SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

INFRA-2011-2.3.5 – Second Implementation Phase of the European High Performance Computing (HPC) service PRACE

PRACE-2IP

PRACE Second Implementation Project

Grant Agreement Number: RI-283493

D12.1
Heterogeneous and Auto-tuned Runtime System

Final

Version: 1.0

Author(s): Christian Perez, Zhengxiong Hou (INRIA), Judit Planas, Rosa Badia, Eduard Aygüadé, Jesus Labarta (BSC), Michael Schliephake (SNIC-KTH), Chandan Basu, Johan Raber (SNIC-Liu), Massimo Guarrasi CINECA), Lasse Natvig (NTNU), Kostis Nikas (GRNET), Krzysztof T. Zwierzyński (PSNC), Seren Soner, Can Özturan (Bogazici University), Renato Miceli (NUIG), François Bodin (CAPS)

Date: 22.02.2013

Project and Deliverable Information Sheet

PRACE Project	Project Ref. №: RI-283493	
	Project Title: PRACE Second Implementation Project	
	Project Web Site:http://www.prace-project.eu	
	Deliverable ID: <D12.1>	
	Deliverable Nature: < Report>	
	Deliverable Level: PU *	Contractual Date of Delivery: 28 / February / 2013
		Actual Date of Delivery: 28 / February / 2013
	EC Project Officer: Leonardo Flores Añover	

* - 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: Heterogeneous and Auto-tuned Runtime System	
	ID:D12.1	
	Version:<1.0>	Status: Final
	Available at: http://www.prace-project.eu	
	Software Tool: Microsoft Word 2010	
	File(s): D12.1.docx	
Authorship	Written by:	Christian Perez, Zhengxiong Hou (INRIA); Judit Planas, Rosa Badia, Eduard Aygüadé, Jesus Labarta (BSC); Michael Schliephake (SNIC-KTH); Chandan Basu, Johan Raber (SNIC-Liu); Massimo Guarrasi (CINECA); Lasse Natvig (NTNU); Kostis Nikas (GRNET); Krzysztof T. Zwierzyński (PSNC); Seren Soner, Can Özturan (Bogazici University); Renato Miceli (NUIG); François Bodin (CAPS)
	Contributors:	
	Reviewed by:	Nuzhet Dalfes, ITU; Dietmar Erwin, FZJ
	Approved by:	MB/TB

Document Status Sheet

Version	Date	Status	Comments
0.1	08/02/2013	Draft	First draft
1.0	22/02/2013	Final version	

Document Keywords

Keywords:	PRACE, HPC, Research Infrastructure

Disclaimer

This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement n°RI-283493. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the Project and to the extent foreseen in such agreements. Please note that even though all participants to the Project are members of PRACE AISBL, this deliverable has not been approved by the Council of PRACE AISBL and therefore does not emanate from it nor should it be considered to reflect PRACE AISBL's individual opinion.

Copyright notices

© 2013 PRACE Consortium Partners. All rights reserved. This document is a project document of the PRACE project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the PRACE partners, except as mandated by the European Commission contract RI-283493 for reviewing and dissemination purposes.

All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.

Table of Contents

Project and Deliverable Information Sheet i

Document Control Sheet i

Document Status Sheet i

Document Keywords ii

Table of Contents iii

List of Figures iii

List of Tables iv

References and Applicable Documents iv

List of Acronyms and Abbreviations iv

Executive Summary 1

1 Introduction 2

2 Task Organization 3

3 Detailed Project Descriptions 4

3.1 The State-of-the-Art in Directive-Guided Auto-Tuning for Accelerator and Heterogeneous Many-Core Architectures 4

3.2 Selection of Task Implementations in the Nanos++ Runtime 7

3.3 Auto-tuning of the FFTW library for Massively Parallel Supercomputers 8

3.4 Auto-tuning 2D Stencil Application on Multi-core Parallel Machines 12

3.5 Generating integral graphs using PRACE Research Infrastructure 14

3.6 Towards runtime-clustering and improved implementations of collective operations in MPI 17

3.7 Energy-efficient Sparse Matrix Auto-tuning with CSX 18

3.8 Power Instrumentation of Task-based Applications using Model-specific Registers on the Sandy Bridge Architecture 20

3.9 An Auction Based SLURM Scheduler for Heterogeneous Supercomputers and its Comparative Performance Study 22

4 Summary 24

List of Figures

Figure 1: Code Variant Performance Comparison. 6

Figure 2: DNADist Time on CARMA and Kepler 6

Figure 3: Execution times for the different schedulers on the PBPI. 7

Figure 4: Efficiency obtained during some tests on PLX cluster, using from 1 to 512 cores. We use 5 different arrays, with size from 256x256 to 8192x8192 points. These tests have been performed both with FFT_MEASURE and FFT_ESTIMATE flags. We can easily see that the efficiency of the FFTW runs decreases suddenly when the number of used cores increase. 10

Figure 5: Computation time of Jacobi benchmark with different data partitioning policies on one node of Curie xlarge nodes 13

Figure 6: Performance of the Jacobi benchmark for 3 different data sizes with different number of nodes and increasing number of processes on Curie thin nodes (N.P: N is the number nodes and P is the number of processes per node). 14

Figure 7: The time of graph generation (G) and eigenvalue calculation and sieving (E) in sequential and in parallel. 15

Figure 8: (a) CSX execution phases from a power consumption perspective. (b) CSX execution power footprint with an increasing number of threads (HyperThreading enabled). 20

Figure 9: Energy efficiency of BlackScholes for with various kernel threading/vectorization balances. 22
Figure 10: Comparison of AUCSCHED and SLURM's own BACKFILL plug-ins. Workload descriptions (a), utilizations (b), fragmentation (c) and spread (d). 23

List of Tables

Table 1: Overview of effort per partner 3
Table 2: Ratio between time of execution of the Fast Fourier Transform using the 2D Domain Decomposition, and the same time using standard Slab Decomposition (Standard FFTW algorithm). Red cells: Time of standard FFTW < Time of 2D decomposition. Yellow cells: Time of standard FFTW ~ Time of 2D decomposition. Green cells: Time of standard FFTW > Time of 2D decomposition. Blue cells: Time of standard FFTW >> Time of 2D decomposition. Please note that the greater is the number of cores, the greater are the performance of the new 2D Domain Decomposition algorithm. 11
Table 3: The number of connected integral graphs c1(n), $1 \leq n \leq 12$ 15

References and Applicable Documents

[1] R. Miceli and al., 2012. AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. LNCS. vol 7782, pp 328-342
[2] OpenMP ARB, "OpenMP Application Program Interface, v.3.0," May 2008.
[3] Cooley, J. W. & Tukey, J., 1965. An algorithm for the machine calculation of complex Fourier Series. *Mathematics of Computation,* Issue 19, pp. 297-301
[4] M. Frigo & S. G. Johnson, 2013, Available at: http://www.fftw.org/
[5] M. Frigo & S. G. Johnson, "The Design and Implementation of FFTW3", Proceedings of the IEEE 93 (2005) 216
[6] CINECA, s.d. *PLX cluster.* Available at: http://www.hpc.cineca.it/content/ibm-plx-gpu-user-guide
[7] CINECA, s.d. *FERMI cluster.* Available at: http://www.hpc.cineca.it/content/fermi-reference-guide
[8] R. Schultz, 2008, "*3D FFT with 2D decomposition*", CS project report, Center for molecular Biophysics, available at https://cmb.ornl.gov/members/z8g/csproject-report.pdf
[9] N. Li & S. Laizet, 2010. *2DECOMP&FFT – "A highly scalable 2D decomposition library and FFT interface"* Cray User Group 2010 Conferences, Edinburgh, s.n.
[10] B.D. McKay, http://cs.anu.edu.au/~bdm/nauty/
[11] H. Lien, L. Natvig, A. Al Hasib, J. C. Meyer, Case Studies of Multi-core Energy Efficiency in Task Based Programs, Proceedings of ICT-GLOW 2012, pp.44-54.

List of Acronyms and Abbreviations

AMD	Advanced Micro Devices
AMG	Algebraic MultiGrid
API	Application Programming Interface
BDO	Block-Diagonal with Overlap
BiCGStab	Bi-Conjugate Gradient Stabilized
BLAS	Basic Linear Algebra Subprograms
BSC	Barcelona Supercomputing Center (Spain)
CEA	Commissariat à l'EnergieAtomique (represented in PRACE by GENCI, France)
CFD	Computational Fluid Dynamics
CINECA	ConsorzioInteruniversitario, the largest Italian computing centre (Italy)
CN	Column-Net
CPU	Central Processing Unit

CSC	Finnish IT Centre for Science (Finland)
CSCS	The Swiss National Supercomputing Centre (represented in PRACE by ETHZ, Switzerland)
CSR	Compressed Sparse Row (for a sparse matrix)
CT	Computer Tomography
CUDA	Compute Unified Device Architecture (NVIDIA)
DEISA	Distributed European Infrastructure for Supercomputing Applications.EU project by leading national HPC centres.
DFT	Discrete Fourier Transform
DHT	Discrete Hartley Transform
DMA	Direct Memory Access
DNA	DeoxyriboNucleic Acid
DRAM	Dynamic Random Access memory
DSA	Digital Subtraction Angiography
EC	European Community
EESI	European Exascale Software Initiative
EPCC	Edinburg Parallel Computing Centre (represented in PRACE by EPSRC, United Kingdom)
ETHZ	Eidgenössische Technische Hochschule Zuerich, ETH Zurich (Switzerland)
ESFRI	European Strategy Forum on Research Infrastructures; created roadmap for pan-European Research Infrastructure.
FEM	Finite Element Method
FETI	Finite Element Tearing and Interconnect
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform of the West (a powerful Parallel FFT lybrary)
FP	Floating-Point
FPU	Floating-Point Unit
FZJ	Forschungszentrum Jülich (Germany)
GB	Giga (= 2^{30} ~ 10^9) Bytes (= 8 bits), also GByte
Gb/s	Giga (= 10^9) bits per second, also Gbit/s
GB/s	Giga (= 10^9) Bytes (= 8 bits) per second, also GByte/s
GCS	Gauss Centre for Supercomputing (Germany)
GÉANT	Collaboration between National Research and Education Networks to build a multi-gigabit pan-European network, managed by DANTE. GÉANT2 is the follow-up as of 2004.
GENCI	Grand Equipement National de CalculIntensif (France)
GFlop/s	Giga (= 10^9) Floating point operations (usually in 64-bit, i.e. DP) per second, also GF/s
GHz	Giga (= 10^9) Hertz, frequency =10^9 periods or clock cycles per second
GigE	Gigabit Ethernet, also GbE
GLSL	OpenGL Shading Language
GNU	GNU's not Unix, a free OS
GP	Graph Partitioning
GPGPU	General Purpose GPU
GPU	Graphic Processing Unit
GRNET	Greek Research & Technology Network (Greece)
GS	Gram-Schmidt
GWU	George Washington University, Washington, D.C. (USA)
HDD	Hard Disk Drive
HE	High Efficiency

HECToR	High-End Computing Terascale Resource
HMPP	Hybrid Multi-core Parallel Programming (CAPS enterprise)
HP	Hypergraph Partitioning
HPC	High Performance Computing; Computing at a high performance level at any given time; often used synonym with Supercomputing
HPCC	HPC Challenge benchmark, http://icl.cs.utk.edu/hpcc/
HPCS	High Productivity Computing System (a DARPA program)
HPL	High Performance LINPACK
IBM	Formerly known as International Business Machines
IDRIS	Institut du Développementet des Ressources en Informatique Scientifique (represented in PRACE by GENCI, France)
IEEE	Institute of Electrical and Electronics Engineers
IESP	International Exascale Project
IPB	Institute of Physics Belgrade
IMB	Intel MPI Benchmark
I/O	Input/Output
ITU-UHeM	Istanbul Technical University – Center of High Performance Computing
JSC	Jülich Supercomputing Centre (FZJ, Germany)
KB	Kilo (= 2^{10} ~10^3) Bytes (= 8 bits), also KByte
KTH	Kungliga Tekniska Högskolan (represented in PRACE by SNIC, Sweden)
LBE	Lattice Boltzmann Equation
LINPACK	Software library for Linear Algebra
LLNL	Lawrence Livermore National Laboratory, Livermore, California (USA)
LQCD	Lattice QCD
LRZ	Leibniz Supercomputing Centre (Garching, Germany)
LS	Local Store memory (in a Cell processor)
MB	Mega (= 2^{20} ~ 10^6) Bytes (= 8 bits), also MByte
MB/s	Mega (= 10^6) Bytes (= 8 bits) per second, also MByte/s
MDT	Meta Data Target
MFC	Memory Flow Controller
MFlop/s	Mega (= 10^6) Floating point operations (usually in 64-bit, i.e. DP) per second, also MF/s
MGS	Modified Gram-Schmidt
MHz	Mega (= 10^6) Hertz, frequency =10^6 periods or clock cycles per second
MIPS	Originally Microprocessor without Interlocked Pipeline Stages; a RISC processor architecture developed by MIPS Technology
MIT	Massachusetts Institute of Technology
MKL	Math Kernel Library (Intel)
Mop/s	Mega (= 10^6) operations per second (usually integer or logic operations)
MoU	Memorandum of Understanding
Mups	Million lattice site updates per second
MPI	Message Passing Interface
NCSA	The National Centre for Supercomputing Applications (Sofia, Bulgaria)
oGPVS	ordered Graph Partitioning by Vertex Separators
OpenCL	Open Computing Language
OpenGL	Open Graphic Library
Open MP	Open Multi-Processing
OS	Operating System
OSS	Object Storage Server
OST	Object Storage Target

oVS	ordered Vertex Seperator
PGI	Portland Group, Inc.
pNFS	Parallel Network File System
POSIX	Portable OS Interface for Unix
PPE	PowerPC Processor Element (in a Cell processor)
PRACE	Partnership for Advanced Computing in Europe; Project Acronym
PSNC	Poznan Supercomputing and Networking Centre (Poland)
QE	Quantum Espresso
QR	QR method or algorithm: a procedure in linear algebra to compute the eigenvalues and eigenvectors of a matrix
RAM	Random Access Memory
RDMA	Remote Data Memory Access
RISC	Reduced Instruction Set Computer
RN	Row-Net
RNG	Random Number Generator
RPM	Revolution per Minute
SAN	Storage Area Network
SARA	Stichting Academisch Rekencentrum Amsterdam (Netherlands)
SAS	Serial Attached SCSI
SATA	Serial Advanced Technology Attachment (bus)
SDK	Software Development Kit
SGEMM	Single precision General Matrix Multiply, subroutine in the BLAS
SGI	Silicon Graphics, Inc.
SHMEM	Share Memory access library (Cray)
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor, also Subnet Manager
SMP	Symmetric Multi Processing
SNIC	Swedish National Infrastructure for Computing (Sweden)
SP	Single Precision, usually 32-bit floating point numbers
SpMxV	Sparse Matrix Vector multiplication
SPU	Synergistic Processor Unit (in each SPE)
SSD	Solid State Disk or Drive
SSE	Streaming SIMD Extensions
STFC	Science and Technology Facilities Council (represented in PRACE by EPSRC, United Kingdom)
STRATOS	PRACE advisory group for STRAtegic TechnOlogieS
TB	Tera (= 240 ~ 1012) Bytes (= 8 bits), also TByte
TFlop/s	Tera (= 1012) Floating-point operations (usually in 64-bit, i.e. DP) per second, also TF/s
Tier-0	Denotes the apex of a conceptual pyramid of HPC systems. In this context the Supercomputing Research Infrastructure would host the Tier-0 systems; national or topical HPC centres would constitute Tier-1
UFL	University of Florida
UNICORE	Uniform Interface to Computing Resources. Grid software for seamless access to distributed resources.
VSB	Technical University of Ostrava
WCSS	Wrocławskie Centrum Sieciowo-Superkomputerowe (Wrocław Centre for Networking and Supercomputing)
WP	Work Package

Executive Summary

Work Package 12 (WP12) "Novel Programming Techniques" performs research and development in four key areas for future multi-petascale and exascale systems. The work in WP12 focuses on auto tuned and automatic techniques to be applied in parallel programming model runtimes (Task 12.1: "Auto-tuned runtime Environments"), performance tools (Task 12.3: "Development environments and tools") and file systems (Task 12.4: "File system optimization"). Furthermore, as it is widely accepted that the key to exploiting future high-end systems will be based on research on new numerical algorithms as well as advancing the parallel processing technology used for higher scalability in numerical applications; consequently WP12 also focuses on research studies exposing more scalability for numerical algorithms (Task 12.2: "Scalable numerical algorithms").

Task 12.1 contributes to improve the support of auto-tuning methods to face the complexity of existing and future large scale systems. It impacts parallel languages, runtime, generic and kernel specific auto-tuning algorithms, multi-core, many-core and multi-node sytems, as well as batch systems and energy consumption measurement methods. In total, five areas organized as individual *projects* were covered. This document is a summary of the projects' results. It contains a brief description of covered topics. Links to PRACE white papers are given for those readers that are interested in more detailed information.

The key research topics investigated in Task 12.1, related to auto-tuned runtime environnement, are the following:

- Language and runtime support for code variant selection
- Kernel specific auto-tuning algorithm (FFTW, 2D stencil, and integral graph)
- Auto-tuning collective operations in MPI
- Energy efficiency
- GPU support in batch scheduler

1 Introduction

High performance systems are getting more and more complex to continue to deliver more performance despite the frequency wall. This complexity is particular high within a node, with the apparition of large multi-core and/or many-core systems, leading to deep memory hiearchies and heterogeneous nodes. Moreover, as energy is becoming a real constraint, the challenge is to deliver the best performance while minizing energy consumption. In addition, more classical but still open challenges such as taking into account the network topology or optimizing job placement remain.

Task 12.1 of WP12 addresses these challenges in five areas:

- *Language and runtime support for code variant selection*: generic runtime environments for parallel platforms have be extended to determine at runtime the best implementation or resource choice (i.e. accelerator or CPU core) for the different computation regions (i.e. task) of a program.
- *Kernel specific auto-tuning algorithm*: specific auto-tuning algorithms have been developed for two important application kernels (FFT and 2D stencil) and the problem of generating integral graphs. They are based on intensive benchmarks to build some kinds of performance model.
- *Auto-tuning collective operations in MPI*: As systems become larger, network topology plays an increasing role in application performance. Hence, we developped topology-optimized implementation of MPI collective operations such as all-to-all, which is in particular important for FFT.
- *Energy efficiency*: As energy becomes a limiting critera, we adressed the issue of developing accurate online energy consumption tools as well as auto-tuning algorithms for sparse matrix computation kernels based on such real time energy measurement.
- *GPU support in batch scheduler*: the batch scheduler is a key element of any large scale system that needs to automatically perform efficient resource selection with respect to job. We addressed the issue of improving the support of heterogeneous node in Slurm.

The deliverable itself is quite concise in order to allow readers to easily identify the projects that are of particular interest for them and to encourage further reading in the accompanying white papers or the referenced publications.

The remainder of the deliverable is organized as follows. Chapter 2 provides a description of the organization of Task 12.1. Chapter 3 provides detailed descriptions of the projects along with obtained results and discussions. Chapter 4 presents the conclusions.

2 Task Organization

Table 1 displays a rough per partner PM distribution of Task 12.1. Around 98 PMs are allocated for this work and experts from nine countries are participating in Task 12.1. In total 9 projects have been completed.

Country	Partner	PMs
France	GENCI	21
Spain	BSC	20
Sweden	SNIC	14
Turkey	UHeM	13
Italy	CINECA	10
Poland	PSNC	5
Norway	SIGMA	6
Greece	GRNET	6.5
Ireland	NUIG	3
	Sum	98.5

Table 1: Overview of effort per partner

These 9 projects cover the 5 key challenges as follows:

- *Language and runtime support for code variant selection*
  - The State-of-the-Art in Directive-Guided Auto-Tuning for Accelerator and Heterogeneous Many-Core Architectures (NUIG & CAPS Enterprise)
  - Selection of Task Implementations in the Nanos++ Runtime (BSC)
- *Kernel specific auto-tuning algorithm*
  - Auto-tuning of the FFTW library for Massively Parallel Supercomputers (CINECA)
  - Auto-tuning 2D Stencil Application on Multi-core Parallel Machines (GENCI)
  - Generating integral graphs using PRACE Research Infrastructure (PSNC)
- *Auto-tuning collective operations in MPI*
  - Towards runtime--clustering and improved implementations of collective operations in MPI (SNIC)
- *Energy efficiency*
  - Power Instrumentation of Task-based Applications Using Machine Specific Registers on Intel Sandy Bridge Architecture (SIGMA)
  - Energy-efficient Sparse Matrix Auto-tuning with CSX (GRNET)
- *GPU support in batch scheduler*
  - An Auction Based SLURM Scheduler for Heterogeneous Supercomputers and its Comparative Performance Study (UHeM)

3 Detailed Project Descriptions

In this chapter, detailed descriptions of the Task 12.1 projects along with obtained results and discussions are provided. All projects contain links to accompanying white papers or publications. All PRACE technical white papers are available on the PRACE-RI web site http://www.prace-ri.eu/white-papers.

3.1 The State-of-the-Art in Directive-Guided Auto-Tuning for Accelerator and Heterogeneous Many-Core Architectures

Supported by: Renato Miceli (NUIG), François Bodin (CAPS Enterprise)

Whitepaper: Renato Miceli (NUIG), François Bodin (CAPS Enterprise), "The State-of-the-Art in Directive-Guided Auto-Tuning for Accelerator and Heterogeneous Many-Core Architectures", PRACE technical white paper.

Automatic performance tuning, also known as "auto-tuning", is a software procedure for selecting one out of several possible solutions to a computational problem. Each of the solutions, called a variant, provides a different algorithmic implementation with varied time and space complexities. In this paper we investigate auto-tuning approaches for improving the performance of applications for accelerator and heterogeneous many-core architectures. We focus on auto-tuning approaches guided by programming directives, also known as code annotations. Programming directives are inserted by code developers to help the compiler or auto-tuner throughout the task of understanding the code and proposing the best variant for the procedures to be tuned. We go over the state-of-the-art in directive-guided auto-tuning for many-core architectures, providing examples and comparing different approaches. In the end, we discuss the technology's commercial uptake, using the CAPS compilers as an example of the current industrial efforts to bringing auto-tuning into the mainstream development cycle.

Auto-tuning is as simple as using software to tune other pieces of software. The objective is to take the task of code tuning from the hands of programmers, in order to release human time to be invested in more skilled activities. Auto-tuning may be executed by the same software to be tuned, in which case we call it a "self-tunable" code; or by an external mechanism – e.g. library, runtime environment, independent application –, which we call an "auto-tuner". In either case, the auto-tuning approach will introduce changes to the algorithmic implementation in order to promote performance gains; some examples of current approaches are the use of novel programming languages, automatic code transformation techniques, programming environments, and language extensions. In this paper we focus on language extensions, specifically programming directives and annotations.

Programming directives, also known as annotations or pragmas, are language constructs generally employed to specify source code metadata. Directives have a dual use: (i) they may be commands, commonly referring to cross-cutting concerns; or (ii) they may be statements, affecting local or global blocks of code, but performing no action. Programmers often insert directives into the code to facilitate code writing or to provide useful information to the compiler. Currently, directives may carry important information auto-tuners may use to understand the source code, the data and dependencies involved, and the key factors that influence application performance. Searching the set of possible variants becomes an easier task with the help of programming directives. Recently, programming directives have been widely adopted in the community of legacy applications, with the advent of programming models that enable the use of multi- and many-core architectures in a seamless way, such as OpenACC.

A many-core architecture is the one that includes a many-core processor. These processors are a specialization of multi-core processors, thus including two or more independent CPUs, the

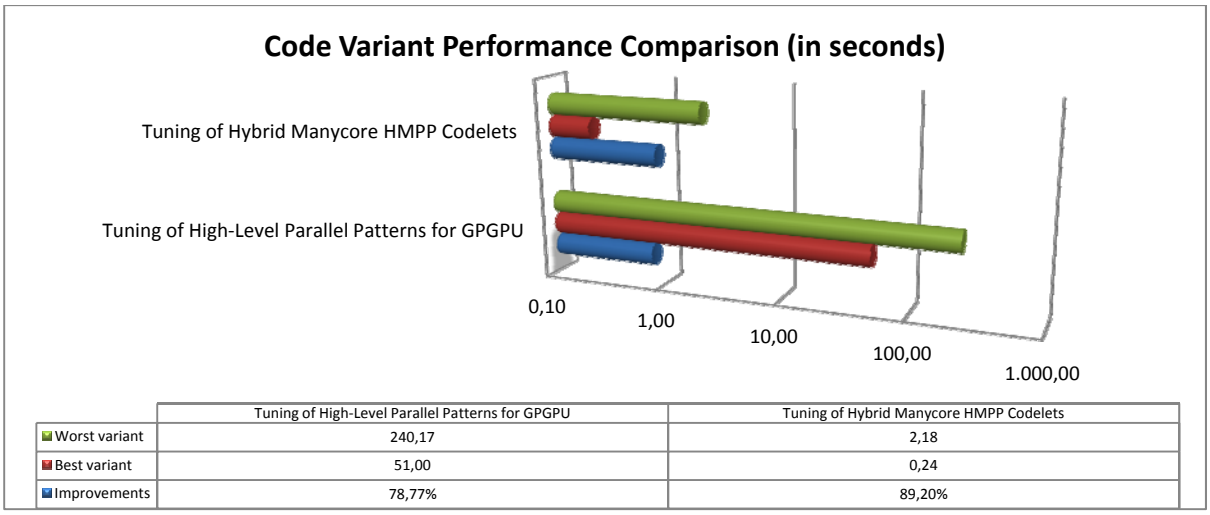
so-called “cores”. However, as opposed to multi-core processors, which contain tens of robust cores, many-core units contain hundreds to thousands of light cores. Many-core units generally come linked to a motherboard via a PCIe port in a machine that already contains a multi-core processor; in this setting, the many-core unit acts as a coprocessor or accelerator. The many-core units’ massively parallelism and the contrast to multi-core processor cores allow applications to benefit from both paradigms depending on the resource usage and the level of parallelism required. It also becomes apparent that balancing resource usage between the main processor and the accelerator is a necessary measure to obtain maximum performance on these heterogeneous many-core architectures. Auto-tuning applications running on accelerators, as well as balancing the load between the main processors and the accelerators, has become a great source of study and interest, both within the academic and industrial communities.

On the academic side, several projects tackle directive-guided auto-tuning for accelerator and many-core architectures. A relevant one is the AutoTune project. AutoTune is a European-funded FP7 project that aims at building an extensible auto-tuning framework for tuning application performance and energy consumption. The framework, called the “Periscope Tuning Framework” [1], instruments, compiles and executes the application to gather measurements, which will guide the tuning process. Besides the framework, the project consortium is working on a set of 3 out of 7 plugins to tune for heterogeneous many-core architectures. The plugins are the following:

- A user-guided tuning plugin: assesses multiple code variants statically defined to decide which variant to apply;
- A tuning plugin for high-level parallel patterns for GPGPUs: focuses on maximizing the throughput of pipeline patterns for many-core heterogeneous architectures by efficiently exploiting CPU and GPU cores of a targeted architecture, especially the replication factor and the buffer sizes of pipeline stages; and
- A tuning plugin for hybrid many-core HMPP codelets: tunes the performance of a codelet compilation, written either using OpenHMPP or OpenACC directives and compiled using the CAPS compiler, by performing source code transformations and assessing and choosing code variants for operations and algorithms used to implement codelets, plus target-specific variables and callbacks.

Currently, the Periscope Tuning Framework (PTF) only requires that the user specify the region where it should tune, plus the tuning parameters whose values should be experimented. A tuning region is defined directly in the code by the programming directives “USERREGION” and “END USERREGION”, which must surround the tuning region. During execution, whenever the tuning region is entered, PTF assigns a different value to the tuning parameter and gather statistics that will enable it to reason about the best code variant for that specific region.

The tuning plugins are being implemented based on tuning techniques for each specific tuning aspect. A preliminary study about the quality of these tuning techniques was conducted and the initial results are displayed in Figure 1.



	Tuning of High-Level Parallel Patterns for GPGPU	Tuning of Hybrid Manycore HMPP Codelets
Worst variant	240,17	2,18
Best variant	51,00	0,24
Improvements	78,77%	89,20%

Figure 1: Code Variant Performance Comparison.

On the AutoTune project, CAPS, a leading provider of many-core technology and services, provides the commercial uptake of the project outcomes. CAPS integrated auto-tuning interfaces and features to the CAPS compiler, allowing the source code to be adapted at runtime according to the underlying architecture. The CAPS compiler takes the source code, written using C/C++ or FORTRAN annotated with OpenHMPP or OpenACC directives, understands special auto-tuning pragmas and generates an executable that captures usage information and runtime metrics. The CAPS auto-tuning driver connects to the executable at runtime to create an optimization space of code transformations and algorithms to explore.

The tuning parameters currently supported by the CAPS compiler are the following:

- Implementations of a same kernel or codelet;
- Runtime configurable parameters for kernel and codelet executions (e.g. number of gangs, workers and vectors);
- Performing code transformations, especially regarding loop transformations (e.g. jam, split, unroll, gridify);
- Implementations of library calls, especially where no one-to-one mapping between libraries with similar features exist.

These auto-tuning approaches allow OpenHMPP and OpenACC codes compiled with the CAPS compiler to take advantage of the performance provided by heterogeneous hardware, without requiring the code to be manually changed. Some preliminary testing was conducted over the bioinformatics application DNADist on CARMA and Kepler architectures. A summary is Figure 2.

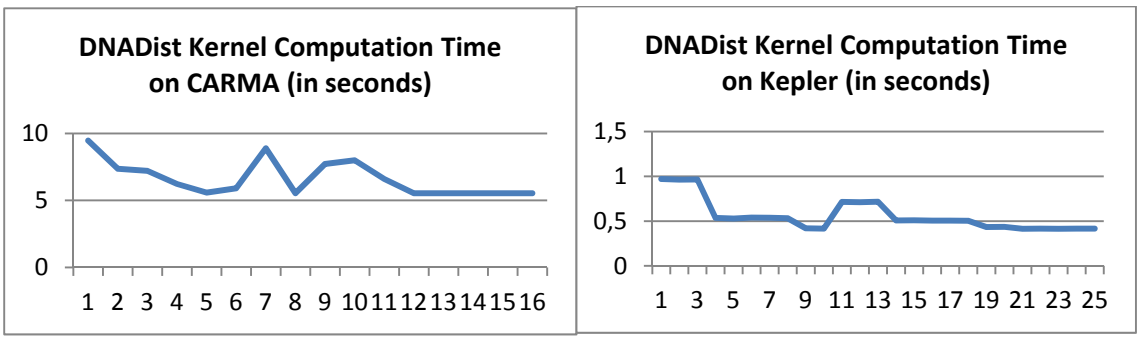


Figure 2: DNADist Time on CARMA and Kepler

3.2 Selection of Task Implementations in the Nanos++ Runtime

Supported by: Judit Planas, Rosa M. Badia, Eduard Aygüadé, Jesús Labarta (BSC, Barcelona Supercomputing Center)

Whitepaper: Judit Planas, Rosa M. Badia, Eduard Aygüadé, Jesús Labarta, "Selection of Task Implementations in the Nanos++ Runtime", PRACE technical white paper

This project presents a set of extensions to the OmpSs framework that shows how the application programmer can expose different specialized versions of tasks (i.e. pieces of specific code targeted and optimized for a particular architecture) and how they will be chosen at runtime to obtain the best performance achievable for the given application.

OmpSs [2] is a task-based programming model and framework that covers different homogeneous and heterogeneous architectures used nowadays and is open to cover future systems designed with new raising architectures.

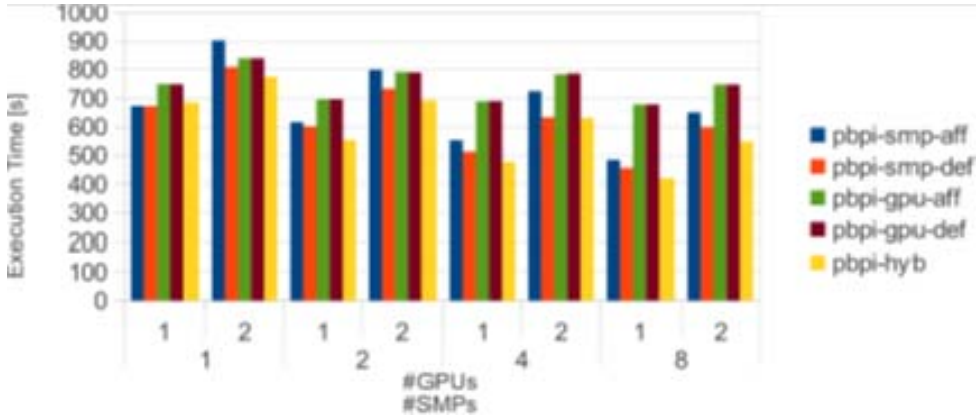


Figure 3: Execution times for the different schedulers on the PBPI.

Generally, there is not a single piece of code that fits all the existing hardware architectures, and even if we find that code, it will not be the best option (in terms of performance, energy consumption, etc.) for all of them. Thus, it is not unusual to find different ways of implementing the same algorithm. This scenario is getting worse with the emergence of new architectures that provide more performance to the applications, and thus, the code improvement and maintenance is getting more complicated and more expensive. In this work, we explored a solution build on OmpSs to alleviate this problem.

The technique evaluated in this work is based on a new scheduling policy for the OmpSs runtime called the *versioning scheduler*. It is based on the already existing dependency-aware scheduler that tries to follow task dependency chains in order to promote data locality and minimize data transfers in a fast and simple way. The *versioning scheduler* keeps and updates several data structures throughout the application execution that collect information related to each set of task implementations. The strategy is based on two different phases from the scheduler's point of view: the initial learning phase and the reliable information phase respectively. The initial learning phase consists of picking task versions from ready tasks in a Round-Robin fashion and distributing them among OmpSs workers. For each task version run, its execution time is recorded.

On the other hand, during the reliable information phase, the scheduler tries to assign each task version to its earliest executor. To make this decision, it takes into account who is the fastest executor of the task version set, but also how busy each worker is by checking each worker's task list. In this phase, execution information is also recorded exactly in the same way as in the previous phase: for each task version, its execution time is computed and its corresponding mean execution time is updated accordingly, so the scheduler is always learning and recording execution information. We consider that the initial learning phase

finishes when the scheduler has enough reliable information about the execution of task version sets.

We have evaluated the work that we have done in order to measure the performance of the presented OmpSs scheduler while running OmpSs applications using SMP and GPU specialized kernels. We run the selected applications with different configurations of numbers of cores and GPU devices to obtain the performance or execution time of each application. The set of applications used in the evaluation were Matrix multiplication, Cholesky factorization, and PBPI (a parallel implementation of a Bayesian phylogenetic inference method for DNA sequence data).

We used MinoTauro machine at the Barcelona Supercomputing Center, a multi-GPU system running Linux with two Intel Xeon E5649 6-Core at 2.53 GHz and two NVIDIA GPUs M2090.

The versioning scheduler (*ver*) is compared against two traditional schedulers, dependency-aware (*dep*) and affinity scheduler (*aff*) when only using the GPUs (*gpu*) and using only the processors (*smp*), and finally, a hybrid scenario that uses the host processor as well (*hyb*).

Figure 3 shows the execution time of each version (remember that lower is better in this chart). We can see that *pbpi-smp* versions run faster than the *pbpi-gpu* versions. This is due to the fact that sending all the computational work of first and second loops to the GPU is not worth, since all the data will have to be transferred back and forth to run the third loop on the SMP workers and memory transfers cannot be overlapped properly due to data dependences.

Furthermore, notice that the versioning scheduler is able to find the appropriate balance between SMP and GPU execution to take advantage and decrease the execution time. Although the amount of data transfers is higher, thanks to the look-ahead scheduler, it is able to overlap more data transfers with computation, so that we can see some benefit.

From our results, we observed that, in most of the cases, the *versioning scheduler* out-performs the other existent schedulers for the OmpSs runtime and at the same time, gives more flexibility to the programmer. Only in a few cases the versioning scheduler slightly slows down the application compared to the other OmpSs schedulers.

With this new proposed scheduler, the programmer can write hybrid applications where more than one implementation for one or more devices (SMP, GPU, ...) is given for her tasks. This feature enhances the programmability of applications and makes its maintenance easier, because the programmer, at any time, can develop a new implementation for an already existent task in her code that targets the same or a different device and that can potentially improve application's performance.

3.3 Auto-tuning of the FFTW library for Massively Parallel Supercomputers

Supported by: Massimiliano Guarrasi, Giovanni Erbacci (CINECA).

Whitepaper: Massimiliano Guarrasi, Giovanni Erbacci (CINECA), "Auto-tuning of the FFTW Library for Massively Parallel Supercomputers", PRACE technical white paper.

Currently, many challenging scientific problems require the use of Discrete Fourier Transform algorithms (DFT, e.g. [3]). One of the most "popular" libraries used by the scientific community is the FFTW library ([4], [5]). FFTW, which is free software, is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and with both real and complex data. The input data can have arbitrary length. FFTW employs O(n log n) algorithms for all lengths. FFTW supports arbitrary multi-dimensional data. This library includes parallel (multi-threaded) transforms for shared-memory systems, and distributed-memory parallel transforms, using MPI libraries. It

does not use a fixed algorithm for computing the transform, but instead it adapts the DFT algorithm to the underlying hardware in order to maximize performance. Hence, the computation of the transform is split into two phases. First, FFTW's *planner* "learns" the fastest way to compute the transform on the selected machine. The planner produces a data structure called a *plan* that contains this information. Subsequently, the plan is *executed* to transform the array of input data as dictated by the plan. The plan can be reused as many times as needed. In typical high-performance applications, many transforms of the same size are computed and, consequently, a relatively expensive initialization of this sort is acceptable. On the other hand, if you need a single transform of a given size, the one-time cost of the planner becomes significant. For this case, FFTW provides fast planners based on heuristics or on previously computed plans. During plan creation, users can choose the method that he/she prefers using the FFT_MEASURE (obtaining a more accurate plan) flags or FFTW_ESTIMATE flags (obtaining the plan faster).

Currently, particularly using small size data arrays, the FFTW libraries have been shown to not scale well beyond a few hundreds of cores. Considering that current PRACE Tier-0 systems consist of several hundreds of thousands of cores, and in order to obtain an access on these systems a good scalability at least up to some thousands of cores will be required Therefore an optimization of the FFTW implementation for massively parallel supercomputer is necessary.

In this section we present the work carried out by CINECA in the framework of the PRACE-2IP project which had the aim of improving the performance of the FFTW library by refining the auto-tuning mechanism that is already implemented in this library. This optimization was realized with the following activities:

- Identification of the major bottlenecks present in the current FFTW implementation;
- Investigation of the auto-tuning mechanism provided in FFTW in order to understand how performance is affected by domain decomposition;
- Introduction of a new parallel domain decomposition;
- Construction of a library to improve the performance of the auto-tuning mechanism.

Since it is necessary to find the bottlencks of the FFTW autotuning algorithm, extensive benchmarking is required. Thus, in the following we will describe these benchmarks, that were performed using two parallel supercomputer available in the CINECA infrastructures: IBM iDataPlex (PLX [6]), and IBM Blue Gene Q (FERMI [7]).

As an example of our initial benchmarks activity, we report the results a scalability test for some 2D pure MPI and hybryd (MPI+OpenMP) DFT executed on the PLX cluster at CINECA in Figure 4.

Summarizing the results of our initial benchmarks on the PLX cluster, we obtained some useful information that we used to improve the performance of FFTW library:

- Using small size arrays, the parallel performance of FFTW degrades greatly when we use more than one node (i.e. when we pass from a shared memory to a distributed memory systems);
- Using more than one node the best performance will be obtained using pure MPI communications;
- If we increase the number of point in the arrays (especially on the first index, that is the index involved in the parallel domain decomposition), the performance significantly improve;
- Generally, it may be disadvantageous using more cores than the number of points of the index involved in the domain decomposition;
- When the size of the first index isn't a multiple of the numbers of cores, the performances worsen.

To overcome the above mentioned limitations, we developed a routine, that improves the auto-tuning mechanism of the FFTW library by changing the number of used cores. A simple scheme of the structure of the algorithm that we implemented in the subroutine is shown in the related whitepaper. This subroutine having as input the size of the index involved in the parallel domain decomposition of the array that we want use (Nx), and the number of available cores (Np), give us the maximum number of cores useable to maximize the performance.

We extended this approach also in the 3D cases where we tried to improve the number of usable cores, using another innovative method: the 2D Domain Decomposition algorithm [8].

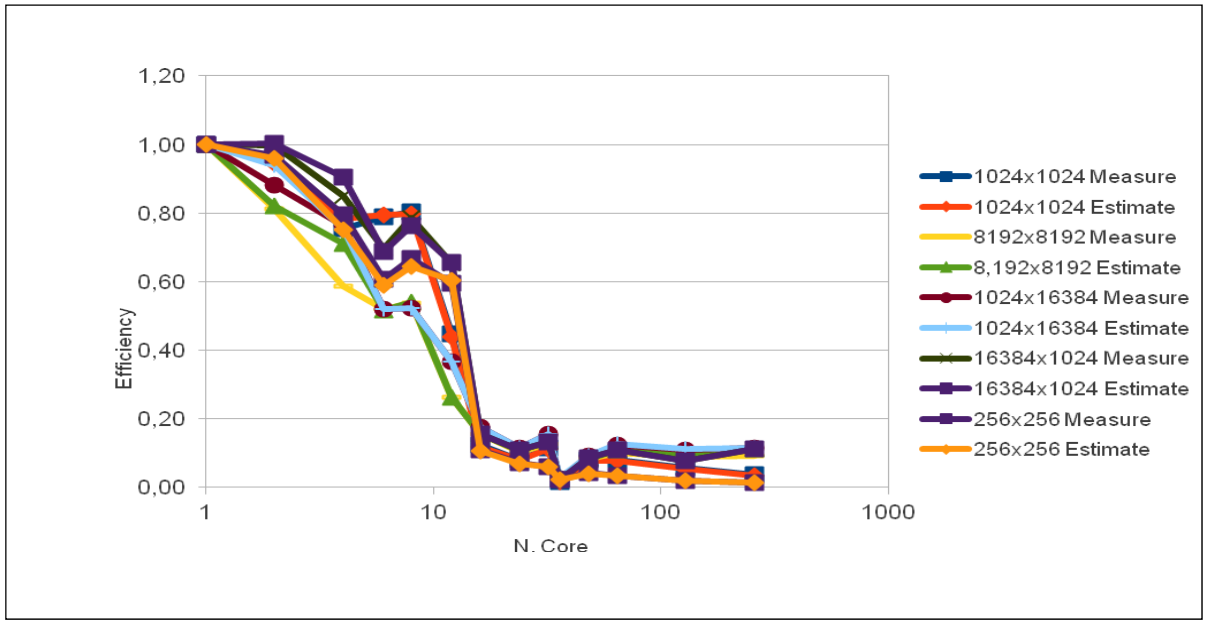


Figure 4: Efficiency obtained during some tests on PLX cluster, using from 1 to 512 cores. We use 5 different arrays, with size from 256x256 to 8192x8192 points. These tests have been performed both with FFT_MEASURE and FFT_ESTIMATE flags. We can easily see that the efficiency of the FFTW runs decreases suddenly when the number of used cores increase.

Currently, for the parallelization on distributed memory machines, the FFTW library uses MPI with a 1D decomposition of the data called *Slab decomposition.* Using this domain decomposition, the parallel computation of the 3D FFT can be divided into three steps;

- 2D FFT (or two 1D FFT) along the two local dimension;
- Global transpose;
- 1D FFT along the third dimension.

This decomposition is faster on a limited number of cores because it only needs one global transpose, minimizing communications. The main disadvantage of this approach is that the maximum parallelization is limited by the length of the largest axis of the 3D data array used. The performance can be further increased using a hybrid method, combining this decomposition with a thread based parallelization of the individual FFTs, but the performance increases only slightly. For example, for a cubic array with N^3 data points, the maximum number of usable cores scales with N in any case.
In order to improve significantly the performance on massively parallel supercomputers, an innovative approach would be necessary. We can reduce this scaling limitation using a 2D Domain Decomposition. In this case, the computation will be done in five steps:

- 1D FFT along the first local dimension;
- Global transpose;
- 1D FFT along the second dimension;

- Global transpose;
- 1D FFT along the third dimension.

Using this method, another global communication is required. Nevertheless, the global transpose requires communication only between subgroups of all nodes. For the same cubic data with N^3 data point, this means that the maximum number of cores scales as N^2, significantly increasing the number of usable cores.

Since the FERMI BG/Q cluster allows us to scale to a larger number of cores than PLX cluster, and it has enough memory to manage 3D arrays, this new domain decomposition algoritm was tested on FERMI, combining the standard FFTW library (for the 1D FFT calculation) with the 2Decomp&FFT library [9].

In Table 2 we report the results of some of the benchmarks. A more detailed description of the algorithms, and the performance obtained is provided in the related white paper.

Number of Data Point	N. of point	Number of Cores				
		256	*512*	*1024*	*2048*	*4096*
128x128x128	2,10e+6	0,35	5,20	14,35	21,63	63,91
256x256x256	1,68e+7	0,12	2,17	4,49	10,66	29,15
512x512x512	1,34e+8	0,10	1,07	2,21	4,65	8,56
1024x1024x256	2,68e+8	0,22	0,44	0,85	1,69	3,42
1024x1024x512	5,37e+8	0,46	0,51	1,01	1,99	3,81
1024x1024x1024	1,07e+9	0,94	1,06	1,17	2,21	4,64
1024x1024x2048	2,15e+9	1,90	2,19	2,53	2,62	5,24

Table 2: Ratio between time of execution of the Fast Fourier Transform using the 2D Domain Decomposition, and the same time using standard Slab Decomposition (Standard FFTW algorithm). Red cells: Time of standard FFTW < Time of 2D decomposition. Yellow cells: Time of standard FFTW ~ Time of 2D decomposition. Green cells: Time of standard FFTW > Time of 2D decomposition. Blue cells: Time of standard FFTW >> Time of 2D decomposition. Please note that the greater is the number of cores, the greater are the performance of the new 2D Domain Decomposition algorithm.

Summarizing the results of the comparison of the two domain decomposition algoritms, we found that the 2D Domain Decomposition algorithm is particularly useful when the number of usable cores exceeds the size of the index involved in the standard parallel decomposition.

Starting from these results it is relatively simple to improve the performance of the FFTW library and enhancing its auto-tuning mechanism. We check if the number of cores exceeds the size of the index involved in parallel domain decomposition, and thus change the parallel domain decomposition algorithm used. A detailed description of the two algorithm used in this case to improve the performance of the auto-tuning mechanism of FFTW will be found on the related white paper.

These algorithms were used to create a library, that can significantly improves the performance of standard the FFTW library on massively parallel supercomputers.

This library was already tested on FERMI cluster, obtaining, as expected, a performance up to 63 times faster than standard FFTW libraries. Thus, the use of these algorithms can highly improve the performance of the FFTW library on modern massively parallel supercomputers.

The 2D Domain Decomposition algorithm could be extended to multi-dimensional systems (more than 3D systems), and in the next moths it will be the subject of further research at CINECA.

3.4 Auto-tuning 2D Stencil Application on Multi-core Parallel Machines

Supported by: Z.X. Hou, C. Perez (GENCI/INRIA)

White Paper: Z.X. Hou, C. Perez, "Auto-tuning 2D Stencil Application on Multi-core Parallel Machines", PRACE technical white paper.

With the increasing development of multi-core petascale or even exascale supercomputers, scalability is a one of the key issues for high performance computing. Currently, there are dozens or even hundreds of cores within one multi-core node. On multi-core clusters or supercomputers, how to get good performance when running high performance computing (HPC) applications is a major concern. There are many parameters to select, which are not obvious for end users. Therefore, an auto-tuning framework is beneficial to the execution of the application.

In this project, we propose and implement some auto-tuning strategies for the execution of stencil HPC applications on multi-core based petascale supercomputers (from PRACE), as well as multi-core clusters (from the French Grid'5000 experimental platform). The fundamental issue is how to make data partitioning and run the application on a number of nodes on a multi-core cluster or supercomputer with good performance. After the analysis of the specific application and probing of resources, running parameters are evaluated on the basis of tuning algorithms and training. Then, the application can be executed with optimized parameters.

A typical 2D Jacobi benchmark and a real NEMO (Nucleus for European Modeling of the Ocean) application were chosen as typical stencil HPC applications. As a first step, we present the experimental results for the Jacobi benchmark in the white paper. For the tuning parameters, we explore two levels: node level and system level. The main tuning strategies include data partitioning within a multi-core node, number of threads or processes within a multi-core node, data partitioning for many nodes, number of nodes in a multi-core cluster system.

There are 3 main policies for data partitioning, i.e. data partitioning by row, by column, or by both of row and column. For data partitioning by both of row and column, there are more choices such as the number of rows and columns on the basis of factorization. Within one multi-core node, when implementing the Jacobi benchmark with C/C++, because array data are stored in the memory by row, usually, it can bring a better performance with data partitioning by row. On a multi-core cluster system, to decrease the communication data between, the processes in different computing nodes, usually it brings a better performance with data partitioning by both row and column, especially, in the case of one core per node. The number for dividing rows (X) and the number for dividing columns (Y) should be as close as possible to the square root of Nn, the number of nodes.

With the increase of the number of cores on a chip, while memory bandwidth per socket may remain constant, memory per core may decrease. Therefore, how to choose the number of threads is a practical issue when running multi-threaded applications. Within a multi-core node, we propose an algorithm to predict the best number of threads for speedup based on the maximum number of threads without saturating memory bandwidth. At the same time, for a multi-core cluster system, we estimate the best number of nodes for speedup on the basis of minimizing the computation time and communication time.

On the basis of the possible optimization space, we provide a hierarchical auto-tuner that explores the optimization space for multi-core nodes of the experimental machines and for multi-core clusters at a reduced concurrency corresponding to the specific machines, e.g. 128 nodes on CURIE thin nodes. In the training step, the best parameters are determined with a small scale of computing resources on a specific machine. Then, a runtime script (not source

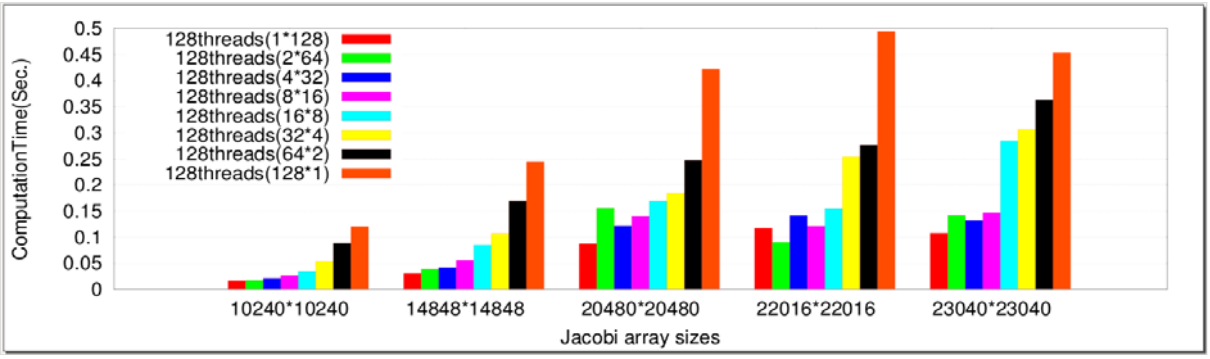


Figure 5: Computation time of Jacobi benchmark with different data partitioning policies on one node of Curie xlarge nodes

code) will be automatically generated for the stencil application on a specific machine. The input parameters of the specific application and a specific multi-core cluster system include Sx (row size of the array), Sy (column size of the array), Nc (the maximum number of cores), Bm (memory bandwidth), Bm1 (the required memory bandwidth with a single thread), PeakFlop/s of one core, Nt (number of total available nodes), Bn (network bandwidth), K1 (the computation time of one cell of the array normalized with respect to the number of used cores). The output parameters include Tbs (the best number of threads for speedup), Nbs (the best number of nodes for speedup), X (the number for dividing rows), Y (the number for dividing columns) for data partitioning.

The main experimental platforms are the Curie supercomputer and some clusters from Grid'5000. The number of cores varies from 2 to 128 cores per node. We designed to carry out the following sets of experiments for the study: (1) Stream benchmark within a multi-core node. (2) Jacobi benchmark application: different data partitioning policies within a multi-core node; different data partitioning policies for many nodes; we run the Jacobi application with different configurations, e.g. increasing the number of threads within a multi-core node (1 node, some threads or processes), increasing the number of nodes within a supercomputer (some nodes and 1 process per node), the same number of nodes with increasing number of processes on each node (a given number of nodes, some processes per node), and different number of nodes with increasing number of processes per node (some nodes and some processes per node). Figure 5 is an example of the measured performance results with different data partitioning policies within one node on a 128-core node of Curie xlarge nodes. In the figure, (X*Y) means the data partitioning method, X is the number for dividing rows, Y is the number for dividing columns of the Jacobi array. Just as expected, partitioning by row usually brings a better performance for the codes implemented in C++, in particular on a large multi-core node.

Figure 6 shows the computation time and speedup of the Jacobi benchmark with different number of nodes and increasing number of processes (some nodes and some processes per node). Given the number of nodes, the performance usually improves with increasing number of processes on each node. It means that the speedup usually increases with the total number of processes. In some cases, there is an exception when using the maximum number of cores per node. There may be a performance decrease compared with less number of processes per node.

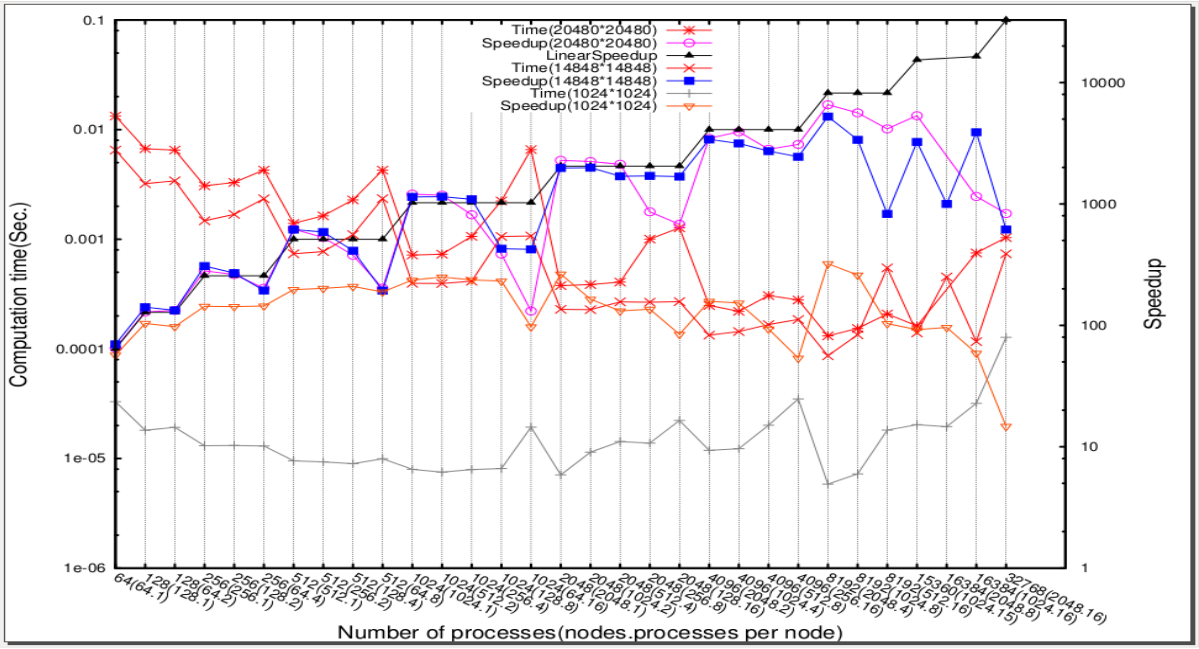


Figure 6: Performance of the Jacobi benchmark for 3 different data sizes with different number of nodes and increasing number of processes on Curie thin nodes (N.P: N is the number nodes and P is the number of processes per node).

For the same number of total processes, using different number of nodes with increasing number of processes per node (given the total number of processes, some nodes and some processes per node) shows that the performance usually decreases with the increasing number of processes on each node. It means that usually one process per node and communication through the inter-nodes network can bring a good performance. It is mainly due to the intra-node communication overhead within one multi-core node. During the experiments presented in Figure 6, we obtained the best speedup when using 8192 processes with 2048 nodes and 4 processes per node.

Currently, on a multi-core cluster or supercomputer, one may choose a number of nodes according to the problem size of the application and the available number of nodes on a given supercomputer. In addition, there may be a constraint for the number of nodes on some machines. To obtain the best performance, all of the parameters need to be tuned. On the Curie thin nodes (with a large number of nodes), we compared the predicted parameters for the best performance with the measured parameters for the best performance. In this case, the performance impacts within a multi-core node were up to 3.1%. The performance impacts on the multi-core cluster were comparatively larger than those within a multi-core node.

3.5 Generating integral graphs using PRACE Research Infrastructure

Supported by: Krzysztof T. Zwierzyński (PSNC)

White Paper: Krzysztof T. Zwierzyński (PSNC), "Generating integral graphs using PRACE Research Infrastructure**",** PRACE technical white paper.

A simple graph $G = G(V, E)$ is called *integral* if all eigenvalues $Sp(G) = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ of its adjacency matrix are integral. Every complete graph of order $n = |V|$ is integral with the spectrum $Sp(K_n) = \{n - 1, (-1)^{(n-1)}\}$. If the maximum degree Δ of graph G is bounded, then the class of connected integral graphs with this bound is finite. For $\Delta = 1$ there is only one such a graph K_2. For $\Delta = 2$ the only such graphs are: $K_3 = C_3$, C_4 and C_6 . The number $c_1(n)$ of connected integral graphs ($1 \leq n \leq 12$) are shown in Table 3. Integral graphs can have possible applications in quantum physics as a model of perfect state transfer (PST). It has been proven that only circulant graphs which are integral can have a property PST (i.e. graphs: K_2, C_4 and

C_6). However, the question which other classes of graphs can have this property still remains open. Good candidates for this class are integral graphs.

There is the ***geng*** algorithm written by B.D. McKay [10] which solves the problem of generation of non-isomorphic graphs of a given order *n*. The process of generation can be split into many (but limited by some constant) independent subtasks, and run parallel. This gives an opportunity to reduce the computation time by the factor of nodes *m*. However, there are two main disadvantages of this solution. The first is that a direct partition into *m* parts gives us a different runtime of ***geng*** for each subtask: it leads to a bad load balancing between nodes. The second is that each partition introduces some overhead connected with use of a queuing system and input and output maintenance. To improve the load balancing we can increase the number of subtask, but this also introduces some extra overhead. It is non-trivial to find the compromise in the number of subtask due to the different time of its executions.

n	1	2	3	4	5	6	7	8	9	10	11	12	Sum
$c_1(n)$	1	1	1	2	3	6	7	22	24	83	113	325	588

Table 3: The number of connected integral graphs c1(n), 1 ≤ n ≤ 12

In the case of dealing with the generation of combinatorial objects (i.e. graphs of some order *n*) and their selection due to some criterion for each algorithm, there is always a limit to the applicability. This is particularly evident in the case of problems for which the size of the search space grows exponentially with *n*. Then, even if we have an algorithm that solves the problem in an acceptable time for the selected *n*, the solution of the problem for *n* greater only by one may require additional treatments in programming, computing and storage resources. Generating connected integral graphs belongs to such class a problems, and size of combinatorial space is growing as 2^(binomial{*n*}{2}) / *n*!.

This is can be done in an acceptable time for the calculation of *n* is highly dependent on the method of combinatorial search space and its size. If the method let us divides the space into separate subclass the appropriate calculation speed can be achieved by using a sufficiently large number of computing machines (for example, if the number of nodes is equal to number of subclasses). Reaching this acceleration depends on the overhead associated with the time of distribution of tasks to compute nodes, the time required to collect the partial results, and whether the duration of the calculations for the subclasses is relatively homogeneous.

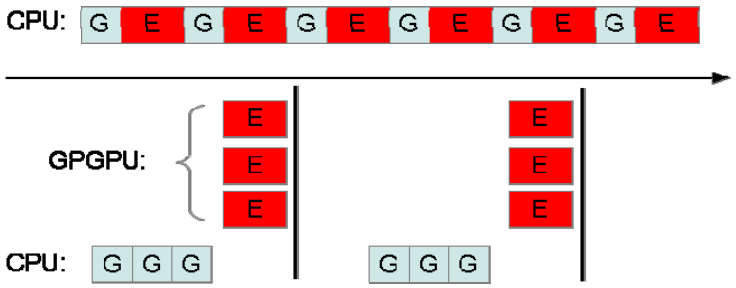


Figure 7: The time of graph generation (G) and eigenvalue calculation and sieving (E) in sequential and in parallel.

The first step of calculation is getting statistical information i.e. minimal, maximal, and average time of execution. For the algorithm ***geng*** also the number of output graphs per subtask can be important. If we predict that computation can take many hours then we can devise the whole process of generation, run it into parts divided into different hosts. When we extend the problem to finding some graphs with given properties (i.e. with integral spectrum) it can be done in two ways: i) generate graphs, save them to file, and then make some

calculations on this data; or ii) checking properties before saving graphs. Adding sieving directly into the generation algorithm we increase the time per graph, but if many of them are eliminated, then we improve the time necessary for saving data.

Using OpenCL technique. Important for the generation is a process for storing information about a graph. The method of calculation of the spectrum of graphs is not only restricted to zero-one, but in general, to real symmetric matrices. This requires a transformation from a binary to a more memory consuming representation. Due to the fact that the time of data transfer from the CPU to the GPGPU is significant, it was decided to send it in binary format. Then the decoding graph is implemented in the GPGPU. This solution speeds up the calculation.

Speedup. The tests were performed on a cluster PC based on AMD Interlagos architecture. Each node of the cluster contains two CPU (24 cores): AMD Opteron™ 6234 2.4 GHz and two GPGPU: NVIDIA TeslaM2050. The time of generation of all graphs of order $n = 10$ on the single node takes only 12 s, but if we add checking and if they are integral it takes 4 minutes (sequential algorithm that use only CPU was used). Using one node and the OpenCL technique for eigenvalue calculation the time of generation and sieving takes 16 seconds, thus we get a 15x speedup. There are 113 integral connected graphs of order $n = 11$. The size of the search space: 1 006 700 565 connected graphs. The calculation (for $n = 11$) on a single node with GPGPU support take 26 minutes. For $n = 12$ only sampling and estimation of the total computation time on a single node have been performed. For $n = 12$ the time of generation on the single node takes approximately 71 hours.

Overhead. The main idea is use GPGPU in such a way, that they calculate spectra of many small matrices in the same time (see Figure 7). It is the simplest way to reduce time complexity. Also small number of changes in the code of the numerical method is needed. Nevertheless, this method change the way of processing graphs - we need first to collect some graphs in memory and then use the sieving method for this group. The size of the group can be chosen arbitrary. When we would use the OpenCL technique for calculation we need to prepare the code of the kernel, compile it, and distribute to all devices (GPGPU or CPU). There are two possible paths of using the kernel. The most flexible case: we have a kernel code written in C99, we compile it to PTX format before the execution binaries are built for every device. Such a solution gives us the worst execution time, because we need to load the compiler into system and run it, and it is a very resource consuming operation. One way to avoid this is using a precompiled version of the kernel source. However, even in that case the OpenCL compiler is loaded into memory and used for PTX input to produce binaries. The other case is that we already have binaries for a given GPGPU device, and we just load it. In such a case using compilation is not needed. The only overhead here is connected with I/O operation when binaries are stored into files.

Conclusions. As a conclusion, the experiments indicate that it is necessary to support the design of algorithms that allow for a self-steering algorithm and its adaptation to the hardware capabilities. As far as possible, the algorithm should be designed as flow control, which can be isolated in blocks, which can be replaced by equivalent codes, in particular the OpenCL kernel code.

In particular, the programmer should specify the method of distribution of tasks and aggregation of results. If an auto-tunnig strategy is also possible try and if the calculations are carried out for too long, replace the calculation of a concurrent or distributed solution if necessary. Mainly this is a situation where the parameter n is growing then even small improvements in the code can lead to significant savings in the time of combinatorial structures generation.

We should also keep in mind the purpose of the calculation, sometimes it is not necessary to find all solutions, and getting even some solutions can be interesting. For integral graphs the search space can also be split into bipartite graphs and non-bipartite graph, regular and non-regular. Those whose automorphism group is one and those that has a matrix representation with the biaxial symmetry. To obtain the graph of a certain class also some graph operations can be used. There are also known families of integral graphs. The division of the search space and the information about already known objects can be also uses to reduce the time. If we already know some integral graphs we can find similar objects both for the same *n* and different, combining, for example, smaller graphs or using a local exchange of vertices. There it is also possible to use in the parallel some search heuristic methods (i.e. evolutionary technique) to obtain almost all or some sample set of integral graphs of a given order. On the other hand, an excessive adjustment of the code leads to the fact that it is difficult to use for other problems.

This problem of generation connected integral graphs can be considered as a benchmark for testing the computational power of a grid. The number of solutions is not so big, thus there is no problem with the output data storage. For $n = 13$ and the current stage of technology the time of generation on a single computer node can take approximately 2 years. For that reason only well tested and tuned methods can complete this task in parallel in a reasonable time even for future exascale systems.

3.6 Towards runtime-clustering and improved implementations of collective operations in MPI

Supported by: Chandan Basu, Johan Raber (SNIC-Liu), Michael Schliephake (SNIC-KTH)

Whitepaper: Chandan Basu, Johan Raber, "Towards Runtime-Clustering and improved Implementations of collective Operations in MPI"", PRACE technical white paper.

Overview. Further performance improvements of parallel simulation applications will not be reached by simply scaling today's simulation algorithms and system software. Rather, they need qualitatively different approaches and new developments that address and reduce the typically non-linearly increasing complexity of algorithms with the use of increasing processor counts.

We presented first results of an activity aimed at improving the performance of collective communication operations of relevance to simulation applications in the white paper "*Towards runtime-clustering and improved implementations of collective operations in MPI*". We focus here on further improvement of applications through more efficient implementations of collective communication operations for large-scale program executions. The development begun in the EU FP7 projects PRACE-1IP and CRESTA and it is continued in PRACE-2IP with further developments, benchmarks on PRACE systems and application of the result in community codes.

We follow two related lines in the development. One line is to use algorithms based on point-to-point communication operations as the starting point for a topology-optimized implementation using recently established capabilities of multi- and many-core processors as well as interconnect technology. Another line aims at improved use of memory hierarchies that influence the performance of communication operations significantly. Many present day supercomputers have multiple level of interconnect bandwidth and latency between processing cores. Message bandwidth and latency between processes running on cores within same numa-zone is generally best followed by cores in the same node, cores connected by same switch, cores connected by higher level switches etc. The message bandwidth and latency hierarchy can have impact on the performance of MPI jobs. Normally, MPI programs do not distinguish between ranks in the same node and ranks across different nodes. This may

lead to over-or under-utilization of weak or strong interconnects respectively. In case of MPI collective operations it is however possible to optimally use the network bandwidth by carefully writing the collective algorithm. In PRACE-1IP we have developed a routine called MPI_Alltoallv_tuned which does all-to-all (vector) operation utilizing bandwidth and latency hierarchy. This routine was tried in the PRACE euroben synthetic benchmark routine called mod2f. The mod2f calculates fast Fourier transform (fft). The standard implementation uses MPI_Alltoallv to take transpose of a matrix. In the modified implementation MPI_Alltoallv is replaced with MPI_Alltoallv_tuned. The performance and scaling of the modified mod2f is greatly improved. In the current work we continue to extend the work. We try to improve the MPI_Alltoallv_tuned interface and we want to try it for more realistic applications.

The results will directly be beneficial to applications and can be used as a building block in a runtime-system aiming at improving application load balance dynamically.

Development activities about topology-aware all-to-all communication. An MPI_Alltoallv_tuned routine has earlier been developed in order to utilize the intra node bandwidth more efficiently. It distinguishes between ranks in the same node and ranks across different nodes. In the present work we are trying to make the interface of MPI_Alltoallv_tuned the same as MPI_Alltoallv. This will make it much easier to use it in a real application. Necessary data structures and algorithms are described in the white paper.

Development activities in order to increase the parallelism in collective communication. The main MPI libraries must serve a broad spectrum of different application classes leavingroom for further optimization in the communication with shorter, latency-bound messages. The specific properties of recent multi-core processor designs, the hardware support for multithreaded programming, and the availability of more powerful network interface cards are used in implementations of collective communication operations in order to use a higher degree of parallelism within these functions. The first results showing performance improvements for short messages are described in more detail in the white paper.

Development activities for automatic network topology identification. The order-of-magnitude difference in bandwidth and latency between intra node memory accesses and inter-node message passing, even over the fastest available interconnect, is used to reveal rank-to-rank pair communications for an MPI based application as being either intra node or inter node. The first results show that the approach can be used indeed to detect a system-topology during the runtime of an application. Details of the used method as well as results from a PRACE system are presented in the white paper.

The possibilities in the context of PRACE 2-IP are used to improve algorithms and the usability of the functions. Developments in the topology identification aim at the improvement of the quality and speed of the identification process. The functions will be used in some communication-intensive community codes in order to demonstrate their benefits.

3.7 Energy-efficient Sparse Matrix Auto-tuning with CSX

Supported by: Jan Christian Meyer, Lasse Natvig (SIGMA), Vasileios Karakasis, Dimitris Siakavaras, and Konstantinos Nikas (GRNET)

Whitepaper: Jan Christian Meyer, Lasse Natvig (SIGMA), Vasileios Karakasis, Dimitris Siakavaras, and Konstantinos Nikas (GRNET), “Energy-efficient Sparse Matrix Auto-tuning with CSX”, PRACE technical white paper.

This whitepaper describes the programming techniques used to develop an auto-tuning compression scheme for sparse matrices with respect to accelerating matrix-vector multiplication and minimizing its energy footprint, as well as a method for extracting a power profile from a corresponding implementation of the conjugate gradient method. More specifically, we employ the CSX storage format in the execution of the CG iterative solution

algorithm (we are using the implementation supplied with the CSX software). CSX offers a mechanism for reducing the matrix preprocessing cost considerably by using statistical sampling of the input matrix; however, we do not employ this mechanism in this preliminary examination. CSX is a compact storage format for sparse matrices that is able to detect and encode in a single representation a diverse set of matrix substructures. A substructure is any regular one- or two-dimensional sequence of non-zero elements inside the sparse matrix. The number of all the possible substructures detected from CSX is indefinitely large and a priori unknown for a specific sparse matrix. For this reason, CSX employs runtime code generation in order to provide high-performance SpMV implementations adapted to the specificities of every matrix. Compared to other storage formats that exploit a single substructure type, e.g., the BCSR format that exploits only dense block substructures, CSX is able to achieve consistent high performance by successfully adapting to a great variety of sparse matrices, ranging from regular ones to those with a rather irregular structure. The supported substructures are horizontal, vertical, diagonal, anti-diagonal, and two-dimensional (row- or column-aligned).

The CG execution using the CSX format proceeds in five phases:

- Matrix loading (either from disk or from CSR)
- Detection of substructures
- Selection and encoding of substructures
- Code generation
- Iteration, bounded by SpMV performance

All steps except for the iteration are performed once. Iteration to solution for a system requires a number of repetitions which is proportional to the rank of the system, potentially amortizing the cost of one-time preprocessing steps by a reduction in the repeated cost of iterations.

All tests in this whitepaper are performed on a quad-core machine featuring a 4-core processor with the Sandy Bridge architecture, admitting explicit control of clock frequency, and energy profiling using RAPL MSR features, as described below. Intel R microarchitectures starting with Sandy Bridge expose Model Specific Registers (MSRs) which provide a running estimate of the energy consumption of the processor. We use a function library developed at NTNU in order to read these registers for continuous intervals of approximately 0.1 seconds. Interval timing is performed using the Linux kernel real-time clock through the *setitimer* system call, which produces a periodic call to a registered signal handler. The CG solver of CSX was augmented with a signal handler function to trap this event and store the running energy status to log files, recording time using the standard *gettimeofday* system call. The use of the latter clock is necessary because it displays a slight drift from the kernel timers, but it is used by the energy counter library in order to convert MSR-internal register values to millijoules. The magnitude of this drift was determined to remain below 0.01s; as this accounts for only 10% of the energy timer's interval, our further discussion is restricted to the time series recorded using *gettimeofday()*. Our periodic recording of energy use over intervals provides a discretized, running estimate of CPU power, which we approximate with a simple linear model, dividing the energy per interval by the recorded time step. This permits us to extract power/time graphs throughout entire CSX runs, which can be related to its stages of execution. Energy consumption is then estimated by numerical integration of this series, using the trapezoid method, in accordance with our linear approximation of energy variations.

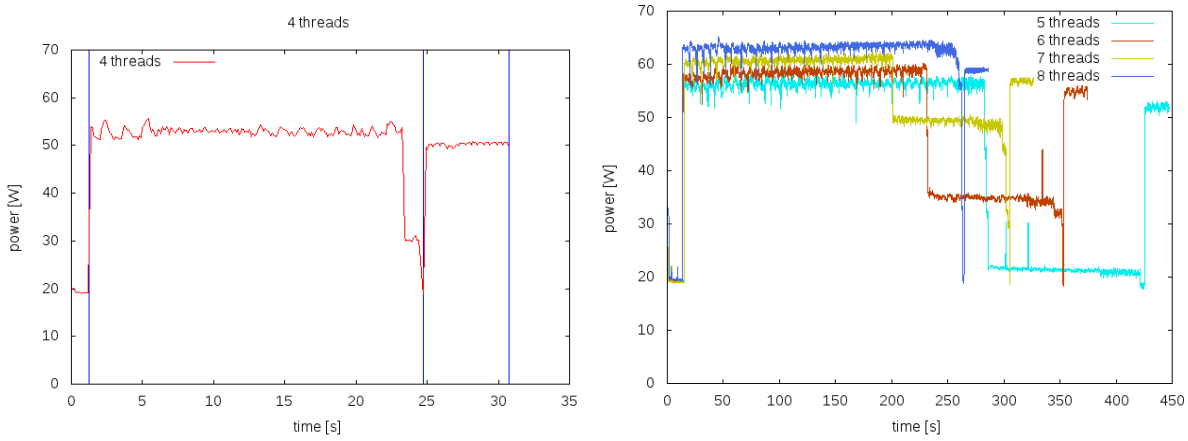


Figure 8: (a) CSX execution phases from a power consumption perspective. (b) CSX execution power footprint with an increasing number of threads (HyperThreading enabled).

Figure 8(a) provides a detailed view of the CSX/CG execution phases as recorded from our energy measurement framework. The vertical, blue lines denote the transitions from stage 1 to 2, from stage 4 to 5, and the program termination. The figure does not discriminate between the intermediate preprocessing stages 2, 3, and 4, as their aggregate cost is of primary interest. The preprocessing stage of CSX compression is quite amenable to parallelization, providing time improvements up to 7 threads, utilizing Hyperthreads at minor additional power, while CSR shows no significant benefit from Hyperthreading. Our results indicate that a constant iteration count of 1024 for stage 5 is sufficient to bring power to a steady state: it makes it feasible to project that energy effects observed at this scale are representative of the sustained power consumption if iterations were continued until convergence. This can enable automatic selection of the energy-optimal configuration from collecting small sample run data prior to execution, as the iteration counts for very large systems can require them to run for significantly longer than these tests. Figure 8(b) shows that the net cost of the preprocessing stage reduces with parallelism and at the same time the efficiency of CSX is growing with increasing thread count. A closer inspection of the results reveals some non-linearities in the relationship between the increases in per-iteration energy vs. the cost of the CSX auto-tuning: the first two Hyperthreads mark an area where the per-iteration cost is slightly lowered, at an increased preprocessing cost.

As witnessed by the results in this whitepaper, our instrumentation is capable of capturing and estimating the energy requirements for a given input set in a fraction of the system's time to solution, suggesting that it can form the basis of an automatic tuning mechanism to select optimal configurations at solver startup time by sampling available alternatives

3.8 Power Instrumentation of Task-based Applications using Model-specific Registers on the Sandy Bridge Architecture

Supported by: Jan Christian Meyer and Lasse Natvig (SIGMA)

Whitepaper: Jan Christian Meyer and Lasse Natvig (SIGMA), "Power instrumentation of task-based applications using model-specific registers on the Sandy Bridge architecture", PRACE technical white paper.

The whitepaper describes the technical side of a research work into the energy-efficiency tradeoffs of task-based execution with vectorization, applying recently available model-specific registers for energy instrumentation. Access to these registers is subject to constraints of processor architecture and operating system design, presenting challenges to leveraging them within a user-space application program. The challenges are presented and a method to address them in an experimental setting is introduced. This method carries restrictions on its applicability, which implies that research methods for applying it must be adapted

accordingly, and the restrictions are briefly surveyed. The described method of instrumentation has already been applied in a research context, and a subset of published results are briefly presented as an illustration of the method's potential. Finally, directions for further work on surmounting the methodological restrictions are suggested.

The model specific registers (MSRs) used in this paper are introduced with the Intel Sandy Bridge architecture, and reflect a running estimate of the amount of energy expended by the processor package, measured to a resolution of microJoules. The primary challenge when applying them in user-space software is that the hardware instruction to read their value requires a privileged mode of execution, effectively requiring it to be executed as part of the operating system. Linux exposes the value of such registers in a virtual device file, which is, by default, restricted to access by a user with administrative privileges. This mechanism is used in the described method.

A secondary challenge to the use of the energy counter registers is that their range is finite before values wrap around, which means that the resolution of the sampling is proportional to a limit on timing intervals that can validly be recorded. The third and final issue discussed, is that the energy estimate is restricted to the package consumption of a single chip, which means that the energy use of a program can only be approximated in terms of that portion of its total consumption accounted for by the processor. These two challenges are addressed by adaptations to the research method rather than technical solutions, by restricting the input cases of the programs used to study the impact of vectorization and thread-level parallelism to sizes where the majority of the execution time is spent addressing an amount of memory which fits the last-level cache. Restricting the size of the input cases also admits a reasonably short execution time, making the instrumentation sufficient to record end-to-end energy consumption for entire program runs. The choice of benchmarks used is guided partly by this size restriction, and partly by the availability and construction of task-based versions of their respective sources. The computational kernels used for testing are the FFTW implementation of the fast Fourier transform, the BlackScholes benchmark from the NAS parallel benchmark suite, and a tiled, task-based matrix multiplication kernel compliant with the level-3 BLAS specification.

Results show an interesting interplay between the choice of parallelization strategy, energy consumption, and validate the effectiveness of restricting the investigation to on-chip problem sizes, through the prediction that all approaches will level in power consumption as the input size grows sufficient to make for bandwidth-bound execution. The clearest example is in the BlackScholes results, reproduced in Figure 9; further evidences are presented in both the whitepaper and the corresponding conference paper [11].

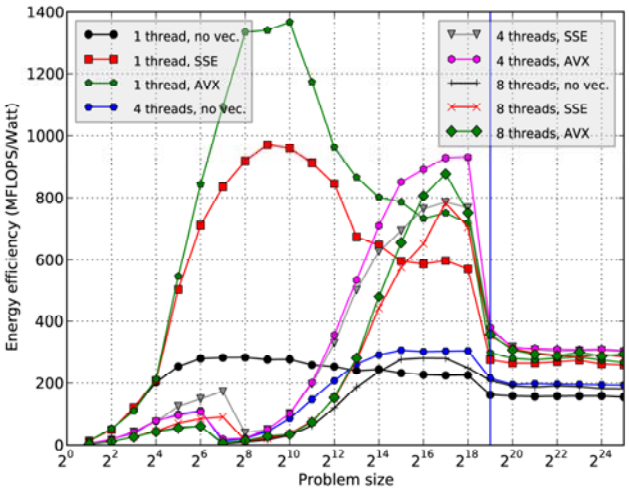


Figure 9: Energy efficiency of BlackScholes for with various kernel threading/vectorization balances.

3.9 An Auction Based SLURM Scheduler for Heterogeneous Supercomputers and its Comparative Performance Study

Supported by: Seren Soner, Can Ozturan (Bogazici University)

Whitepaper: Seren Soner, Can Ozturan, "An Auction Based SLURM Scheduler for Heterogeneous Supercomputers and its Comparative Performance Study", PRACE technical white paper.

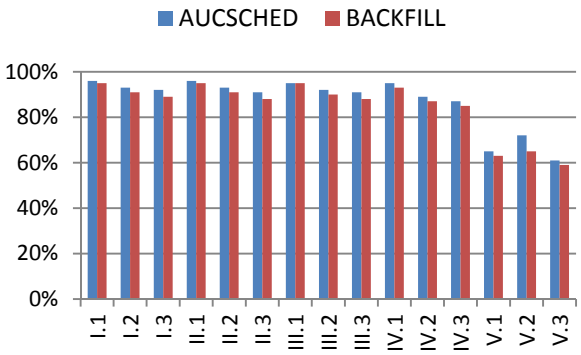
SLURM is a GPL licensed open resource management system that is used on many TOP500 supercomputers. It is estimated by SLURM developers that as many as 30% of the supercomputers in the November 2012 TOP500 list are using SLURM. In particular, it is stated that one third of the 15 most powerful systems in this list use SLURM. These are: No. 2 Sequoia at LLNL, No. 7 Stampede at TACC, No. 8 Tianhe-1A in China, No. 11 Curie at the CEA in France and No. 15 Helios at Japan's International Fusion Energy Research Centre.

This work contributes a heterogeneous CPU-GPU scheduler plug-in, called AUCSCHED, for SLURM that implements an auction based algorithm. This plug-in which builds on the earlier IPSCHED plug-in is available at http://code.google.com/p/slurm-ipsched/. If a job is allocated some resources, then depending on what resources it has been assigned, the job itself may perform tuning to achieve better performance on these resources, for example, by using topologically aware communication. A complementary tuning can also be performed by the scheduler of a resource manager by helping a job to achieve better runtime performance by placing it on resources that will lead to faster execution. Such may be the case, for example, if a communication intensive job is allocated nodes that are in close vicinity to each other. Since a scheduler has access to information about available resources and is the authority that makes allocation decisions, it can enumerate and consider alternative resource allocations for each job. This work considers this complementary approach that aims to tune allocations of jobs at the scheduling level. The work is carried out within the context of linear mapping of jobs to one-dimensional array of nodes. This is the default mode of resource selection in SLURM.

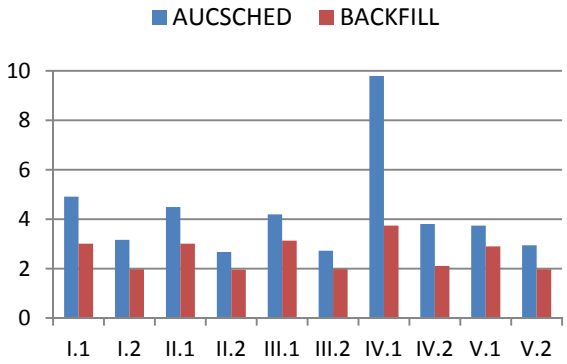
Workload Type	Jobs with core request only	Jobs with node request (4 or 8 cores/node)	Jobs with 1 GPU/node request (1 or 2 cores/GPU)	Jobs with 2 GPU/node request (1 or 2 cores/GPU)
I	100%	0%	0%	0%
II	0%	100%	0%	0%
III	50%	50%	0%	0%
IV	40%	40%	20%	0%
V	33.3%	33.3%	16.6%	16.6%

Workload versions 1, 2 and 3 stand for the fraction of contiguous jobs; 0%, 50% and 100%, respectively (e.g. V.2 stands for workload type V, contiguous job fraction 50%)

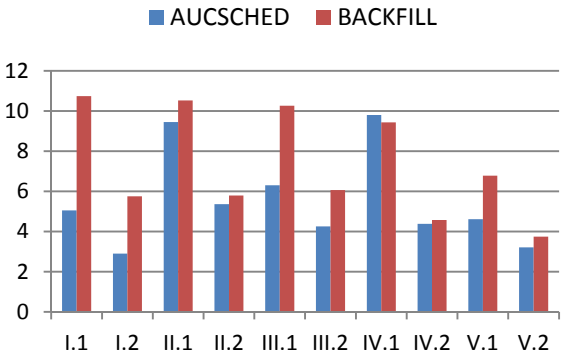
(a)



(b)



(c)



(d)

Figure 10: Comparison of AUCSCHED and SLURM's own BACKFILL plug-ins. Workload descriptions (a), utilizations (b), fragmentation (c) and spread (d).

The implemented auction methodology can be summarized as follows: The auction algorithm takes a window of jobs from the front of the job queue, generates multiple bids for available resources for each job, and solves an assignment problem that maximizes an objective function involving priorities of jobs. The assignment problem is formulated as an integer programming problem and solved using the CPLEX solver. To achieve a topologically aware mapping of jobs to processors, the bids generated include requests for contiguous allocations. Various CPU-GPU synthetic workloads are generated and realistic SLURM emulation tests are performed to compare the performance of the auction scheduler with that of SLURM's own backfill scheduler. Several tests have been performed evaluating 128, 256 and 1024 node clusters. Figure 10 presents the workloads used and the results obtained for the 1024 node system. From the results, the following can be observed:

- SLURM's own and the AUCSCHED plug-in produce high utilizations around 90% when workloads are made up of jobs requesting no more than 1 GPU per node.
- Generally, AUCSCHED has a few percentage points of better utilization over SLURM's backfill plug-in.
- Generally SLURM's backfill is leading to less fragmentation (number of contiguous blocks in an allocation) but AUCSCHED is leading to less spread (stretching of allocation on 1D array).
- When workloads contain jobs that request 2 GPUs per node (i.e. in workload V), it is observed that the system utilization drops drastically to 65-75% range both with AUCSCHED and SLURM's own plug-in. Note, however, that the definition of

theoretical runtime that is used for calculation of utilization is only a lower bound - it is not the optimal value. Computation of the optimal schedule and hence the optimal runtime value is an NP-hard problem. Hence, the following scenarios may be possible: *(i)* The algorithms in both plugins are working nicely but this low utilization may be close to the best that can be obtained due to more complex combinatorial nature of the problem, *(ii)* the algorithms in both plugins need to be improved in order to produce good solutions. Both cases, however, point to the need to further study of scheduling jobs that utilize multiple GPU cards on nodes.

This work formulated job scheduling process as an auction problem in which a window of jobs from the front of the job queue submit bids for the resources they request. Such an approach can help us achieve two main objectives: (1) Development of new scheduling heuristics and software for new heterogeneous supercomputer architectures; (2) Provision of scheduler support for applications that may provide alternative implementations or different resource requirements provided by the programmer/developer. The motivation for objective (1) comes because for many years schedulers were mainly used and optimized for core based systems. But with the recent emergence of heterogeneous systems with accelerator cards such as GPUs, the scheduling problem becomes more complex and more sophisticated. As a result new combinatorial optimization algorithms need to be developed to schedule applications which may use these accelerators. The fact that in the tests with 2 GPU cards per node, utilization values dropped drastically provides evidence for the case that scheduling of such systems need to be studied further. The motivation for objective (2) comes from the fact that mechanisms must be provided by the scheduler to the applications (that may provide alternative implementations and/or different resource requirements) to first express these alternatives and secondly to handle such alternative specifications during scheduling.

The current release of AUCSCHED implemented the internal bid mechanism. The next release will also include SLURM command line directives for enriching the expressiveness of alternative implementations and resource requests.

4 Summary

The main focus of WP12 is to perform research and development on four key areas for future multipetascale and exascale systems: Auto tuned and automatic techniques to be applied in parallel programming model runtimes, performance tools, file systems, and scalable numerical algorithms.

This deliverable reported the results of the first area, auto-tuning systems. Nine projects have produced results.

- Two projects on runtime code variant selection with some language support.
- Two projects on kernel specific auto-tuning algorithm (FFTW and 2D stencil).
- One project on auto-tuning collective operations in MPI.
- One project on improving Slurm scheduler with respect to heterogeneous systems.
- Two projects on auto-tuning with respect to energy consumption.

The first two projects focus on runtime task implementation selection. NUIG and CAPS Entreprise extended the language and runtime support for OpenHMPP and OpenACC to let a user help the system to reduce the space of solutions when choosing which task variant to execute. The work of BSC focused on task implementation selection at runtime to achieve higher performance for a particular hybrid architecture (SMP, GPUs). Morevoer, this feature enhances the programmability of applications and makes its maintenance easier. A new runtime scheduler has been proposed called *versioning scheduler*. Results show that this scheduler is able to out-perform existing schedulers used for the programming model OmpSs.

The next three projects intensively benchmark two important application kernels and the problem of generating integral graphs so as to derive auto-tuning algorithms. First, the work of CINECA focuses on an FFT library (FFTW). They tested the auto-tuning mechanism available in the FFTW library, proposed a simple algorithm which defines the domain partitioning as a function of the performance, and dealt with the 3D case by proposing a simple algorithm, that switches between standard slab decomposition to optimized 2D domain decomposition as a function of performance. Second, the work of GENCI dealt with 2D stencil kernel applications, and in particular with Jacobi. A large set of parameters was benchmarked on a wide variety of machines, ranging from clusters to supercomputers. Then, an auto-tuner algorithm computes the best number of threads, of nodes, and of data partitioning for maximizing speed up in function of a resource model that includes in particular memory and network bandwidth consideration. Third, PSNC work on auto-tuning the generation of integral graphs. They obtained some statistical information related to task runtime as well as output graphs per subtask from the algorithm *geng* that has been ported on a CPU+GPU (OpenCL) machine. A conclusion of the experiments is that it is necessary to support the design of algorithms that allow for self-steering algorithm and its adaptation to the hardware capabilities.

The next project, led by SNIC-Liu, aims at improving the performance of collective communication operations. Two related lines of research have been followed. One line is to use algorithms based on point-to-point communication operations as the starting point for a topology-optimized implementation using recently established capabilities of multi- and many-core processors as well as interconnect technology. A second line has aimed at improving the utilization of memory hierarchies that significantly influence the performance of communication operations. It is in particular applied to an FFT benchmark.

The next two projects focus on auto-tuning with respect to energy consideration. They face the challenge of measuring energy consumption and both addressed it with respect to the model specific registers (MSRs) of the Intel Sandy Bridge architecture. The work of SIGMA and GRNET has focused on sparse Matrix auto-tuning with CSX storage format. It produces instrumentation able of capturing and estimating the energy requirements for a given input set in a fraction of the system's time to solution, suggesting that it can form the basis of an automatic tuning mechanism to select optimal configurations at solver startup time by sampling available alternatives. SIGMA did also work on the technical side of the energy-efficiency tradeoffs of task-based execution with vectorization, applying recently available model-specific registers for energy instrumentation. Results show an interesting interplay between the choice of parallelization strategy, energy consumption, and validate the effectiveness of restricting the investigation to on-chip problem sizes, through the prediction that all approaches will level in power consumption as the input size grows sufficient to make for bandwidth-bound execution.

The last project, led by UHeM, has aimed at improving job scheduling algorithms of the SLURM batch scheduler. It proposes an auction based algorithm that makes use of an integer programming problem formulation to achieve a topologically aware mapping of jobs to processors. Various CPU-GPU synthetic workloads and realistic SLURM emulation tests have showed that the proposed algorithm performs usually better than the default SLURM algorithm.