the kernels execute.

When a kernel is submitted for execution by the host, an *index space* is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a *work-item* and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item. Work-items are organized into *work-groups*. The work-groups provide a more coarse-grained decomposition of the index space. The result is a model that has a strong similarity with the CUDA ones (Figure 6).

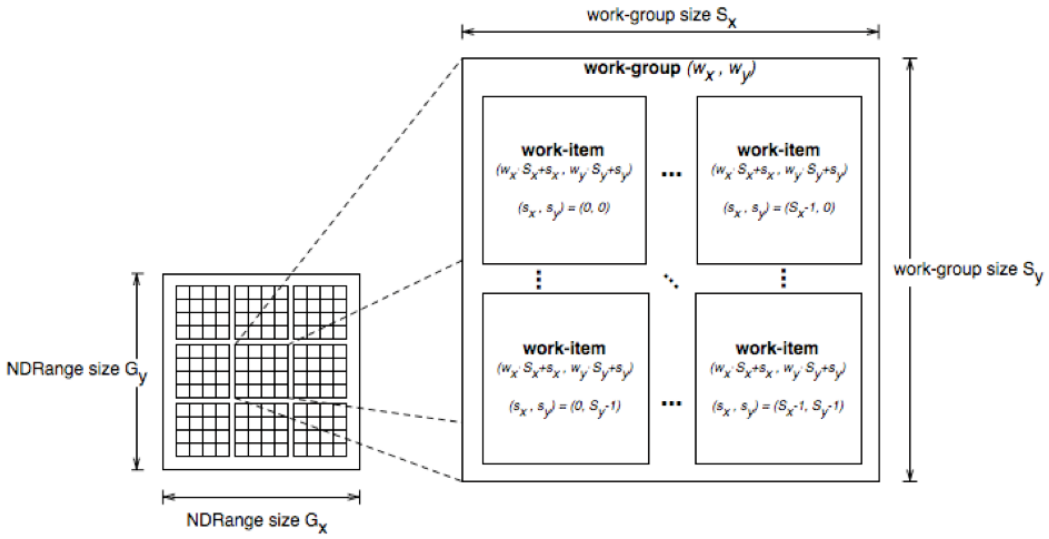

Figure 6: OpenCL Work-Groups and Work-Items organization

A wide range of programming models can be mapped onto this execution model. OpenCL explicitly support two of those models: the *data parallel* programming model and *the task parallel* programming model.

6.3.3 *Memory Model*

OpenCL standard defines four distinct memory regions:

- *Global Memory*: this memory region permits read/write access to all work-items in all work-groups.
- *Constant Memory*: a region of global memory that remains constant during the execution of a kernel.
- *Local Memory*: a memory region local to a work-group.
- *Private Memory*: a region of memory private to a work-item.

The memory regions and how they relate to the platform model are described in Figure 7.

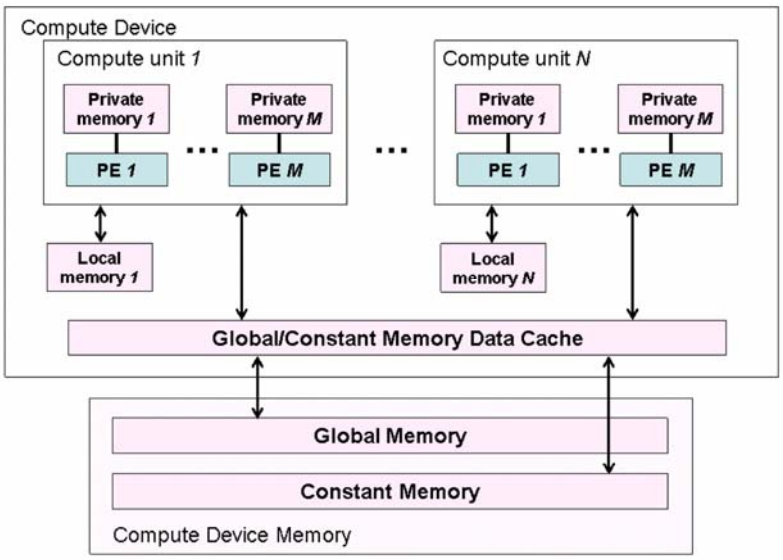

Figure 7: OpenCL Memory Model

The host and OpenCL device memory are, for the most part, independent of each other. This interaction occurs by explicitly copying data (in a blocking or non-blocking way) or by mapping and un-mapping regions of a memory object (always using OpenCL API).

OpenCL uses a *relaxed consistency* memory model (like OpenMP); i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times. Memory consistency is enforced at a synchronization point.

6.3.4 *The OpenCL framework and runtime*

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

1. OpenCL *Platform* layer: The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts.
2. OpenCL *Runtime*: The runtime allows the host program to manipulate contexts once they have been created.
3. OpenCL *Compiler*: The OpenCL compiler creates program executables that contain OpenCL kernels.

The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism. A few constructs and data types are directly taken from OpenGL and some reserved data types that will be advantageous for scientific computing like quads, complex or imaginary data have not made it into 1.0.

6.3.5 *Portability and know limitations*

The standardization of OpenCL has been initiated by Apple and was done together with teams from AMD, Intel, IBM, NVIDIA and software companies like RapidMind and others. Both NVIDIA and AMD have showcased demonstrations of OpenCL on their GPUs in December 2008 at SIGGRAPH Asia. While AMD is planning to replace their hardware-specific GPU programming language Ctm (Close to Metal) with OpenCL, NVIDIA seems to stick to CUDA and implement OpenCL as another language to use CUDA. A first compiler is expected from AMD in 2009. Other companies, like Petapath, are planning to join the Khronos Working Group on OpenCL. Therefore, it seems that the PGI-Compiler might use OpenCL in the future to deliver code that is able to run on all available GPU/OpenCL devices.

6.4 CAPS HMPP

The Hybrid Multi-core Parallel Programming workbench (HMPP, [41]) provides developers with a set of tools dedicated to build parallel hybrid applications that run on many-core systems. These architectures combine general-purpose cores with hardware accelerators (HWAs) such as GPUs (NVIDIA and ATI) or SIMD computing units (SSE2).

The HMPP workbench is a programming environment from CAPS [42] that includes a C and Fortran meta-compiler, hardware-specific code generators and a runtime that use the hardware vendor development tools and drivers.

6.4.1 *Programming interface*

Based on a set of OpenMP-like directives that preserve legacy codes, HMPP directives ensure that the application keeps portable and that hardware specific computations are separated

from the main functionality. For instance, targeted to CUDA, HMPP expresses which parts in the application source should be executed on a NVIDIA card.

The types of HMPP directives are the following:

- *Codelet*: define a function as hardware-assisted.
- *Execution*: specify the codelet remote execution in the program.
- *Data transfers*: data can be uploaded before the execution of the codelet and data download points can be inserted after the execution.

In detail, a *codelet* directive declares a computation to be remotely executed on a hardware accelerator. The favoured type of hardware accelerator can optionally be indicated, otherwise HMPP will take the first available compatible accelerator. For an efficient use of the accelerator, the computation can be specialized to the type of the parameters, their size, value etc. HMPP does not parallelize a sequential code by itself. Furthermore, if the hardware accelerator is not available for any reason, the legacy code still can be executed and the application behaviour is unchanged. Codelet execution can be synchronous or asynchronous.

HMPP can be seen as programming glue between target specific programming environments and general-purpose programming. This is especially useful in combination with different implementations for one codelet, e.g. an accelerated version used only for large vector sizes.

6.4.2 *Description of the runtime*

The HMPP runtime is the dynamic library in charge of the correct execution of the remote procedure calls to the HWA. Bound to the application, this library allocates and initializes the HWA in order to allow the execution of the codelets. It relays communications between the host and the HWA and manages the asynchronous execution of the codelets.

A HMPP application can be compiled with an off-the-shelf compiler and run without any specific runtime to produce a conventional native binary. The HMPP annotated application source is parsed by the HMPP pre-processor to extract the codelet and to translate the HMPP directives into calls to the HMPP runtime. More over, thanks to its dynamic linking mechanism, a HMPP application is able to make use of either a new accelerator or an improved codelet without having to recompile the application source. In this way, a single binary could run on various hardware platforms. HMPP can therefore be used complementary to OpenMP and MPI and is a good solution to accelerate parts of legacy codes. Compilation is done within one line, the HMPP compilers then internally does two main passes, one for the host application and one for the hardware-accelerator code, called *codelet production*.

During the codelet generation, the developer has the possibility to inspect the source code of the codelet and it is able to modify the specific code related to the accelerator before continuing with the next step of compilation and the final linking. HMPP is also able to select at runtime most efficient implementation according to an explicit expression (data size, restrictions, ...) inside the code, an environment variable or a dynamic mapping file.

HMPP generates the real target code, CUDA for example. This permits the expert programmer to have a look at the target code and optimize it manually; so HMPP can be used to create a mainframe of the target code and proceed for manual coding skipping the trivial and time consuming phases of coding (for example in CUDA kernel definitions, copy of memory from and to the host, and so on). Figure 8 shows synthetically how the compile process for CPU+GPU code is composed and how all steps interact with each other.

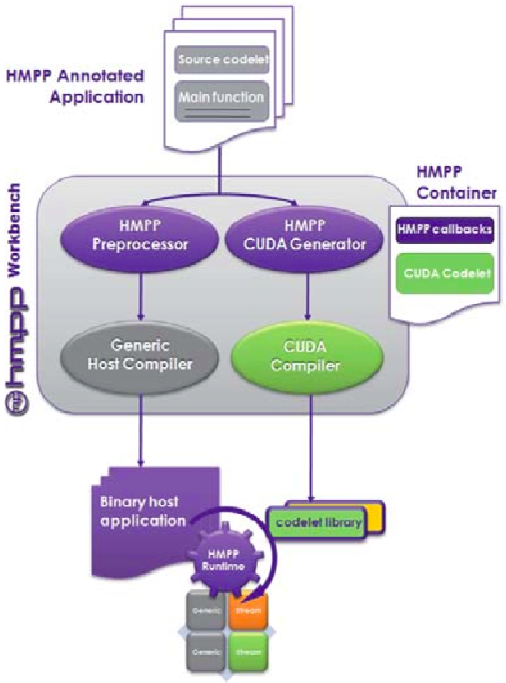

Figure 8: HMPP code generation flow

6.4.3 *Current implementation.*

The HMPP compiler can currently be used to accelerate code on AMD/ATI, NVIDIA GPUs and x86 Multi-core CPUs (using also SSE instructions). HMPP 2.x allows users to pipeline computations in multi-GPU systems and makes better use of asynchronous hardware features to increase performance of GPU accelerated applications. HMPP 2.x fully supports AMD FireStream hardware (supporting BROOK+ 1.0) with a CAL/IL code generator. The addition of an OpenCL code generator is another major milestone; this will give HMPP developers another powerful standard programming option.

The HMPP codelet generators do not handle the full C language but only a subset of C99 standard. Some restriction aims at ensuring a large portability of the code on most HWAs. As for C, the codelet generators do not support the full FORTRAN language. Refer to the HMPP Workbench User Guide to an exhaustive overview of limitations.

At the moment HMPP requires a valid license in order to compile and produce codelets. CAPS HPMM is compatible with GNU GCC and the INTEL icc/ifort compilers.

6.5 RapidMind

RapidMind [43] started in 2004 based on the academic research related to the “Sh” project [44] of the University of Waterloo. The project originated from the idea of doing “general-purpose computing on graphics processing units”, now referred to as “GPGPU”. It started at a time, when the first “programmable” GPUs became available and the only way to program these devices was through the use of shading languages. RapidMind has then subsequently added the Cell processor backend and the x86 backend with the rise of multi-processor CPUs for the consumer market. Today, the "RapidMind Multi-Core Development Platform" is the only commercially available product that allows automatic code generation for multi-core chips from Intel and AMD and for hardware accelerators like GPUs and Cell.

6.5.1 *Architecture and programming model*

The RapidMind Multi-Core Developer Platform adds special types and functions to C++, to allow data stream programming. Data stream programming looks somewhat similar to FORTRAN array operations, for those who are familiar with this construct. However, RapidMind arrays and vectors are much more powerful since the "array operation" is freely programmable. For suitable problems, this allows to write very short and clean programs.

The RapidMind programming model is very distinct from other well known programming and parallelization models. Adapting to this concept needs a real transition, like going from pure C programming to object-oriented C++. Its basic concept, called "data-stream processing", is quite powerful to express parallelism in the language and therefore yield (theoretically) good performancel gains on highly multi-threaded devices. The programmer can easily describe the data dependencies and data workflows and inherently adds all information necessary to do an efficient parallelization. The compiler and the runtime environment then have maximum information to decide how to auto-parallelise the code efficiently and create optimized binary code for the specific hardware detected. RapidMind features a just-in-time compiler and an optimized load balancer. Any information on the existence of threads is perfectly hidden from the user. Figure 9 shows how the runtime Treats the code.

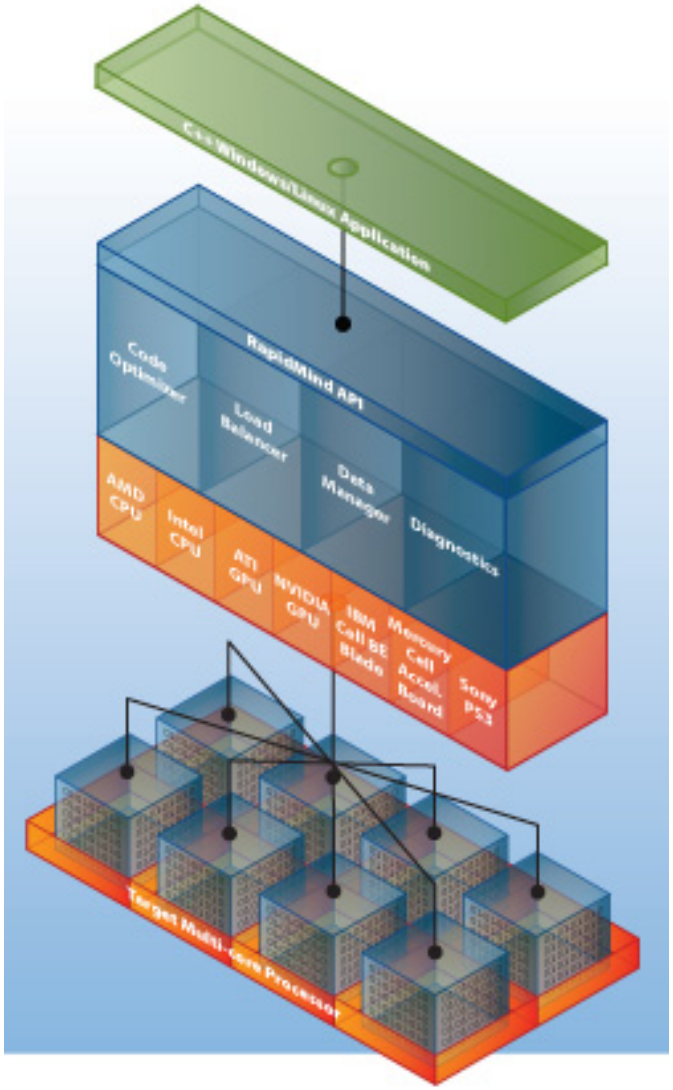

Figure 9: RapidMind runtime

RapidMind pursues the strategy of extracting data-parallelism instead of task parallelism that is the scalable parallelization model for many-core chips. The concept allows automatically parallelization on different hierarchies, e.g. on the SIMD-unit-level and the multi-core level at the same time. It seems that this model could be pushed forward to even replace MPI on the cluster level, although this idea is neither implemented nor on the roadmap.

RapidMind does not offer a math library for BLAS or FFT operation. There has been some investigation in this direction and although there is no actual product, many example codes are available from RapidMind upon request. A publication from 2006 reports on the performance of an SGEMM implementation, a 2D-FFT and a Monte-Carlo algorithm ([45]). The performance achieved in that paper was comparable to the best available GPU implementations at that time. However, since then, RapidMind seems to have not been able to keep pace with the rise of CUDA.

6.5.2 *Portability and known limitations*

RapidMind is purely based on C++ data types and templates. There is no Fortran interface available. C++/Fortran mixed programming is the only way to use RapidMind for Fortran legacy codes, a very error-prone programming model. Theoretically the RapidMind programming model would very well suit the Fortran array operations, but the necessary port will only be done if RapidMind sees a real business case or if someone is willing to pay for it.

In July 2009, RapidMind released v4.0 which is the first version that supports NVIDIA Tesla boxes through the new CUDA backend. Up to now, RapidMind supported GPUs through an OpenGL Shading Language backend, which allowed running on practically every programmable GPU, except Tesla. Tesla has been designed for the HPC market, and although it contains regular GPUs it does neither support graphical output nor shading languages. Starting from version 4.0 RapidMind now supports double-precision accuracy on the GPU, however the current release is not able to reach the performance of the CUBLAS implementation. The next release version 4.1, expected for late Q4'09, will put an emphasis on "CUDA performance".

The latest release is also the first that supports double precision for the PowerXCell8i processor. Although it should theoretically be possible to run any RapidMind code on all supported back-ends, some limitations apply when programming for Cell. There are more portability issues if the code has been tweaked to deliver high performance on the GPU. Our investigation showed that the performance of RapidMind on Cell is generally poor.

The performance of the x86 backend is not so bad, taking into account that firstly, the Intel compiler and Intel MKL are really the leading compiler and math library for Intel platforms. Secondly, the x86 backend is a comparably new backend and has only been introduced in Q4'08. The gap between MKL and RapidMind of a factor of 2 on Nehalem EP for our example codes will hopefully shorten in the future. RapidMind Inc. has recently been bought by Intel and is now working on a merge of RapidMind technology with the quite similar approach of Ct. Ct stands for "C for throughput computing" and is Intel's new language designed for the use of many-core processors.

6.5.3 *Future Developments*

The fact that RapidMind is now part of Intel has lead to many speculations: Some people are concerned, that the RapidMind Development Platform might no longer support non-Intel products, others suspect, that -in the long run- it might not be supported at all. RapidMind Inc. announced that they plan to support RapidMind and stick with the roadmap for the next

releases. However, it seems that including RapidMind technology in Ct might lead to a very powerful new programming language supported by one of the leading chip vendors, which might be able to attract a wider audience and has the potential to become the language for the multi-core era.

Hopefully this product will support other platforms. When looking at the number of NVIDIA GPUs that has been sold, it seems to be mandatory to keep supporting at least CUDA-enabled cards. RapidMind is part of the consortium that released OpenCL and is planning to support OpenCL in one of its later releases.

6.6 CellSs (StarSs programming models family)

StarSs is a parallel programming model for multi-core processors aiming at high portability of application to any target hardware. Ideally it can be extended to other multi-core processors or symmetric multiprocessors since it separates the underlying processor's characteristics from the programmer.

The current implementation of StarSs for Cell processor is called CellSs [46][47]. CellSs makes use of simple annotations (*pragmas*), which represents a tasks' current state, to a sequential code written in C language. Execution of the tasks on the SPE is identified by these annotations without any collateral effects. A source-to-source compiler generates code for both PPE and SPE. The system then executes the tasks concurrently on SPEs and resolves dependency.

There are three types of pragmas:

- *Initialization* and *finalization* pragmas: these pragmas indicate the start (`#pragma css start`) and the end (`#pragma css finish`) of application execution. They are optional and are annotated by CellSs, if not present in the user code, at the beginning and end of the application.
- *Task* pragmas: these pragmas are inserted before a function in an application. They are executed on an SPE (Synergistic Processing Element). It specifies the input, output or inout with size of arrays or matrices, if any.
- *Synchronization* pragmas: the synchronization pragma css wait enables access to the data that are generated by a task. As in OpenMP, there is a barrier directive (`#pragma css barrier`), which forces the master thread to wait for the completion of all generated tasks until now. However, this kind of synchronization is too course grain and in many case can be counter productive. To achieve a finer grained control over data readiness, wait on directive is also available.

The CellSs compiler generates two files: a main program, which is to be run on a PPE and a task code, which is to be run on an SPE. For each of the task code, an adapter function for the annotated functions inside the task code is generated. At the beginning of the main code, the CellSs inserts the following application calls to CellSs runtime: calls to initializing and finalizing functions; calls for registering the annotated functions; and calls to a CellSs runtime library primitive (`css_addTask`) wherever a call to one of the annotated functions is found. The task main code calls these adapters, which is a part of CellSs runtime. These files are then compiled using GCC or IBM XL C compiler to generate a single binary.

The main program of the application (the main thread) runs on PPE. The CellSs runtime generates threads in SPE, which is defined by the environment variable (which is eight by default). The task program is initiated in these threads. Whenever the main program calls

css_addTask, the runtime adds a node representing the task in the task graph. If a data dependency exists between this and previously called task, then an edge between these tasks is added. The master thread also performs parameter renaming (and similar higher order optimization), thereby removing write-after-read and write-after-write hazards, greatly increasing the parallelism of the task dependency graphs. An additional thread called *helper thread* runs on PPE. The helper thread is responsible for scheduling and further managing the task dependency graph. A helper thread selects an idle SPE and assigns a task that has no dependency, to be run in it. The helper thread also specifies the location of the parameters needed for a specific task. The data is transferred between the main memory and the local memory of SPE with the help of CellSs SPE library. Figure 10 summarizes the runtime behaviour.

Once the task completes execution, the task program notifies the helper thread, which then updates the task-dependency graph according to the current situation and schedules new tasks for execution in idle SPEs. In order to avoid inter-task circular dependency while executing program in parallel and to reduce execution time of a bundle of tasks, double buffering is implemented. At the end of the execution of all of the tasks, the DMA out of the output data of the last task in the group is programmed, and the task program waits for all of the DMA outs to finish.

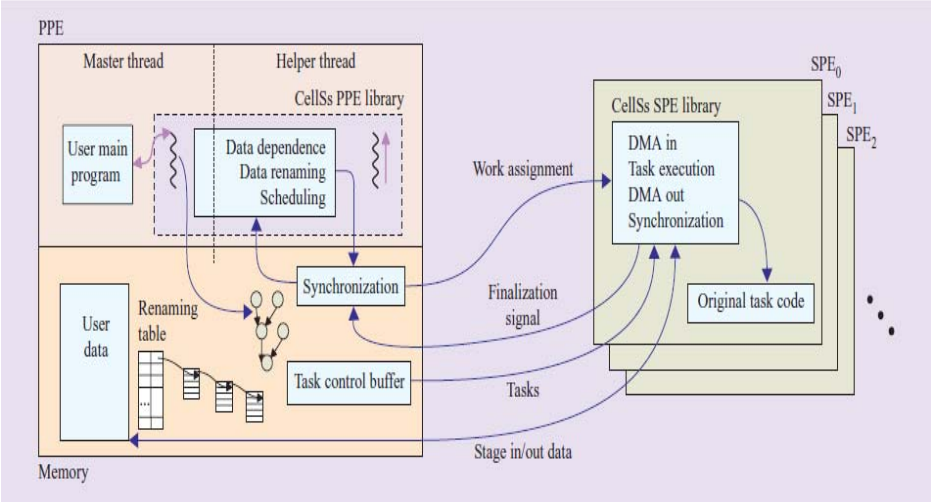

Figure 10: CellSs runtime behaviour

An effective tracing mechanism is implemented on CellSs and the traces are analyzed using Paraver, which provides detailed views of the entire application, as well as individual modules. Traces obtained aid in analysis of the CellSs runtime behaviour.

6.7 Cn for ClearSpeed accelerators

Although ClearSpeed [48] has joined the Khronos Group, which is in charge of standardizing OpenCL and there are plans to create an OpenCL interface for their accelerator boards, Cn is currently the only way to program for the Clearspeed platform.

The Cn language is based on ANSI C with extensions to support data-parallel processing. The main addition to standard C is the definition of mono (scalar) and poly (parallel) data types. Variables, function pointers and entire data structures can be marked as poly; it means that they are then processed in parallel on the array of 96 processing elements (PEs) in the CSX600 on the Clearspeed cards.

The ClearSpeed hardware accelerators are the only accelerators that have been designed for numerical simulations and HPC. They fully support 64-bit double precision and IEEE754

accuracy as well as full reliability through ECC protected memories. Their engagement in HPC is reflected in the graphical debugger and the visual profiler toolset, the most mature profiler that currently exist for hardware accelerators. The last hardware accelerator available is the CXS700 card ([49]).

Clearspeed provides its customers with an optimized subset of Level3 BLAS (DGEMM, ZGEMM, ZGEMM3M, DTRSM) and LAPACK routines as well as library calls for FFTs (CSFFT, 1D, 2D, real and complex.) and a Random Number Generation library to support applications such as Monte Carlo simulations. The supported LAPACK routines are: DGEQRF, DGESV, DGETRF, DGETRS, DORGQR, DORMQR, DPOSV, DPOTRF, and DPOTRS.

7 Assessment of the languages and porting activity

The overall aim of this evaluation is to find highly productive combinations of hard- and software for algorithms used in HPC. In order to accomplish this, an exhaustive comparison of different languages and different accelerators is needed. During the course of the PRACE project, the assessment of languages for hardware accelerators became a joint effort of WP6 and the "*Future Petaflop/s computer technologies beyond 2010*" working package 8 (WP8). WP8 issued a call on prototypes for emerging technologies and after a review and voting process, several prototypes targeting hardware accelerators or languages for hardware accelerators have been accepted, procured and installed. While the focus of WP8 lies on the hardware itself and only touches programmability and productivity issues, people involved in the WP8 prototypes teamed up with WP6 members to allow the assessment of several new emerging languages and the hardware acquired within WP8 at the same time. Detailed information on the different WP8 prototypes will become publicly available in D8.3.2 while the comparison of all languages is described in the remainder of this document.

7.1 The concept of productivity in the HPC context

The combination of performance and programmability is commonly referred to as "productivity" and has been studied during the DARPA High Productivity Computing Systems (HPCS) program. Reviewing the past shows that, although a bunch of high-level languages designed for usability and programmability exists; C and Fortran are mainly used to program high-end machines. C++ and Java are playing -if at all- only a minor role in HPC, although they are used by the majority of programmers worldwide and one might believe that a wide user community is a warrantor for good examples, good teachers, good design patterns, cookbooks, best practices, debugging tools etc. However, the dominance of C and Fortran in HPC reflects one important issue: What, in the end, one of the most important criteria in HPC, is the *performance* of the code. One of the assumptions for most of the "productivity research" carried out within the DARPA HPCS program was that performance is the main goal of every HPC programmer, and indeed 100% of the interviewees mentioned performance when asked about the top issues facing HPC programmers. Other important issues were correctness (90%), fault tolerance (70%), economical costs (20%), and portability (10%) ([50]).

To assess the productivity of a language, a good starting point is to look at software engineering metrics that are being used to determine the degree of difficulty a certain programming model puts on the programmer. In a second step we will use a model to show the trade-off that exists between performance and programmability and we will propose how to assess both during the porting work. Within Task 6.6 twelve languages and environments have been chosen for a detailed evaluation. These are: MPI+OpenMP, UPC, CAF, Chapel, X10, CellSs, CUDA, CUDA+MPI, OpenCL, RapidMind, Cn, CAPS HMPP. Out of these, seven candidates can be used to program for hardware accelerators, namely: CellSs, CUDA, CUDA+MPI, OpenCL, RapidMind, Cn, CAPS HMPP. Again, WP6 worked closely with WP8 to port 3 synthetic benchmarks from the EUROBEN benchmark suite to the target languages. These benchmarks were a dense matrix-matrix multiplication (mod2am), a sparse matrix-vector multiplication (mod2as) and a one-dimensional Fast Fourier Transformation (mod2f). These kernels were selected because are representative of the main computational applications analyzed in Chapter 2, as we can see in Table 7.

Scientific Area	Dense linear algebra (mod2am)	Spectral methods (mod2f)	Sparse linear algebra (mod2as)
Computational Chemistry	VASP CPMD GPAW Siesta CP2K	SIESTA VASP CPMD NAMD CP2K	SIESTA GPAW CPMD HELIUM
Computational Fluid Dynamics		N3D	ALYA N3D Code_Saturne
Condensed Matter Physics	QuantumESPRESSO VASP CPMD CP2K	SIESTA VASP CPMD QuantumESPRESSO CP2K	SIESTA CPMD
Earth and Climate Science		BSIT	BSIT
Life Science		NAMD	
Particle Physics	QCD		
Plasma Physics			TORB

Table 7: Mapping of Scientific applications on basic numerical kernels

This "porting activity" gave us the possibility to try to collect assess productivity information and performance information at the same time.

More information on the used input data sets is available in "Annex A: Reference Input Data Sets (RIDS)" on page 87. "Annex B: Hardware Overview" starting at page 91 describes the hardware that has been used for development and benchmarking. Detailed information on all languages as reported by the programmers is included in "Annex E: Reported Results for all Languages" while an analysis of these results and a full comparison of all languages is given in Section 7.3 "Porting Results".

7.2 Measuring Productivity

To assess the advantages and disadvantages of the new languages and to compare the languages with each other, it is important to take into consideration the findings of the DARPA HPCS program. The papers published from this program suggest various different definitions of and/or approaches to the term "productivity". A common finding is that for the area of HPC, the most important issues that define productivity are "*the time and effort required to write, debug and tune the code, and the performance of the code that results.*" ([55]).

Following the assumptions given in that paper results on the following definition

$$\mathbf{T(P) = I(P) + rE(P)}$$

where:

- **T(P)** is the total time to solution for a given problem P;
- **I(P)** is the implementation time;
- **E(P)** is the average execution time;
- **r** is a weighting factor that reflects the relative importance of minimizing execution time versus implementation time.

We compare the implementation and execution time of $\mathbf{P_0}$ with those of $\mathbf{P_L}$, the equivalent program in a new programming language. The relative power $\boldsymbol{\rho_L}$ of the language, measuring

its ease of use, is the ratio of implementation times, while the efficiency ε_L, measuring performance of the language, is the ratio of execution times:

- $\rho_L = I(P_0) / I(P_L)$
- $\varepsilon_L = E(P_0) / E(P_L)$

Generally, $\rho_L > 1$ and $\varepsilon_L < 1$ for high-level languages, reflecting the trade-off between abstraction and performance.

The "time-to-solution" should be the main focus. The way we will measure performance against time-of-development will allow higher-level languages to show off their potential for novices or for an easy access of accelerators and will give us at the same time an insight on the maximum performance one can achieve when using a certain language. The concept of the "time-to-solution" as introduced here, is only applicable to real applications. However, looking at the amount of languages that is now being promoted for HPC, it was not reasonable to start porting whole applications to the new languages without a solid understanding of the language characteristics and maturity. It was therefore decided to use small synthetic kernels for a first comparative analysis of the languages. The next step for a potential follow-on project should then be a in-depth analysis of a few (less than five) languages for real applications.

In order to assess the performance of a language versus the time of development, we firstly need a mechanism to determine the proper architecture(s) for a given Algorithm A.

> **Assumption I**: *For every algorithm A, which is -more or less- a direct result of a given problem P, exist one or more optimal architectures $Arch_1$ to $Arch_N$.*

The problem of mapping an algorithm to the architecture is not easy but can be solved, e.g. with performance modelling techniques for both the available hardware and the algorithm. It is obvious that the principle design decisions are common throughout all accelerators, e.g. most of them consist of many different simple cores. So, based on the categorization of the seven dwarves, there is only a subset of dwarves that is suited for accelerators. Stressing the differences in the accelerator hardware architectures, one should name this subset and subsequently test each of the algorithms for their suitability for a special accelerator. A few test implementations and an overview of available codes and their performance should guide the programmer to the exact mapping. In the relatively short time frame that was available for this project, this difficult topic could barely be touched. It was postulated that accelerators perform well for algorithms with a high computational complexity.

Following this, the three EUROBEN kernels have been chosen to include one, that is very well suited for accelerators (*mod2am*), one that is merely suited (*mod2f*) and one that is ill-suited (*mod2as*) for current accelerators which all suffer from the bottleneck of transporting data between host and accelerator.

> **Assumption II**: *For a given algorithm A, a given architecture Arch and a given language L, it is possible to gather data to plot performance (in percentage of the peak performance) versus time-to-development. In doing this for all languages L_1 to L_M that are available on a certain architecture Arch, a complete plot will show the effects of the different programming languages per architecture and will –in the end- enable a researcher to choose the optimal architecture and language for his problem and the available time for coding or porting.*

The most important key values will be:

- t_0 : time needed for the first properly version of the code;
- p_0 : the initial performance at time t_0;

- p_{max} : the maximum performance after optimization;
- $\Delta_{perf}/\Delta_{time}$: the higher the slope, the better the productivity gains.

Rough data for these diagrams should be gathered during the porting of the applications. A single diagram might look similar to that:

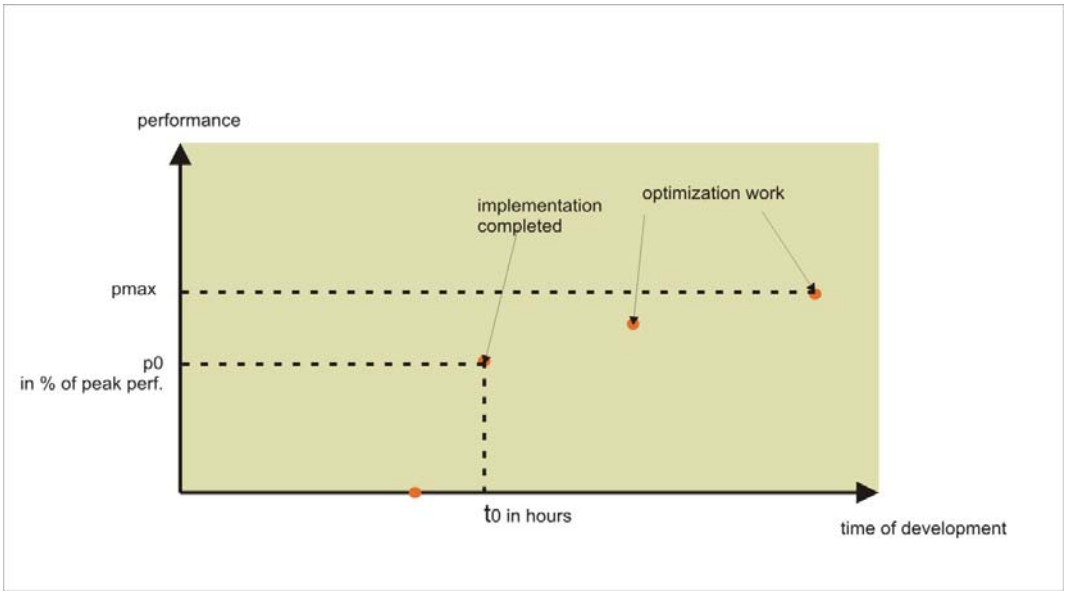

Figure 11: An example of porting diagram (perfomance vs. time-of-development)

All programmers involved in porting the EUROBEN kernels to a new language were asked to report the time necessary for each programming step and the performance improvements of it in so-called "developer diaries". Please refer to "Annex D: Template of the Porting Diary" for a look at the template which has been used to gather this information. It was clear that the individual samples will not be enough to get statistically significant results but these diagrams should give an indication of what happens and what can be achieved with these languages.

7.3 Porting Results

The three EUROBEN kernels mod2am (dense matrix-matrix multiply), mod2as (sparse matrix-vector multiply) and mod2f (1D FFT) should be ported to 14 new languages. This gives a total of 42 attempts to port kernels. They have additionally been coded using the Intel Math Kernel Library (MKL) to obtain baseline performance figures.

Out of the 42 attempts:

- 30 have been partly successful or successful
- 3 were unsuccessful due to compiler bugs or due to unsupported language constructs (CAF/OpenCL mod2as, Chapel mod2f)
- 9 were running out of time (FPGA-VHDL/FPGA-HCE mod2as, CAPS/CUDA+MPI/FPGA-VHDL/FPGA-HCE/OpenCL/RapidMind/X10 mod2f)

For 31 of the ported kernels, the Task 6.6 Reporting Templates (see "Annex D: Template of the Porting Diary") have been handed in, all including more or less detailed developer diaries to judge the productivity of the language.

Porting Status	**mod2am**	**mod2as**	**mod2f**
CAF	successful	not started (compiler bugs)	successful
CAPS HMPP	successful	successful	running out of time
CellSs	successful	successful	partly
Chapel	partly	partly	sdk example used
Cn	successful	successful	successful
CUDA	successful	successful	successful
CUDA+MPI	partly	partly	running out of time
FPGA-VHDL	partly	running out of time	running out of time
FPGA-HCE	partly	running out of time	running out of time
MPI+OpenMP	successful	successful	partly
OpenCL	successful	buggy	running out of time
RapidMind	successful	successful	running out of time
UPC	successful	successful	successful
X10	successful	successful	running out of time

Table 8: Porting status for all kernels

For three of the latest languages, namely *OpenCL*, *Chapel* and *X10*, the performance of the compiler is still very poor. For these three languages the compilers are only proof-of-concepts and should not be used in performance studies at this premature state.

max perf in % of peak perf	System	Measurement based on	mod2am	mod2as	mod2f
CAF	*louhi*	4 cores	72	n.a.	7
CAPS HMPP	*uchu*	1 C1060	79	0.09	n.a.
CellSs	*maricell*	1 PowerXCell8i	79	0.04	2
Chapel	*louhi*	1 core	0.31	0.08	n.a.
Cn	*clearspeed*	1 CSX700	78	0.03	6
CUDA	*uchu*	1 C1060	81	0.87	4
CUDA+MPI	*uchu*	1 C1060	70	0.34	n.a.
FPGA-VHDL (raw)	*maxwell*	1 FPGA	1625 Mflops	n.a.	n.a.
FPGA-HCE (raw)	*maxwell*	1 FPGA	14 Mflops	n.a.	n.a.
MKL	*nehalem*	8 cores	94	3.31	30
MPI+OpenMP	*nehalem*	4x4 (16 cores)	64	2.99	n.a.
OpenCL	*uchu*	1 C1060	2	n.a.	n.a.
RapidMind	*uchu*	1 C1060	20	0.29	n.a.
UPC	*itanium*	4 cores	89	2.87	5
X10	*huygens*	1 core	0.02	0.01	n.a.

Table 9: Maximum performance achieved by the kernels

Since the nature of the languages is very different, there is not one system that could be used for performance comparison of all languages. For most accelerator languages, the WP8 prototype *uchu* from GENCI-CEA has been used, which contains 2 Tesla boxes with 2 GPUs each. For other languages we tried to find systems that allow at least one-to-one comparisons and made use of the PRACE WP5/7 and WP8 prototypes. An overview of the systems that have been used for performance measuring is given in “Annex B: Hardware Overview”. The following table shows the mapping between languages and hardware.

Systems	mod2am	mod2as	mod2f
CAF	*louhi / itanium*	n.a.	*louhi / itanium*
CAPS HMPP	*uchu*	*uchu*	n.a.
CellSs	*cell-cluster*	*cell-cluster*	*cell-cluster*
Chapel	*nehalem*	*nehalem*	n.a.
Cn	*clearspeed*	*clearspeed*	*clearspeed*
CUDA	*uchu*	*uchu*	*uchu*
CUDA+MPI	*uchu*	*uchu*	n.a.
FPGA-VHDL	*maxwell*	n.a.	n.a.
FPGA-HCE	*maxwell*	n.a.	n.a.
MKL	*nehalem*	*nehalem*	*nehalem*
MPI+OpenMP	*nehalem*	*nehalem*	*nehalem*
OpenCL	*uchu*	n.a.	n.a.
RapidMind	*uchu*	*uchu*	n.a.
UPC	*louhi / itanium*	*louhi / itanium*	*itanium*
X10	*huygens*	*huygens*	n.a.

Table 10: Mapping of hardware systems and kernels

7.3.1 *Number of Source Lines*

#Source Lines reported	**mod2am**	**mod2as**	**mod2f**
CAF	350	n.a.	1080
CAPS HMPP	976	979	n.a.
CellSs	305	529	273
Chapel	10	15	n.a.
Cn	1158	1129	615
CUDA	150	350	3000
CUDA+MPI	265	75	n.a.
FPGA-VHDL	1252	n.a.	n.a.
FPGA-HCE	75	n.a.	n.a.
MPI+OpenMP	154	41	279
OpenCL	290	n.a.	n.a.
RapidMind	30	27	n.a.
UPC	550	390	1230
X10	160	294	n.a.

Table 11: Reported number of source lines for all kernels

The reason for measuring the number of source lines of codes was to assess the ease of use of the languages. As described in "Annex C: Software Engineering Metrics", the prevailing view is that the number of source lines gives a rough estimate of the time necessary to program the code, of the readability and of the maintainability of the code. There is a negative correlation between the number of source lines and the number of coding errors.

However, since the nature of the languages is different and the ports have been done by different people in different ways (e.g., some codes made use of BLAS library calls) it has been difficult to measure the number of source lines in such a way that allows a fair comparison of the data. All source codes are available on the PRACE WP6 subversion server, but even counting the number of lines for all versions using tools like CLOC ([56]) proved difficult because some people have been using pre-processor macros to distinguish different versions of the code. Running the pre-processor before counting the number of lines is not reasonable since the number of lines will then contain code coming from highly optimized libraries which should not be measured.

The T6.6 Reporting Template asked explicitly for the number of lines from the programmer. . It was decided to use these figures for the comparison although the given figures show big deviations, which are probably due to the fact that some people reported only the number of lines they had to add to the original reference implementation, while other reported the number of source lines of the most important file, and some people reported the number of lines of the whole directory. For a follow-on project, it is very necessary to specify a unified way to measure the number of source lines. With all these disclaimers please find below an overview of the results.

With all these disclaimers please find below an overview of the results.

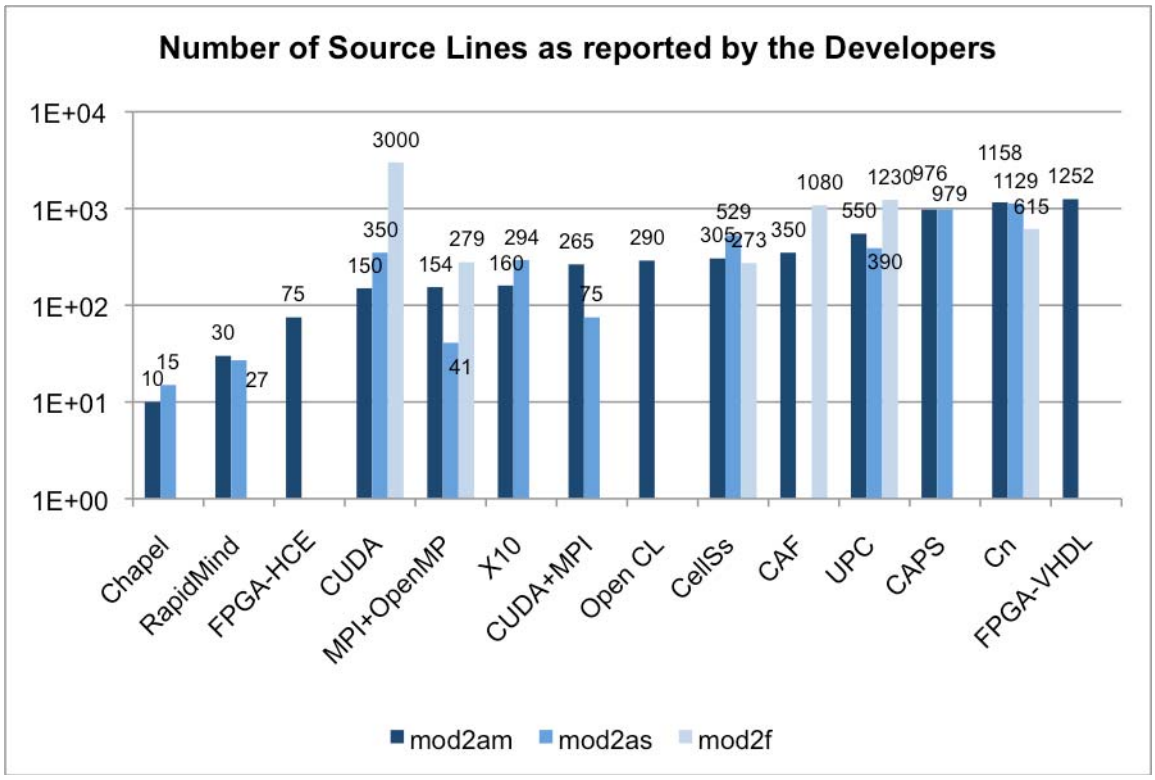

Figure 12: Overview of the number of source lines for all kernels as reported by the developers

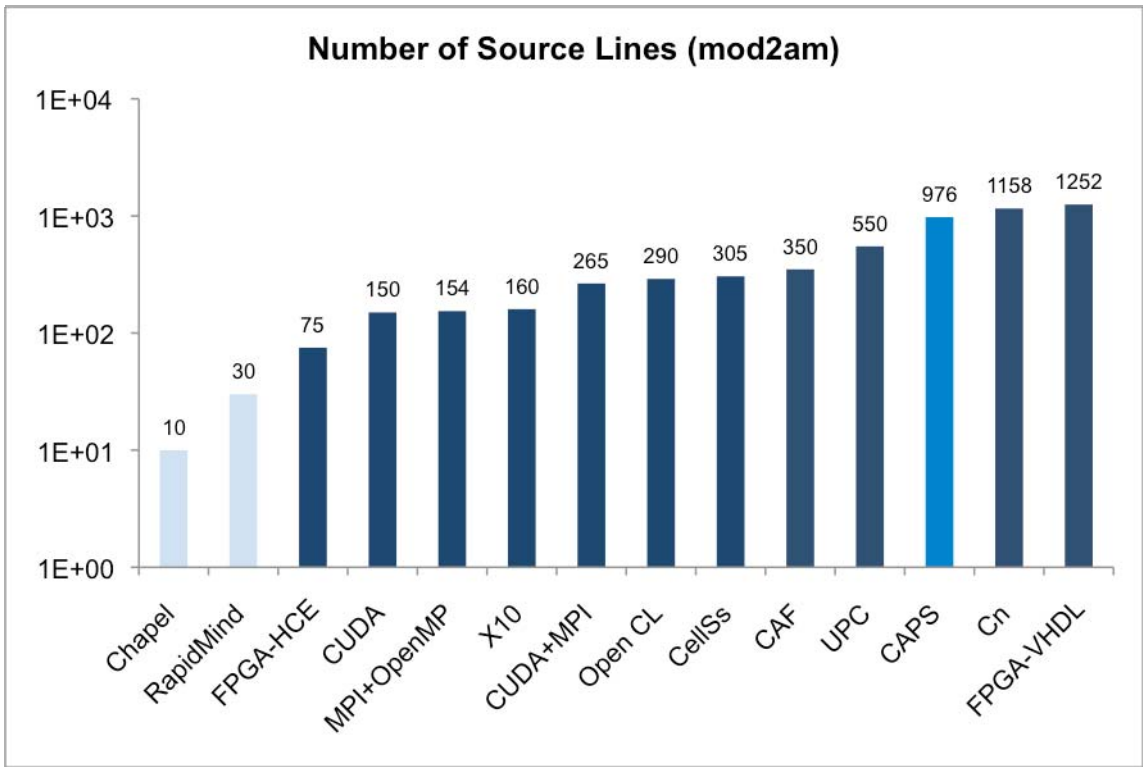

Figure 13: Number of source lines for mod2am as reported by the developers

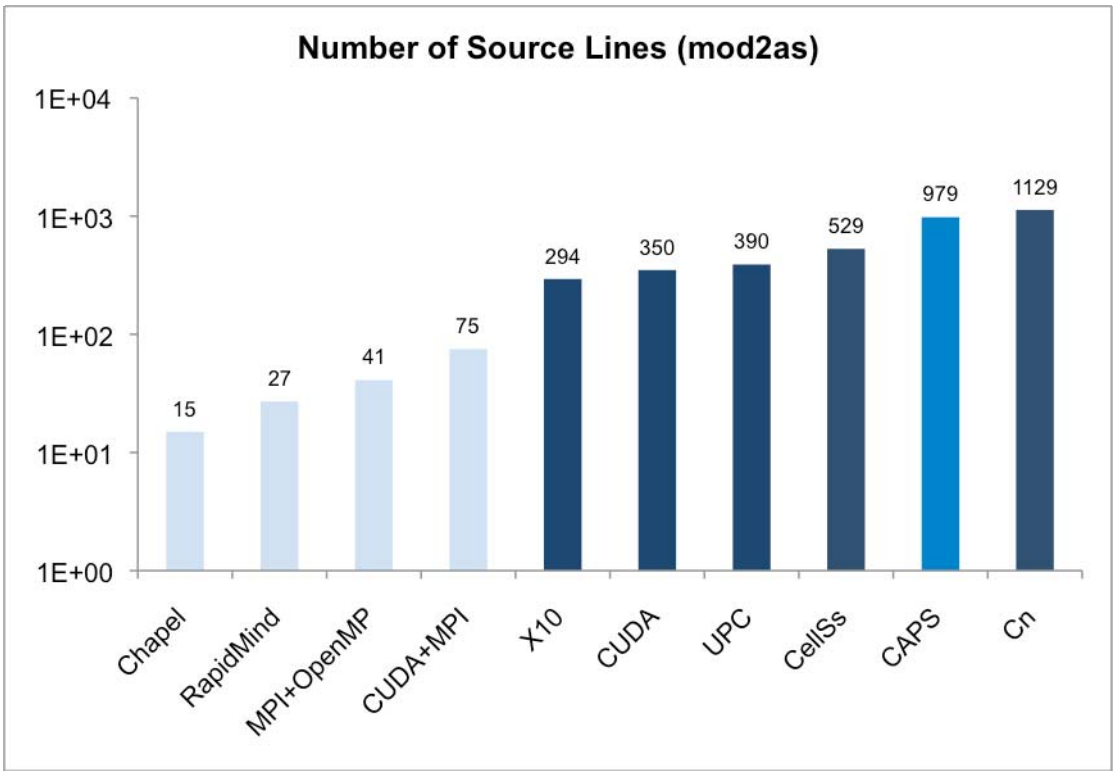

Figure 14: Number of source lines for mod2as as reported by the developers

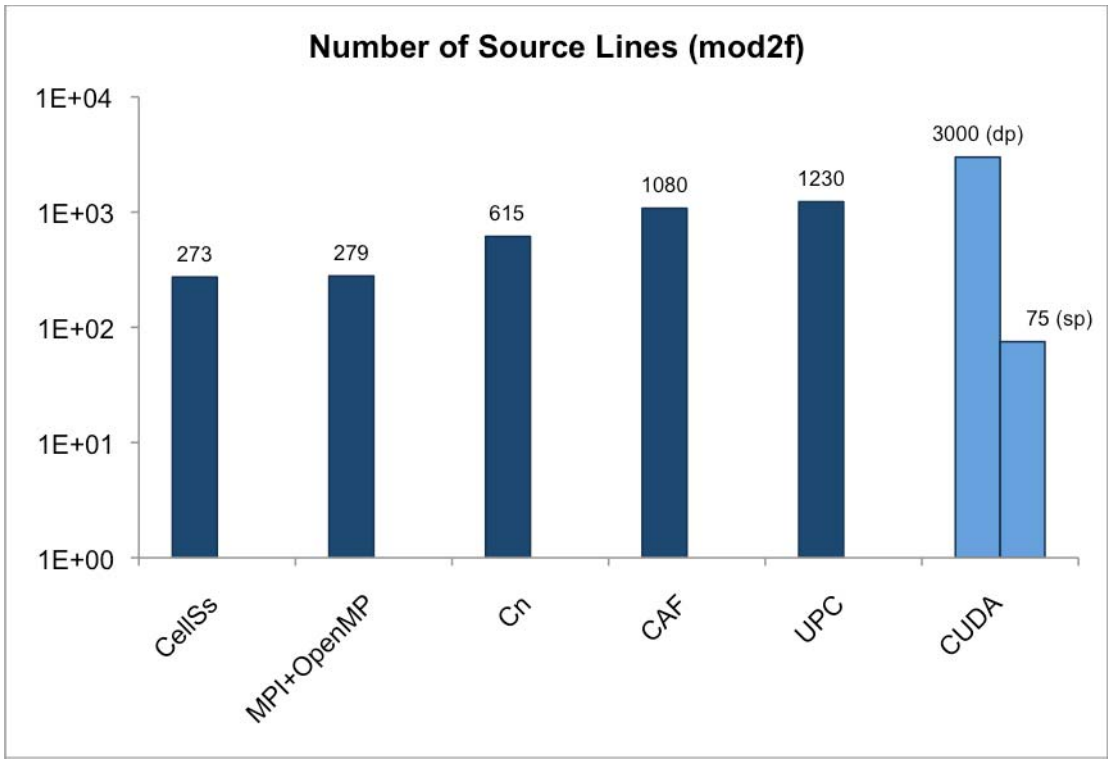

Figure 15: Number of source lines for mod2f as reported by the developers

The overview Figure 12 shows a disturbed image. Having a look at the three figures for each kernel (Figure 13, Figure 14 and Figure 15) reveals some conclusions, even if one has to be cautious about the reported data: numbers in light blue represent very small figures, it seems that the programmers have counted only the number of additional lines of code. Data in middle blue represents singularities of some implementations: the two CAPS HMPP kernels

mod2am and mod2as contain several version in one source file, divided by pre-processor macros. So the actual number of lines of code that is necessary to program in CAPS HMPP is much smaller than the given figure. The CUDA port of mod2f took only 75 lines of code for single precision because one could make use of the 1D-FFT implementation available from CUFFT. Since there was no double precision version of this function, it has to be ported from scratch (using information from a paper available on CUDA Zone). The fast double precision version took 3000 lines of CUDA. All the figures in dark blue seem to be reasonable and comparable.

The trend however, is nevertheless true: the new languages that are designed to be very distinct (e.g., Chapel, RapidMind, CellSs) are the ones that need very few lines, while the ClearSpeed language Cn or VHDL for FPGAs needs many lines of code. It is interesting to see, that CUDA is -at least for the first two kernels- in the range of X10, UPC and CAF.

Another interesting overview is to look at the development time versus the number of source lines (Figure 16, Figure 17 and Figure 18). It is being expected that writing many lines takes much time, so a positive correlation for both should be found. This is partly true, with exceptions for RapidMind and CAPS. RapidMind is an exception since the total development time as reported by the programmer includes a lot of time for testing. The first working version of the code took only around one day. The figures for CAPS HMPP are also misleading since the number of source lines includes many different versions of the kernel and is therefore far too high.

It is interesting to see that both the Chapel and the MPI+OpenMP version have been fast and very short. The OpenCL and CUDA+MPI version have been developed very fast but it is important to know, that they have been ported after a full MPI and CUDA version have been available. They both could build on these versions and adopt them with only slight modifications. The Harwest Compiling Environment (HCE) seems to be a clean solution to program for FPGAs while the VHDL port has been the longest both in terms of development time and number of source lines

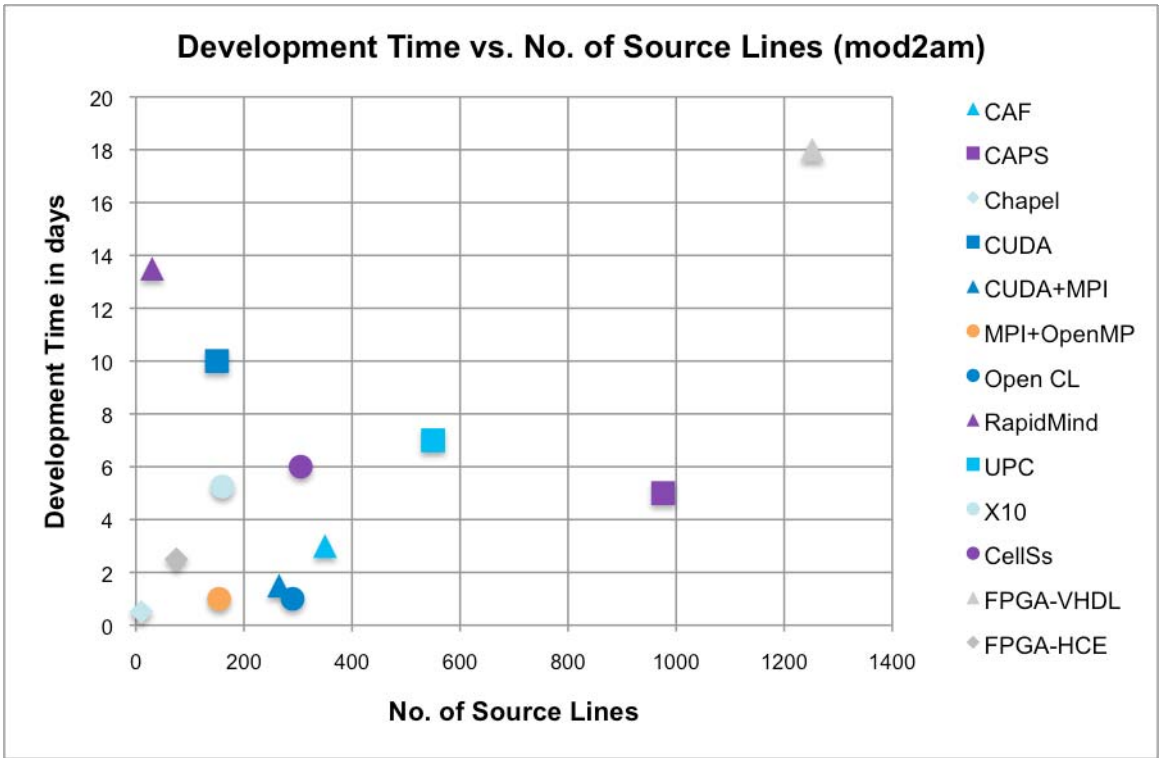

Figure 16: Development Time vs. Number of Source Lines for mod2am kernel

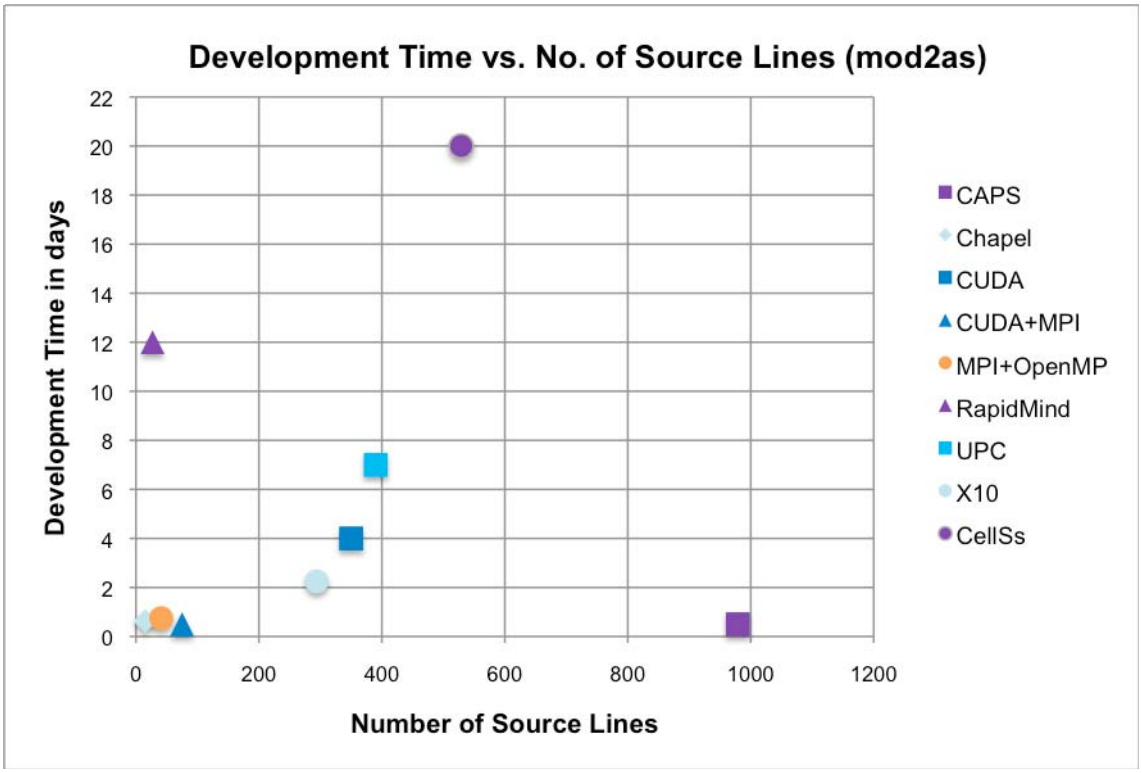

Figure 17: Development Time vs. Number of Source Lines for mod2as kernel

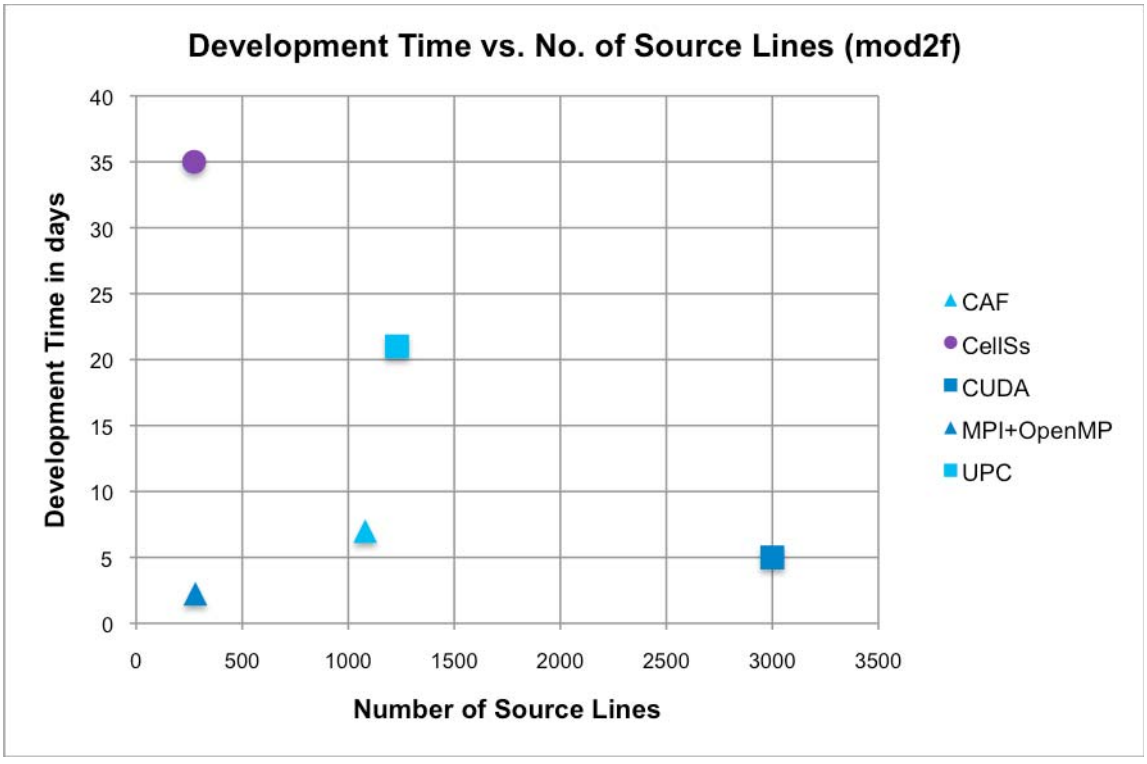

Figure 18: Development Time vs. Number of Source Lines for mod2f kernel

7.3.2 *Development Time*

Development Time [days]	**mod2am**	**mod2as**	**mod2f**
CAF	3	n.a.	7
CAPS HMPP	5	0.5	n.a.
CellSs	3	19	35
Chapel	0.5	0.5	n.a.
Cn	n.a.	n.a.	n.a.
CUDA	10	4	5
CUDA+MPI	1.5	0.5	n.a.
FPGA-VHDL	18	n.a.	n.a.
FPGA-HCE	1,5	n.a.	n.a.
MPI+OpenMP	1	0.75	2.25
OpenCL	1	n.a.	n.a.
RapidMind	18.5	12	n.a.
UPC	7	7	21
X10	5.25	2.25	n.a.

Table 12: Development time for all kernels as reported

Figure 19, Figure 20 and Figure 21 report the development time (in days) for each kernel.

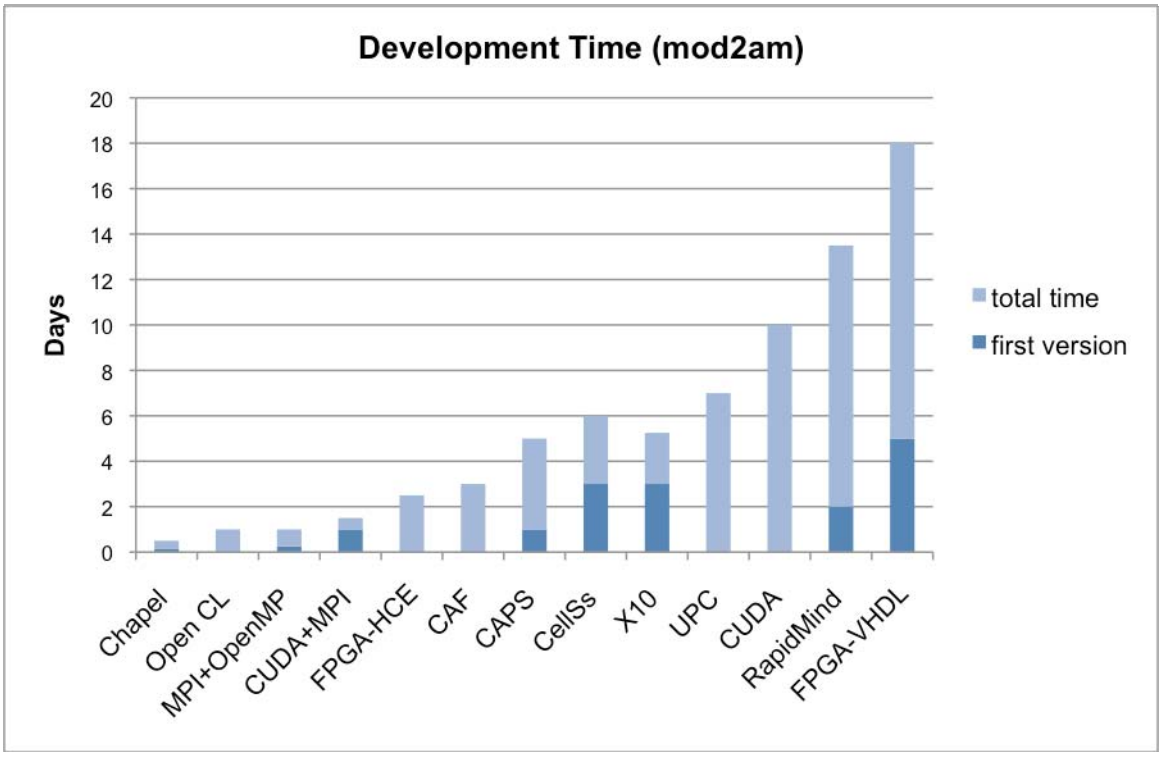

Figure 19: Comparing Development Time of first and last version of mod2am kernel

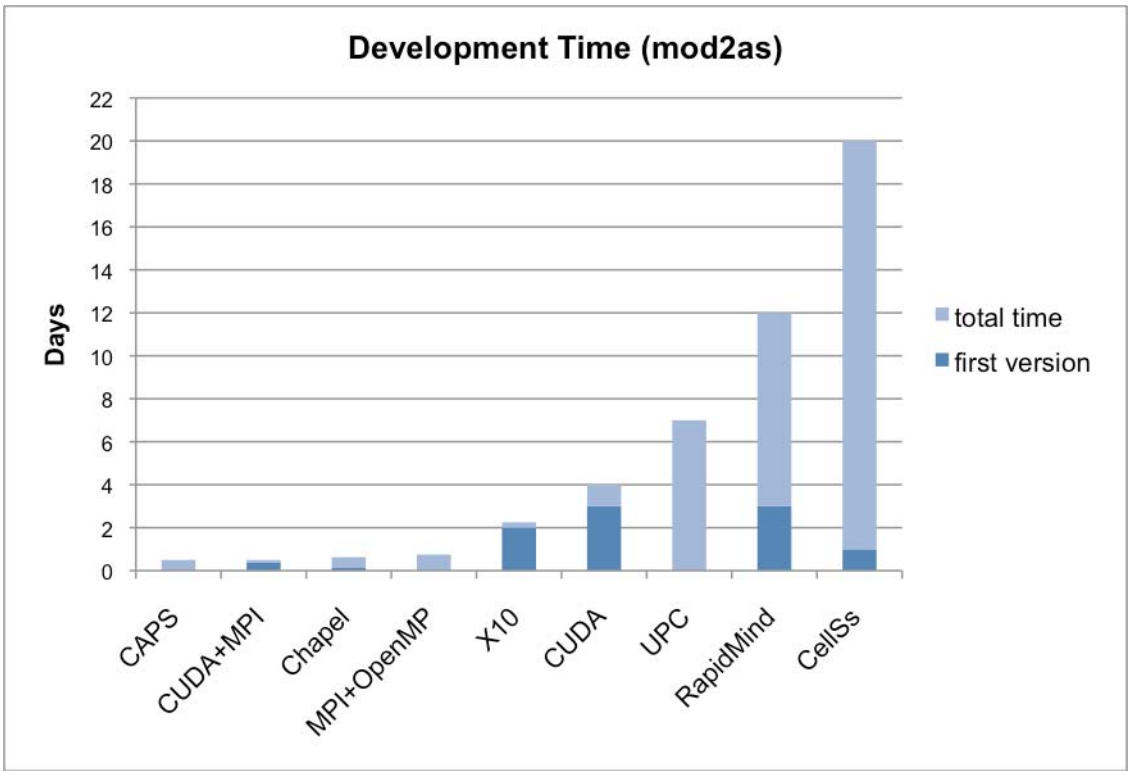

Figure 20: Comparing Development Time of first and last version of mod2as kernel

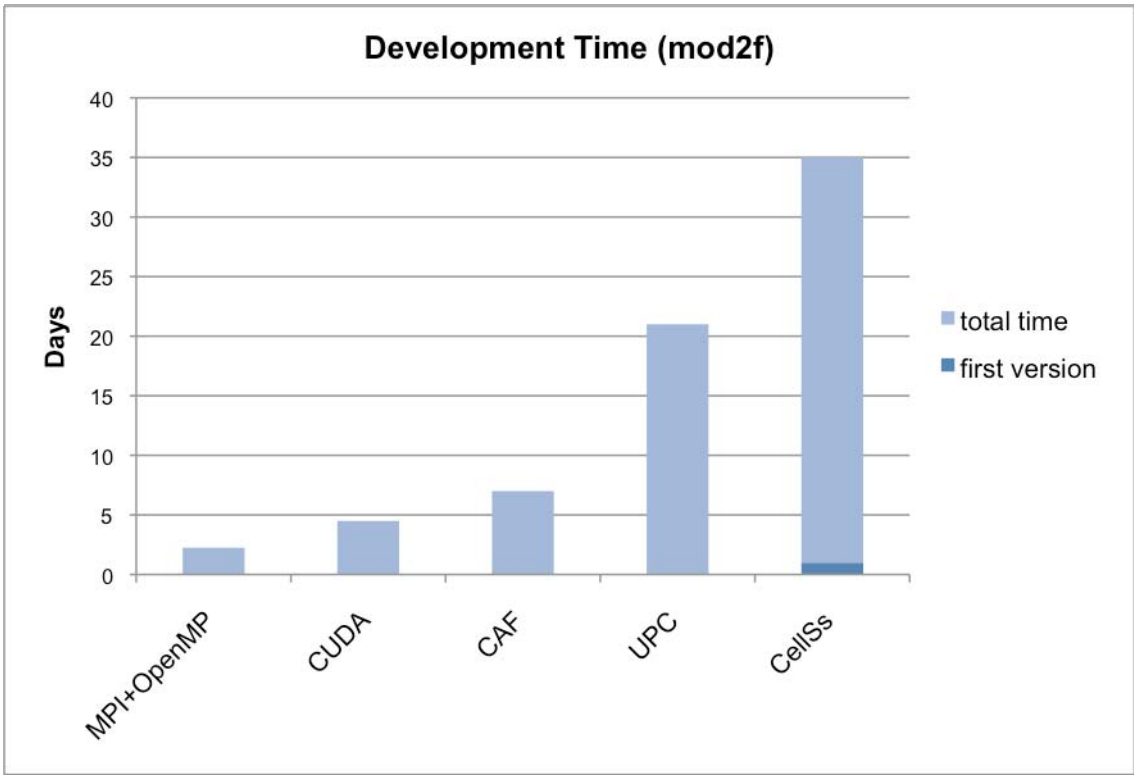

Figure 21: Comparing Development Time of first and last version of mod2f kernel

When looking at the development time spent for the three kernels, the picture is not very clear. This is due to several effects that need a thorough discussion: Firstly, all programmers involved have measured the development time in many different ways. Secondly, some programmers had been language novices when starting the first kernel, while others were already experts. Thirdly, some programmers have not been familiar with some of the mathematical kernels; this is especially true for the 1D-FFT. And last but not least, the pre-

maturity of some compilers lead to prolonged development times as well as poor performance.

In detail, the figures for RapidMind are influenced by the fact that the programmer did report also on time spent mostly in doing performance measurements. The first working version of the code took 1 day for mod2am and 3 days for mod2as. In the case of CAPS and CUDA the responsible programmers were novices when starting with mod2am and already quite experienced for the mod2as port afterwards. The learning curve -from 5 to 0.5 days- for CAPS is quite impressive. The relatively high figures for CAF and UPC are only estimates of the programmer, since his development time was mostly spent in compiler bug tracing. The extremely low figure for CUDA+MPI is due to the fact that the existing CUDA and MPI-C-versions of the codes could be re-used by a programmer who has very deep MPI knowledge. The same programmer did his OpenCL ports using the existing CUDA implementation.

Taking all these remarks into account, the picture becomes clearer and partly satisfies the expectations and hypotheses the group had before starting the project:

- Chapel, MPI+OpenMP, CAPS HMPP and RapidMind are straightforward implementations;
- CAF, UPC and X10 are relatively easy to pick up;
- CUDA needs a bit longer, but for experienced programmers the additional overhead is not too big;
- OpenCL and CUDA+MPI ports are easy if a CUDA version exists;
- A steep learning curve could be seen for CAPS HMPP, CUDA, X10 and RapidMind, which have been the kernels where programmers claimed to be novices.

It is advisable to define the tracking of spent developer time in more detail for a follow on project. It could be of benefit to choose several sub-topics for this, like “reading the documentation”, “coding the first working version”, “doing performance measurements”, “testing the implementation”, “optimization work”, “porting work”, “code debugging”, “compiler debugging”.

Plotting the development time over the achieved performance for the first kernel, the dense matrix-matrix multiplication, in Figure 22 shows for which language the combination of development time and achievable performance is optimal. An in detail performance comparison is given in the following subsection, but we can already see that for the current available hardware the combination of using MKL on the Nehalem architecture is the most efficient. CAPS HMPP performs very well on the Tesla boards.

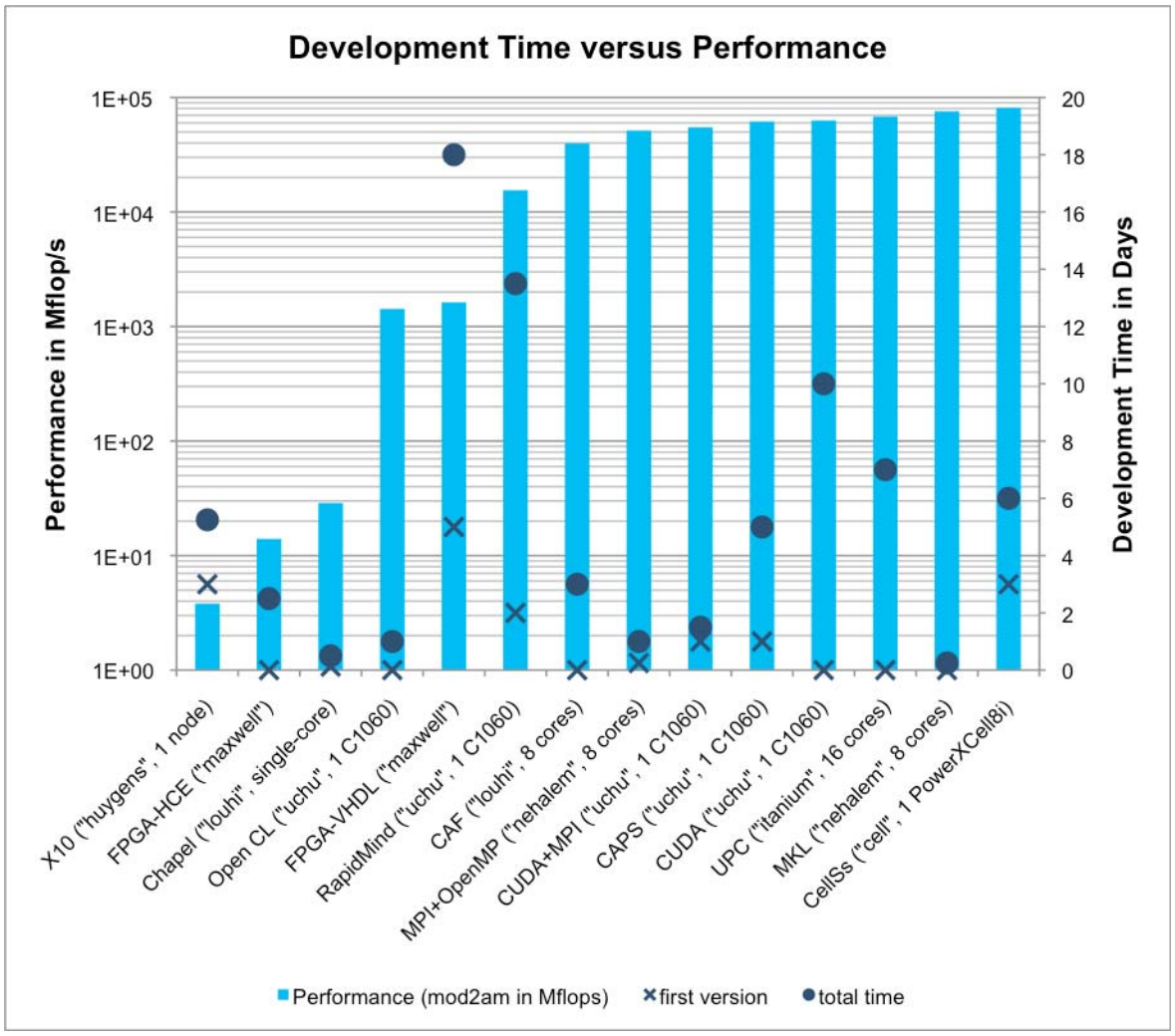

Figure 22: Development Time vs. max Performance for mod2am kernel

7.3.3 *Performance Comparison*

An overview of the achieved performance for all languages is given in Table 13.

% of peak performance	**mod2am**	**mod2as**	**mod2f**
CAF	71.61	n.a.	8.95
CAPS HMPP	78.99	0.09	n.a.
CellSs	79.40	0.04	1.99
Chapel	0.31	0.08	n.a.
Cn	78.13	0.03	6.25
CUDA	80.76	0.87	4.49
CUDA+MPI	70.13	0.34	n.a.
FPGA-VHDL (absolute perf.)	1625 Mflops	n.a.	n.a.
FPGA-HCE (absolute perf.)	14 Mflops	n.a.	n.a.
MKL	93.65	3.31	29.97
MPI+OpenMP	63.57	2.99	n.a.
OpenCL	1.83	n.a.	n.a.
RapidMind	19.85	0.29	n.a.
UPC	88.51	2.87	4.58
X10	0.02	0.01	n.a.

Table 13: Performance comparison of languages in percentage of peak performance

It should be noted that:

- OpenCL and Harwest-C do only support single precision (SP). All other figures given in this section are *double precision* (DP).
- The performance of the FPGA language VHDL and Harwest-C is *absolute performance* in Mflops since it is not reasonable to give a double-precision peak performance for an FPGA.
- The *maximum performance* is the maximum Mflops-rate that could be achieved using the Reference Input Data Set (RIDS) for the kernel, independent from the actual input values. The Mflops-rates were computed using the time necessary for the whole kernel routine - including data transfers from host to accelerator, whenever an accelerator was used.
- It is difficult to compare performance measurements from different architectures. Therefore the *percentage of peak performance* has been used whenever performance on different hardware is compared. For accelerators the percentage of peak performance has been computed based on one accelerator card. For x86 CPUs it has been computed by dividing the achieved maximum performance per core by the theoretical peak performance of the core. Table 14 gives an overview of the peak performance per core or per accelerator for all systems used.

Name	System	Processor	Clock rate * Flop/clock cycle	Peak Perf./core or accelerator
maricell	QS22-blade cluster	PowerXCell8i		102.4 Gflops
clearspeed	Clearspeed CATS units	CSX700		96.0 Gflops
huygens	IBM Power6 cluster	Power6	4.7 GHz*4 Flop/cc	18.8 Gflops
itanium	SGI Altix	Itanium Montecito	1.6 GHz*4 Flop/cc	6.4 Gflops
louhi	Cray XT5	Barcelona	2.3 GHz*4 Flop/cc	9.2 Gflops
nehalem	Intel-based systems	Nehalem EP	2.53 GHz*4 Flop/cc	10.12 Gflops
uchu	NVIDIA Tesla S1070	C1060 GPU		78.0 Gflops

Table 14: Hardware details (peak performance per core or accelerator)

The following figures show graphs with the maximum performance in percentage of peak performance for each kernel. The colouring of the bars, distinguishes figures on different hardware. All orange figures are results obtained on Intel Nehalem EP, all dark blue ones are measured on NVIDIA C1060 GPUs. The light blue figures have been measured on different hardware for which usually only one figure is available.

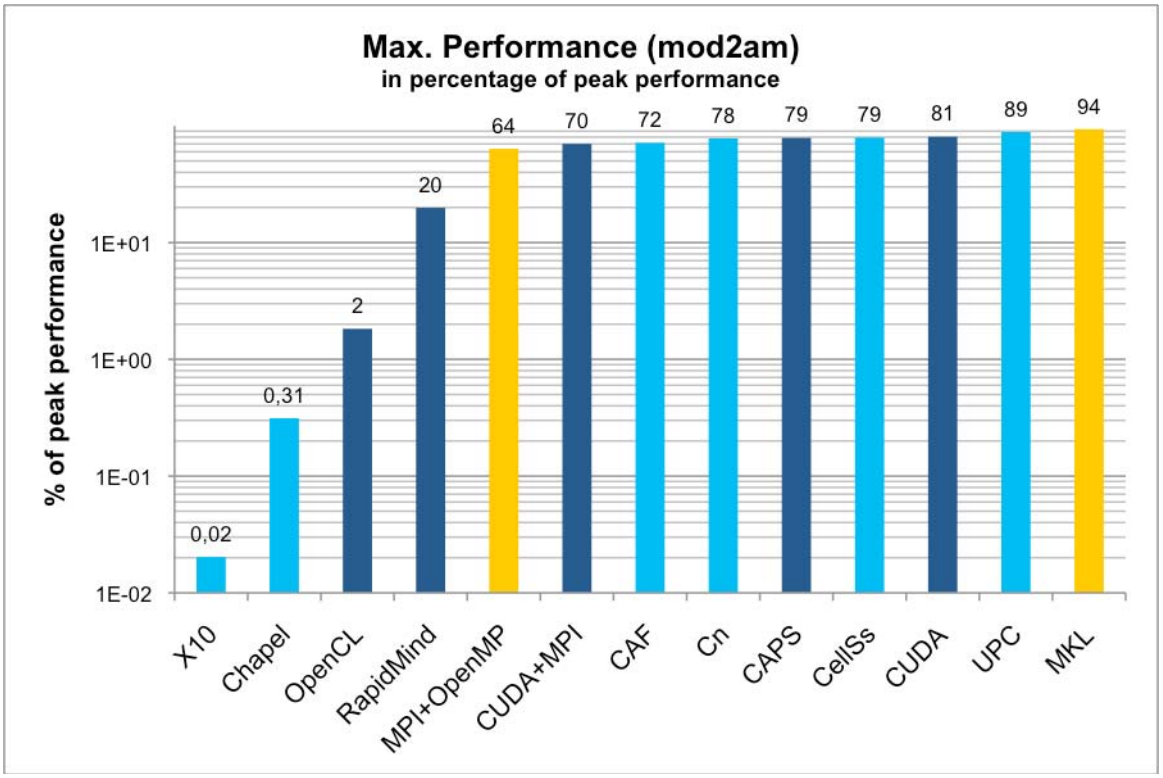

Figure 23: Maximum performance for mod2am in % of peak performance

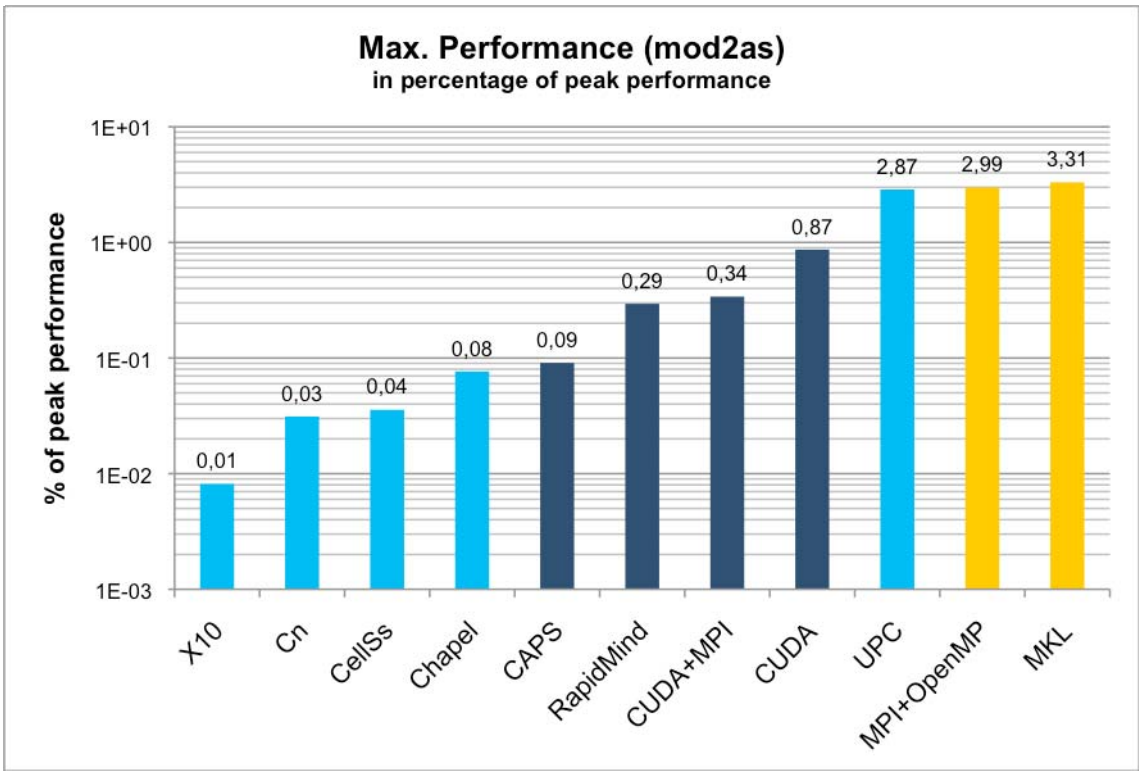

Figure 24: Maximum Performance for mod2as in % of peak performance.

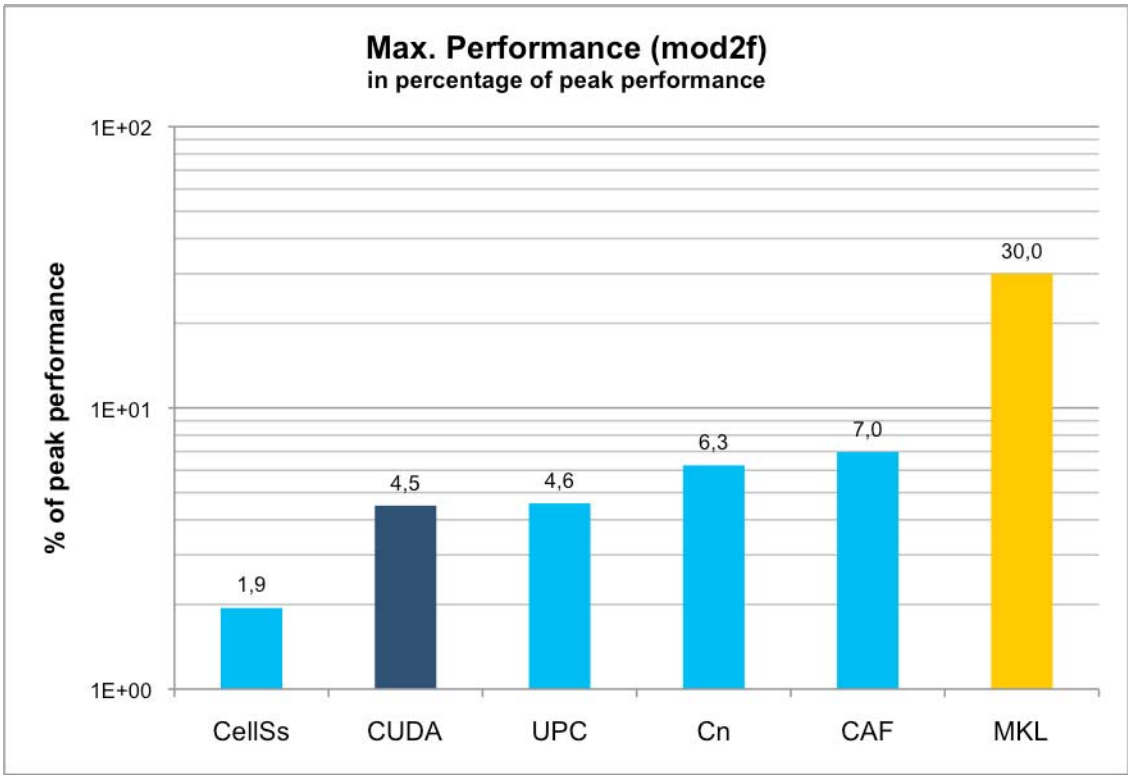

Figure 25: Maximum Performance for mod2f in % of peak performance.

The figures for the FPGA languages could not be included since there is no reasonable definition of a theoretical peak performance for FPGAs. The overall performance of this three years old hardware has not been able to compete with new x86 or accelerator hardware.

The figures for the three latest languages, namely OpenCL, Chapel and X10 have been given for the sake of completeness but show that the compilers are still immature. They indicate that

it might take several more years before highly optimized compilers will be available for these languages but should not be used to prematurely reject these languages. More detailed performance figures for each implementation are given in the Annex "Porting results".

The following in-detail considerations will go from the "traditional" to the more "experimental" languages and programming models.

Comparing MKL versus MPI+OpenMP

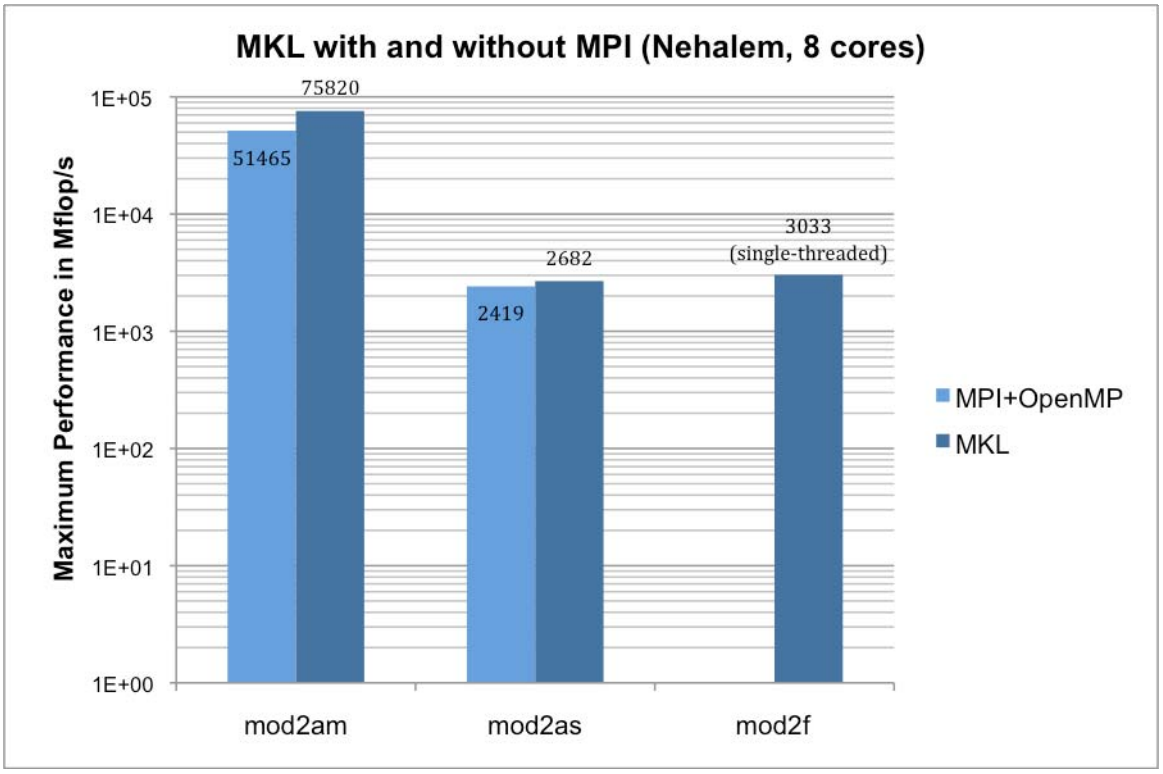

Figure 26: Performance comparison of MKL with and without MPI (MKL vs. MPI+OpenMP)

Comparing the MPI+OpenMP implementation with the reference baseline MKL implementation on 8 Nehalem cores reveals the overhead for using MPI. It is clear that using MPI on one processor makes not much sense, but MPI is necessary when using bigger input data. The MPI+OpenMP implementation has been done in many different ways, but the best performance has been obtained for using MPI between processors and the multi-threaded MKL routines within the cores of one processor. Compared with the pure MKL implementation, the degradation in performance of the MPI+OpenMP implementation is 32% for mod2am and 10% for mod2as. At the time of writing the MKL FFT version was not yet multi-threaded, so all performance figures are measured on only one core.

New and old PGAS languages

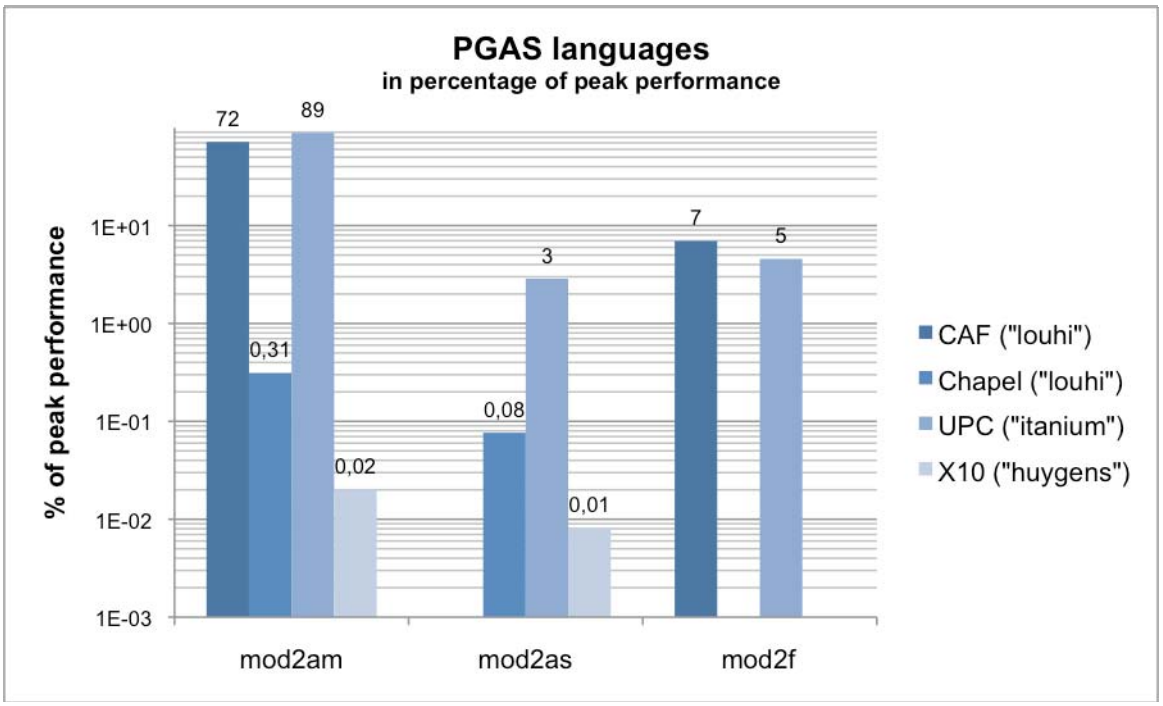

Figure 27: Performance comparison between PGAS languages

A very detailed comparison between UPC and CAF is also subject of work performed in WP8. First results have been published in D8.3.1 and final conclusions will be publicly available in D8.3.2.

GPGPU languages

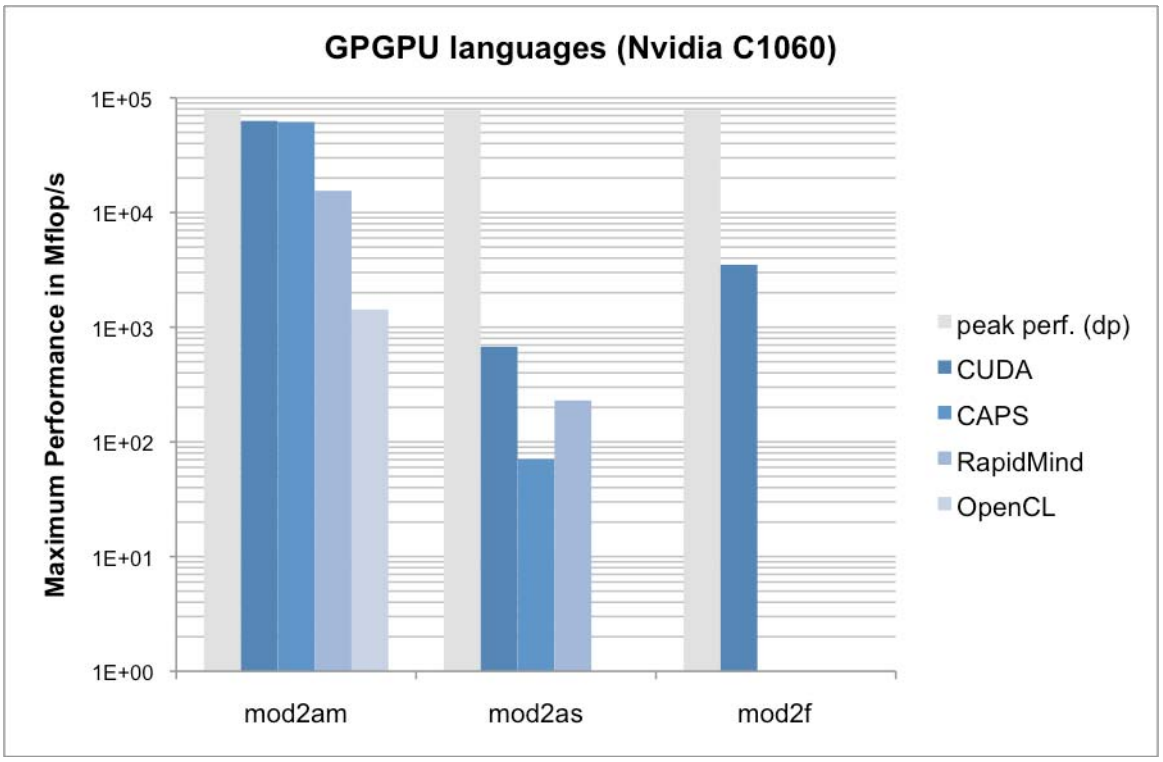

Figure 28: Performance comparison between GPGPU languages

Languages for accelerators

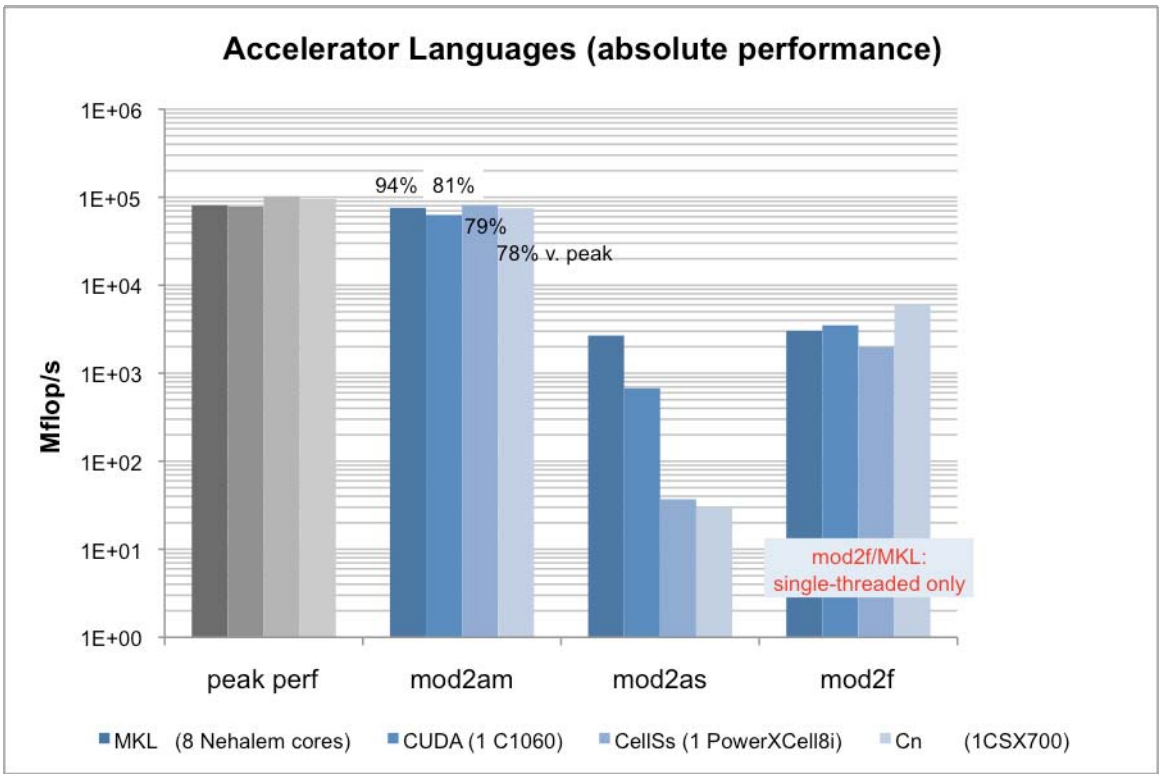

Figure 29: Performance comparison of languages for different accelerators (in absolute values)

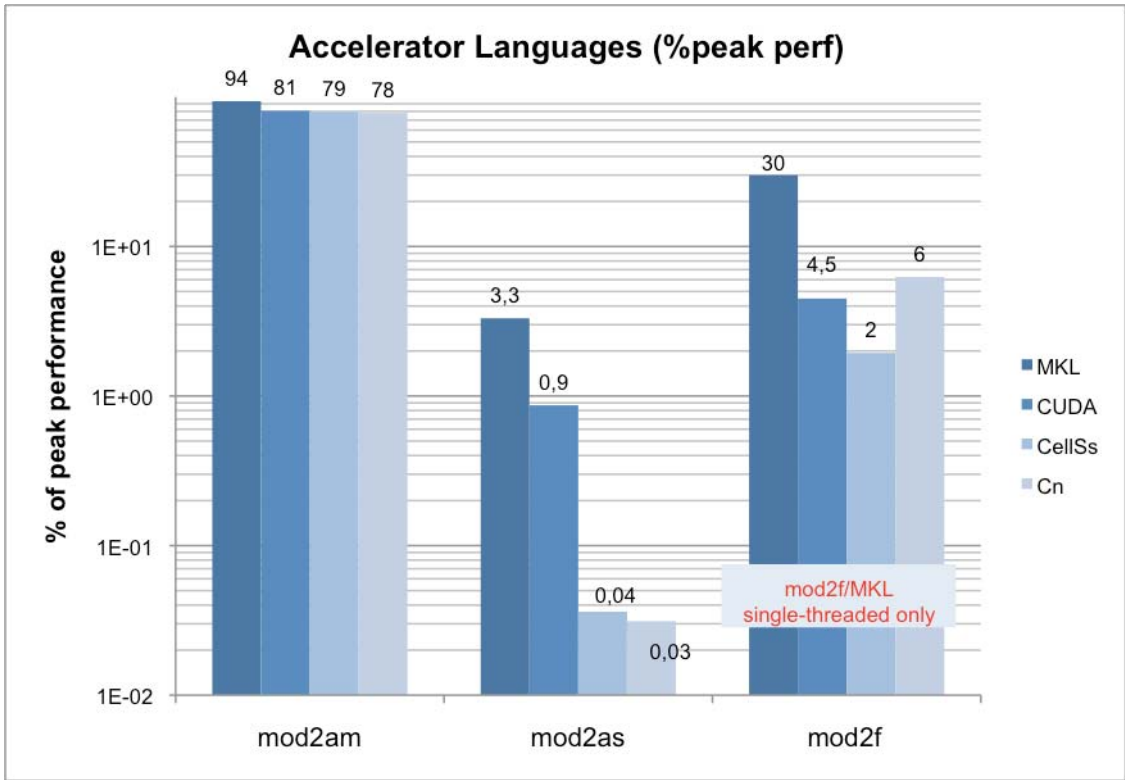

Figure 30: Performance comparison for different accelerators (in % of peak performance)

7.3.4 *Considerations about the reporting template*

To refine the T6.6 reporting template the following improvements should be made:

- Clearly define the way to measure the number of source lines of code e.g., measure only uncommented source lines for the whole directory without the use of pre-processor macros.
- Define in more detail the way to measure performance and ask for more figures e.g., the pure performance of the kernel on the accelerator and additionally the performance including offloading and copying back of data from host to accelerator.
- Add performance measurements which could be used as baseline e.g, the performance of using the CUDA libraries for Tesla cards, the MKL version for Intel systems and whatever is the best optimized vendor library for systems from AMD, IBM, etc.

7.3.5 *Lessons learned*

This subsection includes the lessons learned from the current project and provides conclusions for the further assessment of new languages for a potential follow-on project. It also provides advice for users who are thinking of either starting a new project with one of the new languages, porting existing legacy codes or for those who are in need to make use of accelerators. This subsection is a subjective assessment of the results obtained during the porting of the kernels. It can only be a snapshot of the current state of the art. The assessment starts with a comparison of the PGAS languages; summarizes the results of special accelerator languages and finally discusses the investigation of the many different GPGPU languages. One general drawback of using all these new languages for HPC is the insufficient landscape of debugging and porting tools.

In brief, the performance of the DARPA HPCS program languages Chapel and X10 are insufficient. This is not a big surprise, since both languages are still very immature and the compiler releases are more or less proof of concepts that sometimes do not contain all features of the languages. Furthermore, it has been shown in the past that it is extremely hard for new languages to attract a big enough user community when they have just been designed for HPC. To build highly efficient compilers needs much time, which means for the DARPA HPCS languages that it will be at least several years before these languages can be used for production codes. Porting of the kernels to Chapel has proved that it is possible to design a language that allows a very steep learning curve and a very distinct programming style, which should guarantee nearly error-free codes. The further developments of the new languages should be further assessed using the available ports to benchmark new compiler releases and finish or rewrite those codes which have now been done in a sub-optimal way to circumvent compiler deficiencies.

The performance of the PGAS extensions UPC and CAF to the traditional languages C and Fortran was competitive to current programming languages. The programmer faced some compiler problems and got quite different performance when employing different compilers available on different platforms. The achieved maximum performance is in most cases slightly below the MKL implementation. Even if both languages are often mentioned together, their parallelization model is quite different. While CAF data, which is associated with a different image, has to be addressed in a different way than local data, this is not true for UPC threads. UPC allows addressing data uniformly with the ability to hide data transfers from the user and therefore enables a concise and clean programming style. However, as reported in detail in D8.3.1, to get good performance from UPC it is necessary to stick to a programming model where the data transfers are once again managed by the programmer. This might be partly due to some deficiencies in the compiler but it is doubtful if all these issues can finally be solved in a better compiler. Totally hiding the data transfers from the user needs a high awareness on the side of the user in terms of what the compiler is doing

underneath. Current UPC programs optimized for good performance will then look pretty much like CAF programs, emphasizing a distributed memory programming style. The main improvement achieved by an UPC/CAF code over an MPI code will be the greatly facilitated usage of one-sided communication patterns, which simplifies programming, leads to less error-prone codes and may provide improved performance if the processor is able to optimize the communication structure concurrently with the serial code. However, since a few vendors claim an intention to support the PGAS programming model in the interconnects designs of future systems, evaluation of the classical PGAS languages should continue, using the available kernels to benchmark new compiler releases. Since UPC and CAF are languages that are able to run on many different architectures, these benchmarks could also be useful to decide if one compiler is more mature than the others. A recommendation can be issued to further investigate these languages for real production codes. Note that the PGAS languages have been designed in a manner that they can -in principle- coexist with both currently used parallelization models, OpenMP and MPI.

The review on the results will now go from the accelerator languages for more exotic hardware to the languages for commonly available GPU accelerators. We will first look on the outcomes of the WP8 prototype from EPSRC-EPCC, which compares the use of the high-level language Harwest-C to a VHDL port on a special FPGA supercomputer. Astonishingly, the performance results are disappointing. The assessed HLL compiler Harwest-C seems to be quite good in terms of ease-of-use, but the performance figures are way behind the VHDL port. Even with the relatively complicated VHDL port only a small fraction of the performance of using MKL on Nehalem EP could be achieved. Improvements in the code and the compiler are anticipated for the last deliverable of WP8 D8.3.2. It has been clear before, that FPGAs are not the hardware of choice for double precision codes. The strength of FPGAs lies in fixed precision or bit-manipulating codes which deviate from the normally available 32 or 64bit operator set. Therefore codes that should be ported to FPGAs have to be chosen very carefully and the decision should be made together with experts in embedded computing.

Porting code to Cn, the language for the Clearspeed accelerator cards, has been easier as expected. This is partly due to the chosen benchmark set because for all kernels, except mod2as, example codes or libraries have been available and could be re-used. The ports have been done by a staff member of Petapath, a company that is specialized on Clearspeed products. These are the only ports which have been written by a third party and no developer diaries are available for them. The number of source lines has been reported for the Cn ports and is quite high compared to other codes, even if one considers that the codes make use of library calls. Within WP8 it is planned to port 4 real applications to the WP8 prototype, a report on this activity is expected to be included in D8.3.2. Only then to the usability of this rather different language and accelerator hardware will become clear.

For the Cell platform, no benchmarks using the actual IBM SDK are available. This might be an indication that native Cell programming is very difficult. We have received reporting templates for the use of CellSs. The reported figures give a distorted image. The achieved performance is good, even compared with other accelerators, but the ports have been really time-consuming. Due to the Power PC unit (PPU) of the Cell, getting the codes to run on the accelerator is only a matter of hours or days; but running purely on the PPU will not accelerate the code at all, using the SPUs is mandatory to achieve good performance. As long as the future of the platform is not secured, it is difficult to advise scientists to port their codes to the Cell platform. Without further substantial improvements in the hardware the new GPU generation will very soon outrun the performance of the PowerXCell 8i. The Superscalar concept used for the Cell ports is currently enhanced to run on GPUs. This approach could be revisited in a follow-on project.

GPUs are currently the prevalent accelerator although the have not been used very often for production runs. Several languages exist to make use of GPUs. The language on which most people pinned their hopes is probably OpenCL. Testing the performance of the first available OpenCL compilers has revealed that it is currently insufficient. The language specification is not even a year old, so it might not be a surprise that the performance is very poor. More severe is the fact that the design of the language seems to be insufficient as well. The HPC community hoped that OpenCL could be the language that allows running code on different accelerators which would solve the problem of code maintainability and portability across accelerator devices, but it turned out that to program OpenCL many choices depend on the underlying hardware which prevents a seamless use of other devices. The language is very similar to CUDA and makes some implicit assumptions which are only true for graphic cards. So it seems that the language will stay a GPU language for another couple of years. An HPC compiler expert claimed that it is only feasible to use OpenCL as an intermediate language; higher level languages should be used on top of it and ensure at least portability across GPUs. At the current time it is not advisable to use OpenCL for scientific projects. During a follow-on project the three OpenCL ports should be finished to assure that new compiler releases can be benchmarked on all supported platforms.

Performance-wise the next language is RapidMind. Older publications have claimed equal performance of RapidMind code with the fastest available implementation on the GPU, but the dense matrix-matrix multiplication achieves only a quarter of the performance of the CUDA port. These figures are even worse if applied to the Cell platform, which for one of the codes did not produce correct results. RapidMind's latest version has been the first to support double precision and a CUDA backend (GPU computing was done before using shading languages). The next release will optimize the performance of the generated CUDA code; results of this might be available in time for D8.3.2. Additionally performance measurements of the missing kernel mod2f should be available for the December deliverable, too. Very recently RapidMind has been acquired by Intel to merge the language with Intel's own, quite similar, language Ct - which stands for "C for throughput computing". It remains to be seen if RapidMind will stay an own product or if it will dissolve in Ct. Until this has been clarified, it is not advisable for scientific projects to port their codes to RapidMind.

The investigation of CAPS HMPP showed that there is a painless way of enabling accelerators for a broad community. It has to be mentioned that CAPS HMPP only generates full code for CUDA and SSE. The Cell backend takes only care about data transfers and the kernel routine has to be written by hand inside the so-called *codelets*. Since the NVIDIA cards are currently the predominant accelerator cards, this does not seem to be an issue. CAPS HMPP is by far the most easiest way to port existing code to GPUs. It is the only tool that also allows to port Fortran codes. Ideally, two additional code lines are sufficient to make use of the accelerator. As described in more detail in the WP8 deliverables, for optimal performance a few more lines are necessary, but then the achievable performance is able to compete with CUBLAS. Since the additional lines are inserted as pragma directives, the code stays fully x86 compatible and a performance comparison between the GPU and the host processor can easily be done. This simplifies partly also debugging and provides a powerful tool for rapid prototyping. As soon as a CAPS HMPP port shows a potential for a kernel routine to be accelerated by a GPU it is easy to use CUDA itself to obtain the best performance. It is clear that for a relatively small company as CAPS, users might be concerned that the product will stay in the market. However, since the code stays x86 compliant, one could make use of the tool as long as it is around, and does not face the danger of ending up with a code which is not able to run at all. CAPS HMPP can be recommended for all projects that want to use or need to make use of GPUs. The remaining mod2f port should be finished either in WP8 or in a follow-on project.

The second language that can currently be recommended for those interested in accelerators is CUDA. Even if there is a small danger that its vendor NVIDIA at some point might get out of business or loose interest in CUDA, the language is quite similar to OpenCL. CUDA code could be easily ported to OpenCL if necessary. It is clear that using CUDA for scientific codes is somewhat comparable to using assembler code for the most important kernel routines; with each new hardware or software release small changes in the code are usually necessary. Codes that have delivered good performance on older hardware might suffer on newer hardware and vice versa. The code will definitely grow in size and become harder to maintain. That means that everyone should be clearly aware, whether the code will benefit from the use of accelerators or if there are easier modifications; e.g., optimizing for SSE instructions. Items that should be considered beforehand are:

- "How many lines is the main kernel routine?"
- "How good can it be encapsulated form the rest of the code?"
- "Is the code memory- or CPU-bound?"
- "Is it possible to leave the data on the accelerator or could this data be at least partly "re-used?"
- "How good can the kernel routine run on the relatively special (reduced) GPU cores?"

One of the main arguments for CUDA is definitely its big user community which goes way beyond the HPC or scientific computing community. This user community can be seen as a guarantor for good programming techniques and skilled programmers. The number of sold cuda-enabled GPUs makes them available at a very attractive price. The CUDA SDK is freely available. For a follow-on project CUDA should be one of the languages chosen to evaluate its potential for real scientific codes. Additionally, more and more CUDA-enabled ISV codes are becoming available which should also be investigated. The speedup for those clearly depends on the used input data set. An assessment of this dependency would be very valuable for the community.

The port to CUDA+MPI has proved that using this programming model is easy as soon as one can build on an existing MPI parallelisation and on an available CUDA port of the kernel routine. The performance results vary, for mod2am a nearly perfect scaling is revealed, while mod2as, which is in generally not a good algorithm to be used on accelerators, does not show any scaling at all.

For those who do not want to go the extra mile and make use of accelerators, the good performance of the MKL and MPI+OpenMP ports show that with the new Nehalem processor, the performance gap between CPUs and GPUs has been shrinking. Only when the recently announced new NVIDIA Fermi chips will be available (expected for Q1'10), the gap will again increase and the GPUs will outperform the x86 cores double precision performance at least by a factor of 8. For the next years MPI and OpenMP together with all the available debugging and analysing tools will definitely be the best supported HPC programming and parallelisation models.

Two programming models seem to be missing in this assessment; Firstly, this is Intel Ct. We hope that Ct will be available for investigation in a follow-on project since the concept of the language looks quite promising, the Intel compilers have a good reputation and the language is not targeted solely at the HPC market. It might have the potential to become a commonly used language for many-core processors. Secondly, an evaluation of the PGI compiler with automatic code generation for GPUs is lacking. Both have not been available by the time the assessment of the languages has been started. This shows the rapid development and progress in the field. The performance of the PGI compiler will be assessed by WP8 and results should be available for D8.3.2. Intel Ct will only go into beta-testing shortly before the end of the

year. Both should be included in a follow-on project. The work on the diaries can be further extended to be used for the porting of whole applications to assure that more and more accurate data can be gathered.

It is currently under investigation how PRACE could assess the results produced by projects outside of PRACE. It is necessary that developer diaries for real applications become available for the complete assessment of the productivity and potential of the new languages.

8 Assessment of libraries for petascale systems

Most of the HPC codes around the world rely on external standard libraries to perform standard tasks, and this is true also for most of PRACE codes. Libraries are used by developers to avoid re-inventing the wheel implementing a standard algorithm and to boost the performance of their applications. In fact libraries for HPC most often are highly optimized and implemented in an efficient way. In particular most libraries are so important for performance that vendors offer hardware specific versions (like BLAS and LAPACK) that allow to reach higher fraction of peak performance. As a consequence it is a general optimization practice among HPC developers to write codes specifically designed to use HPC libraries as much as possible. For these reasons libraries are supposed to play an important role in applications development and petascaling. In this chapter we will try to asses the impact they have on petascale systems evaluated, the architectural soundness and the potential for optimizations.

As it is evident from Table 3 PRACE codes use libraries for very different standard tasks, but not all libraries have the same importance in making the application scale in petascale systems, moreover different libraries display different problems and challenges with respect their usage on these systems; then to better analyze them and to find common path and strategies with respect to the porting and optimization of applications, we have grouped the libraries in different categories based on their main characteristics.

Due to the number of different characteristics a library may have, it is not possible to place all libraries in categories with stiff wall, most libraries in fact have characteristics in common with many others even if they cover very different fields of application, this is made evident by the tagging used in Table 2 where each tag represent a specific characteristic. In our analysis we have considered five categories: basic, I/O, high level, communication and special purpose to be relevant for application porting and scalability on petascale systems. For each of these categories relevant features are discussed in what follow.

8.1 Basic Libraries

In this section we analyze those libraries that are used to compute basic and standard tasks like Linear Algebra computations and Fast Fourier Transform (FFT). The term “basic” here does not mean necessarily simple, in fact there are some standard tasks, like matrix inversion, requiring complex algorithms that may lead to subtle numeric effects, especially when scaling out the problem on a petascale system.

Apart from communication libraries, which are analyzed in detail in previous chapters, it is possible to further subdivide basic libraries used in PRACE codes in three categories: Linear Algebra, Fast Fourier Transforms and Low Level, that display the same problems and challenges.

8.1.1 *Linear Algebra*

From the analysis of the PRACE codes it emerges that most of them use linear algebra libraries in floating point intensive sections of the code. This is in common with the large majority of scientific and technical applications running on HPC centres. It is a remarkable fact that for linear algebra libraries a de facto standard exists and it is represented by the BLAS (Basic Linear Algebra Subroutine) and LAPACK (Linear Algebra PACKage) collection of subroutines, whose source code is available for free from netlib.org. BLAS contains subroutines implementing basic linear algebra algorithms like dot product (level 1 BLAS), matrix vector product (level 2 BLAS) and matrix-matrix product (level 3 BLAS),

LAPACK instead contains higher level drivers to implement more complex algorithms like linear system solutions and eigenvalue problems. LAPACK can be used in substitutions of two older libraries LINPACK and EISPACK and requires BLAS when linked to the executable. It is difficult to discuss BLAS without discussing also LAPACK, so even if LAPACK can well be considered high level it will be discussed mainly in this section together with BLAS. Today many vendors offer highly optimized libraries with API compatible with BLAS and LAPACK, so that portability and performance of applications using them is guaranteed. Some examples of these vendor libraries are: ACML (AMD), MKL (Intel), ESSL (IBM), SciLib (CRAY). All of them contain subroutines tuned for one specific class of processors and their performance can reach a high fraction of the peak performance.

Vendor versions of BLAS and LAPACK offer serial, vector and multi-threaded implementations whereas for a message-passing implementation a different set of libraries exists even if they do not contain a counterpart for all the serial driver, these other libraries are called ScaLAPACK, PBLAS and BLACS and will be discussed among the high level libraries below. BLAS are so important for single node performance for so many applications that a lot of work is done by vendors to produce an optimized version for their hardware. LAPACK performance, like for applications, relys mainly on BLAS performance, then from the point of view of future petascale systems it is extremely important to have at least an optimized version of BLAS Luckily a lot of work has been done to produce also vendor independent highly optimized versions of BLAS, among these, the so called GOTO (name of its developer) BLAS and ATLAS (Automatically Tuned Linear Algebra Subroutines), and so they are quite relevant for HPC and in particular for petascale machines. In fact, their source code is available, and they allow reaching performance close to those produced by the vendors, and in some particular range of matrix size they can even display better performances. Therefore it is possible that an application should be linked to these libraries instead of the vendor version to reach a higher performance. Then they should be taken in consideration for petascale systems too. From an application point of view highly tuned BLAS offer the opportunity to make it efficient (on the single node) easily on most architectures, this is particularly true if the application makes intense use of level 3 BLAS.

8.1.2 *Fast Fourier Trasform*

Fast Fourier Transform (FFT), together with linear algebra, is another widely used numerical algorithm in HPC codes. Unlike for linear algebra, for FFT libraries there isn't an established standard for the application interface. Nevertheless for most hardware platforms a specific vendor version exist. So that one of the biggest difficulties in porting code that use an FFT library on a new platform is to change the source code to cope with the specific API. Among PRACE applications the most common vendor FFT libraries are those contained in MKL, ACML, ESSL, SciLib vendor numerical libraries.

A part the vendor FFT libraries, like for linear algebra, some open source FFT library exists, in particular most PRACE applications requiring FFT are interfaced with the FFTW library which, on the other hands, is a sort of emerging standard, in the sense that some vendor libraries are starting to offer the FFTW API together with its own API. On the other hand, the lack of a standard is visible in the FFTW itself, since when the developers switched the major release from version 2 to version 3 they changed the API, so that now there are two version of FFTW being used by HPC applications. In fact there is a good reason why version 2 of FFTW is still alive, it has a parallel driver based on MPI.

Regarding petascaling applications the availability of a highly optimized FFT library is critical to exploit the power of machine cores, and open source libraries like FFTW could have severe problems of performance on non-common hardware platform (like vector

machine). In this case the availability of a vendor or hardware specific FFT library is mandatory to match the performance requirement of petascaling applications.

This fact makes FFT more critical than linear algebra, where a good level of performance, even not optimal, can be obtained with some of the open source versions of the BLAS library. The situation is even worse for parallel FFT driver, where the lack of a standard and the lack of parallel FFT vendor libraries have forced many applications to develop their own parallel driver. This aspect is really critical for petascaling since the scalability problem of FFT has to be coped on the application level. Another possible difficulty connected with the FFT libraries comes from the fact that different libraries do not contain a complete set of subroutines to cover all possible cases of data types (single and double precision, real and complex) and dimensions (1D, 2D and 3D).

8.2 I/O Libraries

Within the next years, platforms will be built with unprecedented parallelism that may, in some case include over a million computational cores. This increased level of concurrency will pose enormous challenges for future I/O systems that must support efficient and scalable data movement between disks and distributed memories.

To tackle all of these problems (scalability, high bandwidth and usability), computer science teams have developed software dealing with parallel I/O that can be used by application scientists. Roughly speaking, parallel I/O is the stack of software that links the application and the I/O hardware. It is divided into multiple layers with distinct roles:

- *Parallel file system* maintains logical space and provides efficient access to data (e.g. PVFS, PanFS, GPFS, Lustre)
- *I/O Forwarding* found on largest system (BG/P for example) to assist with I/O scalability
- *Middleware layer* deals with organizing access by many processes (e.g. MPI-I/O)
- *High level I/O library* that maps application abstractions to a structured portable file format (e.g. HDF5, Parallel NetCDF, NetCDF-4)

In this section, we will focus our investigation on the last two points, which are the only layers that are directly accessible to a user level and compare it to the most commonly used approach, the one-file-per-processor approach with POSIX I/O.

8.2.1 *POSIX I/O*

POSIX is the IEEE Portable Operating System Interface for Computing Environments. It defines a standard way for an application program to obtain basic services from the operating system. POSIX was created when a single computer owned its own file system, which is no longer true with shared parallel file system on modern platforms. Consequently, POSIX I/O hasn't the ability to describe collective I/O access. Moreover, it can be very expensive for a file system to guarantee POSIX semantics for heavily shared files. Nevertheless, it is the most commonly used approach even on massively parallel systems and is considered (often wrongly) to be the most efficient. The worst approach but still implemented in some applications consist of assigning one process to handle all I/O operations. The process, typically rank zero, is responsible for collecting the data from all the other processes and writing it incrementally to the file (or for distributing the data to other processes after it has read the data from the file). This approach not only limits the data size to be read or written (due to the memory size limitation of the responsible process), but also serializes all the I/O

operations and therefore impacts significantly the I/O performance and the scalability. Of course, this approach has to be banned on massively parallel Petaflop/s architectures.

Even on new machines with thousands or tens of thousands of cores, the majority of users still continue to use the traditional POSIX unformatted I/O, where each process of a parallel application writes to its own separate file to store the result of its local computation (often called "the one-file-per-processor" approach). This leads to several drawbacks. First disadvantage, as the number of files is fixed, a restart must use the same number of processes. More problematic, this approach does not scale well and leads to a real data management nightmare when ten or hundred of thousands of files are generated on a petascale machine. Regardless the performance aspect, managing and maintaining such a huge number of files is a challenge in itself. Moreover, if the application has complex data structures, the simple interface may cause significant inconvenience for application users to reassemble the data files for data analysis or visualization purpose.

On large-scale parallel system, it is clear that a solution using a single (or a few) shared file is more appropriate. Table 15 summarizes pros and cons of the POSIX I/O approach.

Pros	Cons
• simple approach already implemented in many applications; • easy to implement (POSIX I/O); • efficient on a limited number of processes;	• does not scale well, not designed for large-scale parallel machines; • a nightmare to manage and maintain the generated files; • data analysis and visualization on generated files may be a very hard task;

Table 15: Pros and Cons of POSIX I/O

8.2.2 *MPI-I/O*

MPI-I/O is part of the MPI-2 standard (1997) and is an I/O interface specification for use in MPI applications. The data model is the same as the POSIX-IO (stream of bytes in a file). The implementation is nowadays widely available on most of HPC platforms. The range of features that is usable is very rich and complete:

- independent and collective I/O;
- non-contiguous I/O with MPI data-types and file views;
- blocking and non-blocking I/O;
- Fortran, C and C++ bindings;
- a system to encode files in a portable format (external32).

MPI-I/O provides a rich interface allowing any user to describe non-contiguous accesses in memory, in file or both and collective I/O. It relies heavily on the concept of MPI derived data-types. As a consequence, this flexibility allows implementations to perform many transformations that could result in better I/O performance (e.g. data sieving, two-phase I/O). MPI-I/O is considered as a low level I/O library and is actually used as a solid basis for more high-level libraries like PnetCDF or HDF5. As such, it should give the best parallel I/O performance. Table 16 summarize pros and cons of the MPI I/O approach.

Pros	Cons
• rich interface giving access to a lot of functionalities; • easy to use and well known approach (same as MPI); • available on almost all HPC platforms; • allow implementation to optimize I/O performance depending on the platform and on the I/O hardware characteristics; • parallel I/O (read/write) on one file; • portable (e.g. the interface); • scalable on large scale parallel machine (depending on the implementation); • often used as the basis for most high level parallel I/O libraries; • well documented;	• a too low level approach; • it is not a portable, self-describing file format; • portability incompatible with performance optimization;

Table 16: Pros and Cons of MPI I/O

8.2.3 *HDF5*

HDF5 (Hierarchical Data Format) is the name of a set of file formats and libraries designed to store and organize large amounts of numerical data. Originally developed at the NCSA, it is currently supported by the non-profit HDF Group, whose mission is to ensure continued development of HDF5 technologies, and the continued accessibility of data currently stored in HDF. It provides

- a versatile data model with a hierarchical data organization in a single file (typed, multidimensional array storage, attribute on dataset, data);
- a portable file format;
- a library with a high-level API (e.g. C, C++, Fortran, …);
- a parallel I/O mode with contiguous or non-contiguous I/O (in memory and file);
- a set of tools for managing, viewing and analyzing the data in the collection.

HDF5 is build on top of MPI-IO. Collective or independent calls are available. The user is responsible for creating a mapping of the in-memory data structure into a representation in HDF5 dataset. Table 17 summarizes pros and cons of the HDF5 library.

Pros	Cons
• portable self-describing file format (tree-like structure) recognized as one of the standards for scientific applications; • high-level parallel I/O interface, with collective call associated with one file; • build on top of MPI-IO (can benefit from MPI-IO optimization, availability); • well documented and maintained; • big community of users and a lot of functionalities;	• Some overheads compared to MPI-I/O; • More complex to implement than MPI-I/O; • High learning curve; • Variable performance and scalability;

Table 17: Pros and Cons of HDF5 library

8.2.4 *NetCDF-4*

NetCDF developed by the Unidata Program of the University Corporation for Atmospheric Research (UCAR) is a popular package for storing data files in scientific computations, widely used in earth sciences. NetCDF consists of both an API and a portable file format. The API provides a consistent interface for accessing NetCDF files across multiple platforms, while the NetCDF file format guarantees data portability. However, this API was originally designed for use in serial code, therefore the semantics of the interface is not designed to allow high performance parallel data storage and access.

NetCDF-4 is jointly collaboration effort done by the groups who develop and maintain NetCDF and HDF5 software to create a software that uses enhancements to the HDF5 data model and format to implement a richer NetCDF data model. The result is intended to combine some of the desirable characteristics of NetCDF and HDF5, while taking advantage of their separate strengths. The NetCDF-4 library provides compatibility with existing.

NetCDF programs and data, additional data modelling abstractions, and features for use in high performance computing, such as parallel I/O.

The latest release at the time of writing is 4.0.1 (March 2009), which provides a full compatibility with NetCDF program and data and adds an upgraded Fortran90 API and performance enhancement. A complete overview of NetCDF-4 and a lot of complementary information and material can be found on the official website. Concerning the parallel I/O feature of NetCDF-4, it relies on the parallel I/O feature of HDF5, thus HDF5 is required and must be installed with the option "-enable-parallel" prior to NetCDF-4 installation. Due to lack of time, neither performance tests nor implementations with NetCDF-4 have been performed, but as the parallel I/O is based on HDF5, we should obtain similar behaviour as for HDF5.

8.2.5 *Parallel NetCDF*

Parallel NetCDF (PnetCDF) has been jointly developed by Argonne National Lab and Northwestern University to fill this lack. The interface builds on the original NetCDF interface and defines semantics for parallel access to NetCDF datasets (same file format). It maintains the look and feel of the serial NetCDF API (same syntax and semantics with prefixing function with "ncmpi_" and "nfmpi"). Build on top of MPI-I/O, it allows for further performance gains through the use of collective I/O optimizations and new flexible data

access functions for non-contiguous memory regions using MPI derived data types. Table 18 summarizes pros and cons of the Parallel NetCDF library.

Pros	Cons
• portable self-describing file format recognized as one of the standards for scientific applications, compatible with serial NetCDF library; • high-level parallel I/O interface, with collective calls associated with one file; • build on top of MPI-I/O (can benefit from MPI-I/O optimization, availability); • has proven to be efficient on some real parallel applications;	• not well documented nor supported; • not so easy to use; • variable performance and scalability; • severe size and data type limitations; • no F90/F95 interface; • lack of flexibility of the file format; • concurrency with NetCDF-4;

Table 18: Pros and Cons of parallel NetCDF library

8.2.6 *Preliminary results testing I/O parallel libraries*

Description of the testbed

In order to assess quality, robustness, efficiency and performance of various parallel I/O libraries (MPI-I/O, PnetCDF, HDF5) on petascale architecture, we have developed a small testbed. Our goal with this testbed is to compare each library with the traditional approach, the POSIX one-file-per-process strategy. Concretely, it is divided in two parts:

- One low level benchmark, the IOR code. IOR is part of the PRACE synthetic benchmark and as such has already been described in [53] and [54]. For our tests, we have chosen to have a constant aggregate file size, whatever the number of active process, with the following parameters:
  - xfersize = 2 MB
  - aggregate filesize = 256 GB
  - mode individual for MPI-I/O, HDF5 and PnetCDF (surprisingly the collective mode seems to give very bad performance)

  Ideally, with these parameters, when the peak performance is reached, the elapsed time (or the throughput) of the test should remain constant.
- One real user application, the code RAMSES. The RAMSES package is intended to be a versatile platform to develop applications using Adaptive Mesh Refinement for computational astrophysics. It is one of the two astrophysical codes of the DEISA Benchmark Suite [52].

In term of I/O, RAMSES generates huge volumes of data for each output. As the application is highly scalable, the number of processes that can be used is very high and therefore data volume is also very large. I/O is therefore a capital part of the application and choosing the right paradigm is critical for performance. Outputs generate two main files, AMR and Hydro. Each file is divided in two sections. For AMR, the first one contains parameters, local variables, information on the mesh structure and various information on the state of the application (progress of the computation, used memory,...).The second one and the largest by far contains information on the mesh structure (indices, linked lists, gravity centers, neighbours lists, index of the father cell, indices of the son cells, domain decomposition). For Hydro, the first section (small in size) contains several global variables whereas the second part contains hydrodynamics physical values for each cell (density, velocities, pressure, ...).

Initially, the original implementation relies on a one-file-per-process strategy. For the purpose of our study, we have implemented new versions of parallel I/O, using MPI-I/O, HDF5 and PnetCDF. For our tests, we used the Sedov_3d 1024 dataset which produces an amount of 96 GB per output.

Hardware description of targeted platforms

The targeted execution platforms are Babel and Vargas, the two most powerful production machines at IDRIS. Babel is an IBM BlueGene/P with 40960 cores (10 racks, 139 TFlop/s peak performance, 20 TB of memory, 16 IO nodes per rack). Vargas is an IBM POWER6 with 3584 cores (68 TFlop/s peak performance, 18 TB of memory). For a detailed hardware description of the two machines see [51]. One particularity of this configuration is that these two machines share a common parallel file system (GPFS) of 800 TB. The file system is built on LUNs (logical disk) which are distributed among 8 racks (DDN 9550). The global throughput of the disks is 16 GB/s. The disks and the two machines are interconnected by 16 I/O nodes (IBM Power6) with a theoretical throughput of 22.4 GB/s.

IOR results and commentaries

All the results presented in the following sections and summarized in the Table 19 correspond to the global throughput of the application in MB/s. "KO" means that the application crashed or did not finish (frozen).

Active process	One-file-per-process		MPI-I/O		HDF5		PnetCDF	
	Write	read	write	read	write	read	write	read
1024	1284	1453	943	1658	238	1151	202	1145
2048	1578	3332	2123	3205	449	2270	360	2178
4096	1079	6757	3466	6418	533	4085	521	3441
8192	648	8127	4066	7719	835	4253	380	3014
16384	269	8222	820	7082	377	4326	KO	KO
32768	58	7306	79	3551	60	4738	KO	KO

Table 19: Parallel IO benchmark using IOR

PnetCDF is the less mature and robust of the three parallel libraries. During our tests, it crashed or froze very often. HDF5 seems to be much more robust but performance is still poor, just a little better than PnetCDF. Finally, MPI-I/O is the only parallel I/O library that appears to be robust and efficient at the same time. On BlueGene/P, it proves to be scalable up to 8192 cores and it always outperforms (up to a factor of 6) the one-file-per-process approach for the writing process (performance is of the same order for the reading process). On POWER6, the results are much more disappointing for MPI-IO. Even if it remains the best of the three parallel libraries either in term of robustness or performance, it is no longer competitive compared to the one-file-per-process approach in term of pure performance. As the parallel file system is shared between the two machines, it can't be blamed for this divergent behaviour. Therefore, the problem is clearly due to the implementation that is not as good on Power6 as on the BlueGene/P, and this shows that a bad implementation can ruin performance even with an efficient parallel file system.

RAMSES results and commentaries

All the results presented in the following sections and summarized in the Table 20.

Active process	One-file-per-process		MPI-I/O		HDF5		PnetCDF	
	Write	read	write	read	write	read	write	read
2048	416	3045	798	706	755	590	KO	KO
4096	237	5589	975	625	863	624	KO	KO
8192	142	4613	KO	KO	KO	KO	KO	KO

Table 20: Parallel IO benchmarking using RAMSES

On a real application, the limitations of PnetCDF are reached very easily, even with this medium size test case. Hence we were unable to run properly even on only 2048 cores. For HDF5, the performance that we get is more or less the same as the one of MPI-I/O, no overhead. For the writing process, they outperform the one-file-per-process approach by a factor varying from 1.8 to 4.1. That is not anymore the case for the reading process where the one-file-per-process approach is 5 time faster. Problems arising on 8192 process for MPI-I/O and HDF5 have to be investigated further as they did not appear with the low level test IOR.

8.2.7 *Conclusion on parallel I/O libraries*

It is a bit disappointing to see that there is not a unique solution to do parallel I/O that is mature, robust, portable, efficient and scalable. PnetCDF suffers from too many problems (internal limitation, lack of robustness and performance) to be recommended. HDF5 appears to be more reliable, but in average on our tests, performance is still unsatisfactory.

Finally, MPI-I/O seems to represent the best compromise in terms of robustness and relative performance. Nevertheless, there are still too many important issues that have to be solved before considering MPI-I/O as the recommended parallel I/O library for petascale machines:

- Variable performances, even on the same machine – For example on BlueGene, poor performance when reading or important variation between collective and individual mode.
- The best scalability, but still limited between 8k and 16k active process on BlueGene for example.
- Maturity and robustness is good, but have still to be improved in order to be really considered as reliable.
- Variable implementation quality, strongly impacting the overall performance (see comparison between P6 and BlueGene/P results).

Of course, this assessment is partial and has to be enhanced and completed by running this testbed at least on some targeted PRACE WP8 prototypes. Nevertheless, it is clear that using parallel I/O libraries on a petascale machine is out of reach in its actual state of development of the various parallel I/O libraries available today, and that alternatives have to be found. Solutions like hierarchical I/O at application level or generalizing I/O forwarding at hardware level are worth the interest.

8.3 High Level Libraries

With the term High Level Libraries (HLL) we refer to libraries that allow to use more advanced and sophisticated numerical algorithms than the basic one, or to libraries that simply require or contain some other libraries. We have classified BLAS and FFT as the main functionality provided by basic libraries. LAPACK is an example of HLL since it both requires a basic library (BLAS) and implements more complex numerical algorithms with

respect to the BLAS basic library. Nevertheless, in our analysis LAPACK is an exception, in fact LAPACK is so strictly related to BLAS that it is more convenient to discuss it together with BLAS.

An important difference between basic libraries and HLL as classified in this survey is that HLL are for the majority parallel or a parallel version exists. This is quite significant because if for single core performance PRACE codes rely mostly on basic libraries, for scalability and petascaling they rely mostly on HLL. So we can certainly target HLL as a key component to investigate and optimize in petascale systems.

Analyzing the list of HLL it is possible to distinguish between "vertical" libraries used in a single application domain (e.g. METIS) and "transversal" libraries used in more application domains (like ScaLAPACK). As a consequence, in a petascale system, a problem with the performance or the scalability of "transversal" HLL will impact many different applications. Then in a petascale system a particular care should be taken to deployment of these libraries.

Like for basic libraries also for HLL there exist some vendor version like ESSP, PESSL from IBM, MKL from Intel, ACML from AMD and SciLib from CRAY, all these contain both basic and HLL libraries, a fact that should make application porting more simple. In reality HLL are not so standard yet as the basic libraries, then, quite often, coping with HLL complicates the porting activity rather than making it easier. Even worse, in some cases vendor specific HLL (e.g. ESSL, PESSL) contains different APIs than other vendors' (MKL, ACML) for the same functionality. Differences at this level create problems during the porting activities form an architecture or vendor (POWER or BlueGene) to another (standard x86 cluster). Possible problems with HLL arise also when the programmer decides to use the parallel extension. Most high level libraries don't provide the underlying communication level (usually MPI) and during the linking step programmers could encounter incompatibilities or unexpected behaviours.

Nevertheless, in general, for application performance and scalability it is correct to try to use HLL in the applications. With respect to petascaling, if some problems arose with a given HLL there are for sure more chances that the problem will be solved, since most probably both the community interested in the given library and the machine vendors are interested to make it working and exploiting the power of the machine.

It is out of the scope of this survey to make a detailed analysis of all HLL libraries, but there are few of them that, for their large importance for performance applications deserve a deeper description and analysis.

8.3.1 *ScaLAPACK*

ScaLAPACK (Scalable LAPACK) is a high level linear algebra library explicitly written for MIMD parallel computation. It implements block-oriented LAPACK linear algebra routines, adding a special set of communication routines to copy blocks of data between processors as required. Similar to LAPACK, a single subroutine call typically carries out the necessary computations. ScaLAPACK is designed for heterogeneous computing and is portable on any computer that supports MPI or PVM.

ScaLAPACK is widely used in PRACE applications. However when the number of processors becomes high, it is not easy to define the right number of processors to use. Tuning the dimensions of the sub-matrices and the number of distributed blocks involved is a key factor to reach scalability. ScaLAPACK does not contain tool that allows the programmer to be unaware of these parameters and could be really expensive to try different combinations in order to find an optimal set of parameters.

Future development of ScaLAPACK could concern improvement for numeric (accuracy of the algorithms and solvers), performance, functionalities (new algorithms or missing features) and engineering aspects (C/C++/Fortran95 bindings). The community around the development of ScaLAPACK is huge and there are a lot of industrial partners like Intel, CRAY, HP, SGI, NAG and others.

8.3.2 *PETSc*

PETSc, the Portable, Extensible Toolkit for Scientific computation, provides sets of tools for the parallel (as well as serial), numerical solution of PDEs that require solving large-scale, sparse nonlinear systems of equations.

PETSc includes nonlinear and linear equation solvers that employ a variety of Newton techniques and Krylov subspace methods. PETSc provides several parallel sparse matrix formats, including compressed row, block compressed row, and block diagonal storage. It also supports 2D and 3D mesh generation, refinement, partitioning, and distribution. Users can incorporate customized solvers and data structures when using the package.

PETSc also provides an interface to several external software packages including BlockSolve95, ESSL, Matlab, ParMeTis, PVODE, and SPAI. PETSc is fully usable from Fortran (with some limitations due to the Fortran syntax), C and C++.

8.3.3 *Vendor libraries*

With the term “vendor libraries” we refer to libraries that are provided by specific hardware vendors for its specific hardware. The most famous are MKL (Intel but also AMD), ESSL (IBM) and LibSci (CRAY). These libraries implement both shared and distributed version of a high number of algorithms, methods and numerical solvers. For example, all vendor libraries have FFT, BLAS and also LAPACK inside.

It is easy to see that vendor specific libraries generally are better than open-source ones. This is especially true for multi-threading routines because it benefits from deep knowledge about the hardware. However, due to the complexity and the “close-source” format, it is not simple to use these libraries. In the case of MKL, for example, the underlying communication layer should be provided as an external and that could create problems of compatibility or unexpected behaviour during runtime if the user will use an unsupported release.

Most of the vendor libraries are available for different platforms but all of them require a license to be used.

8.3.4 *NAG*

NAG is a non-profit software company, started in the 1970, that provides mathematical, statistical and data analysis components for high performance computations. The NAG libraries cover a lot of kernels starting from the dense and sparse linear algebra to optimization, operation research, mesh generation and statistical analysis. Most of these functionalities support parallel or shared-memory environments. The library provides interfaces for Fortran, C, C++, Java, Python and also MATLAB.

NAG libraries are not vendor specific, in fact it is possible to get the libraries for different platforms. However, as most of vendor libraries, it is not free and it requires a license to be used.

8.4 Communication libraries

As shown in Table 1, in the majority of PRACE codes uses MPI as the main layer to allow communication between processes involved in a parallel computation. MPI was widely already discussed in more than one previous section. Only one code, NAMD, uses a different communication library called CHARM++.

CHARM++ is a parallel object-oriented programming language based on C++ developed at the University of Illinois. It is used to write "explicitly parallel" programs and it supports multiple inheritance, late bindings, and polymorphism. CHARM++ is designed around the ideas to be efficiently portable, to reduce and to optimize the latency of communication, to manage transparently dynamic load balance strategies and finally to produce reusable modules. The programming model of CHARM++ is very easy. The computation is split into medium-grained processes (called *chares*). Chares may be easily organized into indexed collections called chare arrays and each chare is mapped to a physical processor (or core) by the runtime (this step is transparent to the programmer). Chares communicate by exchanging information using *messages.* For this reason, CHARM++ is classified as a message-driven language. Messages from a chare to another are collected in a pool and are automatically managed by the runtime.

Features such as application-independent object migration are very difficult to provide in MPI. Adaptive MPI (AMPI) is an implementation of the Message Passing Interface standard on top of the Charm++ runtime system and provides the capabilities of Charm++ in a more traditional MPI programming model. AMPI encapsulates each MPI process within a user-level thread implemented as a Charm++ object. In this way migrating threads and load balancing capabilities provided by the Charm++ framework in a transparent way.

Coming back to the subject of message-passing parallel programming paradigm, communication between processes can be achieved using the BLACS (Basic Linear Algebra Communication Subprograms) open-source library. BLACS exists in order to make linear algebra applications both easier to program and more portable. Key ideas behind the provided functionality are: a standard interface for different message-passing layers; data distributed on a process grid with scoped operations; contexts to identify grids and their characteristics; one-dimensional array-based matrix manipulation and communication; ID-less communication to solve the problem of tag generation in message-passing environments.

The project behind this library was started by Jack Dongarra more than 10 years ago. Although this library is very old and not frequently revised (last patch was release for in 2000), this library is a fundamental piece of other high level library like ScaLAPACK (that is totally based on BLACS communicators) but it can be also used by itself. Some architecture-dependent libraries such as MKL, Parallel ESSL and CRAY SciLib implement their own version of BLACS routines that are fully compatible with the standard open source BLACS interface (or with small differences derived from intrinsic characteristics of the considered library). Actually the presence of the BLACS inside a high level library is probably due to the fact that the mentioned libraries implement also their own ScaLAPACK.

8.5 Special Purpose Libraries

HPC codes do not use only numerical libraries. To perform some tasks like data formatting, operations automatization or build and manage user interfaces, special purpose libraries are used. In our case, the mostly used are libXML, TCL and Tk/Tix. libXML is a toolkit written in C that allows the user to build and parse XML documents. TCL and Tk/Tix are strongly

oriented to the construction of user interfaces. TCL is a scripting language natively written in C. Tk is a platform independent GUI framework for TCL. Both are interpreted.

All these special purposes libraries are not directly connected to the performance and the scalability of a HPC code. We will expect to find these libraries on the next generation of petascale systems but their development is not connected to the evolution of parallel architectures because are mainly used outside this world. However the possibility of compile and use these libraries on the future petascale supercomputer will be guarantee. Hardware and software manufacturers have to not forget to support something that is different of a numerical library but it is crucial to exploit all the features of a consolidated HPC code.

8.6 Conclusion of the assessment

Regarding external libraries used by PRACE applications, the analysis we have performed in this work has shown that different classes of library are relevant for different aspects of applications performance and HPC systems component. Basic (numerical) libraries are of fundamental importance for single core performance in single thread codes and single node performance for multi threads codes. As discussed in Chapter 7 basic libraries are becoming more and more relevant to deal with hardware accelerators, hiding from the programmer the entire burden related to the coding low-level instructions. It is not a surprise that I/O libraries are instead relevant for applications, and in the perspective of a petascale system, they could become a critical application component, since in many cases petascaling the numerical algorithm and the application I/O are different, where not competing, tasks. This is particularly true for parallel I/O libraries. Finally high level libraries turn out to be quite relevant for applications scalability, for better or for worse; in fact applications can certainly benefit from the scalability property of a library but at the same time petascaling of an application can be limited if there is a limit in the library. There are libraries used vertically inside an application domain, and other that are used transversely to application domains, and the latter are obviously more critical in a petascale system since they may impact a broader user community.

We observed that many chip makers are investing a lot to provide high performance libraries allowing applications to exploit a high fraction of the peak machine performance, this especially true for single chip performance. This is a positive thing for HPC applications and developers and the hope is that an equal effort will be applied in providing new highly scalable libraries for next generation systems.

In conclusion most PRACE applications rely on external libraries for performance and scalability. Therefore, a petascale system should provide the libraries or at least it should be possible to install them, otherwise it would be almost impossible for applications to exploit all the power, or exploit it at all.

9 Conclusions and final remarks

The work presented in this Deliverable aims at identifying and analysing the programming models and the software libraries required by petascaling applications in the PRACE implementation phase. Starting from an overview of the state of the art in the field of programming languages, models and software libraries adopted by applications running on nowadays HPC architectures, the activity reported in the Deliverable analyses the evolution of current standards for parallel programming, the next generation programming languages and models, the languages, paradigms and environments for the hardware accelerators. Then the evolution of scientific libraries for the petascale systems is investigated. Furthermore, a specific experimental activity has been addressed implementing three numerical kernels, using twelve of the main programming languages and paradigms under investigation and testing them on some of the prototypes implemented in WP8. To our knowledge, this effort on comparing so many languages on different architectures had never been done before, and represents a useful test bed to evaluate and assess new languages and paradigms for petascaling, trying to use a common methodology, as reported in chapter 7.

The analysis undertaken in this report evidences that a large number of developments are ongoing in the field of programming languages and paradigms for future HPC systems. Different players are proposing many different implementations, but none of them appears strongly consolidated and seems to have a production level maturity. If we consider that historically, especially in the world of HPC, the languages elected for the development of scientific applications are very few (mainly Fortran and C) it seems unreasonable that many other languages will be taken heavily in consideration from application developers in the next future and have a big impact on the computational community.

Probably, among all the languages promoted by the U.S. DARPA HPCS Project, we can note that there are three distinct efforts and, this approach cannot effectively consolidate on the HPC Community and will not reach the main objective of raising HPC user productivity.

The performance of the DARPA languages is still very poor (as the tests on Chapel and X10 demonstrate). This is not surprising, since both languages are still very young and the compilers released so far are more or less proof of concept, which sometime do not even implement all the features of the language.

It appears reasonable, in order to avoid wasting resources, but mainly to try to reach an effective impact for the petascale community, to redirect and focalise in a single action the work addressed by the three major players in charge for these languages. This aspect is important in order to guide the implementation effort at the benefit of the HPC community and improving programmer productivity on next-generation computing systems. We are convinced that a dispersive action can lead to the dead of the language itself.

In order for a new language to hit the ground, we should not underestimate the importance of reaching a critical mass of users, which is already a self-issue in the world of HPC. In other words, if a language remains a proof of concept, used only by a small community of computer scientists, rather than a plethora of application developers (mainly computational scientists from physics, chemistry and engineering disciplines), highly probably the language itself will soon come at a dead-end.

Moreover, it is of paramount importance to simplify the structure of these new languages in order to be appealing for an evolving scientific community. The lessons learned in the past about the ADA language should help now to avoid to repeat the previous unhappy experience.

A different perspective could involve the PGAS languages (mainly UPC and CAF). In facts, these implementations are more consolidated and promising, also because the syntax constructs and the rules are closer to the ones of classical HPC languages.

Even if they will never become a real alternative to the current programming models (based on MPI over C and Fortran), it seems reasonable to speculate that their main characteristics will be well accepted by the scientists involved in developing computational codes. The availability of stable and efficient compilers (hopefully available in the next future) and perhaps HPC systems with interconnections directly supporting the PGAS programming model, will greatly contribute in this direction. Moreover, it will be of paramount importance, in order to consolidate the adoption of languages for the scientific HPC community, the strong and complete adoption of the CAF and UPC syntax and semantics as part of the next ANSI/ISO standards for Fortran and C. In this way it will be possible for the related PGAS programming model to gain much momentum as a novel approach to improving programmability and productivity of large-scale computing architectures.

MPI and OpenMP are arguably the de facto community standards for distributed-memory programming and shared-memory programming respectively. MPI and OpenMP put together distributed-memory (provided by MPI) and shared-memory programming model (provided by OpenMP). While these technologies have served the parallel programming community admirably over the last couple of decades, recent advances in massively parallel multi/many-core architectures have revealed their fundamental limitations in the productive design of high performance codes.

The adoption of the classical programming models based on the traditional message-passing paradigm, seems not scaling efficiently for all petascale applications. With current ubiquity of multi-core architectures, a more promising and efficient approach seems the hybrid programming model based on coupling message-passing and multi-threading. Actually a big effort is in place with the enhancement of MPI plus OpenMP or MPI plus standard threads. The availability of petascale architectures, based on multi/many core processors will greatly benefit from the hybrid programming model in order to effectively scale up the applications.

Hardware accelerators have today a strong vitality and they produce great interest for enhancing performances in HPC architectures. They seem another topic to consider in the direction of petascaling. To efficiently exploit these hardware accelerators, from the more exotic to the nearly commonly available, specific languages or programming environments have been defined and experimented in different applicative contexts, as documented in Chapter 7.

At the current time it is difficult to identify common directions and guidelines in this field, as the technological evolution imposed by the hardware is so rapid and the experimentation on languages and tools has not reached a stable maturity. GPUs are currently the prevalent accelerators, although they have not been used very often for production runs. A bunch of languages exist to program GPUs. A general impression is that all these languages are still ”*hardware driven*” and far from being consolidated. For example, it is clear that using CUDA for scientific codes is somewhat comparable to use assembler code to improve the most important kernel routines.

It is very desirable that some standard will emerge soon in this context. A fixed standard should allow to run codes on different accelerators, solving the problem of code maintainability when the hardware technology evolves. The language on which most people pinned their hopes is probably OpenCL. However the actual compilers, released so far, are still very immature and the performances still very poor.

Another important issue in the direction of portability and performance should be represented by tools like RapidMind and the CAPS HMPP environment, but it is still too early to see what will be their evolution and their impact on the applications development. CAPS HMPP only generates code for few kind of accelerators and RapidMind has been recently bought by Intel to merge the language with the Intel's development toolkit. Probably a good opportunity to improve the programming models for accelerators, could be represented by the availability of compilers adopting specific APIs to allow a semi-automatic code generation for accelerators. Some preliminary work in this direction has been addressed by the PGI compiler which allow the automatic code generation for GPUs.

During the assessment of the libraries clearly appears that different categories of libraries are relevant for different classes of applications in relation to specific performance aspects in order to raise the global performance. Adopting optimized numerical libraries (especially in the case of hardware accelerators) and highly scalable I/O libraries is usually the first mandatory step for an efficient porting of applications to new Petaflop/s systems. Relevant examples in this sense are the PRACE codes selected for the PRACE Benchmark Suite. Most of those codes use basic, high-level and I/O libraries in different combinations, that have a high relevance to achieve target performances.

Moreover, many of the actual numerical libraries implement algorithms showing limitations when moving from teraflop/s to petaflop/s partition sizes. As a consequence, for some of these libraries it will become necessary to implement new numerical algorithms ensuring real high scalability and good numerical stability.

In summary, the Deliverable presents an analysis of the maturity, suitability, applicability of novel programming models, languages and scientific libraries for petascaling. The information is useful also for other WPs (such as WP5, WP7 and WP8) to better address the evolution of applications, but also the architecture and the programming environment for the first and the future generations of Petaflop/s systems.

10 Annex A: Reference Input Data Sets (RIDS)

The working group of people involved in the evaluation of new programming languages and models agreed on four kernel benchmarks that should be used for a comparative analysis of more than 10 languages:

Kernel	Abbreviation	Algorithm
mod2am	am	Dense matrix-matrix multiplication
mod2as	as	Sparse matrix-vector multiplication
mod2f	f	1D-FFT
mod2h	h	Random Number Generator

Table 21: The 4 chosen EUROBEN synthetic kernels.

All kernels have been taken from the EUROBEN synthetic benchmark suite, which has already been used within the PRACE work packages WP5, WP6 and WP7. Since the effort available for porting these kernels to the new languages was quite small, it was agreed that the kernels should be ported in the proposed order, which means that mod2am and mod2as ports are available for (almost) all languages, while a mod2f port exists only for 8 languages, and mod2h has only been ported to ClearSpeed's Cn language, where a special random number generator comes with the SDK.

During the porting phase of the kernels it became evident that for two kernels, namely mod2am and mod2as, the problem size can have a major impact on the observed performance. For the other two kernels, there is hardly any performance sensitivity to the input size for different reasons: Firstly, due to the choice of the FFT type (self-sorting radix-4) the problem sizes are restricted to sizes of 2 to the power of n. Secondly, for mod2h it turns out (as expected) that the performance is virtually independent of the problem size.

To be able to compare the performance measurements for all languages, the group decided to use reference input data sets (RIDS) for each kernel. Aad van der Steen, the author of the EUROBEN benchmark suite and an active member of PRACE WP6 and WP8, proposed the input sizes below which have been used in the remainder of this document for performance measurements.

10.1 mod2am

It has turned out that some problem sizes do either well or not so well on accelerator platforms. As the goal was to capture the "good" as well as the "bad" behaviour of the systems both type of sizes must be represented. Multiples of 96 have been chosen to show the potential of the Clearspeed accelerator cards, while multiples of 512 have been chosen to make optimal use of NVIDIA Tesla GPUs.

The following table has been used as RIDS for mod2am:

No	M	L	N	Repetition
1	10	10	10	100
2	20	20	20	25
3	50	50	50	10
4	100	100	100	5
5	192	192	192	3
6	200	200	200	2
7	500	500	500	1
8	512	512	512	1
9	576	576	576	1
10	1000	1000	1000	1
11	1024	1024	1024	1
12	2000	2000	2000	1
13	2048	2048	2048	1
14	5000	5000	5000	1
15	10240	10240	10240	1
16	12096	576	12096	1
17	12096	1152	12096	1

Table 22: RIDS for mod2am

10.2 mod2as

The sparse matrix-vector multiplication is less dependent on the problem size. This is due to the fact that mod2as is an example for an algorithm with low computational complexity which is not very well suited for the use of accelerators (at least as long as memory has to be transported through the PCI bus).

The RIDS for mod2as is:

No.	Rows	Columns	Non-zeros
1	100	100	350
2	200	200	1500
3	256	256	3276
4	400	400	7000
5	500	500	12500
6	512	512	10485
7	960	960	41500
8	1000	1000	50000
9	1024	1024	57670
10	2000	2000	300000
11	4096	4096	587200
12	4992	4992	996800
13	5000	5000	1000000
14	9984	9984	4485600
15	10000	10000	5000000
16	10240	10240	6000000

Table 23: RIDS for mod2as

10.3 mod2f

The RIDS for mod2f is:

No.	Transform Length	Repetition
1	256	250
2	512	125
3	1024	65
4	2048	35
5	4096	20
6	8192	15
7	16384	10
8	32768	5
9	65536	5
10	131072	3
11	262144	3
12	524288	2
13	1048576	2

Table 24: RIDS for mod2f

10.4 mod2h

The RIDS for mod2h is:

No.	Input size
1	100000
2	200000
3	500000
4	1000000
5	2000000
6	5000000
7	10000000

Table 25: RIDS for mod2h

11 Annex B: Hardware Overview

This annex contains a synthetic description for each PRACE prototype or system used during the porting activity.

11.1 System "clearspeed-petapath" (WP8-NCF)

- Basic Hardware:
  - Intel X5500-based host server
  - Accelerators connected via PCIe Gen.2 16x (8 GB/s)
- Hardware Acclerators:
  - Feynman e740 (4 ClearSpeed CSX700 cards)
  - Feynman e780 (8 ClearSpeed CSX700 cards)
  - CSX700 card:
    - two 96 multi-threaded Array Processors (MTAP), 250 MHz
- Operating System:
  - Scientific Linux

11.2 System “itanium” (LRZ)

SGI Altix 3700

- Basic Hardware:
  - 9728 cores, 512 cores per node
  - Intel Itanium2 Montecito dual core, 1.6 GHz.
  - 4 GByte memory per core
- Operating System:
  - SLES 10

11.3 System "huygens" (WP7-NCF-SARA)

IBM Power6 cluster

- Basic Hardware:
  - 104 nodes, 32 cores per node
  - 128 or 256 GByte memory per node
  - IBM p575 Power6@4.7 GHz
- Operating System:
  - SLES 10 SP2

11.4 System "louhi" (WP7-CSC)

Cray XT

- Basic Hardware:
  - AMD Barcelona, 2.3 GHz quad-cores
  - 2 quad-cores per node in a shared memory
  - 1 GByte memory per core.
- Operating System:
  - CNL, CLE 2.2 UP00

11.5 System "cell-cluster" (PSNC)

- Basic Hardware:
  - Cell Cluster
  - IBM QS22 blades (contains 2 PowerXCell8i)
- Hardware Accelerator:

  - PowerXCell8i
- Operating System:
  - Fedora 9

11.6 System “maxwell” (WP8-EPCC)

- Basic Hardware:
  - 32 IBM Blade Cluster
- Hardware Accelerator:
  - 64 Virtex-4 FPGAs

11.7 System "nehalem" (“ice” WP8-LRZ, “inti” WP7-CEA, “baku” HLRS)

“**Nehalem/ice**”:

- Basic Hardware
  - SGI ICE system
  - 384 cores
  - Intel Nehalem EP, dual-socket, quad-core, 2.53 GHz
- Operating System:
  - SLES 10.2

“**Nehalem/inti**”:

- Basic Hardware
  - Bull cluster
  - Intel Nehalem EP, dual-socket, quad-core, 2.53 GHz, with hyperthreading
- Operating System:
  - CentOs

“**Nehalem/baku**”:

- Basic Hardware
  - Intel Nehalem EP, dual-socket, quad-core, 2.53 GHz
- Operating System:
  - Scientific Linux

11.8 System “uchu” (WP8-CEA)

- Basic Hardware:
  - 4 Bull R422 servers
    - Intel Hapertown CPU with 16GB of RAM
    - connected through PCI-Express 16x to two TESLA servers
- Hardware Accelerator:
  - 2 NVIDIA Tesla, 1.3 GHz
  - 1 Tesla server has 2 C1060 graphic boards
- Operating System:
  - CentOS release 5.3 (Final)

12 Annex C: Software Engineering Metrics

In the field of software engineering a few well-established and easy measureable quantitative metrics to assess the complexity of a certain implementation are used ([57]). The most popular being:

- Physical source lines of code (SLOC)
- Logical source lines of code measured in (NCSS)
- Time-to-solution & others

It is further important to note that some of the metrics listed above are not necessarily tied to a kind of programming language but more to a kind of programming style, e.g. "readability" benefits a lot from good and enough comments and can be accomplished in every language.

12.1 SLOC: Source Lines of Code

The physical measurement of source lines of code (SLOC) originates from the assembler language times, when one line of code consisted of exactly one statement. SLOC is very easy to measure but greatly depend on programming style. Since it is an old metric it has been heavily used in many software-engineering studies. Findings have been, for example that SLOC has an inverse relationship with the defect rate. Furthermore, It has been found that the number of lines programmed in one day stays nearly constant for one particular programmer on one project.

12.2 NCSS: Non-Commented Source Code Statements

A somehow "better" measure for modern high-level languages is the number of non-commented source code statements, sometimes also known as the *logical* source lines of code. It includes all executable source code statements, without regard to the placement of carriage returns or other stylistic elements. Pre-processor directives and comments will be neglected and sometimes even variable declarations are excluded. NCSS can be measured by counting all semicolons in a C-program. "Non-commented source code statements can also be thought of as a measurement of the quantity of source code required to accomplish a given task. Given two versions of a program, the one with a lower NCSS value suggests a cleaner solution." ([58]).

12.3 Time to solution & others

Studies suggest that many other qualitative productivity metrics (e.g. readability, portability, correctness and -perhaps more surprisingly- total effort and therefore time-to-solution) are related to NCSS. One should keep in mind that NCSS has also several shortcomings, especially when it is used for comparison of programmers with different levels of skills.

Other necessary prerequisites for good programmability are programming tools, which play an important role in software development, profiling and optimization for HPC. For most emerging platforms, programming tools tend to be immature considering that most accelerator hardware has initially not been developed for numerical simulations.

13 Annex D: Template of the Porting Diary

Reporting Template for the Evaluation of new Programming Languages

Part I - Basic Information

Developer

Name		Institution	
Email		PRACE Partner	
Telephone		Contributors	… if applicable …

Code

Euroben Kernel (mod2am, mod2as, mod2f, mod2h)	
Other	

Programming Language

(MPI+OpenMP, UPC, CAF, Chapel, X10, CellSs, CUDA, CUDA+MPI, OpenCL, RapidMind, Cn, CAPS, or other)

Language	
Compiler Version	

Hardware Information

Prototype Name	
Basic Hardware	
Hardware Accelerator	
Operating System	
Drivers	

Part II - Developer Diary
Questions? Please contact iris.christadler@lrz.de

Development Time (mins, hours or days)	**Achieved Performance** (Mflops, MLUs, sec)	**Number of Cores** (if applicable)	**Dataset used** (if applicable)	**Comments** (What did you do during this porting step? Why? Problems faced, etc.)

Part III – Porting results

Overall implementation result (Successful/not successful/partly successful)	
Consistency of the numerical results (Are the results the same as for the scalar "C" version of the kernel?)	
General comments	… free format …
Problems encountered	… free format …
Maturity of the standard/compiler	… free format …
Number of lines of source code	… free format …
Suitability of the compiler/ paradigm to implement the kernel	… free format …

14 Annex E: Reported Results for all Languages

The following information has been provided by the programmers who did the actual porting work. The information has been extracted from each Task 6.6 Reporting Templates and has been adapted or unified whenever it seemed appropriate.

14.1 Coarray Fortran (CAF)

Basic Information

Author: Sami Saarinen (CSC)

Euroben Kernel: mod2am, mod2f (mod2as was not started since too many compiler problems encountered during the mod2am port)

Hardware: *louhi*, *itanium*

Compiler Version: cce 7.1.2 on *louhi*; g95 on *itanium*

Developer Diary

mod2am

Development time: 3 days on Cray XT to convert the UPC version to CAF using DGEMM; 1 hour to port the existing CAF version to the SGI Altix system. Achieved Performance:

- Cray XT5: 200 Gflops on up to 64 cores
- SGI Altix: 1 Gflops on 1 core

mod2as

After experiencing severe problems with the CAF compiler on Cray XT during the porting of mod2am (leading to re-producible compute node crashes), the mod2as port was not even started. It should be easy to port this kernel, since now a working UPC version and the compiler environment at LRZ is available.

mod2f

Development time: 7 days on Cray XT to convert the main parts from the UPC version to CAF.

Achieved Performance:

- Cray XT: 5 Gflops on up to 16 cores

Porting Results on Cray XT (CSC)

Overall implementation result: Partly successful

Consistency of the results:
Yes and no. No because GASNet problems (must have GASNET_DISABLE_MUNMAP=1), but still goes haywire above 16 cores for FFT (mod2f).

General comments:

Cray XT cce CAF is NOT ready for production. Lack of process groups in CAF makes coding sometimes harder than in UPC or MPI.

Problems encountered:
- cce CAF compiler has bad optimization problems and code generation errors:
- for example coarray(0:npes-1)[*] does not work, but coarray(1:npes)[*] works okay! does incorrect pattern matching, but if disabled (-Onoopt), the execution may crash, but certainly slows down execution dramatically
- when using two-dimensional codimension, the runtime code seems to crash compute nodes arbitrarily may not be related to two-dimensionality, but source code bug, but still : crash !! (in mod2am)

Maturity of the standard/compiler:
immature

Number of lines of source code:
- *mod2am*: 310 Fortran; 40 C-code (timing utility) , all including empty and comment lines
- *mod2f* : 450 Fortran (main program + transpose code); 630 C-code (major FFT-kernel)

Suitability of the compiler/ paradigm to implement the kernel:
- *mod2am*: poor due to lack of process groups in CAF
- *mod2am*: restricted to certain number of processors (sqrt(NPES) ** 2)
- *mod2f* : good suitability – can be run with the same NPES than with UPC

Porting Results on SGI Altix (LRZ)

Overall implementation result: Successful

Consistency of the numerical results:
Yes

General comments:
- g95 is very strictly conformant CAF compiler
- requires codimension-keyword in coarray declaration

Problems encountered:
- lacks high performance communication i.e. parallel job runs like a snail
- critical – end critical –construct not recognized for unknown reason

Maturity of the standard/compiler:
- good, except missing proper parallel implementation

Number of lines of source code:
See above

Suitability of the compiler/ paradigm to implement the kernel:
- poor due to lack of process groups in CAF
- restricted to certain number of processors (sqrt(NPES) ** 2)

Performance Measurements

mod2am

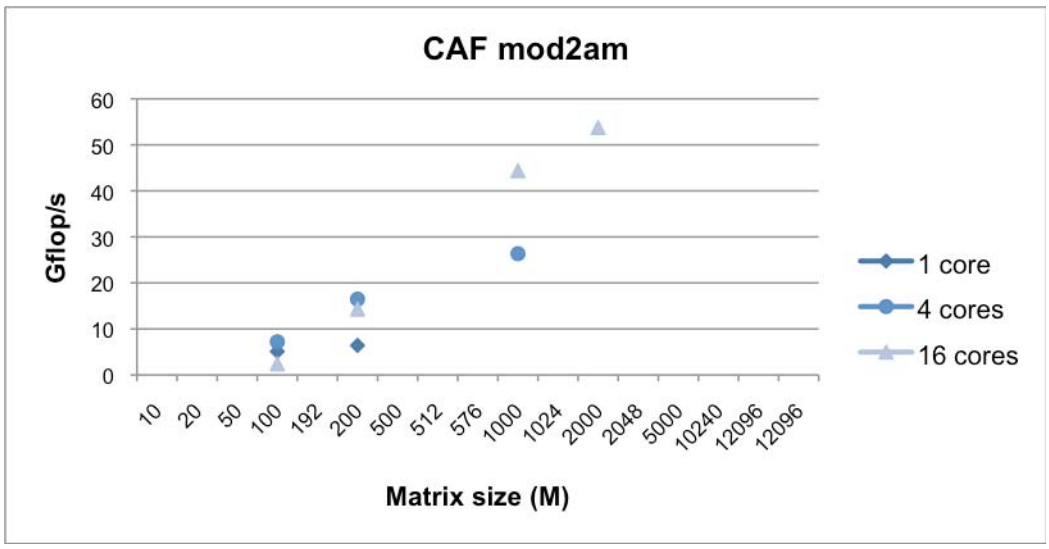

Figure 31: CAF mod2am performances (tests performed on *louhi*)

mod2f

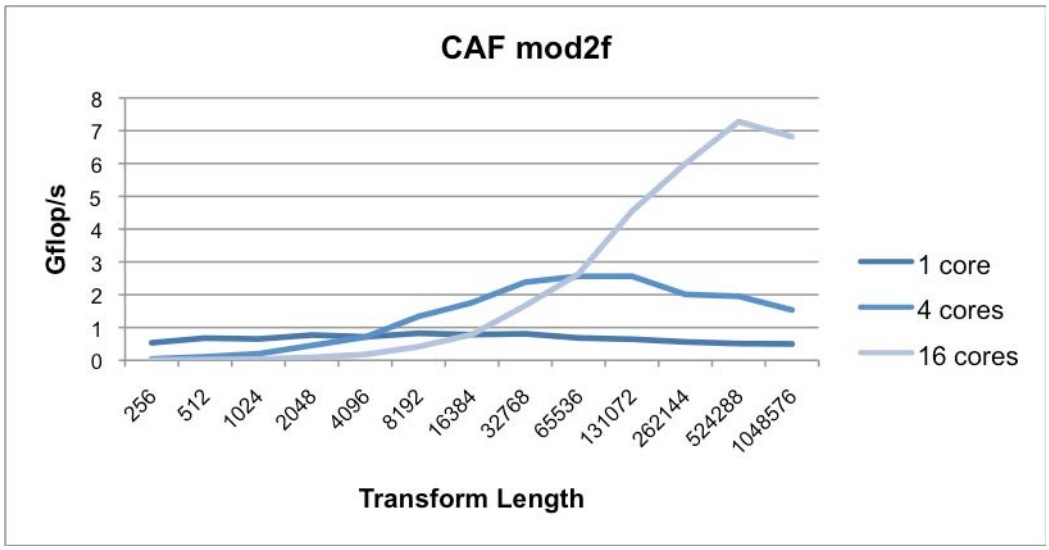

Figure 32: CAF mod2f performances (tests performed on *louhi*)

14.2 CAPS HMPP

CAPS HMPP v2.0.0 has been used in this deliverable. Results for a newer release should be ready for D8.3.2.

Basic Information

Author: Guillaume Colin de Verdière (GENCI-CEA)
Euroben Kernel: mod2am, mod2as, mod2f not started
Hardware: *uchu*
Compiler Version: CAPS HMPP v 2.0.0

Developer Diary

Porting Results

Overall implementation result: Successful

Consistency of the numerical results:
Correct

General comments:
In both cases the initial port was straightforward. Yet the achieved performances were far from the peak performance. It required some tweaking to get decent performances. This shows that a given code, developed for a CPU, will always need to be visited again to get efficient use of a GPU: its internal structure has to be changed according to the hardware capabilities. In this case the compiler is not faulty. Rather, the compiler can be of great help to solve such issues. HMPP does fall into the category of helpful, yet young, tools for GPU computing.

Problems encountered:
- *mod2as*: the reduction was not supported by this release of HMPP. It had to be hand programmed which was somewhat tricky.
- *mod2am*: unrolling and jamming were not functioning properly with older versions of the compiler. The version 2.0.0 fixed the problem and permitted to reach the CUBLAS level of performance.

Maturity of the standard/compiler:
The HMPP compiler is still young in that it has the ability to support more classical constructs such as reductions. The next versions should be really promising in terms of performance and ease of development;

Number of lines of source code:
- *mod2as*: 976 including lengthy comments and various versions of the HMPP port.
- *mod2am*: 979 including lengthy comments and various versions of the HMPP port.

Suitability of the compiler/ paradigm to implement the kernel:
Very good. The HMPP compiler is a great tool to port codes to a GPU. We will see more features with next releases. One should remember that some problems are not suited for GPU usage. In those cases HMPP will still be useful since the CPU can be used as well.

Performance Measurements

mod2am

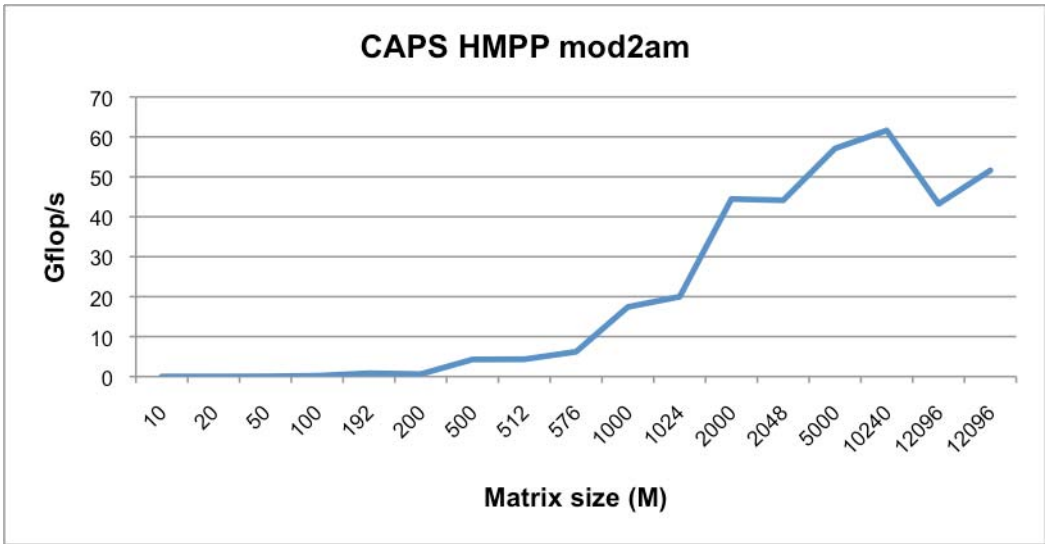

Figure 33: CAPS HMPP mod2am performances (tests performed on *uchu*)

mod2as

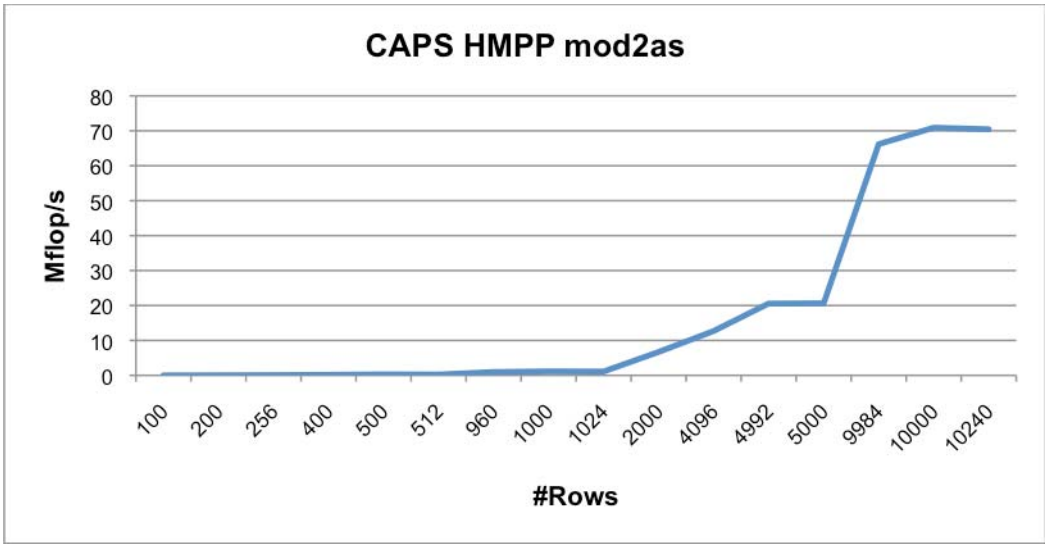

Figure 34: CAPS HMPP mod2as performances (test performed on *uchu*)

14.3 Cell-Superscalar (CellSs)

Basic Information

Author: Ramnath Sai Sagar (BSC), Maciej Cytowski (PSNC)
Euroben Kernel: mod2am, mod2as, mod 2f
Hardware: *cell-cluster, maricel*
Compiler Version: CellSs V2.2

Developer Diary

mod2am

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
3d	35Gflops/s	8 SPUs	960x960 20 times	1)Code with input matrices converted to block 2) Multiplication of the blocked matrices 3) Storing the resultant matrices from block to the original format 4) Works for input matrices multiples of 64 x 64
4d	41 Gflops/s	8 SPUs	1000x1000 20 times	Works for any input matrix size
5d	60 Gflops/s	8 SPUs	1000x1000 20 times	Few optimization especially with respect to the memory allocation
6d	86Gflops/s	8 SPUs	1000x1000 20 times	Removed the conversion to block and conversion from block matrix tasks. Instead , the inputs were stored in the blocks right at the beginning.

mod2as

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
1 day	0.7Mflops/s	8 SPUs	rows:2000 cols:2000 nelmts:300000	(ver 1) Gather of the input vector was done in PPU. The multiplication of a row of a matrix with the input vector was done in a SPU
6 days	12Mflops/s	8 SPUs	rows:2000 cols:2000 nelmts:300000	(ver2) • Gather task to build the input vector for group of rows. • Was done creating a dma list, with each dma transfer of 16 bytes, and then shuffling to the output vector
10 days	2Mflops/s	8 SPUs	rows:2000 cols:2000 nelmts300000	(ver 3) Instead of doing gather and multiplication task for row by row, instead of group of rows. However it had poor performance as the tasks were very small
15 days	30Mflops/s	8 SPUs	rows:2000 cols:2000 nelmts:300000	(ver 4)Reverted back to ver2. Implemented double buffering due to the poor bandwidth. Improved the performance to some extent.
20 days	40Mflops/s	8 SPUs	rows:2000 cols:2000 nelmts:300000	(ver 5) Parsed through the input dma list and created a local table for same vector values, there by avoiding duplicate dma transfers. Improved the performance to some extent

mod2f

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
1h	33.49	1PPU	1048576 (10 times)	Original version of the code running on the Power Processing Unit of the Cell chip – no optimization applied yet. Very poor performance! https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver1/
8h	154.3	16 SPUs	1048576 (10 times)	First try: porting of the implemented algorithm step by step. Here, the first step was implemented: parallel computations of sin and cos table. Additionally the data layout has been changed (now a table of complex double precision numbers is created - good for performance) https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver2/
16h	195.9	16SPUs	1048576 (10 times)	Parallel implementation of the first two radix-4 iterations. Performance gain is rather poor, also for smaller problems. I need to redesign the porting process. https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver3/
32h	130	16SPUs	1048576 (10 times)	Starting from the beginning. Completely new FFT implementation with CSS. No SIMDization here. Many barriers for debug. https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver4/
32h	322	16SPUs	1048576 (10 times)	Code was partially SIMDized. Bit reversal is now performed on-fly (DMA put calls to bit reversed memory addresses) from SPUs (in parallel). https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver5/
16h	369.8	16SPUs	1048576 (10 times)	Full SIMDized code. It should push the performance when after some further tuning. Code need to be further optimized. The computational part is not the most expensive one. Looking for reasons.. https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver6/
32h	677.8	16SPUs	1048576 (10 times)	I've found a reason of poor performance – very inefficient way of reversing bits in 32bit numbers. Changed to fast bit reversing method. The performance of my FFT starts to look good.. https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver7/
32h	867.8	16SPUs	1048576 (10 times)	Looking for some more optimizations. Code inlining and loop unrolling was added to this version. Both of these optimizations were performed by hand. The performance should be better. Looking for reasons... https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver8/
112h	2656	16SPUs	1048576 (10 times)	Some of the FFT symmetry rules was used. Some CSs parallel tuning was performed. SIMDMath library trigonometric functions were inlined. Many other changes. Assumption: the CSs version will be used for M>15 due to task granularity problems for smaller sizes. https://trac.csc.fi/pracewp6/svn/synthbench/eurobenports/cellss/PSNC_ICM/mod2f/ver9/

Porting Results

mod2am

Overall implementation result: Successful
Highly successful, able to reach up to 86% of the peak performance. Was easy to port with the knowledge of C. Efforts had to be taken with respect to the memory allocation.

Consistency of the numerical results:
The results are the same as for the original "C" version of the kernel.

General comments:
A very good programming model. The code overall was easy to port especially due to the knowledge of C. Initial version did not give impressive performance. However, simple restructuring of the input data right at the beginning increased the performance to 86Gfopls/s. Profiling the code with Paraver tool gave an insight on the application execution, which eventually helped in improving the overall performance of the kernel.

Problems encountered:
None, except for the optimization of the code.

Maturity of the standard/compiler:
CSs v2.2 works well on any Cell platform. The language is well documented. Additionally Paraver tool could be used for parallel traces analysis.

Number of lines of source code:
Total:305 lines; Kernel: 35 lines

Suitability of the compiler/ paradigm to implement the kernel:
The paradigm is well suited for performing parallel computations on blocks of data (i.e. running many similar tasks on different blocks of data). Since the matrix multiplication is done is as blocks of sub matrices, it suits the best.

mod2as

Overall implementation result: Partly successful
Performance improvement was not impressive.

Consistency of the numerical results:
The results are the same as for the original "C" version of the kernel.

General comments:
A very good programming model. The code overall was easy to port especially due to the knowledge of C. Initial version did not give impressive performance. . However great deal of knowledge is required about the architecture, especially the memory related issues to improve the performance. Special notice had to be made due to the limited size of local SPE memory and no memory protection.

Problems encountered:
Poor performance and poor bandwidth.

Maturity of the standard/compiler:

CSs v2.2 works well on any Cell platform. The language is well documented. Additionally Paraver tool could be used for parallel traces analysis.

Number of lines of source code:
Total: 529 lines; Kernel: 153 lines

Suitability of the compiler/ paradigm to implement the kernel:
The paradigm is well suited for performing parallel computations on blocks of data (i.e. running many similar tasks on different blocks of data). Sparse matrix multiplication is however not an perfect match over here as the number of multiplications in each of the task differs. Further, as the input vector is scattered across the main memory, the PPE SPE throughput was very low. An experiment of restructuring of the input vector in consecutive memory location improved the performance. However it was not considered for porting for the "fair" policy of porting.

mod2f

Overall implementation result: Partly successful
We failed to optimize the original kernel due to low performance problems, in return we proposed a modified version of the FFT specially designed for the CSs programming language

Consistency of the numerical results:
The results are the same as for the original "C" version of the kernel.

General comments:
CellSs is a nice programming paradigm from a Cell/Multi-core programmer's point of view. Except from the language itself we were able to use Performance Analysis Tool called Paraver which is available for CSs. It could be used to analyze the parallel traces of the CSs application. During the porting process we have learned a lot of important issues regarding CSs and Cell/Multi-core programming itself:

- avoid synchronization, parallelize as much as you can with the use of similar block data structures
- use low level Cell programming techniques (SIMD, DMA transfers)
- choose a good granularity of the CSs task

The final version works with CSs for problems larger than 2^M, where M>15. The initial performance on PPU was very poor: 33,5 MFlops (2^20 problem). The final performance is 2656 MFlops (2^20 problem).

Problems encountered:
Very poor performance of the PPU version of the code. We failed to obtain good parallel performance with original kernel due to difficult memory access pattern and problems with granularity of the CSs task.

Maturity of the standard/compiler:
CSs v2.2 works well on any Cell platform. The language is well documented. Additionally Paraver tool could be used for parallel traces analysis.

Number of lines of source code
273 lines of code

Suitability of the compiler/ paradigm to implement the kernel:
The paradigm is well suited for performing parallel computations on blocks of data (i.e. running many similar tasks on different blocks of data). FFT is not a perfect match here since the number of operations performed on one block of data is limited to some very basic twiddle computations. The granularity of the task seems to be a big problem here, since we are additionally limited by the very small Local Store of each SPU (256KB). To achieve good performance some additional Cell programming techniques like SIMD computations or DMA transfers have to be used. Some of them are partially addressed by the CSs language itself.

Performance Measurements

mod2am

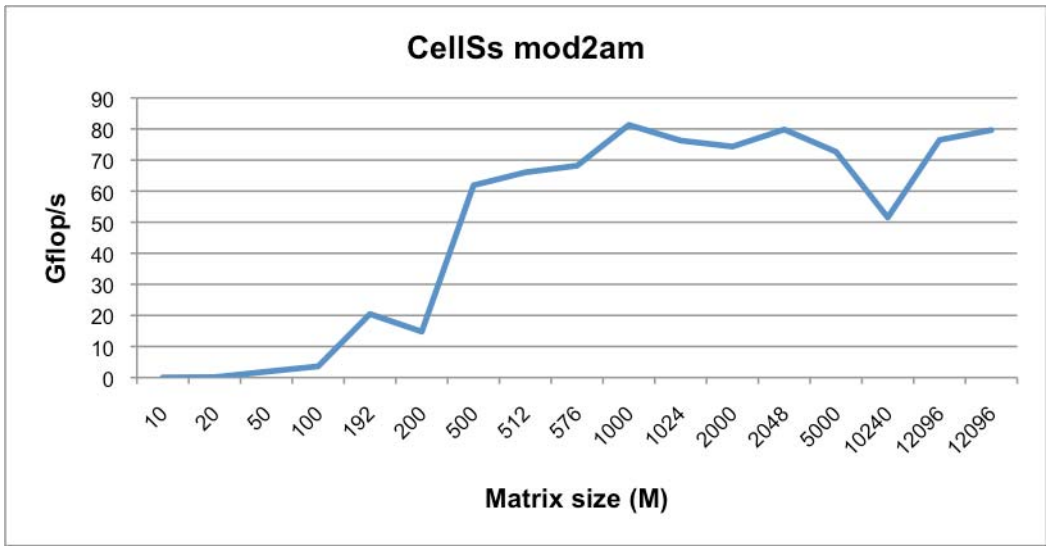

Figure 35: CellSs mod2am performances (tests performed on *maricell*)

mod2as

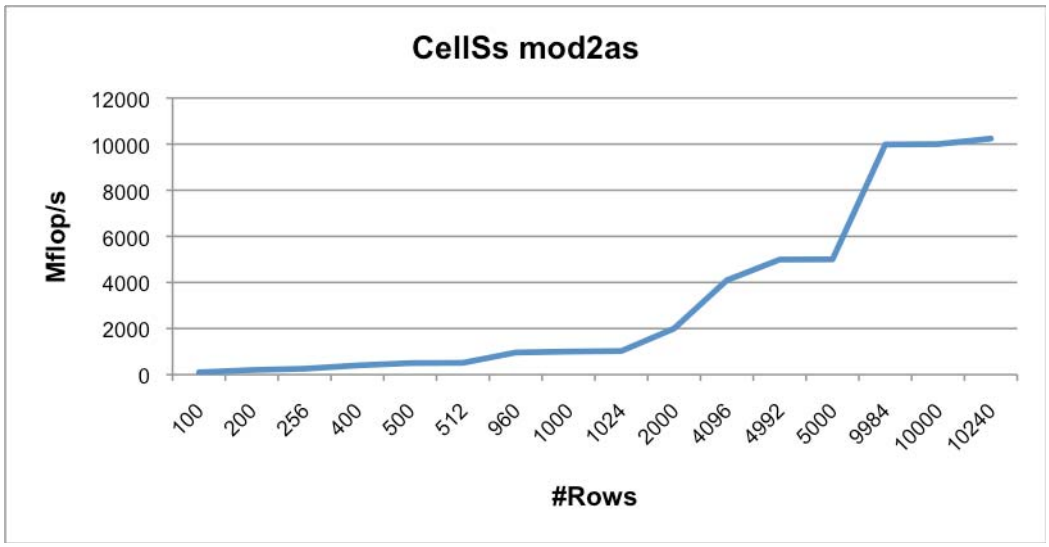

Figure 36: CellSs mod2as performances (tests performed on *maricell*)

mod2f

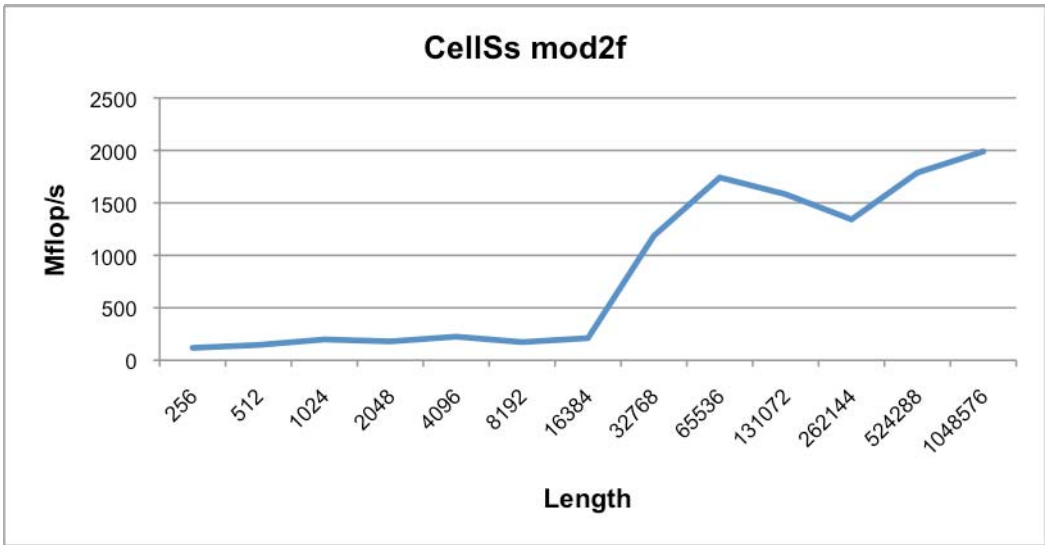

Figure 37: CellSs mod2f performances (tests performed on *cell-cluster*)

14.4 Chapel

As I have stated earlier, there is a big issue with performance using the current release of Chapel, i.e. version 0.9 from early this year. Let me get a bit technical in order to make the problem clear. Chapel introduces two basic concepts: domains and distributions. Domains are basically index set; arrays are defined over these domains. Domains can be arithmetic and structured, i.e. 1/2/3D arrays, but they also may be sparse. Distributions are basically a mapping between domain indices and the location, i.e. node, where array elements corresponding to those indices are stored. In other words, they control how the array is distributed across the cluster. Distributions are provided as part of the standard module library or user defined (which is non-trivial and not documented yet). At declaration time, domains may be associated with a previously declared distribution. Otherwise they use the default distribution, which keeps everything on the master node.

In Chapel loops over arrays (actually domains), as eg the parallel forall loop

```
forall (i,j) in aDomain do
  calculate something
```

are done by calling a method of the distribution associated with a Domain. This method is basically an iterator over all indices and is supposed to yield in parallel/concurrently across the machine (and also support multi-threading on a multi-core node). Similar iterators are used for reductions, etc.

The problem now is, that most standard module distribution's iterators are not mature in the current release. Often they fall back to their serial version. In consequence, the array operation is done on a single node, sometimes even on a single core. This is true for the two distributions I am currently using for the benchmarks making them not scale at all.

Another issue is the fact, that the current Chapel compiler is a prototype only and not supposed to be as performant as a C compiler would. Even with single-core, sequential implementation, I get only a fraction (5%) of the peak performance (I just checked the reference C implementation and am surprised to find, that it is not much better with 7% peak). Using proper Chapel parallel constructs is at least an order of magnitude lower. Together this is most annoying and makes the resultant performance measurements very bad. Usually I am happy if I get near 100MFlops on a single node. In addition, the sparse matrix mult doesn't scale at all currently. The dense matrix-matrix mult scales on a single node across its cores, but not across multiple nodes. In view of this, I don't think it makes sense to report performance numbers in the deliverable. Chapel currently is simply neither expected to perform well nor does it.

On the other hand programming Chapel compared to programming MPI is very productive in terms of time effort. You basically write your program as a serial version. The magic is choosing the proper distributions that is done in the declaration part, not in the algorithmic. In order to increase performance for critical sections, Chapel offers means to optimize data locality and affinity on a case-by-case basis.

The beauty of Chapel approach, is that in most cases you can separate algorithmic considerations from parallelization issues. The body of your for loop will always be nearly the same. I would _suggest_ not to publish performance measurements, at least not for multi-node runs, in D6.6, but instead explain the issues along the lines I did in this mail. That said, I have no problem to run the new data sets.

What I wrote is true for the current release v0.9. The situation is a bit better if you use the Chapel version from svn which is updated several times a days. However, this is a pure development repository and breaks frequently.

Basic Information

Author: Jose Gracia (HLRS)
Euroben Kernel: mod2am, mod2as (mod 2f taken from SDK examples)
Hardware: *nehalem/baku*
Compiler Version: V0.9, standard library module DistCSR.chpl from svn rev 15857

Developer Diary

mod2am

Development Time	Achieved Performance	Number of nodes x cores	Dataset used	Comments
60m		1 x 1	1000x1000x1000	v0; sequential emulating C
30m	230	1 x 1		v1: parallel "forall" loops, inner reduction
30m	1025	1 x 8		v2: array distribution -> multi-core, in principle also multi-node
60m	850	1 x 8		v3: blocked matrix multiply -> data locality

mod2as

Development Time	Achieved Performance	Number of cores	Dataset used	Comments
60m	567	1	2000x2000 @300000	v1: dense matrix, sequential
-	42	1		same as above, normalized to nelmts
60m	2.3	1		V2: sparse subdomains, loop over dense domain
30m	25	1		V3: sparse subdomain, loop over sparse domain
120m	37	1		V4: arrays distributed, workaround iterator, inner reduction; still serial execution
30m	40	1		V5: proper iterator from Chapel svn, still serial

mod2f

mod2f was taken from the SDK examples

Porting Results

mod2am

Overall implementation result: Partly successful

Consistency of the numerical results:
yes

General comments:
Implementation of data parallel problems in Chapel is in principle very simple. Parallelism is achieved through 1) parallel "forall" loops, which allow all cores in a node to work concurrently; 2) "domain distribution" which implement work and data distribution across nodes. However, as of Chapel version 0.9 runtime /standard modules does not really execute the program in parallel and/or distributed across several machines.

Problems encountered:
Sometimes documentation is poor, in particular the language specification. Code examples included in the distribution and presentations/paper on the web are more useful.

Maturity of the standard/compiler:
Specification: ok. Compiler & runtime are still prototypes.

Number of lines of source code:
The actual algorithm is less than 10 lines of code.

Suitability of the compiler/ paradigm to implement the kernel:
Well suited.

mod2as

Overall implementation result: Partly sucessfull

Consistency of the numerical results:
yes

General comments:
Implementation of data parallel problems in Chapel is in principle very simple. Parallelism is achieved through 1) parallel "forall" loops, which allow all cores in a node to work concurrently; 2) "domain distribution" which implement work and data distribution across nodes. However, as of Chapel version 0.9 runtime /standard modules does not really execute the program in parallel and/or distributed across several machines. In principle, however, Chapel provides a standard module for compressed storage row format distributions and should eventually perform well.

Problems encountered:
Syntax for distributed sparse domains was not clear. The Chapel developers at Cray were very helpful in clarifying those issues.

Maturity of the standard/compiler:
Specification is ok. Compiler & runtime are still prototypes.

Number of lines of source code:
The actual code is less than 15 lines.

Suitability of the compiler/ paradigm to implement the kernel:
Well suited, in particular since there is language support for sparse arrays and a standard module for compressed storage row specifically.

mod2f

(mod2f was simply taken from the SDK examples)

Performance Measurements

mod2am

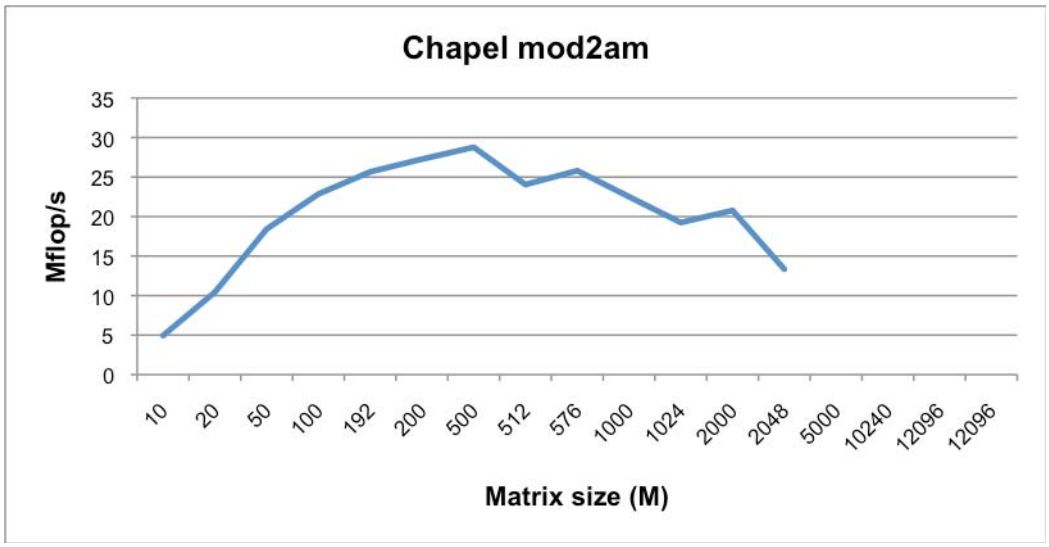

Figure 38: Chapel mod2am performances (test performed on *nehalem*/*baku*)

mod2as

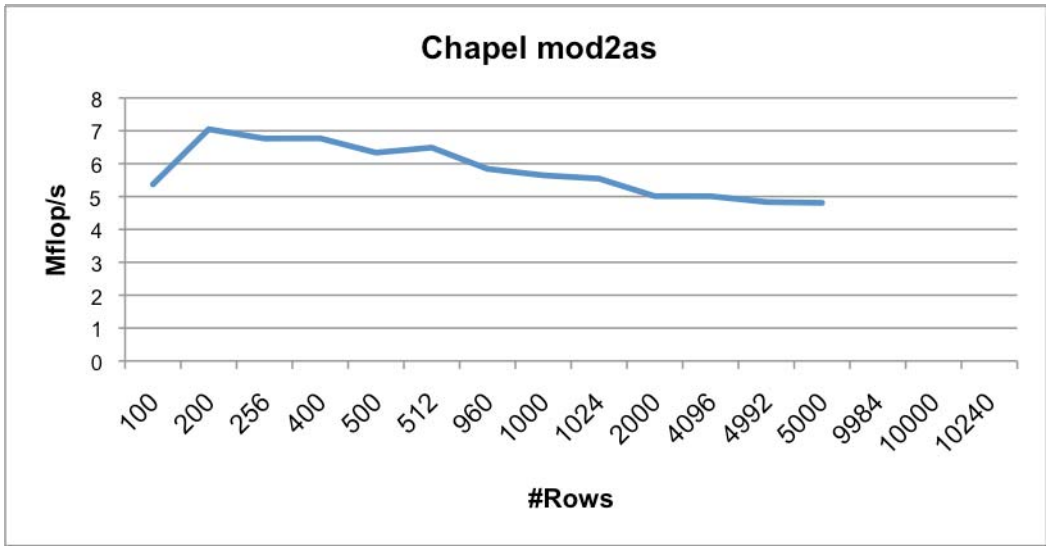

Figure 39: Chapel mod2as performances (tests performed on *nehalem*/*baku*)

14.5 ClearSpeed Cn

The only code for which no library routine is available is mod2as. However, this is so straightforward that there was no call for more than one version: one has to get the data onto the card, distribute rows over the available processors and get the result vector out. The details of aligning the data, transferring data from mono to poly memory, etc. are complicated but known to the PetaPath crew. So, no new insights are coming from that part.

Basic Information

Author: Phil Power (PetaPath), reports handed in from Aad van der Steen (NCF)
Euroben Kernel: mod2am, mod2as, mod2f, mod2h
Hardware: *clearspeed-petapath*
Compiler Version: Cn 3.11

Remark on the hardware: Since the clearspeed-petapath hardware at NCF-SARA was not ready, the test configuration of PetaPath in Bristol was used. This was sufficient to run the kernels.

Developer Diary

For mod2am, mod2f and partly for mod2h, demonstrations codes from PetaPath could be used and needed only adaptation. An estimate about how long the whole code development would have taken is therefore not possible.

mod2am

An adjusted demonstration code was used to generate the results (CSXL library)). So, no problems were encountered.

Achieved Performance: <= 600 Gflop/s in 64-bit precision with 1-8 CSX 700 cards

mod2as

There are no intrinsic problems except the low computational intensity of the problem.

Achieved Performance: 30 Mflops on one card

mod2f

1-D as well as 2-D FFTs were done as a result of modifying existing PetaPath demonstration codes (CSDFT library). However, only on the restricted data lengths that the FFT library allows were run.

Achieved Performance: 6—10 Gflop/s on 1 (2 MTAPs)

mod2h

Instead of only rand64, 4 random number generators from the PetaPath CSRNG library were run: drand48, drnd64, MT2203, and MT19937.

Achieved Performance: 2.72•108 random numbers/s on 1—4 cards, 1—2 MTAPs

Porting Results

mod2am

Overall implementation result: Successful

Consistency of the numerical results:
Yes

General comments:
The CSXL library was used to obtain the results. The results for this kernel were more or less as expected. The architecture is very suitable for this kind of code as all SIMD-type processors are.

Problems encountered:
None.

Maturity of the standard/compiler:
The Cn compiler is around for quite some time. Version 3.11 is very mature and poses no problems.

Number of lines of source code:
1158. Note there are lots of comments lines and of the general purpose timing routine of 180 line only 2 are actually used.

Suitability of the compiler/ paradigm to implement the kernel
See above.

mod2as

Overall implementation result: Successful

Consistency of the numerical results:
Yes

General comments:
The results for mod2as are quite poor as could be expected: All existing SIMD-type accelerators perform badly on reduction operations which a primary constituent of the code. Futhermore, the low computational intensity (ratio of arithmetic vs. memory operations) generally a low performance can be expected. The situation is aggravated by having to transport the data to the accelerator card. It results in a performance that is about 8 times slower than that of a Nehalem core.

Problems encountered:
None

Maturity of the standard/compiler:
Compiler is very mature. No problems in generating the executable from Cn.

Number of lines of source code:
Card codes: 707.
Host codes: 422 (including the host-card communication part).

Suitability of the compiler/ paradigm to implement the kernel:
No problem to implement the code, but presently the hardware cannot support reduction operations well.

mod2f

Overall implementation result:
mod2f was correctly ported without problems.

Consistency of the numerical results:
Yes

General comments:
The results were as expected: a speed of 6 to 10 Gflops for the double precision complex forward FFT on 1 CSX 700 card. This is 5 to 9 times faster than what can be achieved on a single Nehalem processor in C code. This is, however, only for the limited range of transform lengths presently available in the FFT library.

Problems encountered:
None

Maturity of the standard/compiler:
Both compiler and library are very mature and stable.

Number of lines of source code:
Card code: 291; Host code: 324.
However, both parts contain much that is not relevant for the problem posed in mod2f because it is an adaptation of a demonstration code that also performs series of 1-D FFTs and 2-D FFTs.

Suitability of the compiler/ paradigm to implement the kernel:
Quite suitable. However, the range of transform lengths in the library is too limited and should be extended.

mod2h

Overall implementation result: Successful

Consistency of the numerical results:
Yes

General comments:
The performance per e710 (= 1 CSX700) card for uniform random variates is about 2.5 to 3 times faster than a single Nehalem processor. This is due to the low logical operation count vs. the data movement. The PCIe bus turns out to be very well occupied. For Gaussian or Log-Gaussian, or exponentially distributed variates the relative performance can be better as the amount of computation/variate is substantially higher and all can be done on the card.

Problems encountered:
None

Maturity of the standard/compiler:
Very mature and stable.

Number of lines of source code:
Host code: 246; Card code: 408. However, in the host code is the timing code of 192 lines included of which only 2 apply for the main program and in the card code also the code for 3 additional generators is included together with that for rand64.

Suitability of the compiler/ paradigm to implement the kernel:
The kernel can be very well expressed in Cn for this platform. However, the computational intensity for uniform random variates is low. So, the benefit from the parallelism even through the optimised library calls is moderate.

Performance Measurements

No performance data available.

14.6 CUDA

Basic Information

Author: Hans Hacker (LRZ)
Euroben Kernel: mod2am, mod2as, mod2f
Hardware: *uchu*
Compiler Version: CUDA compilation tools, release 2.2, V0.2.1221

Developer Diary

mod2am

Performance achieved: 63 Gflops (DP) on one Tesla.

The developer diary is missing, since the porting activity was started very early within WP8 and the WP6T6 ReportingTemplate was not ready by that time.

mod2as

Performance achieved: 676 Mflops (DP) on one Tesla.

The developer diary is missing, since the porting activity was started very early within WP8 and the WP6T6 Reporting Template was not ready by that time.

mod2f

Develop. Time	Achieved Performance	Dataset used	Comments
45m	~2GFlops (SP) ~1.7GFlops (DP)	2^24	Baseline Nehalem. Rewrite: added option double precision/single precision
2h30m	~18GFlops	2^24	Implemented CUFFT (single precision)
25m	~150MFlops (DP)	2^24	Initial simple CUDA port (double precision)
3h	~190MFlops (DP)	2^24	Rewrite of init-vector from 3 loops to parallel (unrolled) version
25m	-	-	Checking with CUDA-profiler – in order to find bottlenecks
3h45m	~450MFlops (DP)	2^24	Rearrangement of data-structures
4h	~430MFlops (DP)	2^24	replaced double precision operations (*1.0 / *-1.0) with bitflip → rewind :(
5h	~1GFlops	2^24	Maximize loop unrolling in Radix4 Kernels
30m	-	-	Checking with CUDA-profiler – in order to find bottlenecks
1h	~1.3 GFlops	2^24	Added support for memory pinning
3h	~1.6 GFlops	2^24	Changed data structures from __const__ to __shared__ (misunderstanding the CUDA documentation - should not count ! :-))
4h	~2 GFlops	2^24	Rewrite from two input vectors (2x double) to the CUDA datatype double2
30m	-	-	Checking with CUDA-profiler – in order to find bottlenecks
25m	~3.2 GFlops	2^24	Correction of wrong loop-unrolling in one CUDA-kernel
6h	~4.2 GFlops	2^24	Remove shared memory access by substitution (now all values in registers)
4h	~5.8 GFlops	2^24	Using texture cache for data-input reduces access to global memory by ~45%

Porting Results

mod2am

Overall implementation result: Successful

Consistency of the numerical results:
Yes

General comments:
The obvious way to implement this benchmark is to apply the CUBLASDgemm/CUBLASSgemm library calls. As with every accelerator-card one has to move the data to the main-memory of the card. CUDA offers various ways to copy the data. For this application I chose two methods that can be used via a make-switch (documented in the README). Method one is to allocate the host data-structures in the ordinary way with malloc and the same data-structures on the GPU and then copy the data. This works fine however the MMU is involved in every page-request. Method two allocates the host data-structures via a special CUDA call (cudaMallocHost) which marks the allocated pages as non-swappable and allows data-transfer to take place without the interference of the MMU. This method is significantly faster.

Problems encountered:
Since the mod2am-benchmark uses a typical C-'row-major' data arrangement, I used the transpose option for the input matrices. Unfortunately there is no transpose option for the output so I had to write a kernel which transposes the matrix. There is an example code for a matrix transposition in the SDK which I modified for this case. This transposition is done out-of-place by blocking the matrix, transposing the block and storing it back to the main memory of the card.

Maturity of the standard/compiler:
NVIDIA is still adding new features and which needed a few changes to existing code.

Number of lines of source code:
~150 additional lines of code for single/double precision

Suitability of the compiler/ paradigm to implement the kernel:
This kernel is the so called sweetspot for the NVIDIA Accelerator. It single precision it outperforms the MKL/Nehalem version with eight cores. In double precision the Nehelm is only ~ factor 1.2 times faster than one Tesla C1060 card. The double precision MKL version on a 8 core AMD Shanghai is ~ factor 0.08 slower than the CUDA version on Tesla C1060.

mod2as

Overall implementation result: Successful

Consistency of the numerical results:
Yes

General comments:
NVIDIA offers no library for sparse matrix operations. However by searching for a solution on their CUDA Zone, I found the publication “Efficient Sparse Matrix-Vector Multiplication on CUDA” by Nathan Bell and Michael Garland. The paper plus the source-code can be

downloaded there. I took their CSR-kernel and wrote a C-function to fit the needs of mod2as. There was no need to change the data structures. The kernel makes very efficient use of the shared memory within the SM (Streaming Multiprocessor). Because of the warp concept (a warp consists of 32 threads which are all synchronous) they can also do a very efficient reduction. They further put the x-vector into the texture-cache. Because the x-data is reused multiple times this is also very efficient.

Problems encountered:
The main problem with this benchmark is, that the data-transfer as well as the calculation is both O(n). Therefore the PCIe bus limits the overall performance. However if the results of the calculation can stay on the accelerator for multiple iterations the NVIDIA cards are very fast.

Maturity of the standard/compiler:
NVIDIA is still adding new features and which needed a few changes to existing code.

Number of lines of source code:
~350 additional lines of code for single/double precision

Suitability of the compiler/ paradigm to implement the kernel:
This kernel itself would be very well suited for the NVIDIA accelerator. The calculation within the Tesla Card is up to ~ 3x faster than the MKL version on a Nehalem with 8 cores. However the PCIe bus is the main bottleneck and results in a factor of less than 1/5 of the Nehalem.

mod2f

Overall implementation result: Successful

Consistency of the numerical results:
Yes

General comments:
The single precision port was straightforward by the use of CUFFT. The double precision port was more challenging. The latest version of CUFFT (v2.3) now supports double precision and should therfore be as straightforward now as the single precision version. (This could not be tested so far since neither the CEA system nor the LRZ system have installed the latest driver/SDK version).

Problems encountered:
The compiler gives no warning if double precision code is compiled without support for double precision. The values on the GPU are casted into single precision. This results in corrupt data when copied back to the CPU.

Maturity of the standard/compiler:
NVIDIA is still adding new features and this results in a few changes to existing code.

Number of lines of source code:
~75 for SP
~3000 for DP

Suitability of the compiler/ paradigm to implement the kernel:
This kernel is well suited for porting to CUDA. It outperforms the MKL version by ~ factor 3 in single precision and ~ factor 2 for double precision.

Performance Measurements

mod2am

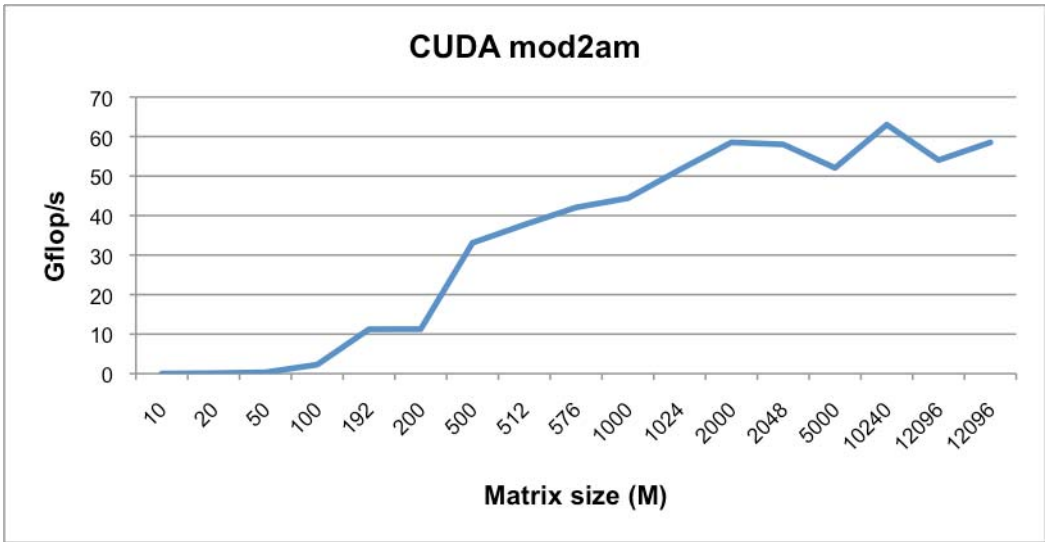

Figure 40: CUDA mod2am performances (tests performed on *uchu*)

mod2as

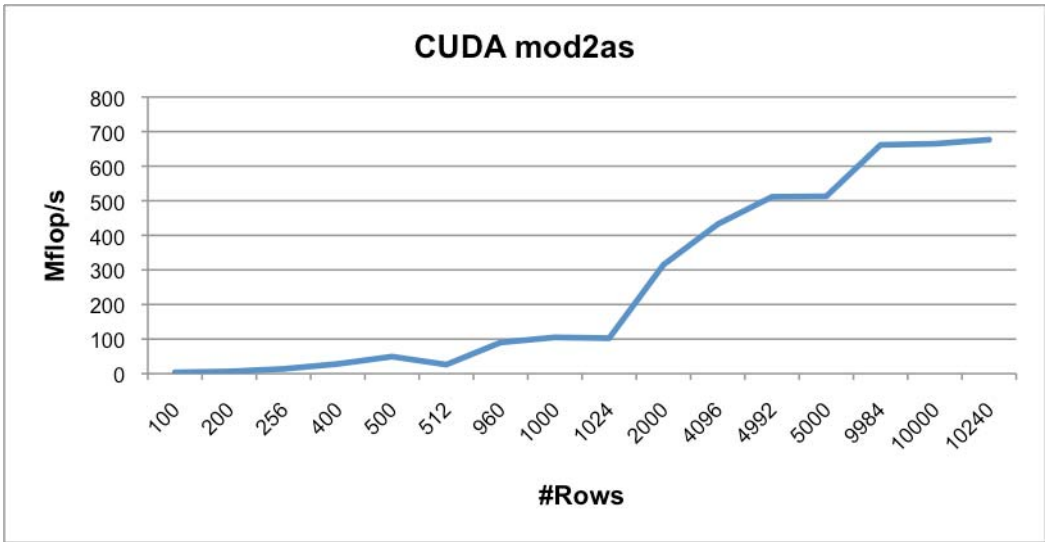

Figure 41: CUDA mod2as performances (tests performed on *uchu*)

mod2f

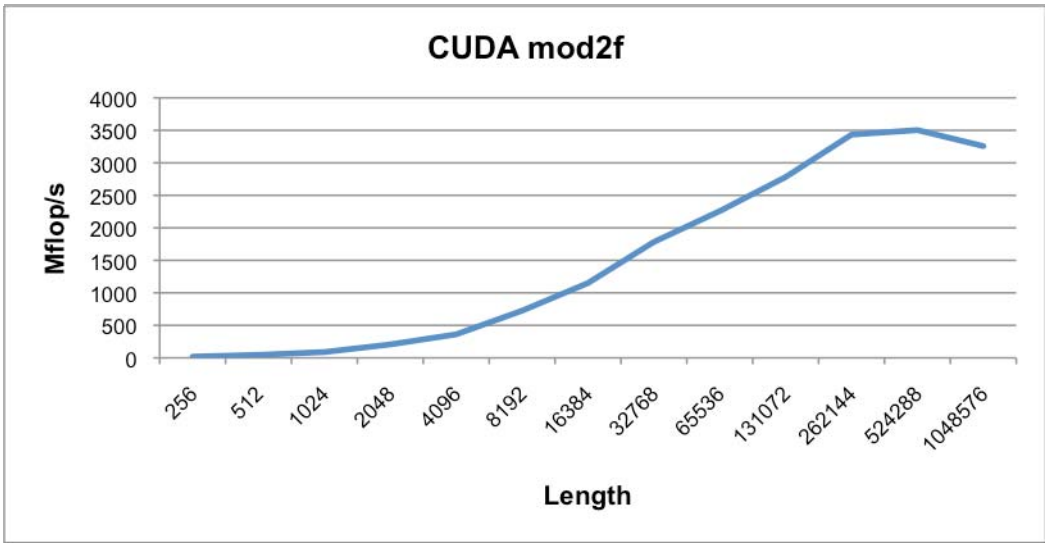

Figure 42: CUDA mod2f performances (tests performed on *uchu*)

14.7 CUDA+MPI

Basic Information

Author: Olli-Pekka Lehto (CSC)
Euroben Kernel: mod2am, mod2as, mod2f
Hardware: *uchu*
Compiler Version: CUDA 2.2 nvcc; gcc 4.1.2; OpenMPI 1.3.2

Developer Diary

mod2am

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
1d	44 GFlop/s <5 GFlop/s 78 GFlop/s 134 GFlop/s 130 GFlop/s	1N 1P 1N 4P 2N 2P 4N 1P 4N 4P	mod2am (10240)	Initial port of mod2am The "1N 4P" run did not complete in reasonable time
30m	41 GFlop/s 67 GFlop/s 128 GFlop/s 135 GFlop/s 219 GFlop/s	1N 1P 1N 4P 2N 2P 4N 1P 4N 4P	mod2am (10240)	Added striping over several GPUs
1h	55 GFlop/s 83 GFlop/s 148 GFlop/s 169 GFlop/s 259 GFlop/s	1N 1P 1N 4P 2N 2P 4N 1P 4N 4P	mod2am (10240)	Added CUBLAS support
30m	56 GFlop/s 53 GFlop/s	1N 1P 1N 4P	mod2am (10240)	Added memory pinning support Did not for internode communication over InfiniBand
3h			mod2am	Attempted to implement CUBLAS-accelerated PDGEMM Unsuccesful due to PBLAS compile difficulties

mod2as

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
3h	364 MFlop/s 266 MFlop/s 535 MFlop/s 288 MFlop/s	1N 1P 1N 4P 4N 2P 4N 4P	mod2as (10240)	Initial port of mod2as
15m	364 MFlop/s 481 MFlop/s 1028 MFlop/s 1062 MFlop/s	1N 1P 1N 4P 4N 2P 4N 4P	mod2as (10240)	Added striping over several GPUs

mod2f

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
1d			mod2f (10240)	Attempted to port based on Fortran+MPI kernel. Unsuccesful due to difficulties in transpose. Gave up as a C+MPI version was to be released.
1d			mod2f	Attempted to port based on C+MPI kernel Ran out of time

Porting Results

Overall implementation result: Partly successful

Consistency of the numerical results:
Yes

General comments:
- Reasonably straightforward to port existing code
- It's easy and effective to assign MPI ranks to multiple GPUs when available
- The striping may also tie ranks unevenly to GPUs on some platforms due to differences in rank placement rules. One must ensure that ranks are assigned in sequentially to nodes.

Problems encountered:
- The performance of a simple port is not sufficient
- Memory pinning when running parallel over InfiniBand with reasonably large dataset caused a hang
- Porting mod2f proved unexpectedly difficult

Maturity of the standard/compiler:
The compiler is mature.

Number of lines of source code:
mod2am
- Normal: 240 lines
- CUBLAS: 140 lines
  - *Striped*: +10 lines
  - *Mempin*: +15 lines

mod2as
- Normal: 50 lines
  - *Striped*: +10 lines
  - *Mempin*: +15 lines

Suitability of the compiler/ paradigm to implement the kernel:
Reasonably suitable: There is likely much further optimization that can be done, especially with minimizing the amount of host-GPU memory transfers

Performance Measurements

mod2am

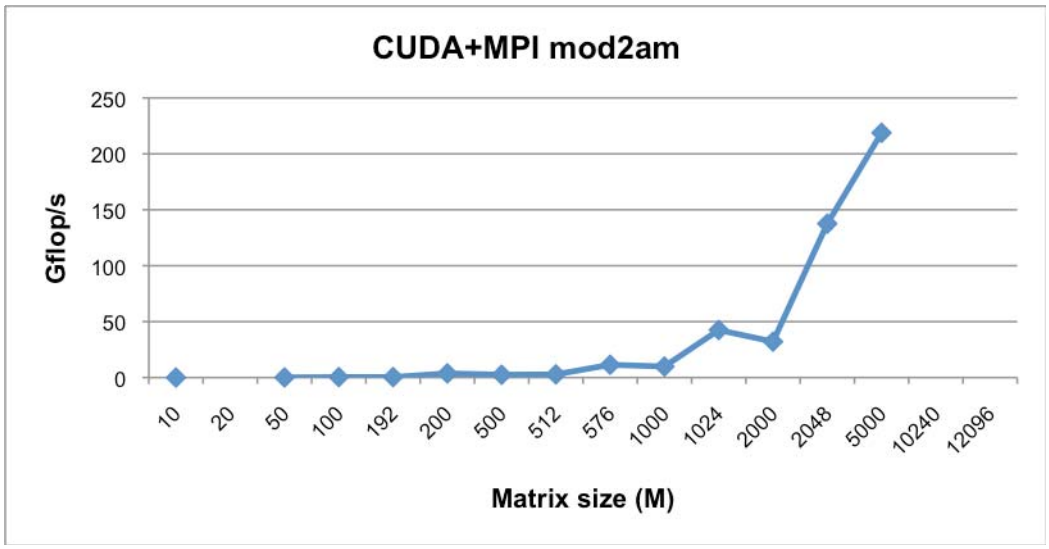

Figure 43: CUDA+MPI mod2am performances (tests performed on *uchu*)

mod2as

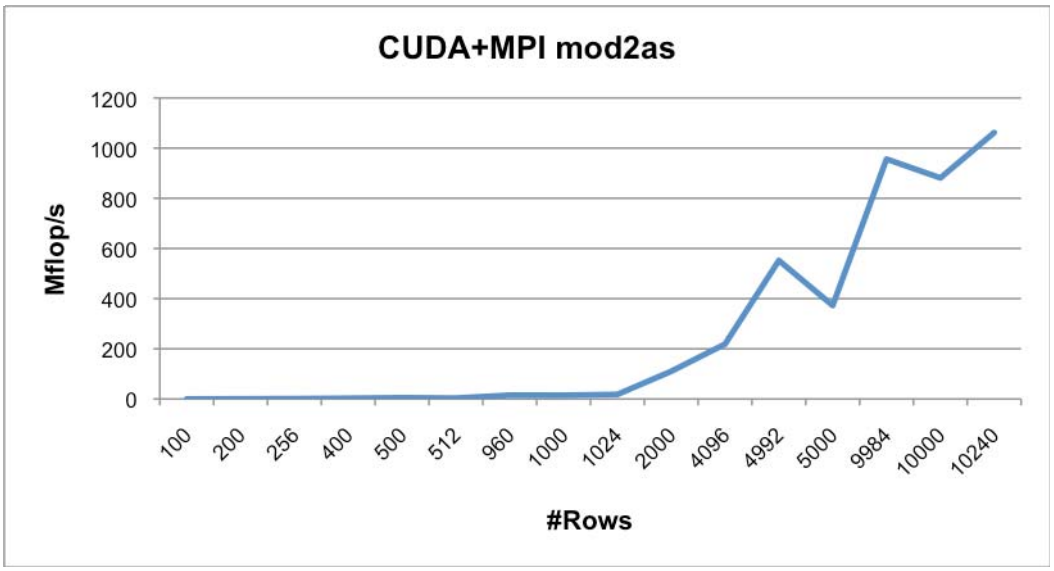

Figure 44: CUDA+MPI mod2as performances (test performed on *uchu*)

14.8 FPGA (VHDL and/or Harwest C)

Basic Information

Author: James Perry (EPCC)
Euroben Kernel: mod2am, mod2as
Hardware: *maxwell*
Compiler Version: HCE 1.4 for Harwest C; Xilinx ISE9.1i for VHDL

Developer Diary

mod2am

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
5d	6.45 Mflops			Initial porting to VHDL. Correct but very poor performance due to memory access pattern.
3d	2,240 Mflops			Switched from external DRAM to internal block RAM. Performed multiple floating point operations in parallel. Faster, but limited to relatively small matrices (256x256 max).
8d	1,846 Mflops			Hybrid VHDL version (storing matrices in DRAM but caching sections in block RAM). Good performance and handles matrices up to 1024 square.
2d	1,625 Mflops			Ported VHDL version to 64-bit double precision
2.5d	14.0 Mflops			Ported kernel to Harwest C

mod2as

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
5d	450			Initial VHDL port – 64-bit double precision.
1d	900			32-bit single precision VHDL version, for fairer performance comparisons with Harwest implementation, which is constrained to single precision.
3d	1.71			Harwest port, returning 32-bit floats.
0.25d	2.76			Harwest port, returning 64-bit integers (which could then be converted to double precision) to host.

Porting Results

mod2am

Overall implementation result: Partly successful

Consistency of the numerical results:
Results are very close to original, occasional small differences due to different order of operations.

General comments:
The kernel was successfully ported to both VHDL and Harwest C. Performance of the final VHDL version is good, significantly faster than running on the host even when the time taken to transfer data over the PCI bus is taken into account. Performance of the Harwest C version

is poor (approximately 40 times slower than the original code). The reason for this is not yet known, but an integer version of the kernel performs 5 times faster, suggesting that the floating-point arithmetic is partly responsible for the slowness.

Problems encountered:
Fitting the VHDL version onto the resources of the FPGA, especially the 64-bit port, was very challenging. Allowing different sizes of matrix (especially non-power-of-2 sizes) would complicate the code, so the final version is a fixed size 1024x1024 matrix multiplier (unlike the original which can handle any size). Handling variable sizes would probably not adversely affect performance.

Maturity of the standard/compiler:
The HCE compiler is still very much under development. It is lacking some features (such as double precision floating point) and a few bugs were encountered during the porting. Xilinx ISE is a mature, proven, and very widely used development tool.

Number of lines of source code:
VHDL: 1252
Harwest C: 75

Suitability of the compiler/ paradigm to implement the kernel:
The kernel is not particularly well suited to implementation using HCE as HCE does not support double precision arithmetic (and its single precision arithmetic seems to be slow). It is also difficult to access large arrays such as the matrices in HCE – this necessitates copying a section at a time to smaller arrays. The floating point and memory bandwidth requirements also mean that the VHDL version does not achieve as big a speed-up over the host as some algorithms can (however, at approximately 2-3x faster, it is still a worthwhile speed-up).

mod2as

Overall implementation result: Partly successful

Consistency of the numerical results:
Results from 64-bit versions are identical to those from original kernel. Results from 32-bit versions are as close as 32-bit precision can get to the original results.

General comments:
Both the Harwest C and VHDL ports worked. However, the Harwest C version is too slow to be of much use, more than 5 times slower than the original C code running on a 2.8GHz Xeon. The VHDL version is very fast but when the time taken to transfer the output back across the PCI bus to the host machine is taken into account, it is also slower than the original code. Therefore the VHDL version would only be an improvement if the random numbers generated were being consumed by another process on the FPGA itself.

Problems encountered:
Harwest C does not support double precision, so the HCE port was limited to 32-bit floating point. It also does not support 64-bit integer arithmetic, making the core of the algorithm (which uses 64-bit integers extensively) rather cumbersome.

Maturity of the standard/compiler:
The HCE compiler is still very much under development. It is lacking some features (such as double precision floating point) and a few bugs were encountered during the porting. Xilinx ISE is a mature, proven, and very widely used development tool.

Number of lines of source code:
VHDL: 633
Harwest C: 88

Suitability of the compiler/ paradigm to implement the kernel:
This kernel is very well suited to VHDL implementation as it consists primarily of bit shifts and exclusive-or which are very efficient and easy to implement in hardware. Additionally, the final floating point scaling can be replaced by a simple bit shift in hardware. The kernel was not so well suited to HCE; HCE's lack of 64-bit integer support meant that the algorithm had to be rewritten to use multiple 32-bit operations instead, and lack of double precision limits the precision of the results.

Performance Measurements

Missing.

14.9 MPI+OpenMP

Basic Information

Author: Filippo Spiga (CINECA)
Euroben Kernel: mod2am, mod2as, mod2f
Hardware: *nehalem*
Compiler Version: Intel(R) 64, Version 10; OPENMPI 1.3.2

Developer Diary

mod2am

(D1= 1000x1000; D2=5000x5000; D3=10000x10000)

Version	Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
v0	10m	D1 = 9.7956e+02 D2 = 6.3040e+02 D3 = 6.3318e+02	1	D1, D2, D3	Starting C kernel (little changes in auxiliary files), some little change
v0.1	5m	D1 = 4.6969e+03 D2 = 3.2568e+03 D3 = 3.2691e+03	1	D1, D2, D3	Starting kernel with the main loop explicit unrolled (4 times)
v0.2	5m	D1 = 3.5492e+03 D2 = 1.8759e+03 D3 = 1.8998e+03	4	D1, D2, D3	Added OpenMP pragma on OUTER loop (i-loop)
v0.3	5m	-	4	-	Added OpenMP pragma on MIDDLE loop, exchanged k-loop with j-loop. This allows SSE vectorization but a evident degradation of performance.
v0.4	5m	D1 = 8.3959e+02 D2 = 1.1216e+03 D3 = 4.3651e+02	4	D1, D2, D3	Added OpenMP pragma on INNER loop (k-loop)
v0.5	10m	-	-	-	Added OpenMP pragmas on both i-loop and k-loop (j-loop and j-loop areexchanged respect to the starting kernel). To use NESTED parallelism set OMP_NESTED=true. Not suppoted by PGI 8.0-6 but fully supported by INTEL 10.1
v0.6	15m	D1 = 1.6539e+04 D2 = 2.9540e+04 D3 = 1.2534e+03	4	D1, D2, D3	OpenMP parallelization is performed by CBLAS2 library. CBLAS is complied with of multi-threading support ACML (libacml_mp.a, v4.2.0).
v1.0	1h15m ~ 1h30m	D1 = 5.4409e+03 D2 = 3.4059e+03 D3 = 3.3349e+04	4 MPI + 4 OMP (16)	D1, D2, D3	Added trivial MPI parallelization: 1D block layout and MPI_Send/MPI_recv. The local matrix-matrix multiplication is parallelized with OpenMP in 4 different ways. Starting from this version, test were performed using the best strategy derived from v0.x steps (__OPENMP_OUTER).
v1.1	20m	D1 = 1.7366e+04 D2 = 9.6143e+03 D3 = 1.3441e+03	4 MPI + 4 OMP (16)	D1, D2, D3	Changed MPI strategy, 1D block layout but using MPI_Bcast.
v1.2	20m	D1 = 1.5272e+04 D2 = 1.1201e+04	4 MPI + 4 OMP	D1, D2, D3	Changed MPI strategy: 1D block layout but using MPI_sendrecv.

[2] http://www.netlib.org/blas

		D3 = 4.3621e+03	(16)		
v1.3	40m	D1 = 1.3558e+04 D2 = 1.1086e+04 D3 = 4.3565e+03	4 MPI + 4 OMP (16)	D1, D2, D3	Changed MPI strategy: 1D block layout but using non-blocking MPI_Isend/MPI_Irecv.
v2.0	2h	D1 = 3.7313e+03 D2 = 7.1081e+03 D3 = 4.1849e+03	4 MPI + 4 OMP (16)	D1, D2, D3	This implementation is based on Cannon algorithm: 2D block (better data distribution and better scaling) using MPI_sendrecv and MPI_Cart* features. Multi-threading parallelization in local block matrix-matrix multiplication is provided by OpenMP explicitly.
v2.0.1	10m	D1 = 4.6977e+03 D2 = 8.4428e+04 D3 = 6.6948e+04	4 MPI + 4 OMP (16)	D1, D2, D3	Same MPI strategy on v2.0 but multi-threading parallelization in local block matrix-matrix multiplication is provided by CBLAS.
v2.1	45m ~ 1h	D1 = 2.3222e+04 D2 = 7.2023e+03 D3 = 4.2226e+03	4 MPI + 4 OMP (16)	D1, D2, D3	Changed MPI strategy: 2D block using non-blocking MPI_Isend/MPI_Irecv to exchange local sub-matrices. Multi-threading is explicit with OpenMP.
v2.1.1	10m	D1 = 7.2162e+03 D2 = 9.7354e+04 D3 = 7.2300e+04	4 MPI + 4 OMP (16)	D1, D2, D3	Changed MPI strategy: 2D block using non-blocking MPI_Isend/MPI_Irecv to exchange local sub-matrices. Multi-threading is provided by CBLAS.
v2.1.2	10m	D1 = 6.5840e+03 D2 = 9.5389e+04 D3 = 6.9531e+04	4 MPI + 4 OMP (16)	D1, D2, D3	Changed MPI strategy: 2D block using non-blocking MPI_Isend/MPI_Irecv but Irecv are preposted to allocate in advance MPI buffers. Multi-threading is provided by CBLAS.

mod2as

(D1= 30000 (3,33%); D2= 30000 (13,33%); D3= 40000 (4,24%))

Version	Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
v0	5m	D1 = 5.74603e+02 D2 = 5.94735e+02 D3 = 5.86538e+02	1	D1-D5	Original C kernel (little changes in auxiliary files).
v0.1	10m	D1 = 7.15820e+02 D2 = 1.11661e+03 D3 = 1.09239e+03	4	D1-D5	Basic parallelization using OpenMP.
v0.2	20m	D1 = 9.51777e+02 D2 = 1.17221e+03 D3 = 1.14962e+03	4	D1-D5	Rewriting of the main loop inside "spvx.c" in order to gain performance. This optimizations works with PGCC compiler, not with ICC. ICC is not able to perform vectorization on the internal loop due to "dereference too complex" error.
v0.3.0	10m	D1 = 1.20627e+03 D2 = 1.42029e+03 D3 = 1.27266e+03	4	D1-D5	Full 0-index CSR, only OpenMP parallelization. Refer to MKL library user guide to understand the differences between this CSR mode and the 1-index.
v0.3.1	10m	D1 = 1.07085e+03 D2 = 1.42131e+03 D3 = 1.40787e+03	4	D1-D5	Full 1-index CSR, only OpenMP parallelization. Refer to MKL library user guide to understand the differences between this CSR mode and the 0-index.

v0.4	30m	-	1	-	Matrix-vector multiplication is provided by Sparse BLAS library (NIST interface3). Unfortunately this library is not multi-threaded!!!
v0.5	1h45m ~ 2h	D1 = 7.90826e+02 D2 = 1.40758e+03 D3 = 1.42811e+03	4	D1-D5	OpenMP parallelization is provided by Sparse BLAS provided by INTEL MKL library 10.2 using 1-index CSR format. MKL has Sparse BLAS 1-2-3 routines and all the routines support multi-threading.
v1.0	45m ~ 1h	D1 = 1.27287e+03 D2 = 1.38739e+03 D3 = 1.37499e+03	4 MPI + 4 OMP (16)	D1-D5	Trivial block-striped partitioning among all processors: sparse matrix is distributed but the dense vector is replicated on all processors.
v1.1	15m	D1 = 9.61996e+02 D2 = 1.41377e+03 D3 = 1.40486e+03	4 MPI + 4 OMP (16)	D1-D5	Same MPI strategy of v1.1 but this version uses INTEL MKL library 10.2 for local computation.

mod2f

Version	Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
v0	10m	-	1	-	Original C kernel (little changes in auxiliary files).
v0.1	40m	-	4	-	Multi-thread FFT provided by FFTW library. Around 20 minutes were spent to compile the FFTW library.
v0.2	40m		4	-	Multi-thread FFT provided by FFTW3 library. Around 20 minutes were spent to compile the FFTW3 library.
v0.3	30m	-	4	-	Multi-thread FFT provided by MKL library. It seems that MKL 1D FFT routine is not multi-threaded…
v1.0	-	-	1	-	the same as "base/C-MPI" but it's a 2D transformation… useless
v1.1	30m	-	1	-	1D distributed FFT using FFTW. No multi-threaded version provided by the library
v1.2	2d	-	1	-	1D distributed FFT using MKL Cluster FFT. I waited to days to fix some problems thanks to the help of INTEL forum support. I have some strange results from timing. I need more time to investigate about it.

Porting Results

mod2am

Overall implementation result: Successful

Consistency of the numerical results:
Expected consistency through different versions starting from base C kernel.

General comments:
Distributed parallelization with MPI is totally explicit. Some parallel libraries, like PBLAS, are specifically oriented for FORTRAN language. The porting activities from BCX to INTI had required only some changes into the makefiles. I have made some changes on all

[3] http://math.nist.gov/spblas

makefiles in order to be as "modular" as possible and easy to modify and update. On both clusters MKL is present. The performances on these clusters are better than BCX. After the WP6.6 deadline, could be interesting to benchmark this architecture to understand how multi-threading with hyper-threading is good. It's interesting to explore the affinity characteristics and to evaluate TBL and cache miss impacts.

Problems encountered:
No major problems were encountered. Small and trivial bugs aroused during the development activity were fixed in less than 2 hours.

Maturity of the standard/compiler:
The INTEL compiler is well tested and known.

Number of lines of source code:
mxm.c: 154 lines. In total 339 lines for the whole directory.

Suitability of the compiler/ paradigm to implement the kernel:
Good.

mod2as

Overall implementation result: Successful

Consistency of the numerical results:
Expected consistency through different versions starting from base C kernel.

General comments:
MKL has its own internal sparse BLAS implementation and it provides both C and FORTRAN interfaces.

Problems encountered:
Using sparse BLAS routines inside MKL libraries require to read on MKL user guide about format convention. It seems trivial but it's not so simple for a developer who never used MKL library

Maturity of the standard/compiler:
Intel compiler is the best solution if you want to use MKL

Number of lines of source code:
getmatvec.c: 41 lines. In total 238 lines for the whole directory.

Suitability of the compiler/ paradigm to implement the kernel:
Good

mod2f

Overall implementation result: Partially

Deeply check of the last version that use MKL Cluster FFT routine is need but we are out-of-time. At this point, I am not able to produce valid performance results of the latest version that use multi-threading MKL library. To complete all the work and the porting activity I need additional time.

Consistency of the numerical results:
I have some problems with the check routine. I would like to change it but at the moment I write a dummy routine.

General comments:
It is really useful to have a high level library that contains both multi-thread and distributed version. However putting functionalities inside a close library could produce problems of compatibility. The usage of MKL Cluster FFT is not easy because MKL has its own interfaces and methods. The implementation of an algorithm based on FFT using MKL Cluster FFT could require reading a lot of documentation.

Problems encountered
During the porting activities I encountered a problem related to DftiCreateDescriptorDM(). Open MPI considers MPI_COMM_WORLD to be a pointer it turns out to be 64-bit long. To avoid this problem INTEL support suggested to use MPI_Comm_c2f. The second problem was related to linking. Due to a bug or an unconventional way of MPICC wrapper to compose the start/end-group link statements, link step failed. I made a trick to allow the compilation using the Makefile.

Maturity of the standard/compiler:
MKL libraries support distributed computation using MPI. Officially MKL requires Open MPI 1.1 or 1.2. However using Open MPI 1.3 I didn't encounter any problems. Seems that MKL 1D FFT functions are not multi-threading. I think this is a strong limitation. INTEL support says that it depends on the version of MKL you are using and the size of the problem if FFT runs in multi-threaded mode or not (see http://software.intel.com/en-us/articles/mkl-threaded-1d-ffts/)

Number of lines of source code:
In total 279 lines for the whole directory.

Suitability of the compiler/ paradigm to implement the kernel:
Good

Performance Measurements

mod2am

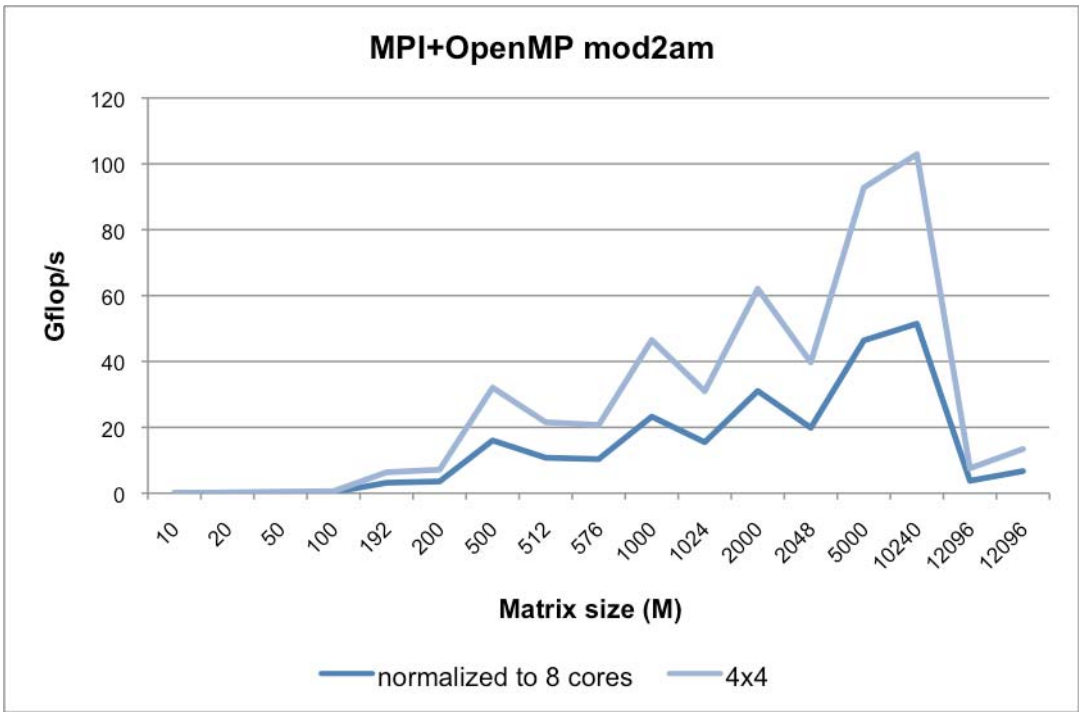

Figure 45: MPI+OpenMP mod2am performances (tests performed on *inti*)

mod2as

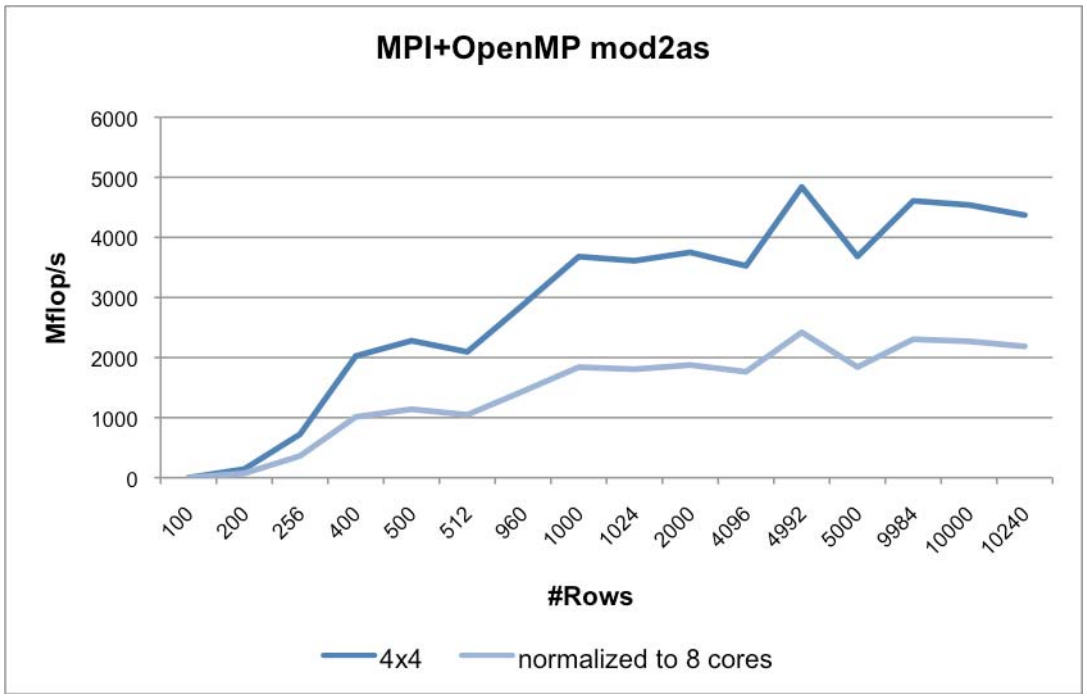

Figure 46: MPI+OpenMP mod2as performances (tests performed on *inti*)

14.10 OpenCL

The available implementations of OpenCL are still in beta-state with the target being the correctness of the implementation and not yet on optimization. Thus, the assessment should focus on productivity and portability to OpenCL instead of performance.

OpenCL development was done with NVIDIA's OpenCL 1.0 conformance candidate release on the Flux cluster. There were several issues with this: Initially it was discovered that the release does not support DP arrays, but interprets them as SP. The benchmarks were modified to use SP floating point. The lack of a software emulation mode, debugging/profiling tools and cryptic error messages made development challenging. The conformance candidate release also requires the older CUDA version 2.2 and a specific driver version. Thus, a dedicated node was assigned to OpenCL development while others were upgraded to CUDA 2.3. Some other minor bugs were encountered which were related to code generation. The OpenCL code was also tested on the GENCI-CEA cluster, but did not work immediately due to the earlier OpenCL beta version that lacked many of the utility library routines used. ATI's OpenCL-implementation (Stream 2.0 beta) was also tested. However, it did not produce the correct results during the initial test.

Basic Information

Author: Olli-Pekka Lehto (CSC)
Euroben Kernel: mod2am, mod2as (buggy), mod2f (running out of time)
Hardware: *uchu* for mod2am and mod2as; *uchu* for mod2am in single precision.
Compiler Version: CUDA 2.2 nvcc, OpenCL 1.0 conformance release

Developer Diary

mod2am

Development time: 1 day to do the initial port of mod2am from CUDA to OpenCL and another hour to port mod2am for use with ATI CPU OpenCL beta, which was unsuccessful due to an unknown bug inside the kernel code

Achieved Performance: 1400 Mflops.

mod2as

Development time: 1 day spent attempting to port mod2as. This was unsuccessful due to an unknown bug inside the kernel code.

mod2f

Port was not started (running out of time).

Porting Results

Overall implementation result: Partly successful

Consistency of the numerical results:
Yes

General comments:
- Reasonably straightforward to port code from CUDA (largely mechanical changes)
- Code samples included with the compiler were useful

Problems encountered:

- Current pre-release version of OpenCL requires a specific NVIDIA driver version which in turn only supports CUDA 2.2
- Support libraries for NVIDIA and ATI versions of OpenCL were different. To create truly portable OpenCL code, one must avoid using them or use ifdefs.
- Double precision was not supported
- Lack of a software emulation mode and a debugger makes debugging very tedious
- Some syntax errors occur occasionally in low-level code generation
- Error handling seems incomplete and error messages can be difficult to decipher
- Occasionally runs lock the device or cause the complete system to hang requiring a reboot
- Lack of proper documentation
- ATI beta version failed to run the OpenCL kernel and no immediate reason was found.

Maturity of the standard/compiler:
The compilers are still very immature.

Number of lines of source code:
250 lines. ATI port added ~40 lines

Suitability of the compiler/ paradigm to implement the kernel:
While the paradigm itself should be quite suitable to implement the kernel, the current implementations are still far too immature for a production environment.

Performance Measurements

mod2am

Important note: Running out of memory for data sets bigger than 5000.

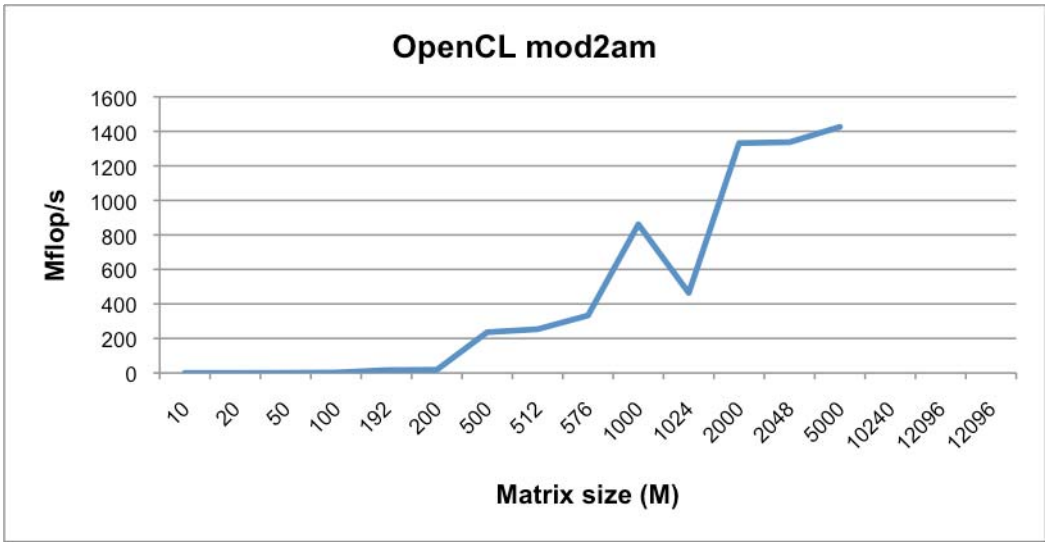

Figure 47: OpenCL mod2am performances (tests performed on *uchu*)

14.11 RapidMind

The performance measurements for this deliverable were made with RapidMind v4.0. This is the first version that supports the 'cuda' backend and therefore the first version that is able to run on NVIDIA Tesla. The target for this v4.0 was CUDA enablement, not CUDA performance. This will be the target for v4.1, which hopefully will be available to include results in D8.3.2.

The double-precision support for GPU and Cell is a new feature. Although the double precision performance was not convincing at all, it has been used to receive comparable results to other languages, especially CAPS HMPP. The single-precision variant of the kernels is faster but still cannot reach the performance of the CUDA libraries.

Basic Information

Author: Volker Weinberg (LRZ)
Euroben Kernel: mod2am, mod2as, mod2f (running out of time)
Hardware: *uchu*
Compiler Version: RapidMind 4.0

Developer Diary

mod2am

Note: This version incorporates MKL, GPU and CELL implementations together.

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
1d		GPU (240 cores)		Tests with RapidMind's SGEMM code available on the RapidMind developer site, https://developer.rapidmind.com/sample-code/matrix-multiplication-samples/rm-sgemm-gpu-5938.zip,
1d		1 CPU		Comparison with host-only version
2d	• incl. data transfer: max. 127 Gflops, • . data transfer: max. 174 Gflops (matrix size=8192)	GPU		Various test runs
1d	MKL with 16 cores: max. 146 Gflops	1-32 CPUs		Implementation + comparison with MKL Version
0.5d		1-32 CPUs		Double precision MKL on host + test runs
0.5d		GPU		Double precision RapidMind on GPU --> no double prec. supported
0.5d		GPU		Test matrix sizes > 8192 --> current impl. limited to sizes < =8192 on GPUs
0.5d				Rewrite EUROBEN source in C++
2d		GPU+CPU		Rewrite RapidMind code in mod2am format + various tests
5d	no perf. increase	GPU		Various optimization attempts (blocking, normalized access)
1d				Integration of simplified sgemm routine in main file
1.5d				Adaption of Makefile, #ifdefs
2d	max. 100 MFlops (matrix size=256)	16 SPU		Various test runs on juicenext@JSC (CELL based)

mod2as

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
3d		GPU (240 cores)		Rewrite the RapidMind Program of mod2am.cpp to handle sparse matrices in CSR (compressed sparse row) format, integration in mod2as code
3d				Rewrite getmatvec.cpp to deliever correct CSR matrices, finally used Hans Hacker's version of getmatvec.cpp
2d	<=50 MFlops for 10000x10000 matrix	GPU		Various Test runs on GPU with RapidMind's glsl backend
1d	MKL: max. 1.1 Gflops for 32 threads (10000x10000 matrix)	1-32CPU		Implementation + comparison with MKL version
1d	max. 1.4 Gflops (10000x10000 matrix), outperforms MKL for fills > 30%	32 CPUs		Various Test runs on CPU with RapidMind's x86 backend
2d	< 1 Mflops	16 SPU		Various test runs on juicenext@JSC (CELL based)

mod2f

Porting Results

mod2am

Overall implementation result: Successfull
Considering n x m matrices with n,m <=8192 and n,m mod 4=0.

Consistency of the numerical results:
consistent

General comments:
Since using 1-tuple RapidMind values to store the matrices does not make optimum use of the GPU registers, the matrices are stored using RapidMind's 4-tuple floating point values Value4f. The current implementation thus limits matrix sizes to multiples of 4.

Problems encountered:
- Double precision not supported for RapidMind's glsl/cell backend,
- the glsl backend restricts 2 dimensional matrices to sizes <= 8192 x 8192,
- running the same (GPU optimized) code on the CELL-based system juicenext@JSC (2 x PowerXCell 8i processors @ 3.2 GHz) gives very poor performance and only works for matrix sizes <= 256 x 256,
- RapidMind's SUSE port (specially built by RapidMind for the LRZ) gives a segmentation fault after the entire glsl program is run via vglrun.

Maturity of the standard/compiler:
No bugs in the x86/glsl RapidMind backend were found.

Number of lines of source code:

Initialization: 8 lines.
RapidMind Program: 17 lines.
Program call: 5 lines.

Suitability of the compiler/ paradigm to implement the kernel:
Best suited if matrices can be completely streamed from/to the host memory to/from the accelerator memory before/after computation.

mod2as

Overall implementation result:
Successfull for x86 backend.
Partly successfull for glsl backend .

Consistency of the numerical results:
Partly consistent.

General comments:

Problems encountered:
- Double precision not supported for RapidMind's glsl/cell backend,
- While the code of the x86 backend gives correct results for all fills of a 10000x10000 input matrix, the glsl backend delievers wrong results for fills > 50% without any error or warning message. (The code has been sent to RapidMind on Aug. 7th and is currently still under inspection.)
- Running the same same code on the CELL-based system juicenext@JSC (2 x PowerXCell 8i processors @ 3.2 GHz) gives very poor performance < 1 Mflops
- RapidMind's SUSE port (rpm specially built by RapidMind for the LRZ) gives a segmentation fault after the entire glsl program is run via vglrun .

Maturity of the standard/compiler:
No bugs in the x86 backend found, glsl backend partly gives wrong results (see above)

Number of lines of source code:
Initialization: 10 lines.
RapidMind Program: 11 lines.
Program call: 6 lines.

Suitability of the compiler/ paradigm to implement the kernel:
Best suited if matrices can be completely streamed from/to the host memory to/from the accelerator memory before/after computation.

mod2f

(still work in progress. Final reports should be available for D8.3.2)

Performance Measurements

mod2am

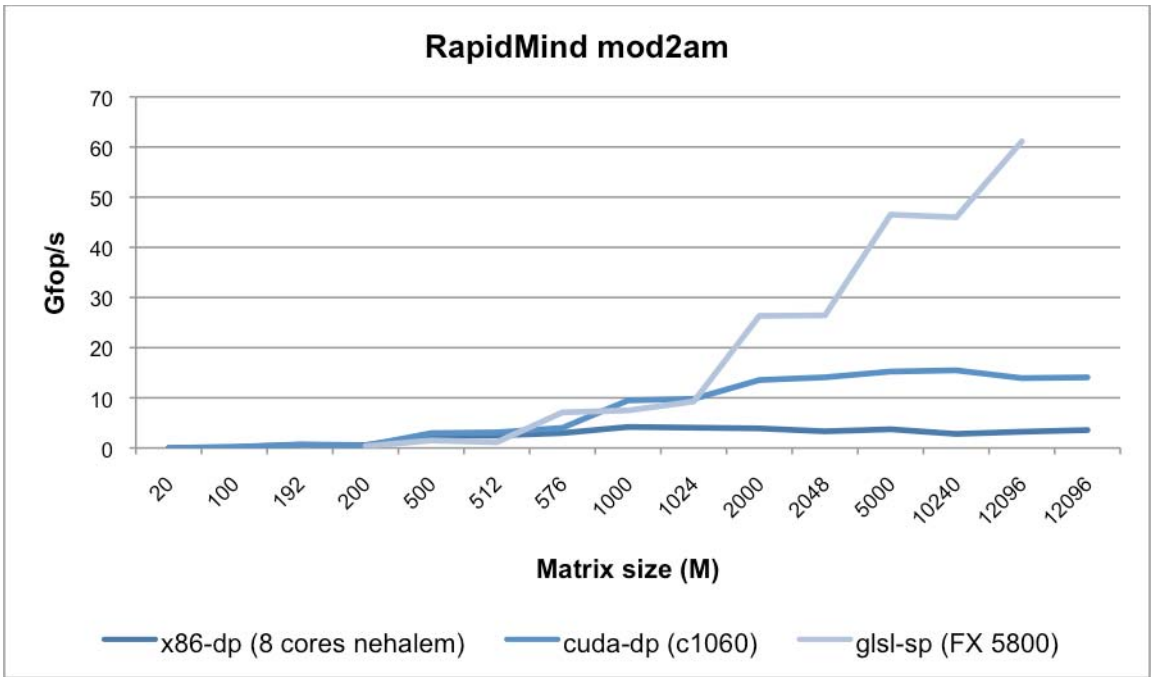

Figure 48: RapidMind mod2am performances (tests performed on *uchu*)

mod2as

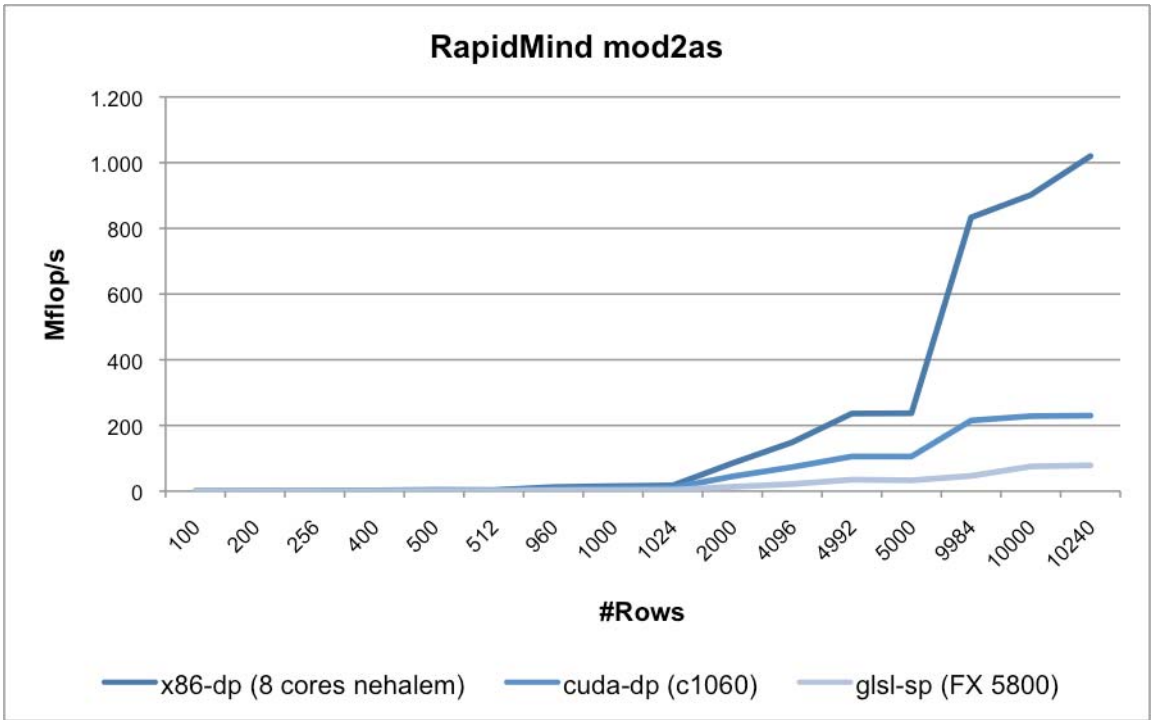

Figure 49: RapidMind mod2as performances (test performed on *uchu*)

14.12 Unified Parallel C (UPC)

One of the key things seemed to be the used of shared [blocksize] for static threads compilations (which I did NOT use) and use of upc_alloc() to imimate shared [blocksize] effect in dynamic threads mode (the mode I used). So the default round-robin allocation (consecutive elements to a different PE) was usually doomed. Also, if one loops over shared-arrays, this may lead implicitly into very small upc_memget()'s being issues. Instead, big blocks, size of shared [blocksize] ought to be communicated in an ideal world. Anyroad, I hope my experiences will spark some discussion inside PRACE. I do NOT think PGAS is dead! It should actually be very alive!

From D8.3.1: "The number of threads can be specified at both compile and runtime, although some restrictions apply in array declaration when the runtime option is chosen." -> This is in fact a major drawback when compared with MPI coding

From D8.3.1: The main objective of CSC was to find the means how to obtain high performance from these two PGAS languages (UPC and CAF) on the Cray XT platform. In UPC, this meant small restructuring of globally shared array allocations. In CAF, this was less of an issue, since the languages syntax offers nearly perfrect means of expressing the data locality.

Basic Information

Author: Sami Saarinen (CSC), Montse Farreras (BSC)
Euroben Kernel: mod2am, mod2as, mod2f
Hardware: *louhi*, *itanium, huygens*
Compiler Version: cce 7.1.2 (initially 7.0.4 and then 7.1.1) on *louhi*
Berkeley UPC 2.8.0 on *itanium*
IBM XL UPC DEV Version (00.00.8000.9999) on *Huygens*

Developer Diary

mod2am

Development Time: 1 week on Cray XT5 to port the code, 1h to recompile and fix the version on the SGI Altix. 1 additional hour to porting on Huygens starting from the CRAY version.

Achieved Performance:

- Cray XT5: 470 Gflops on 256 cores
- SGI Altix: 1600 Gflops on 1000 cores
- IBM Power5: 12,481 Gflops on 64 cores

Notes:

- Pure C-code was translated
- BLAS3-library DGEMM call plugged in
- Many crashes and unexplained failures using the Cray compiler
- A couple of problems in the xlupc compiler where identified. Only workarounds to these problems were introduced in the code.

mod2as

Development Time: 1 week on Cray XT5 to port the code, 1h to recompile and fix the version on the SGI Altix. 1 additional hour to porting on Huygens starting from the CRAY version.

Achieved Performance:

- Cray XT5: 36 Gflops on 256 cores
- SGI Altix: 20 Gflops on 256 cores
- IBM Power5: 6,178 Gflops on 64 cores

Notes:

- Pure C-code translated
- Results with 4 different fills: 1, 3.5, 5 & 10 %
- This was the first UPC I was working ever. After initial hick-ups and poor performance I discovered how to tackle UPC performance.
- A couple of problems in the xlupc compiler where identified. Only workarounds to these problems were introduced in the code.

mod2f

Development Time: 3 weeks on Cray XT5 to port the code, 1h to recompile and fix the version on the SGI Altix.

Achieved Performance:

- Cray XT5: 30 Gflops on 512 cores
- SGI Altix: 50 Gflops on 512 cores
- IBM Power5: -

Notes:

- MPI/Fortran version was translated since MPI/C was not yet available
- Hardest part to get 1D FFT into 2D FFT and to get the transpose working
- Later it was discovered that transpose was in fact working already almost immediately, but because of GASNET + Portals + compiler problems, the results showed up incorrect

Porting Results on System Iouhi (Cray XT)

Overall implementation result: Successful

Consistency of the numerical results:
Correct, but only if GASNET_DISABLE_MUNMAP=1

General comments:
Cray is getting there. Considerable improvements when going from 7.0.4 to 7.1.2 (via 7.1.1)

Problems encountered:
- Allowed private data in shared data structure
- GASNet & Portals still let down
- upc_free() is NOT collective which is strictly according to the standard, but leads into silly problems when deallocating collectively allocated arrays (i.e. allocated via

upc_all_alloc()) as only one UPC-thread can issue upc_free() – the next ones would generate heap corruption messages

Maturity of the standard/compiler:
"What can I say after all these problems?"

Number of lines of source code
mod2am: 550
mod2as: 390
mod2f 1230

Suitability of the compiler/ paradigm to implement the kernel:
Very good

Porting Results on System itanium (SGI Altix)

Overall implementation result: Successful

Consistency of the numerical results:
Correct

General comments:
Very good UPC environment at LRZ.

Problems encountered:
Some small syntax errors in the shared data structures that were okay for some reason on Cray XT.

Maturity of the standard/compiler:
Superb

Number of lines of source code:
mod2am: 550
mod2as: 390
mod2f: 1230

Suitability of the compiler/ paradigm to implement the kernel:
Very good

Porting Results on System Huygens (Power5)

Overall implementation result: Partially Successful

Consistency of the numerical results:
Correct

General comments:
The mod2am algorithm shows poor performance and scalability due to several reasons:

1) Pointer arithmetic is very costly in general and it is one of the limitations of the xlupc compiler. Xlupc compiler developers are aware of it, pointer arithmetic design is being reworked and better performance is expected in the near future.

2) The algorithm could be further improved, here are some comments:
   - Regarding the memory allocation: over-use of upc_alloc damages performance (in the xlupc compiler) the same purpose could be achieved by calling upc_alloc once per thread and redistributing the data with pointer arithmetic.
   - Regarding array distribution, a different layout could be used for a better locality (P x Q cubes where P x Q = THREADS). This change however will be in detriment of programmability.
   - Another bottleneck is in the kernel core. The upc_memgets are translated either to memory copies if shared memory or communication if distributed memory. The i,j,k distribution of the loops leads to 2 memory copies per inner loop iteration $O(n^3)$, which is expensive. I would suggest using pointer cast in shared memory architecture. Or broadcast the required block in a distributed memory architecture, this will imply some extra-memory (programmability will be damaged).
   - A different algorithm could be used, more suitable for distributed memory machines, like Cannon's algorithm. (http://ilpubs.stanford.edu:8090/59/)
   - Finally, xlupc compiler supports a language extension that is a syntax sugar for defining multi-dimensional blocks. The extension is trying to be pushed into the standard. At this point the extension is supported only for static memory allocation.

Problems encountered:
Some bugs related with pointer arithmetic in the xlupc compiler. They were fixed, but there are still others that prevent me to run mod2f.

Maturity of the standard/compiler:
Superb

Number of lines of source code:
mod2am: 550
mod2as: 390

Suitability of the compiler/ paradigm to implement the kernel:
The language has a smooth learning curve, but it does not provide good performance at once. With the appropriate effort a better performance can be achieved.

Performance Measurements

mod2am

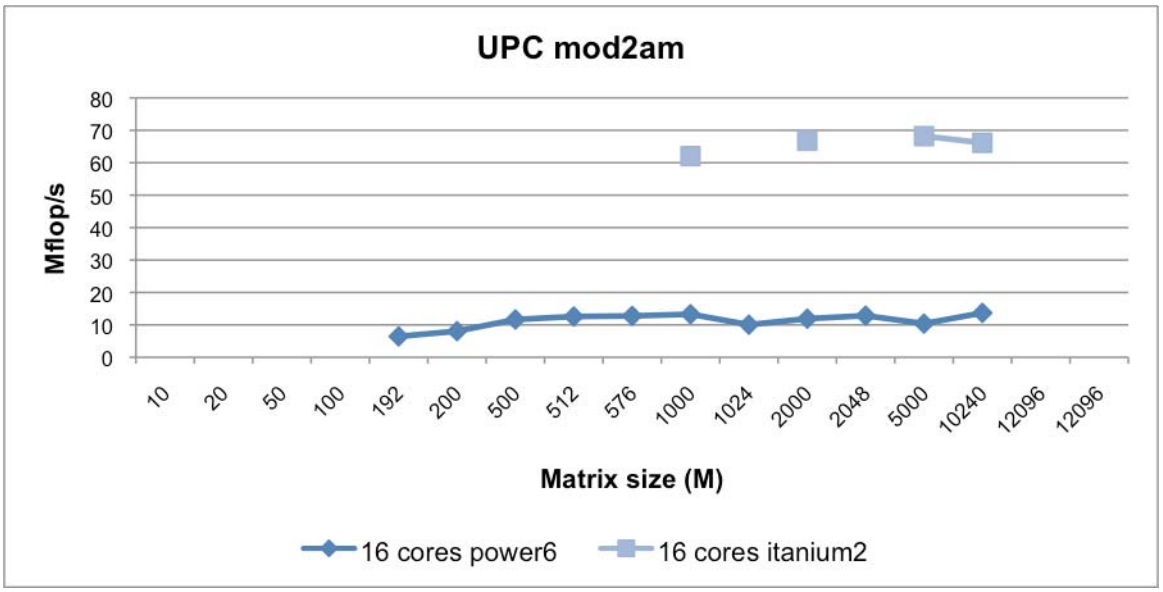

Figure 50: UPC mod2am performances (tests performed on *itanium* and *huygens*)

mod2as

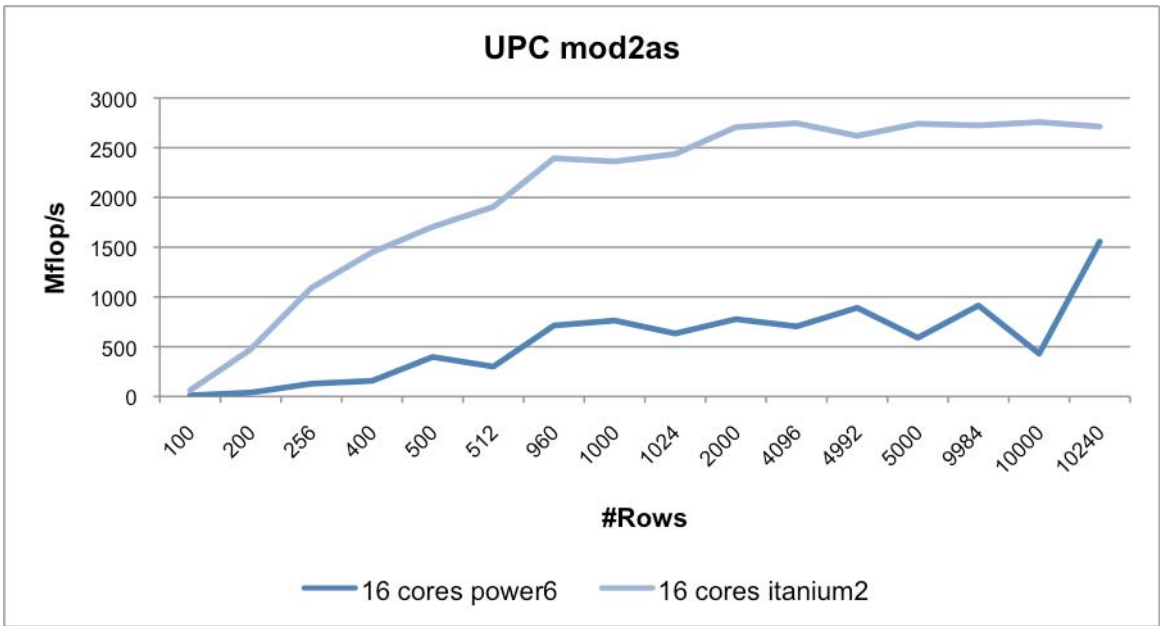

Figure 51: UPC mod2as performances (tests performed on *itanium* and *huygens*)

mod2f

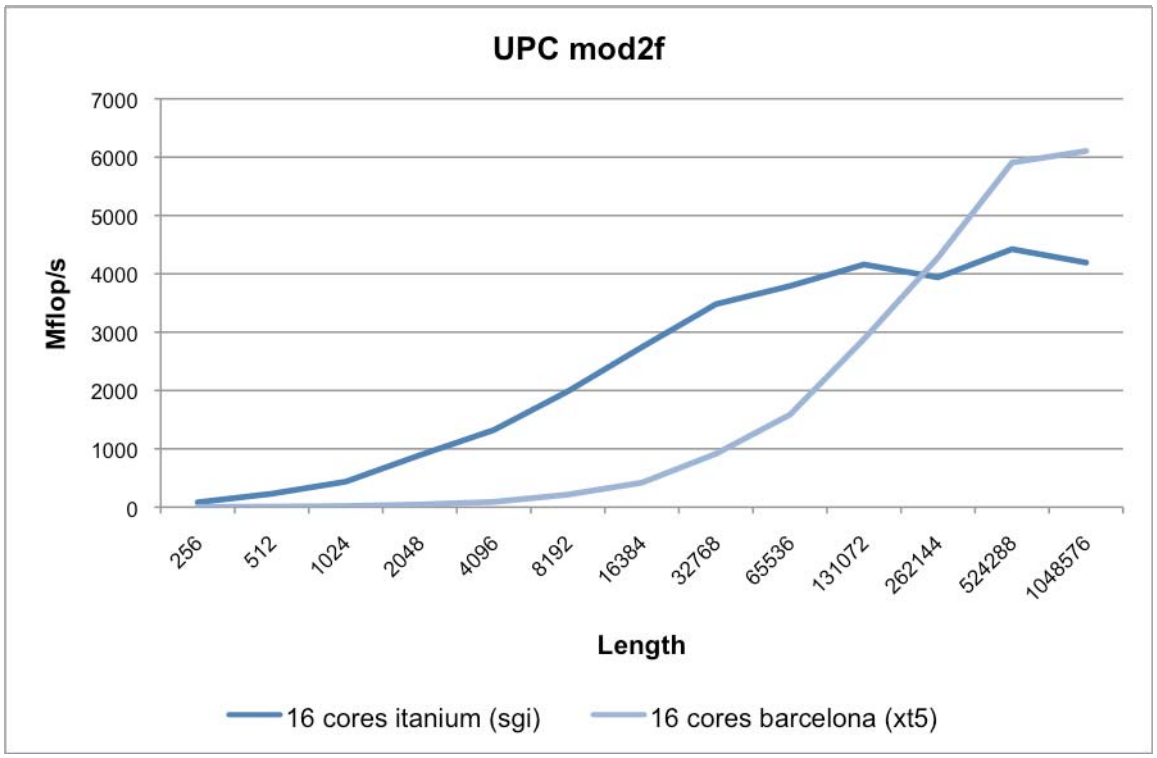

Figure 52: UPC mod2f performances (tests performed on *itanium* and *louhi*)

14.13 X10

Porting the kernels using distributed arrays was not successful. For both mod2am and mod2as a multi-threaded version (using Local Activities in X10 terminology) has been implemented.

Basic Information

Author: Walter Lioen (SARA)
Euroben Kernel: mod2am, mod2as
Hardware: *huygens*
Compiler Version: X10 v1.7.5

Developer Diary

mod2am

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
3 d	n/a		n/a	Started as X10 novice (having significant albeit somewhat rusty Java experience). Studying language report and some tutorial material. Installing X10 and toying around with "Hello World" and some trivial examples.
1d	246.4 s	1	n/a	Minimalistic implementation of matrix multiply as in mod2am.
1d			n/a	More-or-less full implementation of mod2am.
4h			n/a	Feeble attempt to simply read an input file. This can not be found in the language report. There is no API doc. (I tried understanding the (Java) X10 compiler runtime source.) There is no good tutorial example.
2h	23.44 s	32	n/a	Multi-threaded version: basically changing a for loop into a foreach loop. In X10 terminology, this creates multiple "local" "activities". Due to a compiler bug it was not possible to simply "extend" the Mod2amBase class to a Mod2amLocalActivities class where we only have to override the "mxm" method. (It is not possible to use the class constructor.) We could improve on this implementation by blocking the foreach loop limiting the number of threads to the number of available cores, however, this is something that should come natural by using distributed arrays.
3h	9.785 s (base) 4.967 s (multi-threaded)	1 32	n/a	Used the C++ compiler backend instead of the Java backend. Stumbled into some differences. 1. Formatted I/O differs (Java and C++ format strings). 2. An X10 program that compiled and ran successfully using the C++ backend was rejected by the Java backend. The Java backend turned out to be right.
2d			n/a	See (*)

(*) Failed attempt (basically due to both lack of X10 experience and time) to use distributed arrays (the actual PGAS paradigm). Since on the machine at hand we are limited to a single node there is no need to use PGAS. Creating a block distribution is an example of something that is incorrectly described in the Language Report. Clearly this takes extra time to understand. Furthermore, at present there is no possibility to create user distributions. For the matrix-multiply we most probably want to create a 2 dimensional blocking which is

something that is not yet available. The problem is not in writing a syntactically correct X10 version: creating a block-distributed array is more-or-less trivial. Using a single distributed array is also simple, the problem arises when using multiple distributed (or even using 1 distributed array where the other arrays are not distributed). At this point we encounter runtime errors ("bad place exceptions"). Since these exceptions are thrown without much additional info, it is hard to find out what exactly is wrong. On top of this debugging problem the compilation for our simple x10 class takes 1 minute, this also hinders the development / debugging speed.

mod2as

Development Time	Achieved Performance	Number of Cores	Dataset used	Comments
1d				More-or-less full mod2as port.
1d	14.912 s	1	10240	Added a private rand64 class that produces the same result as the rand64 function. The "problem" here is that the current X10 compiler does not support the ULong type as described in the Language Report (probably because Java itself also does not have (native) unsigned integers). This is true for both the Java as well as the C++ compiler backend. However, rand64 can also be implemented using (signed) longs, still producing the same random sequence.
1h	62.732 s	32	10240	Multi-threaded version using "foreach". Due to the thread overhead there is a performance drop.
1h	7.8296 s	32	10240	Multi-threaded version using "foreach". Here we use 32 threads (the number of cores) instead of the outvec dimension (nrows). Due to the thread overhead we only reach a speed-up factor of 2.

Porting Results

mod2am

Overall implementation result: Successful

Successful in the sense that we have single node version that more-or-less scales. (The Java backend uses a thread pool with 64 threads corresponding with 32 cores in a node running in SMT mode. The C++ backend appears to allow 2 – 4 threads). The parallel speed-up when using the Java backend was a factor of 10.5. The parallel speed-up when using the C++ backend was a factor of 2. We did not manage to correctly use distributed arrays (PGAS!) in the available time.

Consistency of the numerical results:

The results are identical, i.e. the "check" function / methods are implemented and intermediate results / work arrays have been checked by hand.

General comments:

The Java backend is extremely slow; the C++ version is a lot faster, however still too slow (both compared to e.g. the portable C version). This of is in line with the experimental state of the language. Performance is possible by linking optimized libraries e.g. ESSL/BLAS dgemm, however, in that case we are no longer looking at the X10 performance.

Problems encountered:

- X10 is an evolving standard. It started as a Java extension, however, it is diverging from Java. The language report is a (very) formal description and does not contain much tutorial material. The current X10 version is the so-called 1.7 language version. Most

tutorials are still based on the 1.5 definition (more Java like). The language report contains some bugs, so you have to fall back on some example programs or even the compiler source. There is also a reference to the API documentation (I assume like the Java API documentation), however this (online?) documentation is not available.
- Compiler errors can be very obscure: when using the Java compiler backend, the X10 code is transformed to Java code and very often a user programming error results in Java compiler error messages on the intermediate file without having a direct clue on what is wrong in your X10 code.
- The C++ backend on Linux allows for "multiple places", however, this does not (yet) work for Linux on Power.

Maturity of the standard/compiler:
This is not meant as criticism, however, the "Experimental" in the title of the language report: "Report on the Experimental Language X10" says all. The language itself is still evolving meaning that code had/has to be changed coming from 1.5 to 1.7. Clearly, X10 should not yet be used for production work; however, it is good to see where we are heading with PGAS languages. Since there is no native compiler, both compiler messages and runtime errors can originate from intermediate files.

Number of lines of source code:
Both versions are 160 lines of code.

Suitability of the compiler/ paradigm to implement the kernel:
Matrix multiplication is very well suited for the PGAS paradigm.

mod2as

Overall implementation result: Successful
Successful in the sense that we have a parallel version that gives a parallel speed-up, however using 32 threads (identical to the number of course) we roughly see a speed-up of 2. This is probably due to the thread overhead and the limited amount of work per thread.

Consistency of the numerical results:
The results are identical, i.e. the "check" function / methods are implemented and intermediate results / work arrays have been checked by hand.

General comments:
For all other comments also for the fields below, we refer to the mod2am diary. An interesting observation is that opposed to the mod2am case, the Java runtime results in better execution times than the C++ runtime.

Problems encountered:
See mod2am

Maturity of the standard/compiler:
See mod2am

Number of lines of source code:
294 lines of code.

Suitability of the compiler/ paradigm to implement the kernel:

Using a (probably slightly re-ordered / padded) distributed sparse matrix seems well-suited for the PGAS paradigm.

Performance Measurements

mod2am

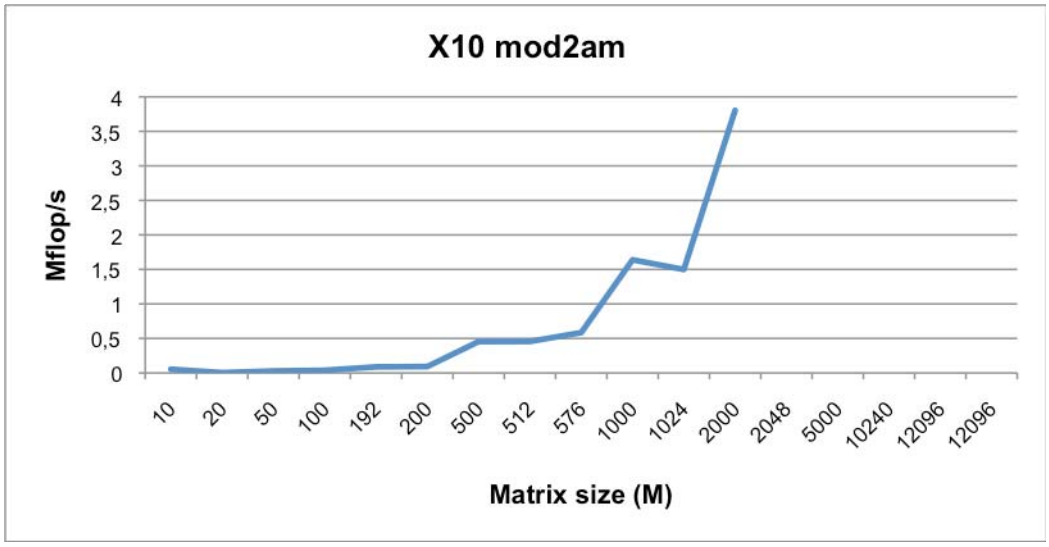

Figure 53: X10 mod2am performances (tests performed on *huygens*)

mod2as

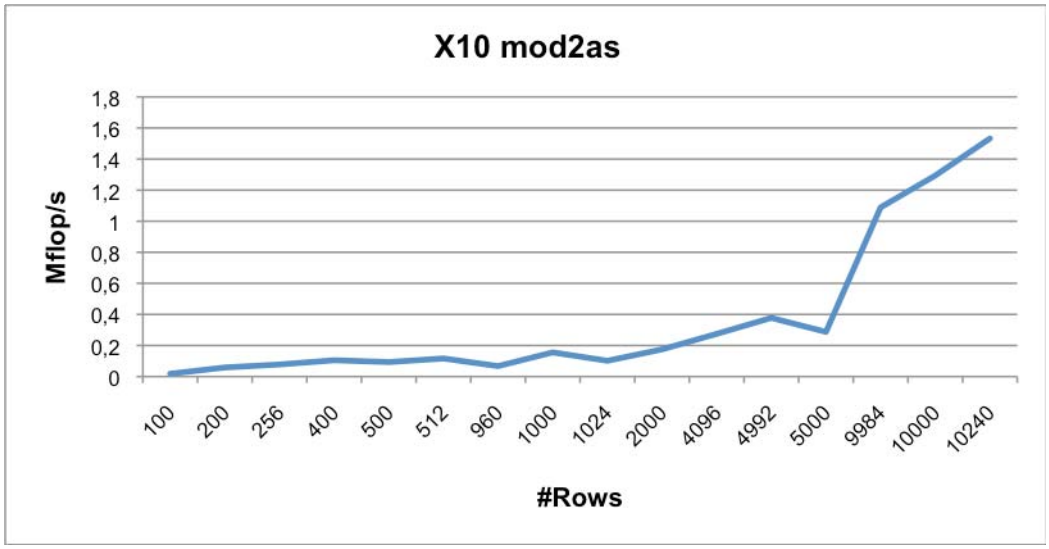

Figure 54: X10 mod2as performances (tests performed on *huygens*)