SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

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

CAPACITIES

PRACE-2IP

PRACE Second Implementation Phase Project

Grant Agreement Number: RI-283493

D12.5
Summary of Novel Programming Techniques Results

Final

Version:	1.0
Author(s):	Jose Carlos Sancho, BSC; Christian Perez, INRIA; Cevdet Aykanat, R. Oguz Selvitopi, Bilkent University; Alberto Miranda, Ramon Nou, Toni Cortes, BSC; Eric Boyer, GENCI
Date:	22.8.2014

Project and Deliverable Information Sheet

PRACE Project	**Project Ref. №: RI-283493**	
	Project Title: Summary of Novel Programming Techniques Results	
	Project Web Site: http://www.prace-project.eu	
	Deliverable ID: < **D12.5** >	
	Deliverable Nature: <DOC_TYPE: Report >	
	Deliverable Level: PU	**Contractual Date of Delivery:** 31/08/2014
		Actual Date of Delivery: 31/08/2014
	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: Summary of Novel Programming Techniques Results**	
	ID: D12.5	
	Version: <1.0>	**Status:** ***Final***
	Available at: http://www.prace-project.eu	
	Software Tool: Microsoft Word 2007	
	File(s): D12.5.docx	
Authorship	**Written by:**	Jose Carlos Sancho, BSC; Christian Perez, INRIA; Cevdet Aykanat, , R. Oguz Selvitopi, Bilkent University; Alberto Miranda, Ramon Nou, Toni Cortes, BSC; Eric Boyer, GENCI
	Contributors:	
	Reviewed by:	Manuel Fiolhais, UC-LCA; Florian Berberich, FZJ
	Approved by:	MB/TB

Document Status Sheet

Version	**Date**	**Status**	**Comments**
0.1	09/08/2014	Draft	
1.0	21/08/2014	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

© 2014 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

List of Figures

List of Tables

References and Applicable Documents

[1] SLURM, "Slurm topology guide," [Online]. [Accessed 2014].

[2] S. S. a. C. Ozturan, "Integer Programming Based Heterogeneous CPU-GPU Cluster Schedulers for Slurm Resource Manager," 2014.

[3] S. S. a. I. K. C. Ozturan, "Extending slurm with support for gpu ranges," PRACE Whitepaper, 2013.

[4] "The Kepler Project," [Online].

[5] J. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers, 1993.

[6] Z. H. C. P. a. V. P. J. Bigot, "A low level component model easing performance portability of HPC applications," *Computing,* 2013.

[7] G. F. E. J. K. K. M. K. D. M. P. N. L. N. C. P. O. R. W. S. G. V. L. a. J. V. F. Desprez, "Supporting Experimental Computer Science," INRIA, 2012.

[8] J. B. a. C. Pérez, "High Performance Composition Operators in Component Models. In High Performance Computing: From Grids and Clouds to Exascale," vol. 20, 2011.

[9] G. C. a. R. M. B. Eagan, "Investigating Performance Benefits from OpenACC Kernel Directives," vol. 25, 2014.

[10] Y. G. M. G. T. W. a. D. P. S. Maleki, "An evaluation of vectorizing compilers," 2011.

[11] J. D. a. D. L. D. Callahan, "Vectorizing compilers: A test suite and results," Los Alamitos, CA, USA, 1988.

[12] L. N. Pouchet, "Polybench/c: the polyhedral benchmark suite," March 2012. [Online].

[13] G. F. F. A. E. B. M. O. a. O. T. J. Cavazos, "Rapidly selecting good compiler optimizations using performance counters," 2007.

[14] J. C. a. M. A. A. E. Park, "Using graph-based program characterization for predictive modeling," New York, NY, USA, 2012.

[15] R. M. A. L. M. G. M. B. A. D. M. Z. C. D. N. a. D. D. V. G. Fursin, "Collective Mind: Towards practical and collaborative auto-tuning. Scientific Programming," vol. 22, no. 3.

[16] G. C. A. S. E. C. M. G. H. H. C. N. S. B. M. S. L. M. a. F. B. R. Miceli, “AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications,” vol. 7782, 2013.

[17] J. F. R. O. C. S. L. Breiman, Classification and Regression Trees, Wadsworth and Brooks, 1984.

[18] C. Basu, “Improving MPI communication latency of Euroben kernels,” PRACE report.

[19] J. R. a. M. S. Chandan Basu, “Towards Runtime Clustering and improved Implementations of collective Operations in MPI,” PRACE report.

[20] “CPMD,” IBM, [Online]. Available: http://www.cpmd.org/.

[21] H. A. Council, “CPMD Performance Benchmark, Profiling and Tuning,” www.hpcadvisorycouncil.com/pdf/CPMD_Performance_Profiling_Intel_x5670.pdf.

[22] G. C. a. D. Balenovich, “Asset flow and momentum: deterministic and stochastic equations,” vol. 357, 1999.

[23] G. C. a. B. Ermentrout, “Numerical studies of differential equations related to theoretical financial markets,” vol. 4, 1991.

[24] J. C. a. E. A. A. Duran, “Evaluation of OpenMP task scheduling strategies. OpenMP in a new era of parallelism,” 2008.

[25] A. Duran, “Sensitivity analysis of asset flow differential equations and volatility comparison of two related variables,” vol. 30, 2009.

[26] A. Duran, “Stability analysis of asset flow differential equations,” vol. 24, no. 4, 2006.

[27] A. Duran, “Overreaction Behavior and Optimization Techniques in Mathematical Finance,” PhD thesis, University of Pittsburgh, Pittsburgh, PA, 2006.

[28] A. D. a. G. Caginalp, “Parameter optimization for differential equations in asset price forecasting,” vol. 23, 2008.

[29] C. Broyden, “The convergence of a class of double rank minimization algorithms, part 1,” vol. 6, 1970.

[30] J. Nocedal and S.J. Wright, Numerical Optimization, New York: Springer Series in Operations Research, Springer-Verlag, 2006.

[31] M. Bartholomew-Biggs, Nonlinear Optimization with Financial Applications, Boston, USA: KluwerAcademic Publishers, 2005.

[32] S. A. a. J. Born, Closed-End Fund Pricing: Theories and Evidence, Boston, MA, USA: Kluwer Academic Publishers, 2002.

[33] e. a. R. Teyssier, “The HYDRO code,” [Online]. Available: https://github.com/HydroBench/Hydro.

[34] R. Teyssier, “The RAMSES code,” [Online]. Available: https://bitbucket.org/rteyssie/ramses.

[35] S. K. Godunov, “A Difference Scheme for Numerical Solution of Discontinuous Solution of Hydrodynamic Equations,” Math. Sbornik,, 1959.

[36] G.-C. d. V. P. W. D. L. P-F. Lavallée, “Porting and optimizing HYDRO to new platforms and programming paradigms–lessons learnt,” PRACE whitepaper, 2013.

[37] “OpenMP 4.0 Specifications,” 2013. [Online]. Available: http://openmp.org/wp/openmp-specifications.

[38] B. U. a. C. Aykanat, “Partitioning sparse matrices for parallel preconditioned iterative methods,” vol. 29, no. 4, 2007.

[39] B. V. a. R. H. Bisseling, “A two-dimensional data distribution method for parallel sparse matrix-vector multiplication,” vol. 47, 2005.

[40] T. A. D. a. Y. Hu, “The university of florida sparse matrix collection,” vol. 11, no. 1, 2011.

[41] U. C. a. C. Aykanat, “Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication,” vol. 10, 1999.

[42] M. O. C. A. O. Selvitopi, “A novel method for scaling iterative solvers: Avoiding latency overhead of parallel sparse-matrix vector multiplies,” vol. 99, 2014.

[43] “CP2K homepage,” [Online]. Available: http://www.cp2k.org .

[44] A. C. X. G. P. K. I. Bethune, ““Million Atom KS-DFT with CP2K,” PRACE whitepaper.

[45] I. Bethune, “Improving the scalability of CP2K on multi-core systems. A dCSE Project,” 2010.

[46] M. U. M. G. A. Kwiecień, “Enabling the CP2K Application for Exascale Computing with Accelerators using OpenACC and OpenCL,” PRACE whitepaper, 2014.

[47] J. V. V. W. J. H. Urban Borštnik, “Sparse Matrix Multiplication: The Distributed Block-Compressed Sparse Row Library,” 2014.

[48] I. B. Fiona Reid, “Evaluating CP2K on Exascale Hardware: Intel Xeon Phi,” PRACE whitepaper, 2014.

[49] [Online]. Available: http://www.khronos.org/opencl/.

[50] [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[51] [Online]. Available: http://www.pgroup.com/resources/accel.htm.

[52] [Online]. Available: http://www.pgroup.com/products/pgdbg.htm.

[53] J. B. a. I. B. N. Brown, “Solving Large Sparse Linear Systems using Asynchronous Multisplitting,” PRACE White , 2013.

[54] L. Kernel, “posix_fadvise man page,” [Online]. [Accessed June 2014].

[55] S. C. a. K. E. VanDeBogart, “Reducing seek overhead with application-directed prefetching,” 2009.

[56] M. Folk, G. Heber, Q. Koziol, E. Pourmal and D. Robinson, “An overview of the HDF5 technology suite and its applications.,” in *Proceedings of the EDBT/ICDT Workshop on Array Databases*, 2011.

[57] R. Nou, J. Giralt and T. Cortes, “Automatic I/O scheduler selection through online workload analysis,” in *9th International Conference on Autonomic & Trusted Computing* , Fukuoka, Japan, 2012.

[58] FORTH, IBM, INTEL, BSC, UPM and Neurocom, “IOLanes EU Project,” [Online]. Available: www.iolanes.eu. [Accessed 2014].

[59] D. Stainton, “linux-ftools - fincore,” [Online]. Available: https://code.google.com/p/linux-ftools/.

[60] S. t. guide. [Online]. [Accessed 2014].

[61] L. E. S. N. S., “MoML.A Modeling Markup Language in XML. Version 0.4,” University of California at Berkeley, 2000.

[62] R. C. Team, R Foundation for Statistical Computing, [Online]. Available: http://www.R-project.org.

[63] C. Broyden, “The convergence of a class of double rank minimization algorithms, part 2,” vol. 6, 1970.

[64] B. K. a. S. Lin, “An efficient heuristic procedure for partitioning graphs,” vol. 49, 1970.

List of Acronyms and Abbreviations

AAA	Authorization, Authentication, Accounting.
ACF	Advanced Computing Facility
ADP	Average Dissipated Power
AFDE	Asset Flow Differential Equations
AISBL	Association International Sans But Lucratif (legal form of the PRACE-RI)
AMD	Advanced Micro Devices
AMFT	Advanced Multilevel Fault Tolerance
APGAS	Asynchronous PGAS (language)
API	Application Programming Interface
APML	Advanced Platform Management Link (AMD)
ASIC	Application-Specific Integrated Circuit
ATI	Array Technologies Incorporated (AMD)
BAdW	Bayerischen Akademie der Wissenschaften (Germany)
BCO	Benchmark Code Owner
BLAS	Basic Linear Algebra Subprograms
BSC	Barcelona Supercomputing Center (Spain)
CAF	Co-Array Fortran
CAL	Compute Abstraction Layer
CART	Classification and Regression Tree models
CCE	Cray Compiler Environment
ccNUMA	cache coherent NUMA
CEA	Commissariat à l’Energie Atomique (represented in PRACE by GENCI, France)
CG	Conjugate Gradient
CGS	Classical Gram-Schmidt
CGSr	Classical Gram-Schmidt with re-orthogonalisation
CINECA	Consorzio Interuniversitario, the largest Italian computing centre (Italy)
CINES	Centre Informatique National de l’Enseignement Supérieur (represented in PRACE by GENCI, France)
CLE	Cray Linux Environment
CORBA	Common Object Request Broker Architecture
CP	Coarse Problem
CPU	Central Processing Unit
CPMD	Car–Parrinello Molecular Dynamics
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)
CUDA	Compute Unified Device Architecture (NVIDIA)
DARPA	Defense Advanced Research Projects Agency
DDN	DataDirect Networks
DDR	Double Data Rate
DEISA	Distributed European Infrastructure for Supercomputing Applications. EU project by leading national HPC centres.
DGEMM	Double precision General Matrix Multiply
DIMM	Dual Inline Memory Module
DMA	Direct Memory Access

DNA	DeoxyriboNucleic Acid
DP	Double Precision, usually 64-bit floating point numbers
DRAM	Dynamic Random Access memory
DSL	Data Description Language
DT	Decision Tree
EC	European Community
EESI	European Exascale Software Initiative
EoI	Expression of Interest
EP	Efficient Performance, e.g., Nehalem-EP (Intel)
EPCC	Edinburg Parallel Computing Centre (represented in PRACE by EPSRC, United Kingdom)
EPSRC	The Engineering and Physical Sciences Research Council (United Kingdom)
eQPACE	extended QPACE, name of the FZJ WP8 prototype
ETHZ	Eidgenössische Technische Hochschule Zuerich, ETH Zurich (Switzerland)
EUABS	European Unified Applications Benchmark Suite
ESFRI	European Strategy Forum on Research Infrastructures; created roadmap for pan-European Research Infrastructure.
EX	Expandable, e.g., Nehalem-EX (Intel)
FC	Fiber Channel
FFT	Fast Fourier Transform
FFTW	Fastest Fourier Transform in the West
FETI	Finite Element Tearing and Interconnect
FEM	Finite Element Method
FHPCA	FPGA HPC Alliance
FP	Floating-Point
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FTI	Fault Tolerance Interface
FZJ	Forschungszentrum Jülich (Germany)
GASNet	Global Address Space Networking
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)
GDDR	Graphic Double Data Rate memory
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 Calcul Intensif (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
GPGPU	General Purpose GPU
GPU	Graphic Processing Unit
GS	Gram-Schmidt
GWU	George Washington University, Washington, D.C. (USA)

HBA	Host Bus Adapter
HCA	Host Channel Adapter
HCE	Harwest Compiling Environment (Ylichron)
HDD	Hard Disk Drive
HE	High Efficiency
HET	High Performance Computing in Europe Taskforce. Taskforce by representatives from European HPC community to shape the European HPC Research Infrastructure. Produced the scientific case and valuable groundwork for the PRACE project.
HLCM	High Performance Composition Operators in Component Model
HMM	Hidden Markov Model
HMPP	Hybrid Multi-core Parallel Programming (CAPS enterprise)
HP	Hewlett-Packard
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
HT	HyperTransport channel (AMD)
HWA	HardWare accelerator
IB	InfiniBand
IBA	IB Architecture
IBM	Formerly known as International Business Machines
ICE	(SGI)
IDRIS	Institut du Développement et des Ressources en Informatique Scientifique (represented in PRACE by GENCI, France)
IEEE	Institute of Electrical and Electronic Engineers
IESP	International Exascale Project
IL	Intermediate Language
IMB	Intel MPI Benchmark
I/O	Input/Output
INRIA	French Institute for Research in Computer Science and Automation
IOR	Interleaved Or Random
IPMI	Intelligent Platform Management Interface
ISC	International Supercomputing Conference; European equivalent to the US based SC0x conference. Held annually in Germany.
IVP	Initial Value Problem
IWC	Inbound Write Controller
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)
LAD	Assembly Description File
LBE	Lattice Boltzmann Equation
LINPACK	Software library for Linear Algebra
LLNL	Laurence Livermore National Laboratory, Livermore, California (USA)
LQCD	Lattice QCD
LOOCV	Leave-one-out cross Validation
LRZ	Leibniz Supercomputing Centre (Garching, Germany)
LU	Lower Upper Decomposition
LS	Local Store memory (in a Cell processor)

MATMUL	Matrix Multiplication
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	MetaData 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
MIC	Many Integrated Core
MIF	Maximum Improvement Factor
MIPS	Originally Microprocessor without Interlocked Pipeline Stages; a RISC processor architecture developed by MIPS Technology
MKL	Math Kernel Library (Intel)
ML	Maximum Likelihood
MoML	Modeling Markup Language
Mop/s	Mega (= 10^6) operations per second (usually integer or logic operations)
MoU	Memorandum of Understanding.
MPI	Message Passing Interface
MPP	Massively Parallel Processing (or Processor)
MPT	Message Passing Toolkit
MRAM	Magnetoresistive RAM
MTAP	Multi-Threaded Array Processor (ClearSpead-Petapath)
MTTI	Mean Time to Interrupt
mxm	DP matrix-by-matrix multiplication mod2am of the EuroBen kernels
NAS	Network-Attached Storage
NCF	Netherlands Computing Facilities (Netherlands)
NDA	Non-Disclosure Agreement. Typically signed between vendors and customers working together on products prior to their general availability or announcement.
NoC	Network-on-a-Chip
NFS	Network File System
NIC	Network Interface Controller
NUMA	Non-Uniform Memory Access or Architecture
NYSE	New York Stock Exchange
OpenCL	Open Computing Language
OpenGL	Open Graphic Library
Open MP	Open Multi-Processing
OS	Operating System
OSS	Object Storage Server
OST	Object Storage Target
PaToH	Partitioning Tools for Hypergraph
PCIe	Peripheral Component Interconnect express, also PCI-Express
PCI-X	Peripheral Component Interconnect eXtended
PGAS	Partitioned Global Address Space
PGI	Portland Group, Inc.
PETSc	Portable, Extensible Toolkit for Scientific Computation
pNFS	Parallel Network File System
POSIX	Portable OS Interface for Unix
POMP	OpenMP Monitoring Interface
PPE	PowerPC Processor Element (in a Cell processor)

PRACE	Partnership for Advanced Computing in Europe; Project Acronym
PSNC	Poznan Supercomputing and Networking Centre (Poland)
P2P	Peer to Peer
QCD	Quantum Chromodynamics
QCDOC	Quantum Chromodynamics On a Chip
QDR	Quad Data Rate
QN	Quasi Newton
QP	Quadratic Programming
QPACE	QCD Parallel Computing on the Cell
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
RFA	Radio Frequency Ablation
RISC	Reduce Instruction Set Computer
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
SLURM	Simple Linux Utility for Resource Management
SM	Streaming Multiprocessor, also Subnet Manager
SMP	Symmetric MultiProcessing
SMT	Simultaneous Multithreading
SNIC	Swedish National Infrastructure for Computing (Sweden)
SP	Single Precision, usually 32-bit floating point numbers
SPE	Synergistic Processing Element (core of Cell processor)
SPH	Smoothed Particle Hydrodynamics
SPU	Synergistic Processor Unit (in each SPE)
SpMV	Sparse Matrix Vector Multiplication
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
STT	Spin-Torque-Transfer
SURFsara	Dutch national High Performance Computing & e-Science Support Center
SVM	Support Vector Machine
TARA	Traffic Aware Routing Algorithm
TB	Tera (= 240 ~ 1012) Bytes (= 8 bits), also TByte
TCO	Total Cost of Ownership. Includes the costs (personnel, power, cooling, maintenance, ...) in addition to the purchase cost of a system.
TDP	Thermal Design Power

TFlop/s	Tera (= 1012) Floating-point operations (usually in 64-bit, i.e. DP) per second, also TF/s
Tier-0	Denotes the apex of a conceptual pyramid of HPC systems. In this context the Supercomputing Research Infrastructure would host the Tier-0 systems; national or topical HPC centres would constitute Tier-1
TSVC	Leave-one-out Cross-Validation
UFM	Unified Fabric Manager (Voltaire)
UNICORE	Uniform Interface to Computing Resources. Grid software for seamless access to distributed resources.
UPC	Unified Parallel C
UV	Ultra Violet (SGI)
VALT	Vectorization and Loop Transformation
VHDL	VHSIC (Very-High Speed Integrated Circuit) Hardware Description Language
WCSS	Wrocław Centre for Networking and Supercomputing

Executive Summary

This Work Package performed research and development on the programmability of future multi-petascale and exascale systems. In particular, it was focused on main four areas, auto-tuned runtime environments, scalable numerical algorithms, fault tolerant tools, and file system optimization for exascale systems. A total of 16 research projects are reporting in this document covering multiple different techniques.

On auto-tuned runtimes environments a total of six projects were exploring different auto-tuning techniques at different granularities. For example, it goes from a coarse technique that auto-tunes the job scheduler down to the finest granularity when auto-tuning Intel vector instructions. Specifically, on SLURM job scheduler we explored the case capability to do topologically aware mappings of jobs on hierarchically interconnected systems. On the other hand, three projects were focused mostly on the programming language exploring one these projects the case of controlling dynamically the number of threads allocated to OpenMP parallel regions to decrease the overall wall-times; and the rest were focusing on improving of workflow executions through meta-model methods and the other using a component based approach to improve the 3D FFT. Additionally, on project was focused on compilation utilities to improve the vectorization of their codes. And finally, the last project was focused on improving the collective communications operations, specifically the *MPI_All_to_All*.

On scalable numerical algorithms, it was explored new algorithms or techniques that improve the scalability of existing algorithms. Eight projects were also focused on this research line. with several diverse parallel techniques such as utilization of GPU and MIC accelerators, message-passing paradigm and shared memory constructs. There are algorithmic approaches to increase the efficiency of widely used numerical algorithms (such as Conjugate Gradient (CG)) as well as adaptive parameter determination and utilization. The target applications and libraries include HYDRO, PETSc, CP2K and FETI.

On the other hand, on fault tolerant tools, it was shown the performance of the fault tolerant tool called FTI based on application-based checkpoint/restart. The overhead of using this tool could be very low at large scale. It was reporting less than 6% on HYDRO at 9,600 cores.

And finally, on file system optimization it was showed the performance of user hint guided I/O prefetching which is seen a scalable technique for accessing I/O at exascale systems. In fact, it was shown that anticipating user reads, with small information from the user, can produce great benefits in terms of performance. In addition, it was shown than the memory usage could be reduced with the use of an advanced filtering technique.

1 Introduction

This Work Package has been performed research and development on the programmability of future multipetascale and exascale systems. Specifically, it was focused on the following four areas of research which corresponds to the tasks where this Work Package was organized as well,

- Task 12.1: Auto-tuned runtime environments. Runtime environments for parallel platforms were investigated to provide auto-tuning capabilities in order to automatically find the best implementation for application codes.

- Task 12.2: Scalable numerical algorithms. This task explored new algorithms exposing much higher asynchrony, overlap between communication and computation, locality awareness than current practice.
- Task 12.3: Development environments and tools. It addressed the problem to provide efficient tools in order to deal with failures at large scale parallel system. The impact to the performance to applications when using a fault tolerant tool is evaluated at large scale.
- Task 12.4: File system optimization. It was focused to address performance issues of file systems at exascale in order to reduce the perceived request latency and improve the overall performance when a high volume of I/O requests are expected in such as environments.

This document provides a summary of the results achieved in each of these tasks during the third year. WP12 was initially plan for two years, but at the end was extended for an additional year. This is the only deliverable produced during the third year for WP12. Along with this deliverable it was produced a series of whitepapers that detailed in a more extended way the results presented in this document.

Task 12.1 and 12.2 were organized into several projects. In particular, task 12.1 was focused on six different projects focusing on auto-tuning various parts of the systems such as job scheduler, processor's vector instructions, or OpenMP applications. All of these projects are reported in this deliverable. Additionally, five whitepapers were produced to describe in more detail the results obtained.

Similarly task 12.2 was also composed on multiple independent projects, a total of eight. Results achieved of these projects are summarized in this document. In addition, five of these eight projects have also produced a whitepaper.

On the other hand, the other last tasks 12.3 and 12.4 were organized into only one project each. For this reason, they did not provided a whitepaper and the results are fully presented in this document. The techniques explored during the third year area on both tasks are not a continuation of the previous techniques explored during the first two years of WP12. For the third year they explored fresh new ideas. In the case of task 12.3, it was explored the use of fault tolerant tools which are of vital of importance for exascale systems. And on task 12.4, it was explored prefetching techniques in order to reduce the huge pressure of I/O operations in exascale environments as well.

In summary, this deliverable is reporting work for a total of 16 projects focusing on important aspects to improve the scalability of codes at large scale such as automatic optimization, algorithms, fault tolerance, and file systems.

Section 0 summarizes the results achieved on Auto-tuned runtime environments. Section 3 summarizes the results achieved on Scalable numerical algorithms. Section 4 provides results of a fault tolerant tool on a large scale system. Section 5 describes a new technique based on I/O prefetching in order to improve file systems at large scale. And finally, Section 6 concludes this document.

2 Auto-tuned Runtime Environments

2.1 Introduction

This chapter details the work done during the one year extension of Task 12.1. It is a follow up of D12.1 that described the work done on *Heterogeneous and Auto-tuned Runtime System* during the first two years. Six projects were extended and are covered by this chapter. This chapter 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. All but the last project have been published as PRACE white paper.

This six projects aims at making HPC systems more efficient by improving optimization support, in particular through the use of auto-tuning. They contribute to optimize various HPC elements: OpenMP applications, SLURM scheduler, Kepler workflow engine, component based 3D FFT, compiler auto-vectorization, and All-to-all collective communication. Here is the list of these projects that are then further detailed in this chapter.

- *Auto-tuning of OpenMP applications on the IBM Blue Gene/Q*
  - It deals with a library called SOMPARlib which capable of controlling dynamically the number of threads allocated to OpenMP parallel regions.
- *Topologically Aware Job Scheduling for SLURM*
  - It deals with a new AUCSCHED3 SLURM scheduler plug-in that has a capability to do topologically aware mappings of jobs on hierarchically interconnected systems.
- *Self-improving workflow models for generating of combinatorial objects designed in the Kepler Project System*
  - This work proposes some dedicated methods that can improve the execution time of workflows based on decision trees and the replication of some actors in the workflow.
- *Evaluating Component Assembly Specialization for 3D FFT*
  - It deals with the design and evaluation of component based assemblies of 3D FFT computation to ease code specialization while maintaining performance.
- *Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics*
  - This work aims to provide techniques for determining the best way to optimize certain codes regarding vectorization, with an end goal of guiding the compiler into generating optimized code without requiring developer expert knowledge.
- *Scalable algorithm for network bandwidth / latency hierarchy detection and designing topology aware collective routines and testing performance in CPMD application*
  - This project aims to develop a scalable code identifying performance tiers reflecting the network topology to improve MPI ALL-TO-ALL scalar/vector collective routine such as for example for the quantum chemistry code CPMD.

2.2 Auto-tuning of OpenMP applications on the IBM Blue Gene/Q

WP179: *Auto-tuning of OpenMP applications on the IBM Blue Gene/Q system*
Authors: *Maciej Cytowski, Maciej Szpindler (ICM, Univ. of Warsaw)*

Modern high performance computing architectures are based on multi-core and multi-threaded computing nodes. The mixed MPI and OpenMP programming is currently a reference model for obtaining high scalability on large computing systems. In such a model, MPI processes contain many OpenMP parallel regions. Scalability and performance of those parallel regions may differ between various computing systems and between each run of the code. The control of the number of threads used by different OpenMP regions, by users of the HPC systems, is very often limited to setting a single environment variable – `OMP_NUM_THREADS`. We have developed a tool called SOMPARlib which is based on OpenMP Monitoring Interface (POMP) and is capable of controlling the execution of various OpenMP parallel regions introduced in computational codes during run time. The tool is particularly useful in the case of architectures that introduce the multithreading mechanisms like Simultaneous multithreading (SMT) or Hyper-Threading (HT).

The library in its current version provides support for codes of a specific computational structure. The main object of our interest are those simulation packages for which computations are divided into a large number of iterations, e.g. molecular dynamics packages, cosmological simulation codes and many others. Main simulation loop may contain numerous OpenMP parallel regions. It may also call an external function (e.g. library calls) which may contain its own OpenMP parallel regions.

When code is linked against SOMPARlib and executed the following occurs:

1. SOMPARlib initialization phase
   a. memory required by SOMPARlib is allocated
   b. maximum number of threads available for a single process is found
   c. parallel modes (i.e. different number of threads used during testing phase) are defined
2. First iteration - detection phase
   a. structure of the main simulation loop is detected
   b. all OpenMP parallel regions within a single iteration are detected
3. Next N iterations (where N = number of parallel modes) - testing phase
   a. performance of parallel regions in all defined parallel modes is measured
   b. the best parallel mode for each of the OpenMP regions is selected
4. All remaining iterations - computing phase
   a. all remaining calculations are carried out with number of threads set individually for each of the OpenMP parallel regions
5. SOMPARlib cleanup phase

During the initialization phase SOMPARlib allocates memory required for storing performance measurements data. The maximum number of threads available for current process is checked by calling the function `omp_get_thread_limit()`. Based on this number the available parallel modes are determined. The first parallel mode is always related with the use of all available threads. If the number of threads assigned to a given parallel mode is divisible by two, the next parallel mode will be selected by reducing the number of threads twice. Otherwise, the number of threads is reduced by one. For example, if the maximum number of threads available to a single process is equal to 64, there will be 6 parallel modes considered during the testing phase, corresponding to 64, 32, 16, 8, 4 and 2 threads per process. However, if the maximum number of threads available to a single process is equal to 24, SOMPARlib will select 5 parallel modes corresponding to 24, 12, 6, 3 and 2 threads per process. The most effective setup is achieved when the maximum number of threads available to a single process is a power of two, which is also a very natural choice for many of today's HPC platforms.

In the next phase, the computations are started. During the first iteration of the main simulation loop all OpenMP parallel regions within the loop need to be detected. SOMPARlib is able to automatically detect a loop structure, but only for simple loops, wherein each OpenMP parallel region is called at most once during each iteration. In such cases the detection of the loop structure ends when SOMPARlib encounters the first OpenMP parallel region again, i.e. in the beginning of the second iteration. In all other cases, the SOMPARlib API needs to be used in order to mark the beginning and end of the iteration.

Next N iterations of the main simulation loop, where N is equal to the number of parallel modes, are used to gather necessary performance measurements of all OpenMP parallel regions in the loop. Starting from the parallel mode with the largest number of threads SOMPARlib measures execution time of all parallel regions. These measurements are repeated for subsequent parallel modes with decreasing number of threads. To prevent a significant slowdown of the testing phase we have decided to implement an additional checkpoint. If in two consecutive parallel modes the execution time of a given OpenMP parallel region increased 1.5 times, then we recognize that this change of performance is the effect of scalability. Therefore, in such situations we do not continue to search for the best parallel mode for the given OpenMP region, since we expect that consecutive time measurements will be worse than the previous ones. When appropriate parallel modes have been selected for all regions, the simulation is continued, and before each subsequent OpenMP region the appropriate number of threads is set by calling the `omp_set_num_threads()` function.

SOMPARlib was created primarily as a case study and therefore has some important limitations. In the current implementation, we assume that the amount of work corresponding to each OpenMP parallel region is at the same level in all the iterations. Furthermore, each iteration of the main simulation loop must always contain the same OpenMP parallel regions executed in the same order. Assumptions about the structure of the program are so restrictive mainly due to technical limitations of the POMP interface available on the Blue Gene/Q architecture. When it comes to handling different OpenMP standard functionalities, currently only classical parallel regions and parallel loops are supported. In particular, SOMPARlib does not support OpenMP tasking.

Usage of the SOMPARlib on the IBM Blue Gene/Q system is rather simple. Recompilation of the code is only necessary in the case of applications that use the SOMPARlib's API. Programs need to be also linked with the library, e.g.:

```
$ bgxlc_r -qsmp=omp program.o -o program.x -lxlsmp_pomp -lsompar
```

Thanks to the POMP implementation available in IBM compilers, SOMPARlib is able to control OpenMP parallel regions included within external libraries. SOMPARlib is able to detect and control OpenMP parallel regions defined within external libraries. Most importantly, no additional code modifications or recompilation of those libraries is required.

Functionality of the SOMPARlib is shown based on the benchmark program specially written for this purpose. Benchmark code is made up of 512 successive iterations each consisting of five steps with different computational footprint. In the first step of the main simulation loop an N-body type computations are carried out. For all of 32768 particles in three-dimensional space distance and interactions between them are calculated. The loop over particles is parallelized with single OpenMP pragma. In the second step, the matrix multiplication is calculated with two square matrices of size 256 x 256. On the Blue Gene/Q system we use the OpenMP parallelized DGEMM available in the ESSL SMP library. Third step of the benchmark is an OpenMP implementation of a sorting algorithm applied to randomly generated sequence of 1048576 floating point numbers. The implementation is based on the `qsort` function available in the standard C library, which is applied in its thread-safe version to equal subsequences of the original data. The resulting sorted sequences are then merged into the final result. In the fourth step of the benchmark we use the FFTW (v.3.3.2) library compiled with OpenMP support to compute the 3D FFT of a 512 x 512 x 512 grid data. The last step of the main simulation loop is the LU factorization of a sparse matrix. For this purpose we use the SuperLU-MT library and an example sparse matrix Fidapm11 of size 22294 x 22294 and 623554 non-zero elements obtained from the Matrix Market.

The most important result of presented test was the decrease of overall walltime for benchmark runs. To compare performance we have executed two benchmark runs with one of these runs compiled and controlled by the SOMPARlib. We compare the results obtained with the use of SOMPARlib with standard executions in available SMT modes. On the Blue Gene/Q platform we have executed the program on 16 (SMT1), 32 (SMT2) and 64 (SMT4) threads. The actual number of threads used by the application was chosen by setting the `OMP_NUM_THREADS` environment variable. In this way we try to mimic a situation where in order to obtain the highest scalability computations are executed with the use of a single MPI process per node and maximum number of threads available within node. The increase of performance was measured for all SMT modes; results are presented in Table 1.

Parallel modes	16 (SMT1)	32 (SMT2)	64 (SMT4)
Improvement	49.91%	21.37%	7.62%

Table 1: Performance results for all SMT modes

2.3 Topologically Aware Job Scheduling for SLURM

WP180: *Topologically Aware Job Scheduling for SLURM*
Authors: *Seren Soner, Can Ozturan (Bogazici University)*

SLURM is a popular resource management system that is used on many supercomputers in the TOP500 list. SLURM provides two primary modes of operation for topology-aware job placement in order to reduce network contention: One mode for hierarchical interconnects like a tree (or a fat tree) and another mode for three-dimensional torus architectures. In this work, we contribute a new AUCSCHED3 SLURM scheduler plug-in that has a capability to do topologically aware mappings of jobs on hierarchically interconnected systems.

SLURM identifies the lowest level switch in the hierarchy that can satisfy a job's request and then allocates resources on its underlying leaf switches using a best-fit algorithm [1] AUCSCHED3 is based on our previous auction based scheduling algorithm of AUCSCHED2 [2] [3]. AUCSCHED3 does the following: (i) It generates bids for topologically good mappings of jobs onto the resources and (ii) it adjusts the priorities of the jobs slightly without changing the original priority ordering of jobs so as to favor topologically better candidate mappings.

Effectiveness of the new AUCSCHED3 plug-in is tested on a three level (levels 0, 1 and 2) hierarchically interconnected 1024 node system with 16 cores and 3 GPUs on each of its nodes. In the workloads used for testing, there are five types of jobs (named A, B, C, D, E) with each type making different resource requests as shown in **Table 2**. The workloads are made up of various percentages of these job types and are shown in **Table 3**.

Job Type	Job Description
A	only x cores
B	x cores on y nodes
C	x cores on y nodes, 1 GPU on each node
C'	x cores on y nodes, 1 to 3 GPUs on each node
D	x cores on y nodes, 2 GPUs on each node
D'	x cores on y nodes, 2 to 3 GPUs on each node
E	x cores on y nodes, 3 GPUs on each node

Table 2: Job types in the workload

When testing SLURM's Backfill plug-in, jobs of type A, B, C, D and E are used. AUCSCHED2 and AUCSCHED3 also provide support for generic resource ranges (which is not available in SLURM/Backfill). Such a feature can be useful to runtime auto-tuning applications that can make use of variable number of generic resource such as GPUs. Job types C' and D' and workloads 5' and 6' are the same as their counterparts but use GPU ranges. Therefore, we only test them using AUCSCHED3.

		Percentage of Jobs				
Workload ID	**Number Of Jobs**	**A**	**B**	**C**	**D**	**E**
1	350	100	0	0	0	0
2	2095	100	0	0	0	0
3	350	0	100	0	0	0
4	2095	0	100	0	0	0
5	350	20	20	20	20	20
6	2095	20	20	20	20	20
5'	same as 5, but uses GPU ranges					
6'	same as 6, but uses GPU ranges					

Table 3: Workload types and job distributions

To test our new AUCSCHED3 plug-in, we conduct emulation tests. We are able to retrieve topology related information of allocated jobs and hence we can evaluate goodness of allocations. The results are analyzed using the following performance measures: (i) *Lowest level common switch* (the lowest level common switch from which all the nodes allocated to a job can be reached), (ii) *Spread* (the distance from the first node to the last node allocated to a job), and (iii) *Utilization*.

Average lowest common switch level and the average spread measures are plotted in Figure 1. To get a better insight, the distribution of jobs over lowest common switch levels of all jobs for all workloads and for 5,6,5', 6' workloads are given in Figure 2(a) and Figure 2(b) respectively.

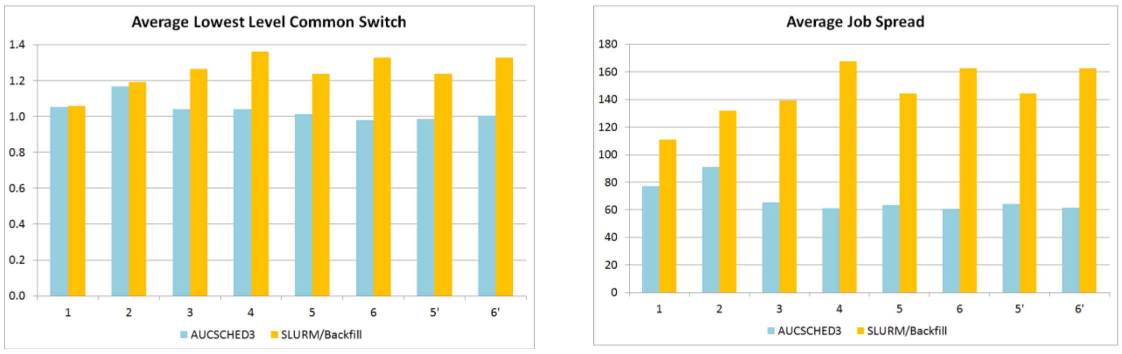


Figure 1: (a) Average lowest level common switch (b) Average job spread

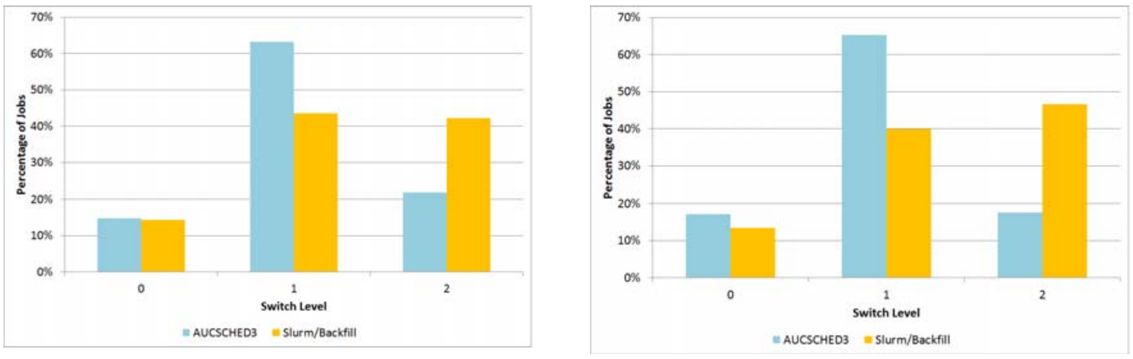


Figure 2: Distribution of jobs over switch levels for all workloads (a), for 5, 6, 5' and 6' workloads (b)

The results obtained on the emulated 1024 node system show that AUSCHED3 plug-in is able to generate better topological mappings than SLURM/Backfill. It can do this while keeping system utilization levels even higher than that of SLURM/Backfill in the case of workloads 1, 2, 5, 6, 5', and 6'. In the case of workloads 3 and 4, utilizations drop slightly by 4% and 5% respectively. However, considering the fact that the switch levels of AUCSCHED3 mappings are lower, the execution times are likely to be shorter due to faster communication and hence the differences in utilizations in these cases are likely to be smaller. Overall, AUCSCHED3 is able to generate both topologically better mappings of jobs and achieve higher system utilizations especially in workloads involving jobs that request both CPU and GPU resources.

2.4 Self-improving workflow models for generating of combinatorial objects designed in the Kepler Project System

WP181: *Self-improving workflow models for generating of combinatorial objects designed in the Kepler Project System*
Authors: *Krzysztof T. Zwierzyński (PSNC)*

We consider the problem of design of self-improving meta-model of workflow of jobs that is sensitive on the change of the computational environment. As an example of searched combinatorial objects some classes of integral graphs are used. We propose some dedicated methods that can improve the execution time of workflow based on decision trees and the replication of some actors in the workflow.

The Kepler Project [4] is based on the Ptolemy II System, a platform supporting multiple models of computation, i.e.: Synchronous Data Flow, Dynamic Data Flow, Process Network, Discrete Events, and Continuous Time. The Kepler Project provides a graphical user interface and a run-time engine that can execute workflows either from within the graphical interface or from a command line. Workflows can be nested, allowing complex tasks to be composed from simpler components. Kepler Project workflows are defined in the Modeling Markup Language (MoML) [5].

Let $G = (V, E)$ denote a *simple graph* with a nonempty vertex set V of a cardinality $n = |V|$, and a set of pairs of vertices called edges E where loops are forbidden. The *distance matrix* ($Dis(G)$) of G is an all-pairs shortest path matrix consisting distances between all pairs of vertices in the graph G. The set $Sp(G) = \{\lambda_1, \lambda_2,\ldots, \lambda_n,\}$ of graph eigenvalues of the adjacency matrix A is called the *spectrum* of graph G. The combinatorial goal was to find all graphs of given order n with distance matrices that have integral eigenvalues.

The workflow model of the computation in the Kepler Project is a bipartite digraph $W = (A \cup R \cup InP \cup OutP, LP \cup LR)$, where A is set of Actors (units that can perform some computation; in particular Composite Actors). The set R contains vertices that correspond to relations. *InP* is the set of input ports and *OutP* is set of output ports, respectively. *LP* is a set of pairs (*a*, *p*) where *a* ∈□*A*, and *p* ∈*InP* ∪ *OutP*. *LR* is a set of pairs (*r*, *p*) where *r* ∈*R*, and *i* ∈ *InP* ∪ *OutP*. The workflow shown in the Figure 3 represents a simple pipeline workflow: the output from A1 is the input for A2; the output of A2 is the input for A3. The relations R1 and R2 correspond to the arrow.

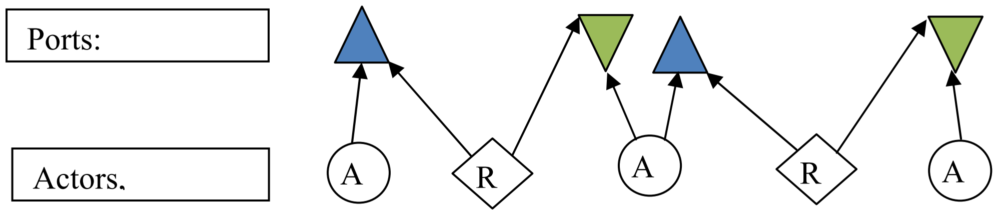


Figure 3: The model of computation (○: A – Actors; ◊: R – Relations, ▼□ InP Input ports; ▲: OutP Output ports)

The main goal is to describe the possibilities of transforming workflows in such a way that the runtime of new one should be shorter than the old one. From that reason we define some meta-workflows that can take some statistic data of executions of some set of workflows and chose the best one, or define a decision tree which workflow use for a specific values of parameters of the combinatorial problem to solve.

Figure 4 shows the Kepler Project workflow that generates graphs using the *G*(*n*, *p*) model. The value *p* is an edge probability. The graphs are stored in the Prologue database as compound terms. The main benefit of this web-service oriented solution is that we can also put into the Prologue database some producing rules that add some new compound terms derived from the facts in the database.

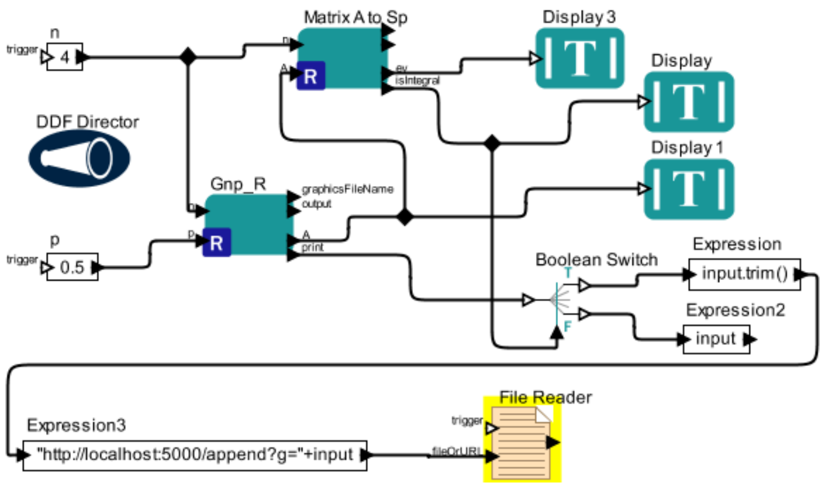


Figure 4: Usage the G(n, p) model and the Prologue database (n = 4 and p = 0.5)

Using the meta-workflow shown in the Figure 5 we can obtain statistics about the runtime of some programs.

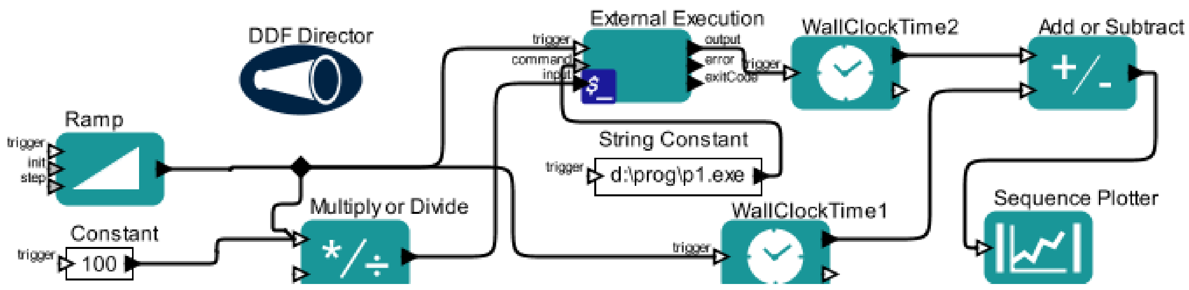


Figure 5: The meta-workflow about runtimes of a given program in the function of task size.

If some calculations for instances of the problem can be done independently then to improve the runtime we can make the replication of the main Actor in the workflow. The useful limit of the replication can be obtained experimentally. The replication technique is also a method that can auto-tune workflows for new generations of computers where for each processor more cores will be available. However, that schema of replication has a limit that comes from the bottleneck of the task scheduler.

We can also introduce in a workflow some *Map-Reduce* technique. Figure 6 displays one of the methods that introduce a *decision tree* (DT) into the Kepler Project workflow. The *Expression* Actor compare vectors of parameters values *X* of the task with the vector of thresholds values *T*. It results in some decision class.

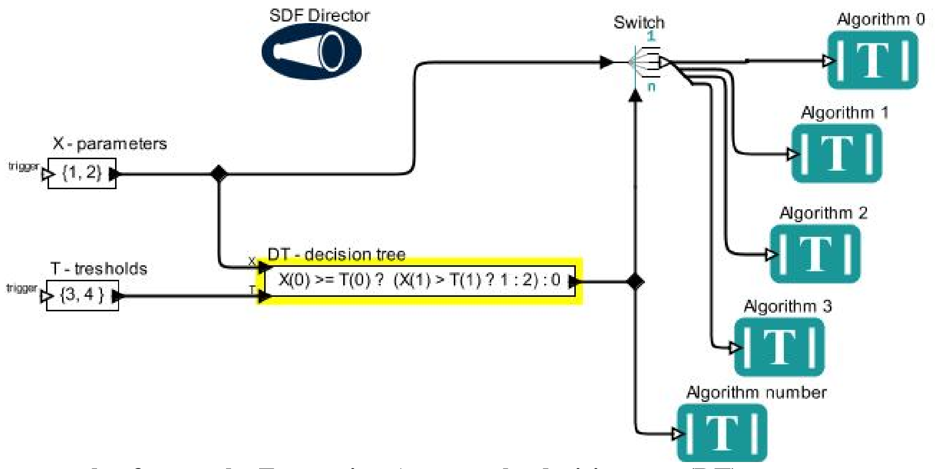


Figure 6: An example of usage the Expression Actor as the decision tree (DT)

To build a DT we can use J.R. Quinlan’s algorithm called *c4.5* [5]. To improve the thresholds in the DT we can during the execution of workflow run additionally in parallel way some other algorithm chosen randomly from the sequence. If it finishes the execution before the algorithm chosen by DT, then we can add this information to the training set, and build a new DT. Instead of choosing another algorithm randomly we can use information about thresholds values. If the parameter value is equal to the threshold value in the node close to leafs of DT than we can run in parallel both branches. If we distribute computation over the network of heterogeneous hosts, we can also use DT in the job scheduling algorithm to find the best host that can make the fastest computation of our task.

The same strategy can be used in the case when the task can be divided into two or more parts (subtasks). Then in the case when one big task is working to long, we can run in the delay technique its equivalent subtasks. This can be done hierarchically for subtasks.

2.5 Evaluating Component Assembly Specialization for 3D FFT

WP182: *Evaluating Component Assembly Specialization for 3D FFT*
Authors: *Jérôme Richard (LIP/Univ. Orléans), Vincent Lanore (LIP/ENSL), Christian Perez (LIP/Inria)*

The Fast Fourier Transform (FFT) is a widely-used building block for many high-performance scientific applications. Efficient computing of FFT is paramount for the performance of these applications. This has led to many efforts to implement machine and computation specific optimizations. However, no existing FFT library is capable of easily integrating and automating the selection of new and/or unique optimizations.

To ease FFT specialization, we have evaluated the use of component-based software engineering, a programming paradigm which consists in building applications by assembling small software units. Component models are known to have many software engineering benefits but usually have insufficient performance for high-performance scientific applications.

This work is based on L^2C [6] a general purpose high-performance component model, and it studies its performance and adaptation capabilities on 3D FFTs. Experiments show that L^2C, and components in general, enables easy handling of 3D FFT specializations while obtaining performance comparable to that of well-known libraries. However, a higher-level component model is needed to automatically generate an adequate L^2C assembly

The L^2C model [6]can be seen as an extension of modular compilation or as a low level component model that does not hide system issues. Indeed, each component is compiled as an object file. At launch time, components are instantiated and connected together according to an assembly description file (LAD) or to an API. L^2C supports various features like memory sharing, C++/FORTRAN procedure invocations, message passing with MPI, and remote procedure calls with CORBA thanks to provides/uses port and MPI ports (MPI communicator sharing). Components can also expose attributes used to configure component instances. A L^2C assembly descriptor file contains a description of all component instances, their attributes values, and the connections between instances. Each component is part of a process and each process has an entry point (an interface that is called when the application starts).

We have designed and implemented in L^2C various assemblies to analyze how L^2C can be used to implement distributed 3D-FFT based on the use of global transpositions. First, we have first designed a basic 3D-FFT assembly using 1D decomposition. Then, we have improved it with some optimizations from the literature. Optimizations have been applied in three stages to highlight different component model features: i) replacing a component implementation with a more optimized implementation (transpose implementation), ii) using component attributes for heterogeneous platforms tuning, and iii) global assembly adaptation to implement computation/communication overlapping and 2D decomposition.

We have evaluated the component based approach in terms of performance and adaptability through some L^2C assemblies. Performance and scalability are evaluated on up to 512 cores on homogeneous and heterogeneous architectures. Adaptability relates to the easiness to implement the various optimizations, and to how much code has been reuse.

Experiments have been done on multiple clusters of the Grid'5000 experimental platform [7].These clusters are Griffon, Graphene, Edel and Genepi. They are made of quad-core dual processor but Graphene whose nodes have a quad-core mono-processor. For heterogeneous tests, we have used the Genepi cluster and the Edel cluster. Both clusters are connected to the same InfiniBand network. However, they have different processors which make them suitable for heterogeneous experiments.

Assembly Name	*Decom-position*	*#Transposition*	*Hetero.*	*Library Name*	*Decom-position*	*#Transposition*	*Hetero.*
1D 2t xz	1D	2	No	**FFTW**	1D	2	Not used
1D 1t yz	1D	1	No	**FFTW 1t**	1D	1	Not used
1D 2t yz	1D	2	No	**2DECOMP 1D1t**	1D	1	Not available
1D 2t yz blk	1D	2	No				
1DH 1t yz	1D	1	Yes	**2DECOMP 1D2t**	1D	2	Not available
1DH 2t yz blk	1D	2	Yes				
2D 3t	2D	3	No	**2DECOMP 2D**	2D	3	Not available
2DH 3t	2D	3	Yes				

Table 4: Assemblies and reference libraries used in experiments

Table 4 summarized the L^{2}C assemblies and the reference FFT libraries that are used in experiments. All experiments involve complex-to-complex 3D FFTs. We have not used L^{2}C assemblies with overlapping because their implementation is still ongoing. The FFT libraries used as reference are FFTW 3.3.4 and 2DECOMP 1.5. All libraries are configured to use a complex to complex 3D FFT without overlapping (as for L^{2}C assemblies) using FFTW sequential implementation and double precision floating point. All implementations use `FFTW_MEASURE` planning. The compiler is gcc (version 4.7.2) and the implementation of MPI is OpenMPI (version 1.8.1).

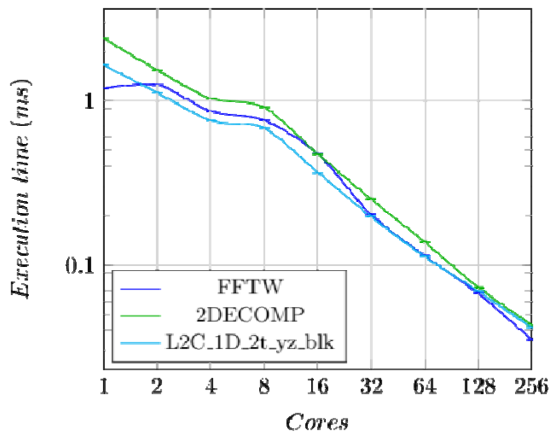


Figure 7: Experiments for a 2563 matrix, 1D decomposition, and two transpositions on Griffon.

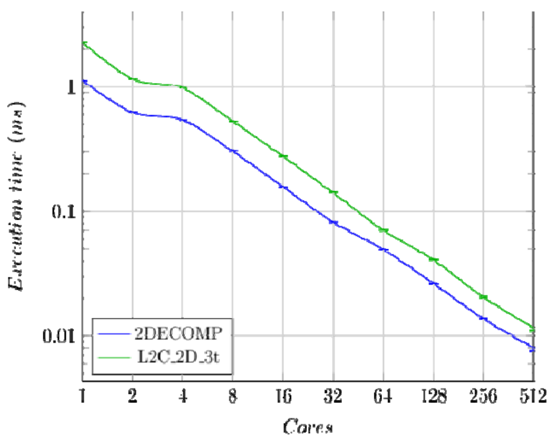


Figure 8: Execution time for a 2563 matrix, 2D decomposition, and one transposition on Graphene.

Figure 9 and Figure 7 shows that L^{2}C assemblies can be as efficient as reference libraries for 1D decomposition. However, as shown in Figure 8, the L^{2}C assembly needs more optimization to compete when using 2D decomposition.

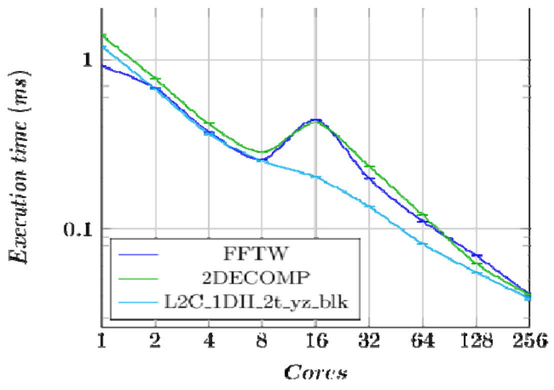


Figure 9: Experiments for a 2563 matrix, 1D decomposition, and two transpositions on a heterogeneous cluster (Edel+Genepi).

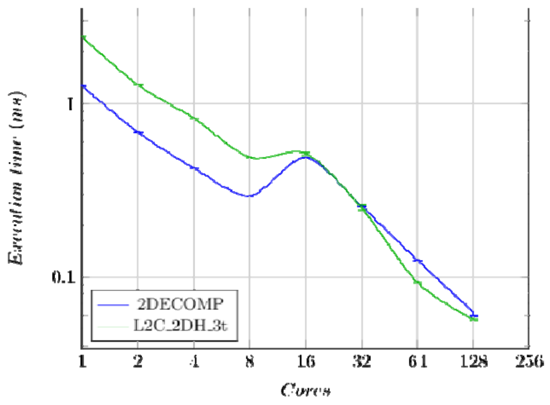


Figure 10: Experiments for a 2563 matrix, 2D decomposition, and one transposition on a heterogeneous cluster (Edel+Genepi).

Figure 9 and Figure 10 have been obtained on a heterogeneous cluster made of two homogeneous clusters (Edel and Genepi) for 1D and 2D decomposition respectively. The goal of these experiments is to show that components enable to easily adapt an assembly to support heterogeneity and that of course improves performance.

Assembly Name	*C++ Lines of Codes*	*Code reused*	*Assembly Name (cont.)*	*C++ Lines of Codes (cont.)*	*Code reused (cont.)*
1D 2t xz	927	-	**1DH 1t yz**	983	80%
1D 1t yz	929	77%	**1DH 2t yz blk**	1097	72%
1D 2t yz	929	100%	**2D 3t**	1067	87%
1D 2t yz blk	1035	69%	**2DH 3t**	1146	69%

Table 5: Total number of lines for the various versions of the 3D FFT application

With respect to reuse, **Table 5** shows code reuse (in terms of number of lines of C++ code) between some of L^{2}C assemblies. Reuse is the amount of code that is reused from the assemblies list higher in the table. Overall, our L^{2}C implementations are much smaller than 2DECOMP or P3DFFT (respectively 11570 and 8118 lines of FORTRAN code); that is also because they implement more features.

With respect to adaptation, which is the goal of this work, components enable lightweight and specialized assemblies. Several optimizations from the literature have been implemented, taking advantages of code reuse, component replacement in assemblies, and component attribute tuning. Other optimizations require the implementation of new components. The specialization process allows to reuse most of the base components (69% to 100% reuse) without any modification.

With L^{2}C, assembly descriptions need to be rewritten for each specific hardware. As it is fastidious and error-prone, such descriptions should be automatically generated. This is one of the purposes of HLCM [8], a high level component model. HLCM also aims at automating assembly generation. To this end, one direction is to rely on 3D FFT performance models.

2.6 Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics

WP183: *Performance Improvement in Kernels by Guiding Compiler Auto-Vectorization Heuristics*
Authors: *Miceli (NUIG/URennes), Christian Lalanne (NUIG), Michael Lysaght (NUIG), Michael Browne (NUIG), William Killian (UDel), EunJung Park (UDel), Marco A. Vega (UDel), John Cavazos (UDel)*

Vectorization support in hardware continues to expand and grow as we still continue on superscalar architectures. Unfortunately, compilers are not always able to generate optimal code for the hardware; detecting and generating vectorized code is extremely complex. Programmers can use a number of tools to aid in development and tuning, but most tools require expert or domain-specific knowledge to use. In this work we aim to provide techniques for determining the best way to optimize certain codes, with an end goal of guiding the compiler into generating optimized code without requiring developer expert knowledge. Initially, we study how to combine vectorization reports with code generation and iterative compilation and summarize our insights and patterns on how the compiler vectorizes code. Our code generation and compilation utilities can be further used by non-experts in the generation and analysis of programs. Finally, we leverage the obtained knowledge to design a Support Vector Machine classifier to predict the speedup of a program given a sequence of optimization. We show that our classifier can predict the speedup of 56% of the inputs within 15% over- and 50% under-prediction, with 82% of these accurate within 15% both ways.

Based on previous experience [9] we developed two utilities in order to help with version generation for iterative compilation. **`autovec`** is a source-to-source compiler which translates a simplified directive language to a compiler-specific directive language (e.g. Intel Compiler, CAPS HMPP Compiler, PGI Compiler). **VALT** (vectorization and loop transformation) performs iterative compilation among a set of optimizations to apply more than once in a given program. These utilities can help non-experts in the generation and analysis of codes; they have been used here to understand the inner workings of the compiler's vectorization strategies.

`autovec` directive	**Intel-specific pragma**
`permute`	*generate each version*
`vl(x)`	`simd vectorlength(x)`
`always`	`vector always`
`ivdep`	`Ivdep`
`none`	<nothing>

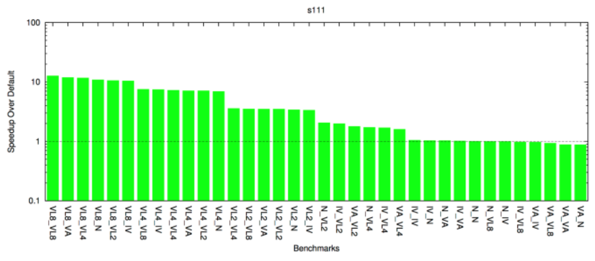
Table 6: `autovec` directive support and translation

VALT directive	**Intel-specific pragma**
`vector(default)`	*<no code emmited>*
`vector(none)`	`novector`
`vector(always)`	`vector always`
`vector(ignore)`	`ivdep`
`vector(aligned)`	`vector aligned`
`vector(temp)`	`vector temporal`
`vector(nontemp)`	`vector nontemporal`
`vectorsize(x)`	`simd vectorlength(x)`
`loop(unroll(x))`	`unroll(x)`
`loop(jam(x))`	`unroll_and_jam(x)`
`loop(nofusion)`	`nofusion`
`Loop(dist)`	`distribute_point`

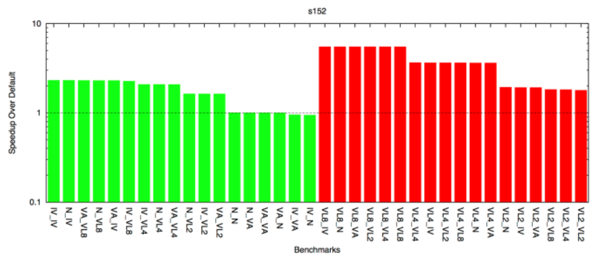
Table 7: VALT directive language translation for Intel-specific pragmas

To evaluate the Intel compiler's built-in vectorization heuristics, two sets of benchmarks were used to determine performance improvement: *Test Suite for Vectorizing Compilers* [10], with 151 loop nests, is an extension and modification of a test suite for vectorizing FORTRAN compilers in the late 1980's [11] and *Polybench/C 3.2* [12] with 30 micro-kernels, stems from

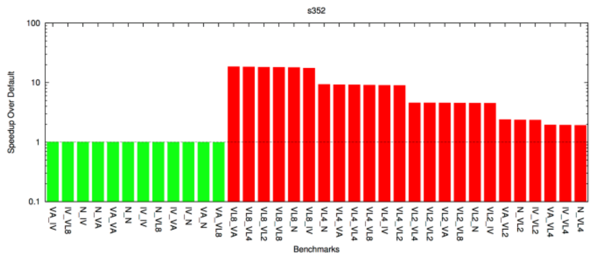
Pouchet's work with Polyhedral compilers. We studied the vectorization reports generated by the Intel compiler and compared the original benchmark's vectorization report to the vectorization reports from each version generated by **`autovec`** and **`VALT`**. With this we identified patterns and trends followed by the vectorization heuristics. Figure 11 shows sample performance analysis of 4 different loop nests with varying level of correctness.



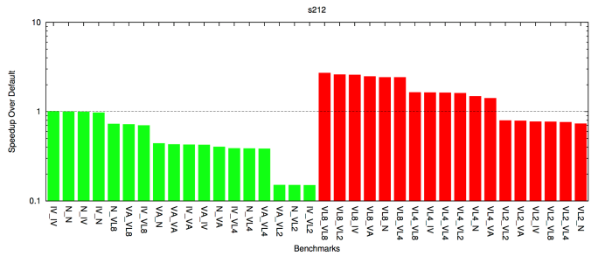
(a) Good speedup observed; no invalid code generation



(b) Good speedup observed with invalid code faster



(c) No speedup observed; invalid code generation faster



(d) No speedup observed; invalid code generation

Figure 11: TSVC loop nest optimizations and speedup comparison (valid code is green; invalid is red).

We were also able to classify TSVC loop nests into a four different categories:

1. *Non-vectorizable* – Loop nests not vectorizable. 16 benchmarks (11%) were not vectorizable as indicated by the optimized version vectorization report and minimal/non-existing speedup observed.
2. *Known vectorization pattern* – Loop nests which could be vectorized by the compiler with minimal additional speedup observed after optimizing. 69 benchmarks (46%) fell into this category. This suggests that overall the Intel compiler is able to vectorize code well with its built-in heuristics although they are not always optimal.
3. *Inner-loop vectorizable* – Loop nests not initially vectorized well but better optimized with a **`#pragma simd`** directive placed in an inner loop. 12 loop nests (8%) were inner-loop vectorizable with a speedup of at least 2×.
4. *Outermost-loop vectorizable* – Loop nests not initially vectorized well but better optimized with a **`#pragma simd`** directive placed in the outermost loop of the loop nest. 54 loop nests (35%) were outermost-loop vectorizable with a speedup of at least 2×.

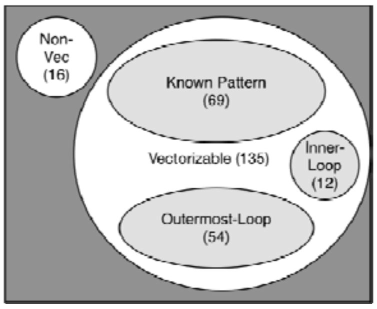


Figure 12: Categorization of TSVC loop nests.

SVM-Based Speedup Predictor

Given the speedup information based on different optimizations for a collection of loop nests, we designed a support vector machine (SVM) classifier to automate the prediction of benchmark speedups given an optimization sequence. We used the 151 loop nests from TSVC as training data for our predictor model. For training we specified our feature vector as consisting of 45 performance counters [13] [14] normalized to the total of instructions executed, the

Percentile	Type	Count
0.15	Under	45
0.50	Under	15
1.00	Under	12
2.00	Under	16
> 2.00	Under	27
0.15	Over	24
> 0.15	Over	12

Table 8: Speedup Predictor Accuracy

speedup of vectorized code over non-vectorized code, the optimization bit vector, and the speedup of optimized code over vectorized code. We tested our trained models with leave-one-out cross validation (LOOCV). Our results in Table 8 show that 84 of the loop nests were accurate within 15% over-prediction and 50% under-prediction. 5 of the over-predictions were on non-vectorizable loop nests. The predictor could not accurately predict speedup for over 44% of the loop nests. For 46% of the loop nests, the predictor was accurate within 15%. Analysis of the types of loop nests and the predicted speedup did not show correlation between the types of benchmarks which were under- and over-predicted. In the future, we can reuse this predictor in auto-tuning frameworks [15] [16].

2.7 Scalable algorithm for network bandwidth / latency hierarchy detection and designing topology aware collective routines and testing performance in CPMD application

No whitepaper
Supported by: *Johan Raber, Chandan Basu (SNIC-LiU)*

The goals of this project are to develop a scalable code identifying interconnect performance tiers reflecting the interconnect topology. This information subsequently is to be used in forming MPI collective groups for use in improved MPI ALL-TO-ALL scalar/vector collective routine of our design. It has also been our goal to demonstrate the viability of our approach on a popular quantum chemistry code, CPMD, which relies heavily on `MPI_ALLTOALL` in its implementation.

The identification of interconnect performance tiers has been done using statistical tools, most prominently Classification and Regression Tree models (CART) [17] using a two-phased approach. In the first phase, ping-pong latency data from selectively chosen, and representative, nodes and ranks of the system are used to build a statistical decision tree where rank pair data such as ping-pong latency, bandwidth and message rates can be used to categorize the relation between them, for instance whether they share memory controller or they reside on different nodes. In practice this traversing of the decision tree is done through a series of comparison operations on the input data record, the simplest case of which is shown in Figure 13.

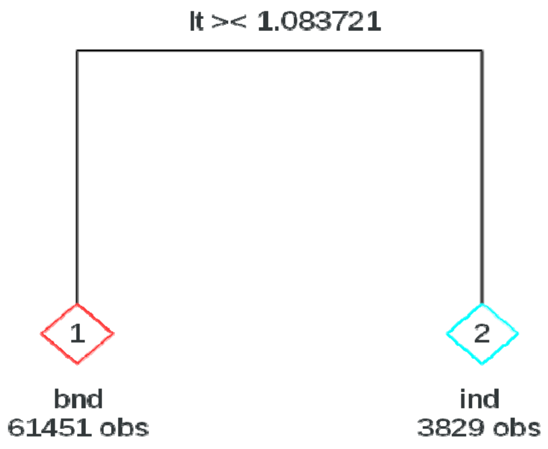


Figure 13: Graphical output from the statistical software R.

The first phase is intended as a one-time effort for any given cluster (or other system) and sets up the categorization criteria to be used in the second phase, the runtime classification and MPI group forming.

In the work presented in this report, we have chosen to measure small message ping pong latency. This because every switch an MPI data package passes incurs a latency overhead whereas it does not necessarily incur a detectable bandwidth drop, which should allow the detection of the number of switches an MPI package passes. This is shown in Figure 14 that displays measurements on two popular interconnect topologies as kernel density plots.

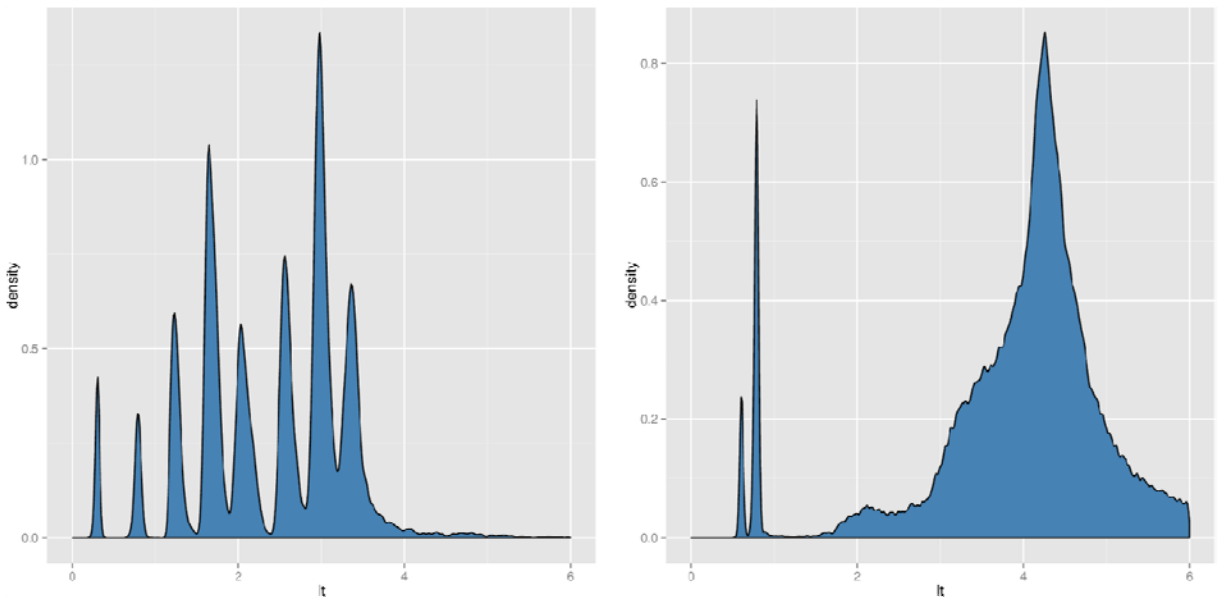


Figure 14: Kernel density plots of rank pair latency data from the Triolith commodity cluster.

The data collection is based on the OSU micro benchmark code (`osu_latency`) for small message latency and our adaptation of it for our purposes also contains a fix to handle the problem of the very common outliers in the measurement data. The outliers are caused by network congestion as well as OS incurred interrupts. The way we deal with this is to simply increase the number of samples taken per rank pair data collection record and to also drop 20% of the head and tail extreme values sampled. Dropping the head values is possibly not necessary since there is a hard physical limit to how low they can be and they should therefore not be considered as “noise” to be dropped.

To investigate the feasibility and scope of our approach on different interconnect topologies we have performed measurements on two important classes of supercomputers, a commodity cluster using InfiniBand interconnect in a fat tree topology, and also a Cray system using their proprietary Gemini interconnect in a 3D torus topology. The results are depicted in Figure 14 and they merit some comments. The first two peaks (from the left) in the left plot arise from intra node rank pairs, the first of which come from rank pairs sharing memory controller. Next comes a group of three peaks arising from rank pairs on different nodes sharing the same InfiniBand switch and then comes the last group of three peaks stemming from rank pairs on different nodes residing on different InfiniBand switches. The fine structure of these groups arise from the computer architecture of present day two-socket Intel Sandybridge based servers where the low latency peak consists of rank pairs where both ranks reside on the CPU socket controlling the PCI express bus hosting the InfiniBand HCA. The high latency peak in these groups consists of rank pairs on sockets not controlling the IB HCA.

The right plot of Figure 14 has much less well defined peaks except for the first two. These stem from intra node rank pairs and their relative size difference comes from the two-socket nodes and the AMD Magny-Cours CPU which has two memory controllers per socket resulting in four memory controllers per node. Beyond these two well defined peaks lies a “data smear” devoid of clear features. This looks like a characteristic of the 3D torus topology the cause of which we have not investigated. Likely, our approach cannot be used to good effect on this supercomputer architecture beyond identifying intra node rank pairs, switch jump count detection seems out of reach.

We have earlier worked on a topology aware ALL-TO-ALL vector routine [18] [19]. This routine detects intra-node and inter-node topology and forms intra-node and inter-node MPI communicators. In our implementation ALL-TO-ALL communication is achieved by first an ALL-TO-ALL across inter-node ranks followed by several ALL-TO-ALL across intra-node ranks. As intra-node bandwidth and latency are better than that of inter-node this gives better ALL-TO-ALL performance. We have shown before [18] [19] that performance of our ALL-TO-ALL vector routine is better than the normal `MPI_ALLTOALLV` routine. Our earlier implementation of ALL-TO-ALL had a different interface than `MPI_ALLTOALLV`. In addition, it required some initialization routine to be called whenever a new dataset is to be sent. The usage of this routine therefore required some changes in the application source code. In this project we have extended our work on ALL-TO-ALL routine. We have made the interface of our ALL-TO-ALL vector routine the same as that of the `MPI_ALLTOALLV` routine. We have created a FORTRAN interface of our ALL-TO-ALL vector routine and we have also created topology aware ALL-TO-ALL scalar routine. With these enhancements these routines can now directly replace `MPI_ALLTOALL` / `MPI_ALLOTOALLV` in any C / FORTRAN program without any code change.

We have tested our ALL-TO-ALL routine in the CPMD code [20] CPMD is an ab-initio molecular dynamics code written in FORTRAN. It is well known that CPMD code performance is dominated by MPI_ALLTOALL time [21]. For compiling CPMD with our ALL-TO-ALL routine we added the extra preprocessor flag `-DMPI_ALLTOALL=MPI_ALLTOALL_TUNED` at the compile time, where `MPI_ALLTOALL_TUNED` is the name of our ALL-TO-ALL routine. At link time we add the library containing the `MPI_ALLTOALL_TUNED` routine. We have used gcc 4.7.2, Intel MPI 4.0.3.008 and Intel MKL 11.0.4.183 for compiling CPMD code.

For our benchmarking we have used the CPMD Si512 Inp-1 benchmark. We measure the total run time `t_ref` and `t_tuned` respectively for "reference CPMD binary with MPI_ALLTOALL" and "tuned CPMD with MPI_ALLTOALL_TUNED". We define *speedup = (t_ref - t_tuned) / t_ref*. Figure 15 shows the %speedup of the tuned version of CPMD. For small number of ranks the tuned version is slightly slower than the reference version of CPMD which is expected as our approach is expected to give speedups for wide runs. We see that for 256 rank and 1024 rank runs the tuned version is more than 60 % faster. However for 512 rank run there is no speedup. This anomaly may be due to some change in internal algorithm in the `MPI_ALLTOALV` at this size.

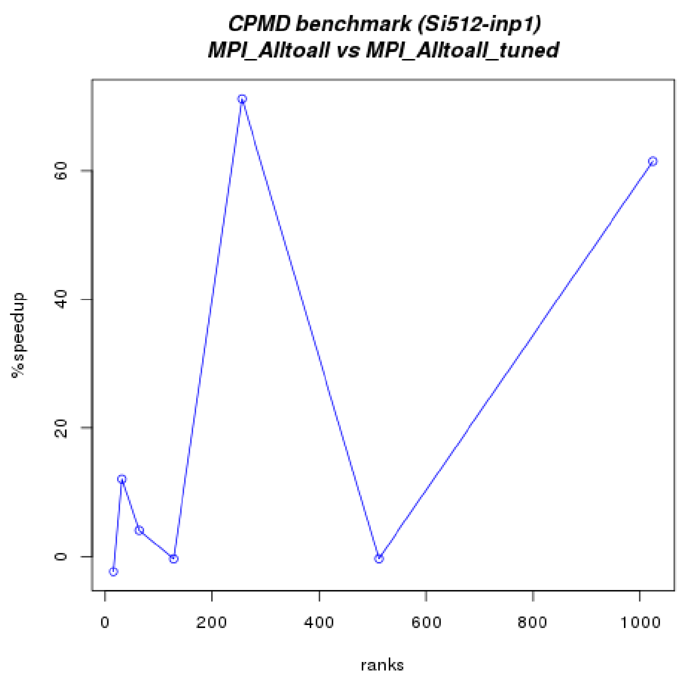


Figure 15: CPMD benchmark. The speedup is defined as (t_ref - t_tuned) / t_ref.

It seems clear that our approach to performance tier detection is viable for at least the commodity cluster class of supercomputers where it is capable of producing very finely resolved data to form MPI groups. Its usefulness is therefore limited by how well an MPI application developer can utilize this information in the setup of the calculation and not by the precision of the performance tier detection. It is more than capable enough to serve our goal for using it with our improved MPI ALL-TO-ALL routines. In our current implementation of ALL-TO-ALL routines we have not integrated the information from network tier detection. At present we are only detecting intra-node and inter-node network tier. The replacement of our `MPI_ALLTOALL_TUNED` routine in the CPMD code shows speedup in general. We will continue working on further improving our topology aware collective routines.

2.8 Conclusion

During this one year of extension, six projects successfully contributed to Task 12.1 that is about auto-tuning HPC systems. They all aimed to automatize the efficient usage of HPC systems Indeed, petascale and post-petascale systems are too much complex to rely on non-experts to optimize their usage. The results of these six projects have concerned various aspects of an HPC system.

- *Language*
    - The library named SOMPARlib has shown to be able of controlling dynamically the number of threads allocated to OpenMP parallel regions to decrease the overall walltimes for benchmarks runs.
- *Job scheduler*
    - An improved version of the AUCSCHED SLURM scheduler plug-in (AUCSCHED3) achieves better mappings and higher system utilization by using topologically aware mappings on hierarchically interconnected systems.
- *Workflows*
    - Meta-model based dedicated methods relying on decision trees and the replication of some workflow actors have been design and integrated into Kepler to improve the execution time of workflows.

- *Component models*
    - *This work has shown that equivalent performance, better code reuse, and much easier specialization of 3D FFT codes can be obtained by using a component based approach.*
- *Compiler*
    - A method and tools (code generation and compilation utilities) have been proposed to let non-experts improve the vectorization of their codes to improve their performance.
- *Collective communications*
    - A method to detect tier performance and its inclusion into an MPI All-to-all implementation have been proposed and its benefit in term of performance improvement have been shown on a quantum chemistry code (CPMD)

3 Scalable Numerical Algorithms

3.1 Introduction

Numerical algorithms play a vital role for solving several different big problems encountered in physics, chemistry and mathematics. With the advent of modern supercomputers with thousands of cores and enabling parallel tools that utilize numerical algorithms, it became necessary to reconsider efficiency of these tools running on emerging large-scale systems. This section contains the works that are aimed towards improving parallel performance of the numerical algorithms with several diverse parallel techniques such as utilization of GPU and MIC accelerators (Sections 3.2, 3.6 and 3.8), message-passing paradigm (Sections 3.2, 3.3, 0, 3.6, 3.7 and 0), and shared memory constructs (Section 0). There are algorithmic approaches (Sections 0 and 3.6) to increase the efficiency of widely used numerical algorithms (such as Conjugate Gradient (CG)) as well as adaptive parameter determination and utilization (Sections 3.7 and 0). The target applications and libraries include HYDRO, PETSc, CP2K and FETI.

The first work in Section 3.2 focuses on a cellular automata algorithm that models fluid flows, called the Frish-Hasslacher-Pomeau model. It improves the performance of this algorithm using multiple GPUs with OpenCL. Comparing this approach to a previous work with OpenACC, it is shown that OpenCL has more potential for improvement.

Section 3.3 presents a novel hybrid distributed memory algorithm for nonlinear parameter optimization with parameter pools. This algorithm finds its application in a nonlinear dynamic system which is exemplified as the asset flow differential equations. The optimization objective is to determine the parameters for which the differential equations produce the best fit using daily market prices and net asset values. The presented algorithm can handle large number of parameters and obtains smaller error with better improvement factor.

The work in Section 0 aims to improve the performance of the Coarse-Grain version of HYDRO application by introducing OpenMP tasks into this tool. Using the task dependency concept introduced with OpenMP 4.0, a subdomain synchronization scheme is realized. With this approach, almost perfect scalability is attained.

The work in Section 0 tries to improve the parallel performance of sparse linear iterative solvers through proposing a novel scheme that completely avoids the communication latency overhead of parallel sparse-matrix vector multiplication operations. Basically, the solver is organized in such a way that two different types of communication operations (P2P and collective communications) can be performed simultaneously. Although the proposed methodology is realized in CG solver, the reorganization method is applicable to most of the Krylov subspace methods. The proposed computational rearrangement scheme has no potential to introduce numerical instability to the solver. The simultaneous communication is realized by embedding P2P messages into the collective communication messages. With this approach, it is shown with the large-scale experiments on a Cray XE6 and IBM BlueGene/Q machine that the CG solver can be scaled much better.

Yet another approach for scaling CG iterative solver is presented in Section 3.6. In this work, a pipelined CG solver is used to overlap more computation with communication and is implemented within the FETI application. Besides, the performance of MAGMA LU dense direct solver is evaluated on GPU and MIC accelerators.

Section 3.7 focuses on Finite Element Method (FEM) simulation of thermal and electrical fields. In an effort to reduce the simulation time, rather than using a uniform discretization of the time interval, an adaptive time-stepping scheme is utilized. With the objective of discovering feasible values for the threshold parameters on both structured and unstructured meshes, an IBM BlueGene/P computer is used and the results indicate that the time-stepping approach leads to better scalability and decreases the number of both inner and outer iterations drastically.

Another work that takes advantage of GPU accelerators is summarized in Section 3.8. Specifically aiming at DBCSR library (responsible for performing sparse matrix multiplications) of the CP2K application, the applicability and performance of OpenACC and OpenCL parallel tools are illustrated by comparing them to BLAS, SMM, and MATMUL.

The final work in this subtask focuses on asynchronous solution of sparse linear systems and is explained in Section 0. Aiming at solving the linear elasticity equations in a solid cuboidal block of material, the proposed solver is realized in PETSc tool. In addition, an automatic and adaptive scheme is proposed to determine the parameter for dynamically adjusting local convergence criterion. It is shown that this auto-tuning approach yields better performance compared to the one that uses the fixed parameter.

3.2 FHP library multi-GPU extension with MPI and OpenCL

WP184: *Multi-GPGPU Cellular Automata Simulations using OpenCL*
Authors: Sebastian Szkoda (WCSS, IFT UWr), Zbigniew Koza (IFT UWr), Mateusz Tykierko (WCSS, IIAR PWr)

The aim of this research is to examine the possibility of parallelizing the Frish-Hasslacher-Pomeau (FHP) model, a cellular automata algorithm for modelling fluid flow, on clusters of modern graphics processing units (GPUs). To this end an Open Computing Language (OpenCL) implementation for GPUs was written and compared with a previous, semi-automatic one based on the OpenACC compiler pragmas (S. Szkoda, Z. Koza, and M. Tykierko, Multi-GPGPU Cellular Automata Simulations using OpenACC, http://www.prace-project.eu/IMG/pdf/wp154.pdf). Both implementations were tested on up to 16 Fermi-class GPUs using MPICH3 library for inter-process communication. We found that for both of the multi-GPU implementations the weak scaling is practically linear for up to 16 devices, which suggests that the FHP model can be successfully run even on much larger clusters. Secondly, while the pragma-based OpenACC implementation is much easier to develop and maintain, it gives as good performance as the manually written OpenCL code.

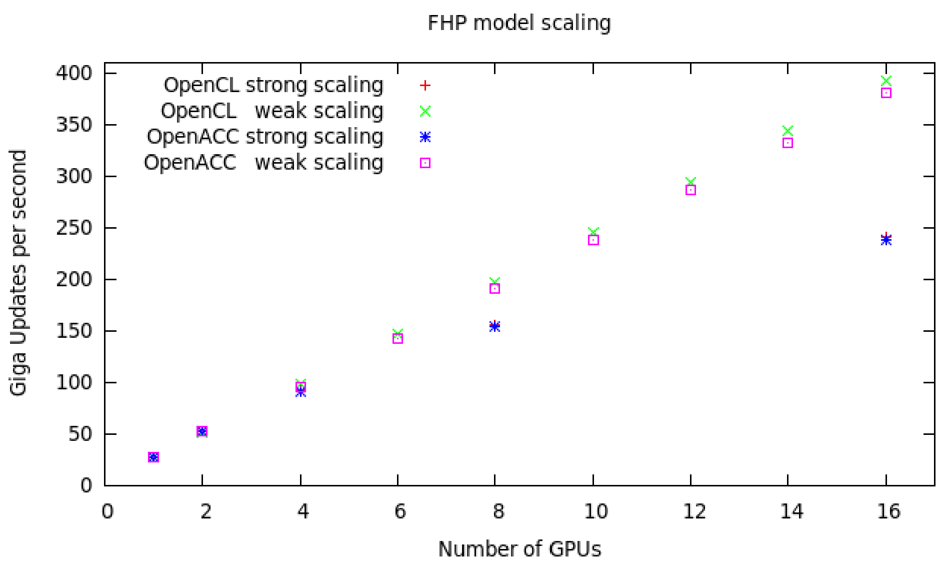


Figure 16: The strong and weak scaling on 16 nodes each with 2 NVIDIA M2050 cards.

Using 8 professional HPC computing nodes with two Nvidia M2050 units each, we could go up to 400 GUps (giga lattice updates per second). The result is one order of magnitude faster than for the single GPU implementation, which still is 4 times faster than the fastest CPU implementation (OpenMP + SSE) on a computing node with Intel Xeon X5670, considering weak scaling. The results for the weak and strong scaling for up to 16 GPUs are shown in Figure 16. The weak scaling is almost linear for both implementations. Figure 16 also presents results for an alternative OpenACC, Multi-GPU implementation, presented in a previous work. Performance results of both implementations are equal, but it is important to remark that the multi-spin implementation considered here is perfectly suited for SIMD devices which allow semi-automatic OpenACC porting to be equally efficient as the hand-written code. This situation is expected to be rare.

3.3 Scalable Parallel Nonlinear Parameter Optimization Algorithm with Parameter Pools

WP185: *Scalable Parallel Nonlinear Parameter Optimization Algorithm with Parameter Pools*
Authors: Ahmet Duran (UHEM-ITU), Mehmet Tuncel (UHEM-ITU)

In this project, we propose a new hybrid algorithm and implement it using MPI. In particular, we study a scalable parallel nonlinear parameter optimization algorithm with parameter pools for a nonlinear dynamical system called the asset flow differential equations (AFDEs) in $\Re^4$. AFDEs have been developed and analyzed by Caginalp and collaborators since 1989 [22], [23] [24], [25], [26]. The algorithm is applicable for parameter optimization of the related nonlinear dynamical system of differential equations with thousands of parameters.

Duran [27] introduced a serial algorithm called the asset flow optimization forecast algorithm. An inverse problem involving parameter optimization for AFDEs was used in order to forecast near term market returns by following an out-of-sample procedure. Duran and [28]. A quasi-Newton (QN) weak line search with the Broyden–Fletcher–Goldfarb–Shanno formula [29] and their semi-dynamic initial parameter pool are utilized in conjunction with daily market prices and net asset values to determine the parameters for which the AFDEs yield the best fit for the previous n days in the optimization procedure. They use nonlinear least-square technique with initial value problem (IVP) approach by focusing on the MP variable P since any real data for the other three variables B, $\zeta 1$, and $\zeta 2$ in the dynamical system is not available explicitly. The gradient (∇ F (x)) is approximated by using the central difference formula, and step length s is determined by the backtracking line search [30].They [28] construct a pool of initial parameters K_i chosen via a set of grid points in a hyper-box. They select an initial parameter vector from the initial parameter pool because the optimization success of quasi-Newton method in the algorithm depends on the initial parameter. Besides the fixed part of various initial parameters, the dynamic part of the pool is updated by adding successful parameters so that they keep a pool of different and most recently used candidate parameters. It is a feasible dynamic multi-start approach without a convexity assumption for their semi-unconstrained optimization problem. The parametric sensitivity analysis was performed by Duran [25]. Later, Duran studied the stability analysis of the AFDEs, in three versions, analytically and numerically.

There are several challenges while studying numerical parameter optimization of the nonlinear dynamical systems. For example, some initial parameters may lead to singularities in the AFDE during parameter optimization process. Our implementation handles these kinds of problems. Moreover, we apply nonlinear optimization technique for arbitrary conditions with various initial parameters in a challenging financial application. Furthermore, for optimization methods using derivatives in a nonlinear model it is important to start the iteration close enough to the potential global minimum to get rid of being caught in a local minimum. There is no strategy that will guarantee the number of necessary iterations to discover the neighbourhood of the global optimum [31]. Therefore, we need sufficiently large number of initial parameters systematically via high performance computing.

In this study, we propose a parallel nonlinear parameter optimization algorithm. One of the novel components of the former algorithm was the presence of the dynamic initial parameter pool that contains most recently used successful parameters, besides the various fixed parameters from a set of grid points in a hyper-box. Therefore, it has dependencies on the most recently used successful parameters.

We use fixed initial parameter pool with more number of parameter vectors so that we can get rid of the dependencies. Unlike the serial algorithm, the new algorithm has a classified initial parameter pool with partitions that can generate different curves having behaviours such as almost steady, uptrend, downtrend, strong uptrend and strong downtrend. In Stage 2, each core performs curve fitting by using its own initial parameters and the steps in the serial algorithm [28] are followed to find the local optimal parameters.

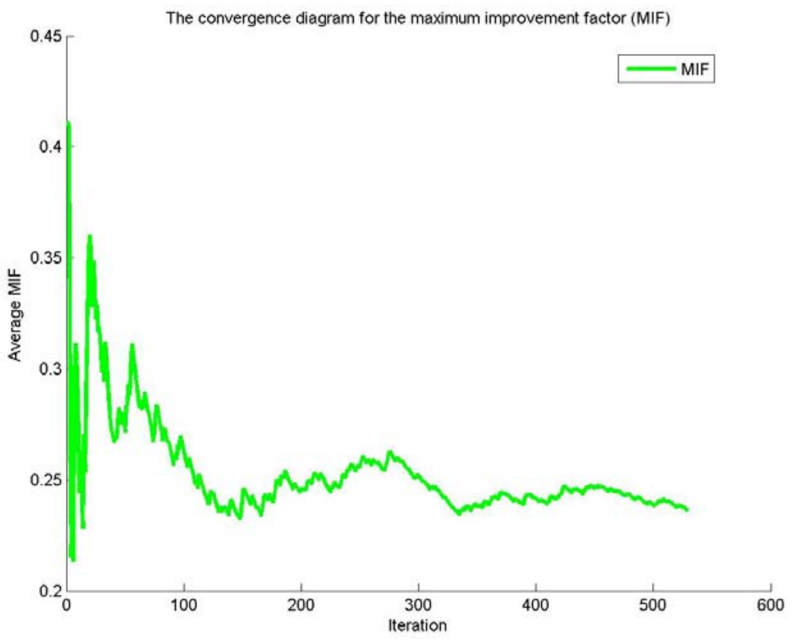


Figure 17: Monte Carlo simulation of the MIF for curve fitting of Price_1k_v2.

<u>Algorithm. The parallel nonlinear parameter optimization algorithm</u>

Stage 1. Obtain classified initial parameter pool having partitions that can generate different curves having various behaviours.
Stage 2. Apply pool partitioning for parallelism. Each core should find the local optimal parameter(s) by using its local initial parameters.
Stage 3. Find the global parameter(s) that can minimize the nonlinear least squares error.

We generate time series pairs as proxy to market price and net asset value by using random walk simulation where the volatilities of the time series are similar to those of real closed-end funds traded on NYSE [25]. See Anderson and Born [32] for more information about the closed-end funds.

Table 9 describes the time series, their volatility behavior and ranges. Table 10 shows the wall time for testing the parallel nonlinear parameter optimization algorithm for 128 cores on the

Linux Nehalem Cluster. Table 11 illustrates the Monte Carlo simulation results for the parameters, the average number of QN iteration, the average nonlinear least squares error and the average maximum improvement factor (MIF) where MIF is used to measure the performance of the optimization process and it is defined as the ratio of the final nonlinear least squares error to the initial nonlinear least squares error. Generally, the smaller MIF corresponds to a better performance, which depends on the closeness of the initial parameter to the optimal one as well. For example, Figure 17 shows the convergence diagram of the MIF for curve fitting of Price_1k_v2.

Time series	Standard deviation	Max	Min
Price_1k_v1	3.94	65.68	48.19
Nav_1k_v1	2.25	58.03	48.32
Price_1k_v2	5.01	67.50	48.77
Nav_1k_v2	2.38	63.94	53.68
Price_1k_v3	2.66	61.68	49.38
Nav_1k_v3	1.81	58.78	49.99
Price_1k_v7	3.63	67.31	51.82
Nav_1k_v7	2.91	61.55	48.45

Table 9: Description of the time series

Time series	Wall clock time (s)
Price_1k_v1	28071.28
Price_1k_v2	27677.68
Price_1k_v3	28421.29
Price_1k_v7	25723.34

Table 10: Wall clock time for 128 cores on the Linux Nehalem Cluster available at UHeM

Time series	Parameters				Average number of QN iteration	Average NLS error	Average MIF
	c_1	q_1	c_2	q_2			
Price_1k_v1	1.9301	18.5099	18.4881	44.5936	166.97	0.0175	0.2109
Price_1k_v2	2.0564	20.8872	16.5765	55.3533	160.32	0.0310	0.2362
Price_1k_v3	2.0438	16.0870	17.5011	39.4577	156.14	0.0171	0.2126
Price_1k_v7	1.5148	21.7521	15.9028	48.8719	135.57	0.0411	0.2859

Table 11: Monte Carlo simulations results

We find that the new parallel algorithm having 512 initial parameter vectors in the classified pool that can generate different curves outperforms the sequential parameter optimization algorithm using dynamic initial parameter pool having up to 80 initial parameter vectors. We obtained smaller nonlinear least squares errors, better maximum improvement factor, and curve fitting for more curve segments, by the advantage of using sufficiently large number of initial parameters methodically.

3.4 Introducing OpenMP Tasks into the HYDRO Benchmark

WP186: *Introducing OpenMP tasks in the Hydro benchmark*
Authors: Jérémie Gaidamour (IDRIS/CNRS), Dimitri Lecas (IDRIS/CNRS), Pierre-François Lavallée (IDRIS/CNRS)

HYDRO [33] is a mini-application which implements a simplified version of RAMSES [34] a code developed to study large scale structure and galaxy formation. HYDRO uses a fixed rectangular two-dimensional space domain and solves the compressible Euler equations of hydrodynamics using a finite volume discretization of the space domain and a second-order Godunov scheme with splitting direction technique.

The OpenMP "Coarse-Grain" version of HYDRO described in [35] exploits data locality by using a 2D domain decomposition of the global domain. The implementation is very similar to the MPI version of HYDRO as each thread is responsible for the computation of a local subdomain. The algorithm used on each subdomain is described in Figure 18. For the interface cells, synchronization between threads is needed, as a thread must read (step (1) of the algorithm in Figure 1) the initial values of its neighbour subdomains (ghost cells) before neighbour threads update them (step (3) of the algorithm). Before step (3), threads have to wait until step (1) of the two neighbour domains is complete. This thread-level synchronization is a barrier involving threads grouped in threes, the implementation of this barrier involves flushing the buffer with the OpenMP FLUSH.

```
U(i , j) is the 2D grid buffer for conservative variables.
For each time step n :
- Apply boundary conditions
- PARALLEL LOOP:  For each column j of a subdomain:
        (1) READ - Copy the values of the j column into a 1D temporary buffer.
            The buffer holds conservative variable values of the previous time step ($U_{n-1}(:,j)$).
        (2) COMPUTE - Compute the new grid values ($U_n(:,j)$) only from the temporary buffer
            (ie: compute primitive variables, solve Riemann problem at cell interfaces and compute incoming fluxes).
        (3) WRITE - Copy $U_n(:,j)$ in global U.
- PARALLEL LOOP:  For each row i of a subdomain:
        […]
```

Figure 18: 1D Godunov time step routine in the column and row direction (pseudo-code).

The version 4.0 of the OpenMP specification [36] introduces the concept of tasks with dependencies. The goal of this work was to introduce OpenMP tasks in the "Coarse-Grain" version of HYDRO. We aim at implementing the subdomain synchronization we described previously using task dependencies.

Figure 19 and Figure 20 show how we implement the symchronization using task dependencies. A unique task is responsible for reading all the interface cells between two domains. Every WRITE kernel directly depends on a single COMPUTE kernel and the dependency graph can be further simplified by merging COMPUTE

Table 12 and Table 13 show a scalability evaluation of the OpenMP versions of HYDRO. The experimental platform is the supercomputer Ada (CNRS/IDRIS-GENCI) composed of IBM System x3750 M4 compute nodes.

The scalability of the "Coarse-Grain" version is nearly perfect due to efficient cache utilization. Threads work fully in parallel on their own portion of data and no time is wasted at the barriers. This performance gain comes at the cost of a greater code complexity.
The scalability of the new "OpenMP tasks" version is not as good as the finely tuned "Coarse-Grain" version since we are no longer in control of the affinity between threads and subdomains.

The OpenMP specification does not define how the scheduling of tasks should be done by runtime systems [37] and very little information is available concerning the actual strategy of our OpenMP library. As for now, it is not possible to select the task scheduler strategy at the user level but it would be useful to be able to give hints to the scheduler in the same way we control the scheduling of parallel loop or the CPU affinity of threads in OpenMP.

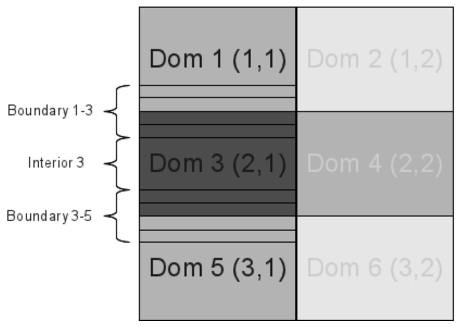


Figure 19: The domain decomposition with the domain coordinates and a simplified description of the interfaces.

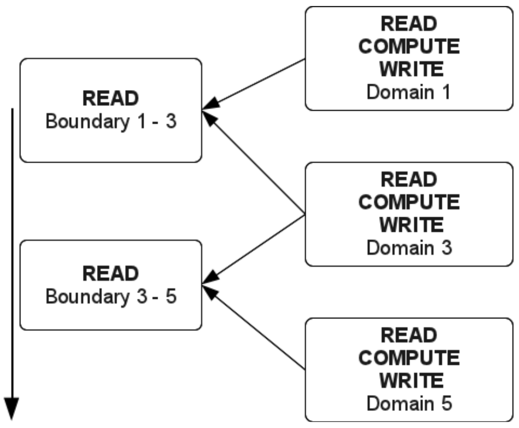

READ Boundary 1 - 3
READ Boundary 3 - 5
READ COMPUTE WRITE Domain 1
READ COMPUTE WRITE Domain 3
READ COMPUTE WRITE Domain 5

Figure 20: Dependency graph of the implemented algorithm.

	Time (s)	Speedup	Efficiency
1	**377.15**	1.00	100.00 %
2	**188.41**	2.00	100.09 %
4	**98.63**	3.82	95.60 %
8	**52.18**	7.23	90.35 %
16	**24.72**	15.26	95.36 %
32	**11.91**	31.67	98.96 %

Table 12: Scalability of the “Coarse-Grain” version of HYDRO

	Time (s)	Speedup	Efficiency
1	**376.81**	1.00	100.00 %
2	**188.99**	1.99	99.69 %
4	**98.65**	3.82	95.49 %
8	**56.72**	6.64	83.04 %
16	**32.96**	11.43	71.45 %
32	**20.01**	18.83	58.85 %

Table 13: Scalability of the “OpenMP tasks” version of HYDRO

The tasking model allows expressing a complex algorithm with ease and using tasks was less intrusive than implementing the algorithm with busy-waiting. We plan to extend this work with a more complete analysis of the performance and by investigating specialized task-based programming environments and runtime systems.

3.5 Reducing Synchronization Overhead of Conjugate-Gradient-type Parallel Iterative Solvers

WP187: *Reducing Synchronization Overheads in CG-type Parallel Iterative Solvers by Embedding Point-to-point Communications into Reduction Operations*
Authors: R. Oguz Selvitopi (Bilkent), Cevdet Aykanat (Bilkent)

Iterative solvers are widely used and adopted to solve sparse linear systems of equations on modern large-scale parallel systems. In these systems, the communication requirements of the solver generally become the main bottleneck for obtaining a good scalable performance. For this reason, the coefficient matrix is usually processed in a pre-processing phase which involves partitioning of this matrix to reduce the communication requirements. In the literature, the most often used and optimized communication metric is the communication volume [38], [39].

In iterative solvers, there are two types communication that are repeated through all iterations:

- *Collective communication operations*: This type of communication is used to gather the results of the inner product computations at all processors and requires all processors to join the communication. The MPI equivalent of this operation is the MPI_Allreduce (hereafter referred to as ALL-REDUCE) with the summation being the reduction operator.
- *Irregular point-to-point (P2P) communication operations:* This type of operation is used to communicate the entries of the input and/or output vector of the sparse-matrix vector multiplication (SpMV). The irregular sparsity pattern of the coefficient matrix causes irregular task-to-task interaction between parallel processes. They are generally performed by simple MPI primitives, e.g., MPI_Send , MPI_Recv, and their variants.

We devise a computational reorganization method to perform P2P and collective communication operations simultaneously. This allows the synchronization points in a single iteration of the solver drop from two to one for a single pair of SpMV and its follow-up inner product(s). We use a modified Conjugate Gradient (CG) iterative solver to show the validity of the proposed methods. We use 1D partitioning of the matrix and test the solver on JuQueen and Hermit up to 2048 cores with the matrices selected from UFL sparse-matrix collection [40]. The proposed computational rearrangement scheme has no potential to introduce numerical instability to the solver, because it is mainly based on performing a negligible amount of redundant computation per processor.

In parallelization of CG, the coefficient matrix is generally row-wise decomposed and distributed among the processors. Without computational reorganization, there are two separate communication: (i) one of them is the P2P communications prior to SpMV computations and (ii) the other one is the collective communication after the local inner products. This parallel algorithm has two synchronization points due to these P2P and collective communication phases. In the proposed alternative parallelization with computational reorganization, the input vector of the SpMV computations is not formed with the P2P communications but it is formed with the help of the other vectors. Instead of communicating this input vector, the output vector is communicated and it is augmented with the entries that are received from other processors. This augmented output vector is then subjected to the same linear vector operations to perform the augmented input vector of the SpMV, requiring no further communication. This reorganization enables P2P communications to be performed right after collective communication operations, reducing two separate communication phases into one.

We realize the opportunity provided by the computational reorganization by performing these two types of communication phases in a single one. The P2P and collective communications are performed simultaneously by embedding messages of P2P communications into the communication pattern of the algorithm used for the ALL-REDUCE algorithm. In other words, the latency overhead due to the P2P communications is completely eliminated by using the messages that are already transmitted for ALL-REDUCE.

Embedding messages of P2P communications into collective communications have the following implications:

- Startup costs of all messages due to P2P communications are completely avoided.
- An exact bound on the maximum and average number of messages is provided, which is $\log P$ for a parallel system with P processors. This is a significant advantage and is actually the key factor in obtaining a good scalable performance at large processor counts.
- Communication volume increases due to the store-and-forward scheme required by the embedding.
- Embedding scheme requires buffering due to the store-and-forward scheme.
- There is a trade-off between avoiding latency costs and increasing communication volume. Here, the former is favoured, because, as will be shown with the experiments, message latency becomes the dominating factor in determining the communication costs with increasing number of processors.

The store-and-forward scheme used in embedding contents of P2P messages into the messages of collective communication operations may increase communication volume. If total number P2P messages is low, this can be a bottleneck in obtaining a good scalable performance. We present two heuristics to further reduce this increased communication volume. Objective of both of the mapping heuristics is to keep the pairs of processors that communicate a large volume of data close to each other. The closeness notion here refers to the communication pattern used for the ALL-REDUCE algorithm. Both of the heuristics are Kernighan-Lin (KL) [41] type of algorithms which try to find a good mapping by a number of successive swap operations:

- *KLF*: Use full neighbourhood information with $P(P-1)$ possible swaps.
- *KLR*: Restrict the search space to the processors that directly communicate, thus reducing the number of possible swaps to $P\lg P/2$.

For more detail on these heuristics, refer to [42].
We compare four schemes in our experiments (i) Conventional parallelization of conjugate gradient solver (*CONV)*, (ii) Alternative parallelization with computational reorganization (*EMB*), (iii) Alternative parallelization with computational reorganization and mapping algorithm *KLF* (*EMB-KLF*), (iv) alternative parallelization with computational reorganization and mapping algorithm *KLR*. We used PaToH [41] to partition all matrices prior to execution. Two parallel systems are used in the experiments: Cray XE6 (XE6) and IBM Blue Gene/Q (BG/Q). The obtained speedup results for the **pcrystk02** matrix with 968,583 nonzeros are illustrated in Figure 21.

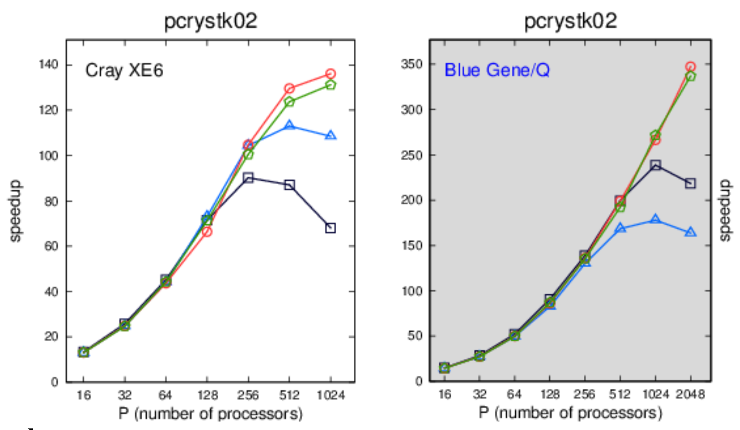


Figure 21: Speedup values.

With increasing number of processors, the proposed three schemes usually scale much better compared to conventional parallelization. These speedup values validate that startup costs become more important with increasing number of processors and to obtain a good scalability performance, it is paramount that latency should be considered as a separate stand-alone optimization objective. The results of this task have been recently accepted for publication in the prestigious *IEEE Transactions of Parallel and Distributed Systems* journal. Details and more test results can be found in this publication [42].

3.6 Scalability improvement of the projected conjugate gradient method used in FETI domain decomposition algorithms

WP198: *Scalability improvement of the projected conjugate gradient method used in FETI domain decomposition algorithms*
Authors: Tomáš Kozubek (VSB), David Horák (VSB), Václav Hapla (VSB), Lubomír Říha (VSB)

FETI (Finite Element Tearing and Interconnecting) type domain decomposition methods are powerful tool for constructing numerically and parallel scalable solvers for real linear and nonlinear engineering problems combining iterative and direct solvers. In many cases the resulting algebraic formulation leads to a quadratic programming problem with convex constraints. Solvers for such problems including our own FETI method called Total FETI are developed at IT4Innovations, VSB-Technical University of Ostrava. We have implemented many of these solvers (TFETI, etc.) into an in-house **FLLOP** (FETI Light Layer On top of PETSc) library being implemented as an extension of the PETSc framework. It is primarily used for solving constrained quadratic programming (QP) problems on parallel computers. To allow other packages to use FLLOP solvers a general C array based FLLOP_AIF interface has been also implemented.

FETI methods use the Lagrange multipliers λ to enforce equality constraints (gluing conditions) in the original primal problem: $\min \frac{1}{2} u^T K u - u^T f \;\; s.t. \;\, Bu = o$. The primal problem is then transformed into significantly smaller and better conditioned dual equality constrained problem: $\min \frac{1}{2} \lambda^T F \lambda - \lambda^T d \;\; s.t. \;\, G\lambda = o$, with $F = BK^+B^T$. This problem can be solved by means of projectors $P_G = I - Q_G$, $Q_G = G^T(GG^T)^{-1}G$ and CG method applied to the projected system $P_G F \lambda = P_G d$. For this dual problem the classical estimate of the spectral condition number by Farhat, Mandel, and Roux is valid, i.e. $\varkappa(P_G F | Im\, P_G) \leq C\frac{H}{h}$, with H denoting the decomposition and h the discretization parameter. To be able to fully utilize the massively parallel computers it is essential to maximize the number of subdomains (decrease H) which leads to the reduction of the subdomain size. This improves the performance due to following reasons: (1) a subdomain stiffness matrix size is reduced which speeds up both factorization and subsequent pseudo-inverse application; and (2) the conditioning is improved which reduces the number of iterations. The negative effect of the large number of subdomains is the increase of dual and null space dimension which slows down the coarse problem (CP) solution (solution of the system $GG^T x = y$). Therefore in this case the bottleneck of the TFETI method is the application of the projector which dominates the solution time. Several parallel direct solvers have been already tested for CP processing in other Work Packages.

The main objective of our work within the extension of WP12 PRACE-2IP is to achieve better scalability of FLLOP's FETI solvers. A performance evaluation of two new techniques is presented: (1) a novel pipelined implementation of CG (**PIPECG**) method in PETSc and (2) a **MAGMA LU** solver running on following many-cores accelerators: GPU Nvidia Tesla K20m and Intel MIC Xeon Phi 5110P.

The following tests have been performed:

1. the performance comparison of CG vs. PIPECG when applied to the following problems:
   a) primal block-diagonal, assembled ($A = K, b = f$),
   b) primal decomposed and penalized, unassembled – PFETI ($A = K + \rho B^T B, b = f$),
   c) dual decomposed, unassembled – TFETI ($A = P_G F, b = P_G d$).
2. the performance evaluation of the following functions from the MAGMA library designed for many-core accelerators (GPU and Intel MIC):
   a) LU factorization,
   b) solve function,

   running on (1) multicore CPU only ; (2) combination of CPU and GPU; and (3) combination of CPU and MIC. LU factorization and triangular solves are the essential functions used for the solution of the CP ($A = GG^T, b = y$).

The numerical experiments have been performed using **Anselm** (Bull cluster at IT4Innovations – used for GPU and MIC tests with MAGMA library) and **Sisu** (Cray XC30 at CSC Helsinki – used for large scalability tests of PIPECG algorithm). For the numerical testing a loaded **elastic cube** is used. To be able to test our FETI solvers on large scale problems, that are expected to run on exascale machines, we have used a parallel PermonCube benchmark developed by our team at IT4Innovations. It enables to generate data of large-scale problems decomposed into thousands of subdomains in parallel.

Numerical experiments for tests 1a – 1c were done with **PIPECG** implementation available in PETSc 3.4. The main idea of PIPECG is "talk less and work more", in this case it is the hiding the communication needed in two dot products computation behind the matrix by vector multiplication, some additional work with several additional auxiliary vectors compared to CG is necessary. The success of PIPECG profits from the balancing of these two operations, if the multiplication dominates and dimension of the problem is huge, i.e. also the dimension of additional vectors is huge, then PIPECG can have even worse performance than the general CG in our case. PIPECG brings significant time savings for cases 1a and 1b, if the number of subdomain elements is less than 11^3- see Figure 22. PIPECG is in both cases worse than CG for large problems as the benefit from the overlap of matrix by vector multiplication and two dot products computation is eliminated. The first operation exceeds significantly the second one and an additional work with longer vectors is more and more significant. In case 1c there is no performance difference between the general CG and the pipelined CG algorithm due to excessive communication contained in the dual operator $P_G F$:

(1) multiplication by a constrained matrix gluing the subdomains together; (2) multiplication by the transpose of this matrix; and (3) application of the projector. With respect to our current analysis it seems that pipelined CG will be ideally suited for the parallel solution of the TFETI CP ($A = GG^T, b = y$) – it is very promising and it will be the topic for the further research.

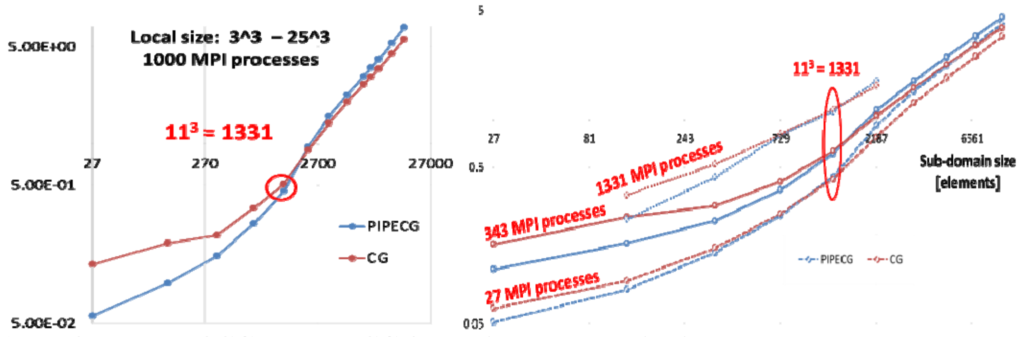


Figure 22: Performance of CG vs. PIPECG for various subdomain sizes.

The performance evaluation of the **MAGMA LU** dense direct solvers applied to CP was evaluated on a single compute node of Anselm supercomputer. Three hardware configurations have been tested: (1) CPU only (16 OpenMP threads – two 8-core CPUs); (2) CPU + GPU (16 OpenMP threads + 1 Tesla K20m GPU), and (3) CPU + MIC (16 OpenMP threads + 1 MIC - 240 OpenMP threads). Concerning the largest CP having dimension 24,576, the following observations have been made: (1) LU factorization: CPU + MIC is 2 times faster than CPU (22.5 sec); CPU + GPU is 4 times faster than CPU (11.3 sec); CPU only (46.1 sec); (2) 100 calls of the triangular solves: CPU + MIC is 3 times faster than CPU (8.29 sec); CPU + GPU is 1.02 times faster than CPU (22.75 sec); CPU only (23.29 sec). To conclude, the combination of CPU + MIC is more efficient if large number of solve calls is required while CPU + GPU is better if solver can find a solution in smaller number of iterations.

3.7 Computer modeling and simulations in strongly heterogeneous nonlinear media

No whitepaper
Authors: K. Georgiev, N. Kosturski, S. Margenov, Y. Vutov

This work concerns the Finite Element Method (FEM) simulation of thermal and electrical fields in strongly heterogeneous nonlinear media on structured (voxel) and unstructured (tetrahedral) meshes. Mass and heat transfer and coupled electrical processes involved in the radio–frequency (RF) hepatic tumor ablation are considered. Instead of a uniform discretization of the considered time interval, an adaptive time-stepping procedure is applied in an effort to decrease the simulation time. The procedure is based on the local comparison of the Crank Nicholson and backward Euler approximations.

The new results in the development of the computer models and their parallel implementations include: *a*) a scalable high-performance adaptive time stepping algorithm for simulating the radio-frequency ablation for treatment of liver tumors; *b*) implementation and tuning of the new computer modules on the IBM Blue Gene/P computer in Sofia both on structured and unstructured meshes; *c*) founding a suitable set of values for the algorithm threshold parameters.

The minimally invasive treatment called radio-frequency ablation (RFA) guided by imaging techniques, the doctor inserts a thin needle through the skin and into the tumor. High-frequency electrical energy delivered through this needle heats and destroys the tumor. The circuit is closed with a ground pad applied to the patient's skin. The right procedure parameters are very important for the successful killing of all of the tumor cells with minimal damage on the non-tumor cells.

Computer simulation on geometry obtained from a magnetic resonance imaging (MRI) scan of the patient is performed.

The IBM Blue Gene/P computer, located at the Bulgarian Supercomputing Center, is used for the simulations and numerical experiments with the new adaptive time stepping algorithm. This machine consists of two racks, 2048 Power PC 450 based compute nodes, 8192 processor cores and a total of 4~TB random access memory. Each processor core has a double-precision, dual pipe floating-point core accelerator. Sixteen I/O nodes are connected via fiber optics to a 10~Gbps Ethernet switch. Number of runs were done both using 128 and 1024 processors. Uniformly refined mesh was used for the runs on 1024 processors.

The experimental results show that the new algorithm is scalable. The tests allowed us to find some suitable parameters and showed the practical usefulness of the developed solver for such kind of computer simulations. One can observe that the computing time is decreased more than three times, the number of outer iterations is decreased from 420 to 71, and the number of inner iterations decreases from 2233 to 535.

3.8 Optimization of CP2K DBCSR library for GPU with OpenCL

No whitepaper
Authors: Mariusz Uchroński (WCSS), Marcin Gębarowski (WCSS), Agnieszka Kwiecień (WCSS), Franciszek Klajn (WCSS)

P2K [43] is an open-source application designed for atomistic and molecular simulation of solid state, liquid, molecular and biological systems. A scalability of CP2K has been tested and has shown good results [44]. The code is written in Fortran 95 and parallelized mainly with MPI but in some parts, also with hybrid techniques, like MPI/OpenMP [45] and CUDA. Recent works report promising results also for OpenACC and OpenCL implementations of a DBCSR library [46], [47] and show a successful usage of the Intel Xeon Phi accelerators [48].

The main goal of this project is to optimize the DBCSR library, which performs sparse matrix multiplications, using OpenCL [49] and check the possibility to improve the OpenACC [50] implementation. OpenCL and OpenACC are both open standards and have been successfully used in previous works on the library. During this project, we worked on CP2K v2.4.

Introducing OpenACC into the code requires usage of a compiler which understands the OpenACC pragmas. There are only three such compilers so far, all commercial, delivered by PGI, CAPS and Cray. The CP2K code compilation with PGI [51] has been reported as problematic [46] , and to fully utilize the DBCSR OpenACC port within the application some additional work had to be done. We have performed a code compilation attempt with PGI 14.1 and with a few changes of the source code managed to provide a working solution. The PGI 14.1 does not fully support Fortran 2008 extensions, so every call to the built-in erfc function had to be changed to its implementation delivered in the other part of the CP2K code. Such code worked fine, but only if it was compiled with no optimization. A compilation with anyone of the optimization flags, e.g. "-fastsse", produced executables which ended with a segmentation fault. Further analysis with the PGDBG debugger [52] revealed that the PGI Fortran compiler is not able to correctly allocate the memory for temporary arrays used in an array multiplication:

```
ALLOCATE(a(n,n),w(n),work(lwork),STAT=ierr)
CPPostcondition(ierr==0,cp_failure_level,routineP,error,failure)
a(1:m,1:m) = MATMUL(TRANSPOSE(umat(1:n,1:m,l)),MATMUL(hmat(1:n,1:n,l),umat(1:n,1:m,l)))
```

To resolve the issue a further modification of the code was required. An explicit declaration and allocation of temporary arrays has led to a successful run after compilation with optimizations:

```
ALLOCATE(a(n,n),b(m,n),c(n,m),w(n),work(lwork),STAT=ierr)
b(1:m,1:n) = TRANSPOSE(umat(1:n,1:m,l))
c(1:n,1:m) = MATMUL(hmat(1:n,1:n,l),umat(1:n,1:m,l))
a(1:m,1:m) = MATMUL(b,c)
```

At the same time we attempted to use the CAPS Compiler Suite in version 3.4.4 to compile CP2K code with OpenACC pragmas. The compilation was performed as specified in the CAPS documentation: FC = hmpp -d gfortran. A part of the code was compiled successfully, but some of the Fortran syntax was not recognized by belfort (part of the suite), populating the following:

```
belfort: [Error BGFT0071] Illegal module procedure list_timerenv_init at
/cp2k/src/list.F, line 30
```

The code which caused the error is as follows:

```
INTERFACE list_init
    MODULE PROCEDURE list_timerenv_init, list_routinestat_init, &
        list_callstackentry_init
END INTERFACE
```

The code is syntactically correct and is compiled successfully with GNU and PGI compilers. The error made it impossible to use the CAPS compilers for CP2K and its OpenACC extension, as an alternative for the PGI. In addition, in May 2014 an official statement of CAPS has been announced to all its customers, that the company will be closed due to financial problems, and no new licences or support will be provided after the end of June, 2014.

An analysis of the initial OpenCL implementation [46] revealed that the approach to memory allocation and data distribution need to be changed to better utilise the GPU architecture. As a result a new OpenCL kernel has been implemented for the DBCSR library, and integrated into the application code, as an OPENCL driver. A specific function for data partitioning has been implemented and the portions of data sent to the GPU has been enlarged which reduced the transfer times. The data distribution between the work groups on the GPU allows usage of the shared memory to store elements of the matrices needed during a multiplication. The block multiplication algorithm in the DBCSR library requires using of memory locks (semaphores), when executed on GPUs. This is undesirable, as threads writing to the same memory area are blocking each other, slowing down the execution. A solution for this is a specific assignment of small block matrices to threads within the kernel, in such a way that to each work group a unique fragment of the output matrix is assigned. The index table for data distribution was sorted and slices distributed in such a manner that two groups do not work on the same slice of C matrix, if possible. The number of conflicts between working groups has been reduced, but the semaphores are still used, to ensure that only one work group can write to the slice. It is planned to work further on removing the conflicts and semaphores completely.

The experimental results for the DBCSR library are presented in Table 14. We compare the OpenCL and OpenACC ports with BLAS, SMM and MATMUL drivers from CP2K.

Nblocks	BLAS	SMM	MATMUL	OPENACC	OPENCL
10k	0.68	0.49	0.95	3.52	0.16
40k	1.15	0.98	1.92	3.95	0.99
90k	1.63	1.47	2.88	4.26	1.34
160k	2.16	1.96	3.83	4.85	1.70
250k	2.60	2.45	4.81	5.48	2.08
360k	3.09	2.94	5.83	6.01	2.51
490k	3.61	3.43	6.71	6.53	2.84
640k	4.10	3.94	7.68	7.07	3.26
810k	4.63	4.44	8.65	7.54	3.77
1000k	5.07	4.95	9.61	7.98	4.09

Table 14: DBCSR library tests results for different drivers (Exec times in seconds)

Results presented in Table 14 show that difference between execution times for OpenCL and SMM driver (drivers with shortest execution times) increases with the problem size (Nblocks). The OpenACC performs worst for small problem sizes but it is better than MATMUL for bigger problems.

We have also used a dbcsr_mm test from the CP2K distribution, and the results are presented in Table 15.

Input	BLAS		SMM		MATMUL		OPENACC		OPENCL	
	S	Mflops	s	Mflops	S	Mflops	S	Mflops	s	Mflops
dbcsr_mm	2.98	770.6	2.94	781.5	29.54	77.4	150.83	14.9	3.04	755.22

Table 15: CP2K tests results for dbcsr_mm test case

Execution times of the application for OpenCL, BLAS and SMM drivers shown in Table 15 are almost the same. The results obtained with OpenACC are worse than OpenCL in every case, as there is a limited possibility to influence the data transfer and memory allocation strategies with pragmas. An attempt was made to introduce the pragmas outside the small matrix multiplication loops to limit the unnecessary data transfers, but the compiler-generated kernels have produced wrong results. Further work could combine OpenACC pragmas (for matrix multiplication) with CUDA kernels (for data preparation), to achieve a better performance.

The BLAS used in the testing was ACML 5.3.1, FFTW 3.3.4, and the PGI 14.3 compiler (with compilation flags –fastsse –tp amd64 and –acc).

For development and testing we used the Supernova system, located at Wrocław Centre for Networking and Supercomputing (WCSS). Supernova serves as a Tier-1 machine within PRACE infrastructure. For testing we used fat node with four sixteen-core AMD Opteron 6274 processors with 256 GB of memory and two NVIDIA Tesla M2075 (448 cores, 6 GB of memory) per node. For OpenCL code development we also used a node with two NVIDIA GTX 480.

3.9 Asynchronous solution of sparse linear systems

No whitepaper
Authors: Mark Bull (EPCC)

In third year, we have continued the work reported in [53], on solving sparse linear systems using asynchronous multisplitting algorithms.

Firstly, we have implemented a more realistic, and somewhat more complex test case, which solves the linear elasticity equations in a solid cuboidal block of material. The physical system is discretised into cubic elements interconnected at nodes. The nodes are numbered $1, \ldots, n$ and each node has three degrees of freedom (in x, y, and z dimensions). The system of equations that is solved is:

$$Kx = f$$

where x is a $3n$ vector of displacements, f is the vector of forces acting on each of the nodes in each degree of freedom, and K is the global stiffness matrix which links the forces to the displacements. In the finite element approach each node is only coupled to neighbouring nodes. In this model each node has a maximum of 27 neighbours (nodes which share an element), so there are a maximum of 81 non-zero entries per row. As the number of nodes grows, K becomes increasingly sparse. We have validated our solver against PetSc and a bespoke finite element code for this test case.

Secondly, we have worked on a scheme for automatic run-time tuning of a key parameter in the algorithm. For each iteration it is necessary to decide how accurately to solve the inner block system before exchanging halo data again.

This can either be done by choosing a fixed number of iterations of the Krylov solver, or by iterating until the block residual has been reduced by a given amount. In [53] we showed that choosing a fixed number of inner iterations for the entire solve is not straightforward. If the number of iterations is too small, then too much time is spent exchanging inaccurate data; if it is too big then time spent obtaining spurious local accuracy is wasted. Moreover, the optimum value changes as the system converges.

We dynamically adjust the local convergence criterion as follows: we select a local relative residual as the criterion instead of a number of iterations, since this appears to be more robust. We assess the local progress of the algorithm by calculating the *local progress rate* as the local relative change in residual per second for the latest inner solve, and smooth this value both over outer iterations and over neighbouring blocks. We then adjust the local convergence criterion to try to maximise this local progress rate.

Figure 23 shows the results of using our autotuning scheme compared with fixing a number or inner iterations. We can see that the initial convergence rate is about five times faster with autotuning compared to its counterpart which does not use autotuning.

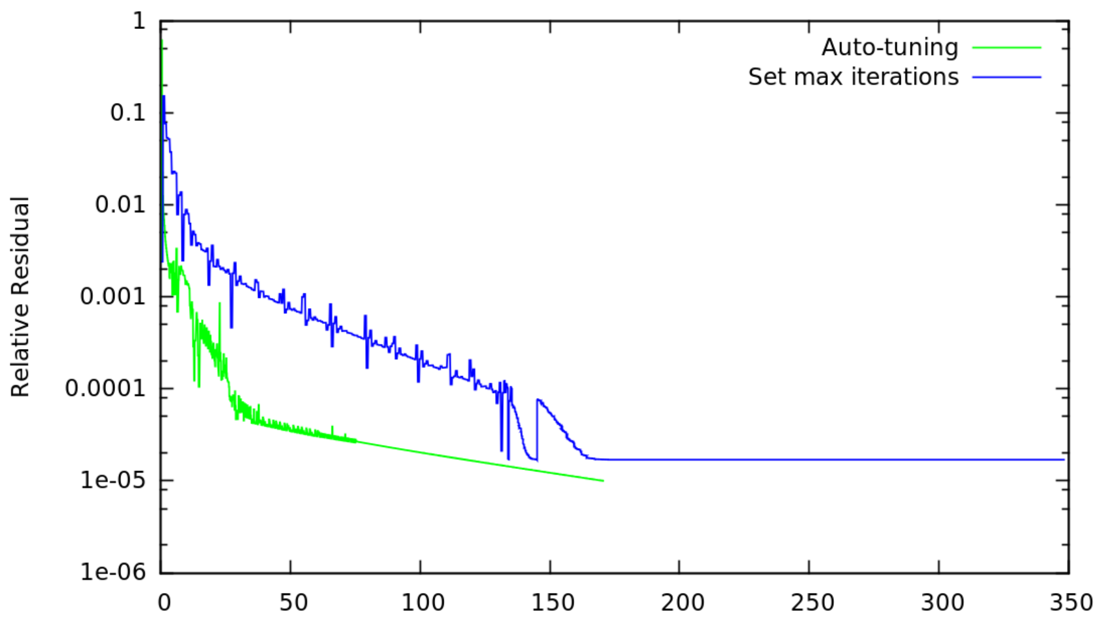


Figure 23: Convergence rate for asynchronous multisplitting.

3.10 Conclusion

Subtask 12.2 has successfully exploited various applications and libraries that utilize numerical algorithms such as HYDRO, PETSc, CP2K and FETI. The performance of various numerical algorithms is improved via using new parallel paradigms within accelerators or shared memory constructs and novel algorithmic strategies. GPU and MIC accelerators are utilized with the OpenACC and OpenCL enabling parallel constructs. Besides, shared memory parallelism with OpenMP is exploited to improve efficiency of HYDRO application. The approaches for numerical algorithms also include distributed memory parallelism, example applications including PETSc and FETI.

The 8 projects in Subtask 12.2 have resulted with 5 whitepapers and 3 detailed reports. In addition to this deliverable, all whitepapers will be available online and will prove useful for users facing similar algorithmic challenges. In summary, Subtask 12.2 has provided new guidelines, algorithmic approaches and adaptive methodologies to improve the performance of numerical algorithms on modern large-scale systems.

4 Development environments and tools

4.1 Introduction

Future high-end HPC platform will induce constraints associated with their huge amount of components. Introducing checkpoint in HPC application has to be addressed, dealing with the threat of an unaffordable overhead .

Fault tolerance is clearly stated as a critical issue for multi-petascale and future Exascale systems. According to the EESI1 roadmap, the current approach that consists in saving the full execution state on remote file system is responsible for significant overhead in Petascale systems and will not scale at the Exascale level. In this prototype, we have tested the performance, scalability, and overhead of a new approach called Advanced Multilevel Fault Tolerance (AMFT) combining existing application based checkpointing and multilevel checkpointing capabilities provided by using different storage hierarchies from local storage (standard HDD, hybrid HDD/SSD, regular SSD and optimized SSD) to "remote" parallel file system.

This work is a follow up of D9.3.4 from 1P-WP9 AMFT, as part of the objectives of 2IP-WP12.

4.2 FTI : Basic Description

Fault tolerance and application resiliency will be a key issue for next multi-petascale and Exascale system as identified by many recent reports including IESP[1] and EESI[2]. As the evolution of the networks and the bandwidth of the parallel filesystems will not scale as needed, it will be impossible to checkpoint a full system image at an appropriate frequency (for dealing with a low expected MTTI[3]).

One solution consists in implementing application-based checkpoint/restart (in order to reduce the footprint of the checkpointed data to save, just the key variable states) and to use in a smart way the different levels of storage hierarchies available on HPC systems for performing asynchronous high frequency checkpoint restart.

Application based checkpoint restart can be realized by coding explicit subroutines into source codes for storing pertinent data or through directives for assisting smart runtime systems in saving pertinent data structures.

The FTI middleware (Fault Tolerance Interface) co-developed by the INRIA-Illinois joint laboratory on Petascale computing and Tokyo Institute of Technology will be used as the multilevel checkpointing middleware.

The objectives of this prototype are to assess on different hardware platforms the interesting potential of FTI and AMFT on new profiles of applications coming from the PRACE benchmarks, the newly EUABS (European Unified Applications Benchmark Suite) or applications proposed by community codes from 1IP-7.2 or 2IP-WP8.

Criteria like the amount and complexity of work to adapt the target applications, performance and scalability of such applications, overhead of using FTI and its level of maturity will be

[1] International Exascale Software Project : http://www.exascale.org/iesp/IESP
[2] European Exascale Sotware Initiative : http://www.eesi-project.eu
[3] Mean Time To Interrupt

assessed in order to envision "industrializing" and making it as a standard package of the PRACE software stack.

The FTI middleware co-developed by the INRIA-Illinois joint laboratory on Petascale computing and Tokyo Institute of Technology was used as the multilevel checkpoint middleware. FTI is a research prototype composed of a programming interface and a runtime environment. FTI is a portable package and its implementation is totally agnostic to the target application since FTI API functions can be used by simply linking with the FTI library.

FTI can be used with applications already featuring application level checkpointing or with applications that do not provide any support for fault tolerance. To adapt applications featuring application level checkpoint/restart the programmers replaces the checkpoint and restart calls existing in the application by FTI checkpoint and restart function calls. For other applications, the programmer replaces checkpoint/restart using the FTI API in the same way one would implement application level checkpoint but avoiding the complexity of the multilevel resiliency, garbage collector and metadata management. In addition, FTI proposes several configuration parameters that can be easily set up in a configuration file.

FTI is based upon protection levels as described below,

- L1 : FTI performs checkpointing on local storage without any concern to resist a failure of this support.

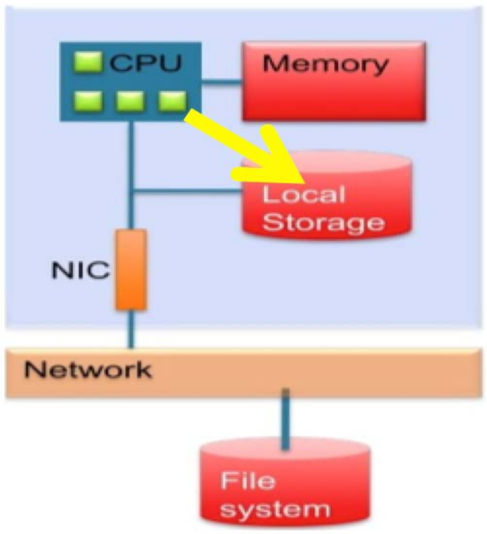


Figure 24: FTI – L1 "Basic write".

- L2 : FTI does a local checkpoint and duplicates to a partner node. This mode allows an application to resume from any failure involving one node in each partnership.

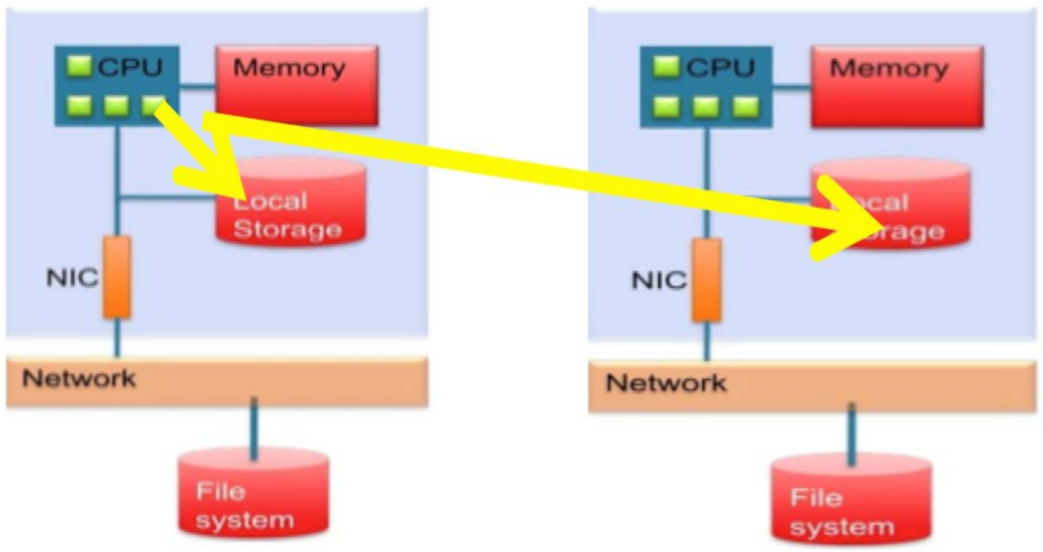


Figure 25: FTI – L2 "Partner Copy".

- L3 : FTI dispatches checkpoints to a group of multiple nodes. The backup is encoded with other processes using a Reed-Solomon algorithm.

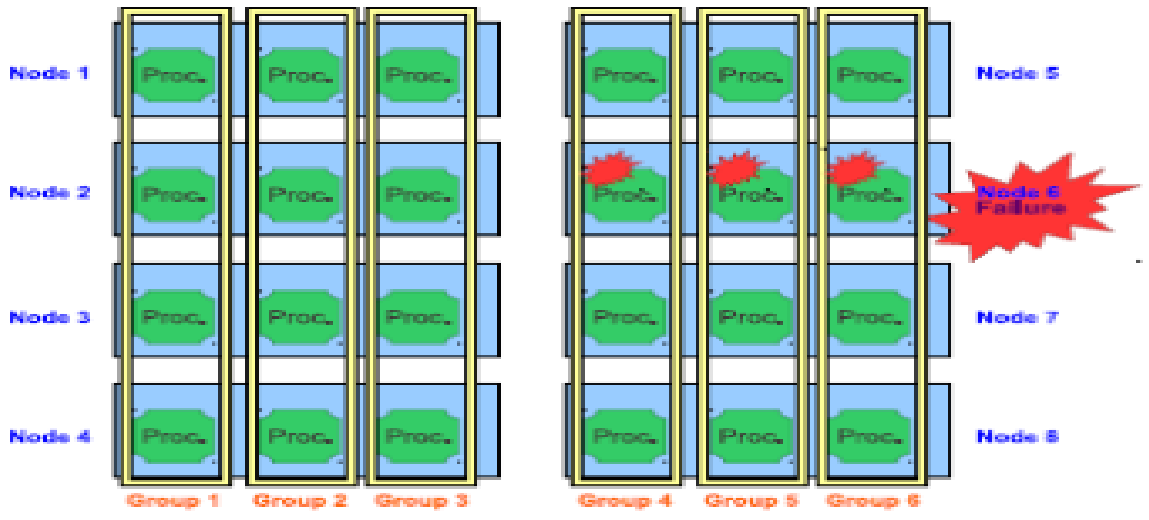


Figure 26: FTI – L3 "Node group checkpoint" – Redundancy with Reed Solomon encoding.

- L4 : FTI writes the checkpoint on the parallel file system (PFS). To hide the cost of the backup level, the local storage system is used as a buffer to write the data on PFS asynchronously.

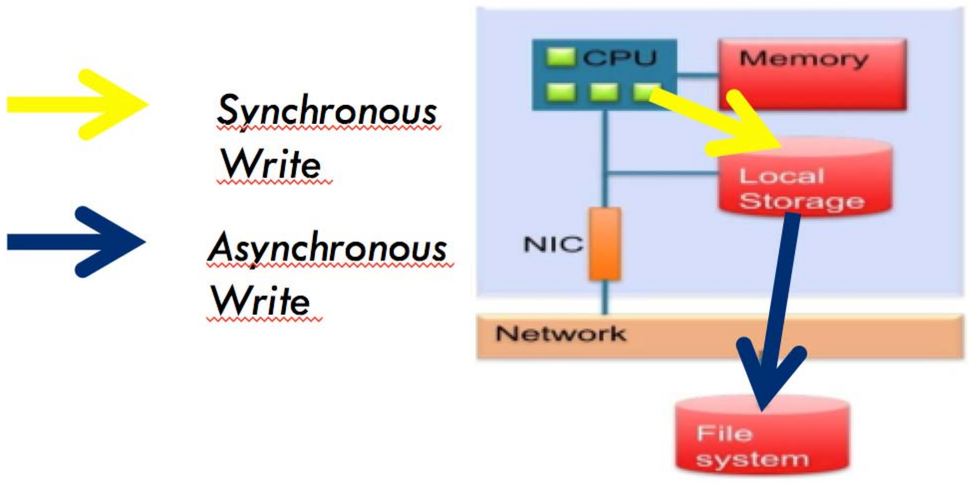


Figure 27: FTI – L4 "Node group checkpoint" – Redundancy with Reed Solomon encoding.

The use of FTI with all these levels makes sense when SSDs are available as local storage. As a matter of fact, they insure permanency of data after a power off that a classic memory does not offer. This allows a faster recovery than with the PFS alone, especially if a big number of nodes are involved.

FTI enables to generate checkpoints more frequently on the first three levels and to reduce the frequency of more expensive checkpoints at level 4, while ensuring effective protection against hardware failures. Thus, the global impact of the implementation of protection can be reduced in comparison with classical mechanism based upon the PFS alone.

FTI is a collection of few simple functions to call within the application to protect:

- int FTI_Init (char _configFile, MPI_Comm globalComm). This function will initialize FTI.
- int FTI_Protect (int id, void _ptr, long size). It stores a pointer to a variable that needs to be protected.
- int FTI_Snapshot (). This function takes an FTI snapshot or recover the data if it is a restart.
- int FTI_Finalize (). This function closes FTI properly on the application processes.

Besides measurements, requirements and new features interesting for production applications were decided among partners.

- A Fortran interface was necessary (only C was available). This has been done by CEA/MdlS
- Python was removed to improve portability
- FTI did not formerly handle restart when no failure occurred for multiple stage jobs (protection files were flushed on normal end), a flag has been added to keep protection for a scheduled restart.
- Checkpoint frequency was based upon the number of iterations. This is inconvenient from a production perspective as iteration may be barely predictable or may vary. Duration based checkpoint was implemented.
- A synchronous mode was implemented with no requirement for a dedicated process. This mode lacks advanced features but it allows a better portability on machines with restrictions on the number of processes per node.
- FTI was based upon a dedicated MPI task on a per node basis, this may lead to a waste of resources. A thread-based version has been implemented for synchronous mode and extension to other levels is under study.

- The API was redesigned to get more information about the protected variables. This will allow future developments for corruption detection and data compression.
- To figure out how to use FTI without SSD, a “memory map” (mmap) mode is on study.

4.3 Experiments

Measurement was done using weak-scaling. Calibration was necessary to ensure the best compromise between test duration, size and frequency of protection and measurable impact of multilevel protection.

Weak scaling scheme (grid size / number of cores) up to 9600 cores on the CURIE system.

grid size	core for hydro
50000*100000	600
100000*100000	1200
100000*200000	2400
200000*200000	4800
300000*200000	7200
400000*200000	9600

Table 16: Hydro weak scaling : Grid size / number of cores

To put a focus on FTI impact and to represent future high checkpoint traffic expected, frequency has been set to every 6 minutes, each task writing a 255MB file. Several levels of FTI were assessed ranging from no checkpoint to PFS synchronous mode (figure below).

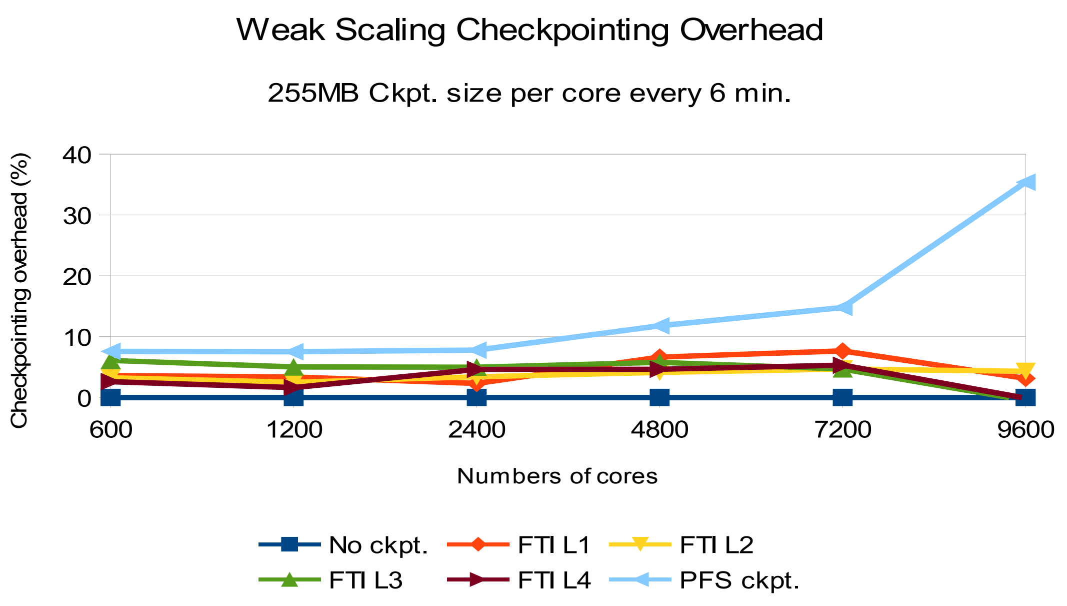


Figure 28: Hydro – FTI : Overhead level measurements.

The overhead induced by FTI Hydro remained for all tests below 6% and broadly constant as that induced by conventional backups on the file system (PFS, Lustre in our example) increased exponentially with the scaling (light blue curve).

To analyze these results it is necessary to take into account the variability of the measurements. As a matter of fact, variability of results on a machine in concurrent production can often reach 5%. This explains in particular the graph changes in the classification of different levels of FTI : they have quite small differences in cost. For example, the level 1 should be theoretically faster. We can say that these results are from real world experience.

4.4 Conclusion

The concept introduced by FTI is application oriented and is a good answer to future multi petascale and exascale HPC facilities.

- FTI handles « how » and takes the best advantage of the HPC platform
- The application integration focuses on « what »
- Easy to implement on already checkpointed applications
- Embedded added value by FTI hardly integrable by application implementation

5 File system optimization

On this chapter we present the architecture and the different tests that we have done on user hint guided I/O prefetching as part of the WP12 of the PRACE project. In Exascale HPC environments, it is important to be able to successfully prefetch clients' I/O in order to reduce the perceived request latency and improve the overall performance. Nevertheless, due to the high pressure on the resources that the large numbers of clients of Exascale environments deliver, it is also important that this prefetching uses as less memory as possible, and also avoids issuing more I/O operations than strictly required. In this kind of scenario, the elements that know how an application will issue the I/O operations it requires, are either application itself or the user of the system. Thus, it seems clear that allowing a richer communication between them and the underlying storage will provide benefits.

5.1 Introduction

The page cache is an important element concerning the performance of an Operating System (OS), since it either stores memory pages with the contents of a file, or memory pages directly requested by an application. The former have a lower priority than the latter, and this causes that pages associated to a file may be discarded when the application requires additional memory. Unfortunately, keeping file pages in memory is important to reduce the perceived request latency and avoid accessing the storage device when a data page is needed again (note that accessing data in the page cache can accelerate I/O time by a factor of 100x when compared to a disk access).Additionally, as the number of client applications competing for the OS resources increases, and the amount of available memory grows scarce it is not always easy for the OS to determine which file pages should be discarded from the page cache and which should be kept. Since both are typical situations of large Exascale environments, we designed a user-guided prefetching mechanism that allows the user to specify the I/O access patterns of an application, so that it is possible to identify which data pages are important, and which can be safely discarded. To make the prefetching more flexible, we introduce the concept of minimal time to process, so that future blocks can be prefetched in advance at the right time. Further extensions are possible, like adapting those times to what is really happening (taking into account congestion issues, or CPU-intensive processes. Nevertheless, these extensions fall out of the scope of this work, and we will not describe them here.

As we have mentioned, the out-of-the-box prefetching that is done by the OS is exceedingly simple: unless a POSIX_FADV_RANDOM flag is indicated by the application (using the posix_fadvise interface [54]), the OS assumes that subsequent request will follow a sequential access pattern and simply tries to fetch a few more sequential blocks than requested by the upper layers. If or when the OS detects that the requests issued are no longer sequential, the prefetching process is stopped. Note that, in order to avoid overloading the system, the prefetcher does not try to get a lot of additional blocks from the devices, with the most recent Linux kernels only reading at most 32 blocks per file in advance. However, those additional blocks are not discarded, but rather kept in the page cache, which may cause memory consumption issues.

For all these reasons, a more versatile prefetching library and some modifications on the kernel may suppose a big improvement in the I/O stack over two topics: cache memory usage as we can reduce it to the minimal needed and performance, as we can prefetch better than the OS non-standard patterns (big blocks or non-sequential).

We can summarize the start of the art with Reducing Seek Overhead with Application-Directed Prefetching [55]. The paper offers a library and some changes in the kernel to provide prefetching. Our proposal focuses on the user part and avoids kernel modifications, since such modifications are generally a bad idea as they introduce additional difficulties in the deployment of the solution.

5.2 Contributions

We introduce two novel proposals in order to improve the performance and reduce the memory consumption of cluster nodes in Exascale environments. The first proposal introduces the concept of user-provided *minimal time to process* inside the prefetching mechanism, which represents the estimated minimum CPU time between two different I/O requests. With this information at hand, read requests can be anticipated and the prefetcher act in consequence. As a result of using this information, we obtain a reduction of the memory used for prefetched data blocks, with some additional improvements on the performance of the operations.

The second proposal, on the other hand, is an *advanced filtering feature*. This feature is able to understand how user's data is stored and apply a filter to decide if the data is going to be useful or not. For example, users could use this feature to filter a set of climate raw data by temperature, discarding non-interesting data in the background, and keep in the page cache only the data that is higher than, e.g., 10ºC. With this feature, which is similar to HDF5 filters [56] or indexed data, the prefetching library can keep in the page cache memory only the data that is actually useful, returning to the user a record with "dummy" data if the data requested does not pass the filter. Notice that this *dummy record* is defined by the user itself, and only serves to mark that a filtered data access did not succeed. As such, the library can discard the read data block, thus saving on long-term memory usage. This feature will work on raw, unstructured data when no index or other fast selection methods are available (e.g., scan workflows).

Our proposal avoids making changes in the kernel since it is difficult to export the ideas for the different kernels/OSs available. Additionally, it has been shown on previous works that such changes are a barrier to install or test the proposed techniques on different environments, as it happened for instance with IOAnalyzer [57] in the IOLANES [58] project, forcing the creation and maintenance of a user-mode version. Thus, we introduce a new a library that acts as a wrapper for normal read/open/seek system calls, and that allows the appropriate user hints to be provided in order to implement our extended prefetching mechanism.

To allow users to appropriately characterize the I/O access patterns of their applications, our library extends the standard open operation list of parameters with an extra IODefinition structure that is used to specify the application's I/O for that particular file (see Table 17). Once the library receives this I/O characterization (plus some hints that could be inferred using machine learning), it creates several background *fetcher threads* that anticipate the read operations of the applications, and load the appropriate data in the page cache so that it is readily available. Once used, if the data blocks are not going to be reused, they are discarded, rather than kept in the page cache (please refer to Figure 29 for a high-level scheme of the library components).

NAME	EXPLANATION
Mode	Single read, fetcher, or filtering
StartPos	Starting position
RecordSize	Record size
EndPos	Ending position
Stride	Bytes to jump between records
Rate	Time to wait between record reads (msecs)
FieldPos	Field position inside a record (bytes)
FieldType	Field type : Integer, float, double
FieldOp	Field operation : gt, lt, eq
FieldValue	Value to use with FieldOp to filter
FieldDummy	Dummy value to be returned to the user

Table 17: List of the fields in an IODefinitionstructure

Notice that the key of user-hinted I/O is the interface with the library, a richer communication channel between clients and the filesystem that allows a detailed specification of the expected I/O applications will do. Table 17 shows the fields introduced by the IODefinition in our current prototype, which the users of the library can use to define the I/O for a particular file. The fields in a white background are those used in the basic user-guided I/O mode of our library, while the fields in a gray background are those used for the advanced filtering feature. As can be observed, the basic user-guided I/O requires the *start* and *end* offset within the file, the size of the *record* (i.e. the amount of consecutive data to be read), the *stride* (the space between consecutive reads that the application will jump) and the *rate* at which the file should be read. Notice that the Rate field specifies how much time the fetcher threads should wait until the next record is read. Advanced filtering however, requires more fields; these include the position of the target variable inside the record (in bytes), the type of the variable and the operation to apply on it (e.g., "> 10"). Finally, we also require a value to mark the dummy fields that do not pass the filter, to allow for a fast identification in the application level. Note that all these fields could be replaced with a Data Description Language (DSL), for example)

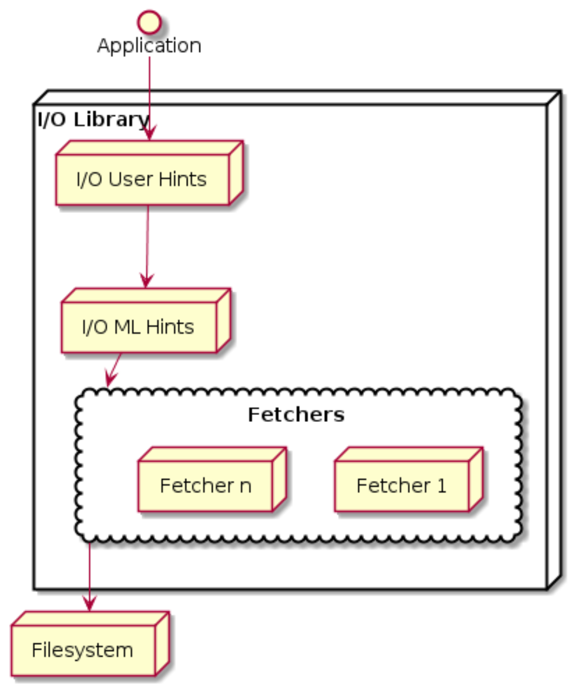


Figure 29: I/O library architecture

All this information is used to configure the read pattern for the fetcher threads, which will load the page cache in time for the application to use the data.

Our first proposed prototype, shown in Figure 30, included a memory manager where read pages were stored in order to have a clear view of what was indeed needed to be cached and what not, thus allowing the library to support filesystems that do not have the readahead system call (or similar) implemented. Nevertheless, this organization had some problems. As can be observed in Figure 31, even though with this proposal the library keeps prefetched data blocks in user memory, the OS also keeps an additional copy in the page cache: when a fetcher thread asks the OS for a page on disk, the read result is stored in the page cache and then it is *copied* to the I/O library memory. Although this is not a problem for the library, as we can mark the pages as not usable, we are stressing the system unnecessarily.

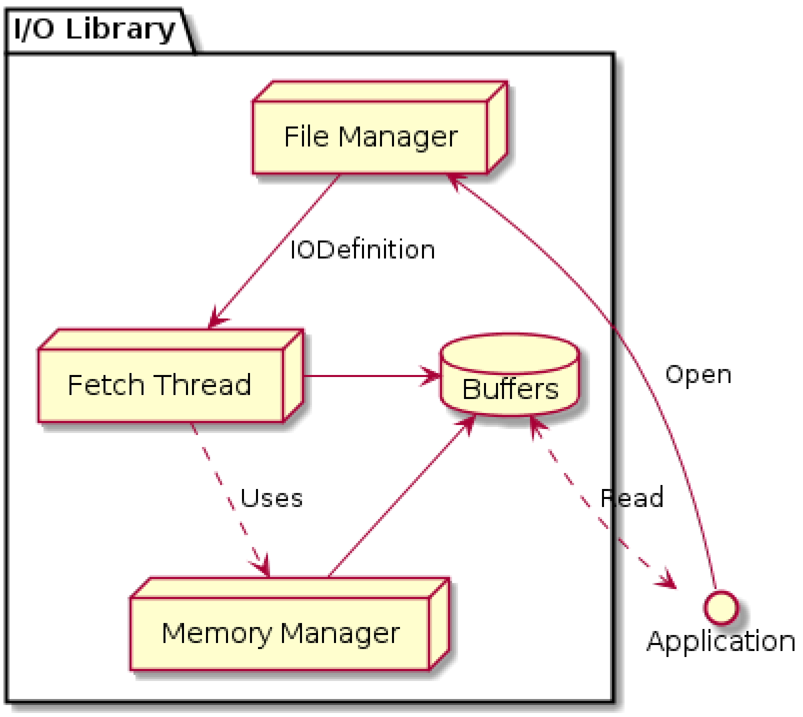


Figure 30: Discarded proposal with Memory Management

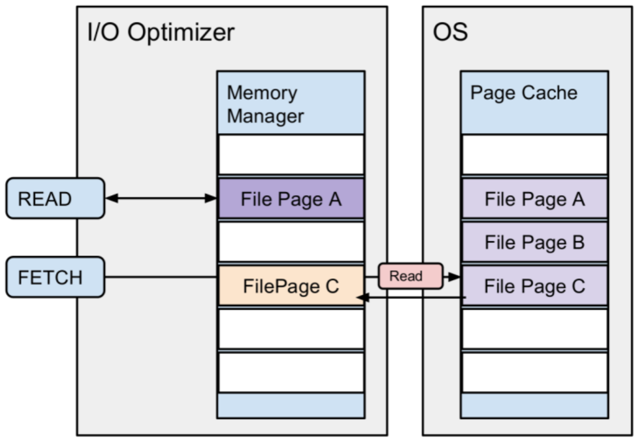


Figure 31: Memory duplication between page cache and memory manager

After some preliminary tests, we decided to remove the memory manager and instead make use of the readahead system call, thus effectively using the kernel's page cache as a buffer for the prefetching library. Using the page cache as a buffer only requires one kernel → user space interaction, since the data can be transferred directly from the kernel buffers to the application buffers. This option removes all the complexity of having to keep a block cache in the user side. Since the library does not have control over which pages are in the page cache (as the OS will drop them without notification), to the library uses the fincore [59] utility to keep track of which pages of a file may have been dropped, and act accordingly. For example, if the OS removes some pages due to memory pressure, the fetcher thread may need to be delayed.

When the user opens a file, the prefetch library looks up the user-provided IODefinition structure and creates a *background prefetcher thread* with the information contained in it. In order to illustrate the inner works of the library, let us consider the situation where a user opens a file and requests it to be prefetched with stride. In this situation, the created background thread first needs to identify which is the sequence of records that the user will require. Once this is done, the thread then issues a readahead call for each of record in order to store it in the page cache, jumps to the next one and waits the required time defined by the user to match the desired I/O rate. Notice that since the information will only be read once (as it is specified inside the IODefinition class), when the user reads it a DONTNEED fadvise is applied to the page marking it as a candidate to be removed from the page cache. If the user reads faster than the background reader, the background fetcher detects it and jumps to a more advanced position.

We also propose an advanced prefetching mechanism that allows to define the record structure via the IODefinition structure and to add some filtering instructions. For example, imagine that the application stores its data as an aggregation of the structures (i.e. records) shown in Table 18.

Variable	**Size**
Char[256]	256 bytes
Int	4 bytes
Other	1024 bytes

Table 18: Sample record

With this record, the library allows users to define a record size of 1284 bytes (the total sum of all the fields) and a field of interest (FOI) that represents an integer at offset 256 (within the record). This FOI is useful, for instance, in order to specify detailed filtering tasks, so that e.g. only records with a FOI integer value greater than 100 will be useful for the application. The *background prefetcher thread* scans the different records and applies the specified operation. If the record passes the filter, it will already be in the page cache (as the library needs to read it in order to apply the filter operation). If the filter fails, the record is removed from the page cache by issuing a fadvise call with the DONTNEED flag. Note however, that in order to have improvements on the application side, the library needs to keep an updated bitmap with the records that have passed or not the filter. Using this bitmap (or similar structure), we can offer to the client a dummy record on the failed ones. This dummy record, known by the application will reduce the prefetch memory usage and may increase the performance. This bitmap or index, is also essential to offer a filtered or virtual view of the file, as in traditional SQL views, avoiding operations on the client (like testing that the record is a dummy or not).

5.3 Evaluations and Experimental Results

In order to test the benefits of our proposal, we devised a test application to measure the memory used in the page cache. This test application mimics the READ_DATA → PROCESS_DATA → READ_DATA access pattern common to most HPC scientific applications. As we will see in this section, our proposal obtains some performance benefits for big requests (> 512 KiB), and large memory benefits in all the cases evaluated. The test environment for these experiments uses a Intel Core 2 Quad CPU Q9300 with 4 GB and a standard SATA drive ST31500341AS. The kernel version is a 3.14 without modifications.

Scenario 1 – Memory usage of 1 process

In this scenario, we want to show the reduction on memory used for the page cache on the case of 1 single process. Although memory will be freed if needed, it helps to understand what will happen when there are a large number of processes. For this experiment, we measure the memory usage of a test application that performs 2000 reads of 1MiB records from a file, uses 100ms of process time for each record read, and a 400KiB offset stride between records. We evaluate the two following configurations: standard kernel reads with prefetching (STD) and user-guided reads with the proposed prefetching library (USER HINTS). As can be observed in Figure 32 the memory cost of the USER HINTS method is close to 0, while read operations with STD method keep all the accessed blocks in the page cache even if they are only used once. Also note that, despite the fact that improving performance is not the main objective of the library, reads using the original kernel prefetching need nearly 10% more time to complete the tests than those using the library. The reason for this lies in the fact that the user-informed library is more effective at prefetching striding patterns: whereas the kernel keeps reading data blocks sequentially until it notices the offset stride, the library is able to stop reading immediately and move the file pointer to the next offset.

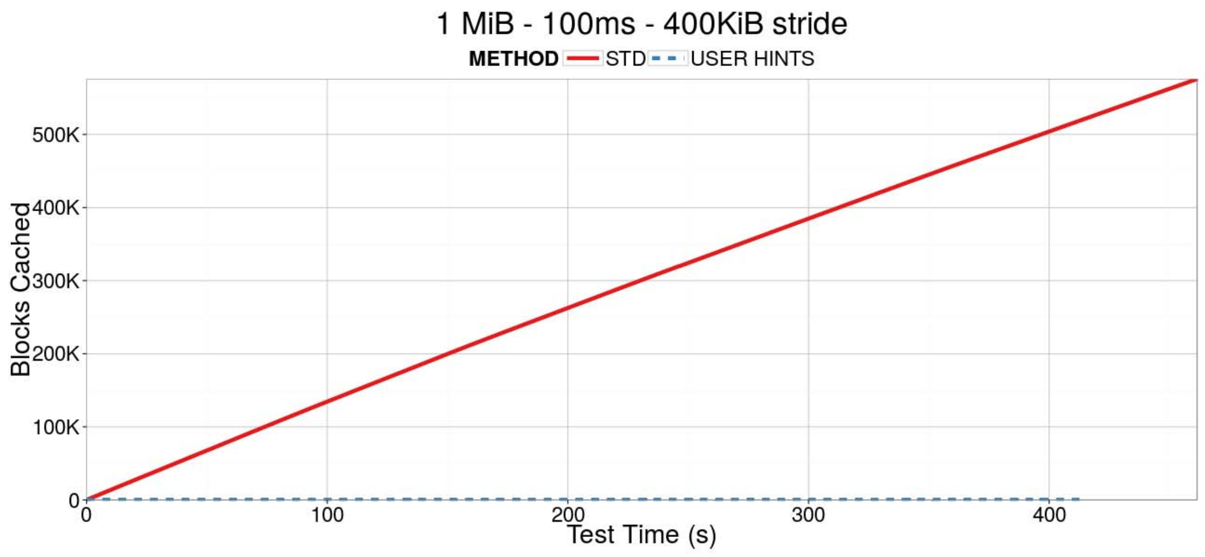


Figure 32: Memory utilization using 1 process with different read techniques.

Scenario 2 – 64 processes, informed sequential access

In this experiment we measure the memory usage of the page cache of 64 concurrent processes performing informed reads on 64 independent files. Each process performs 100 read operations that follow a sequential access pattern. In order to properly evaluate the benefits of the library, we measure the impact of different record sizes (512KiB, 1MiB, 2MiB, and 4MiB), and different CPU processing times (1ms, 10ms, and 100ms). We also include the results for a pure sequential access pattern (NO STR) and a strided access pattern with an offset of 400KiB (STR).

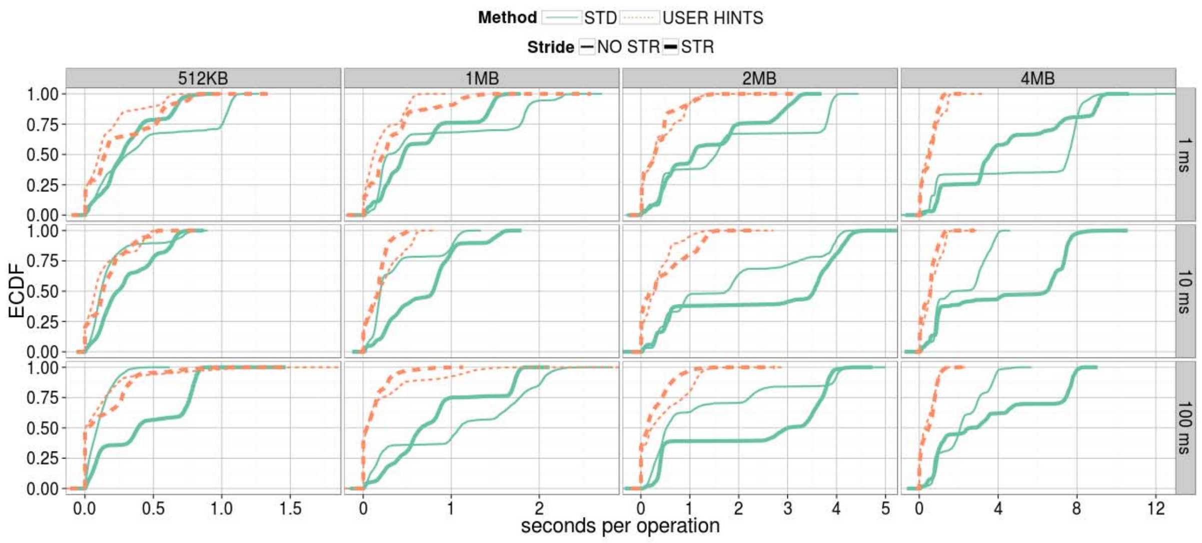


Figure 33: ECDF with each operation response time.

Figure 33 shows the results of this evaluation. The figure is divided horizontally by the block request size (512KiB-4MiB) and vertically by the minimum time needed to process the requested block (1ms-100ms). Each subplot shows the *empirical cumulative distribution function* (ECDF) for the operation response time, both for STD and USER HINTS reads. It can be observed how, in most of the experiments, the USER HINTS method offers a better response time than standard prefetching. For record sizes of 512KiB and purely sequential access, however, the USER HINTS results are similar to the standard prefetching. Additionally, the plots also show that the difference between strided and non strided access for the USER HINTS method is small. Since the library knows in advance how data needs to be accessed, the I/O cost of disk seeks can be overcome at the fetcher level, thus leading to very similar response times.

Focusing on non-strided access, we aggregated the measured cached data kept in the page cache for all 64 processes, and plot it in Figure 34. Each x-axis represents the duration of the test, since the memory usage of the page cache will change as the test progresses. It can be clearly observed how the STD method offers an increasing memory usage. Note that the memory usage is irregular for higher block sizes since the OS may eventually need to do some evictions. For the USER HINTS method, the memory usage is kept low for the entire test's length. This is the expected behavior, since the library is able to evict data pages when they are no longer needed, keeping the memory usage contained. Note that although these evictions may not be needed, it is convenient for the library to mark the pages with the DONTNEED flag. The kernel will, normally, evict old pages automatically, but the behavior could change under memory pressure situations or under different kernels.

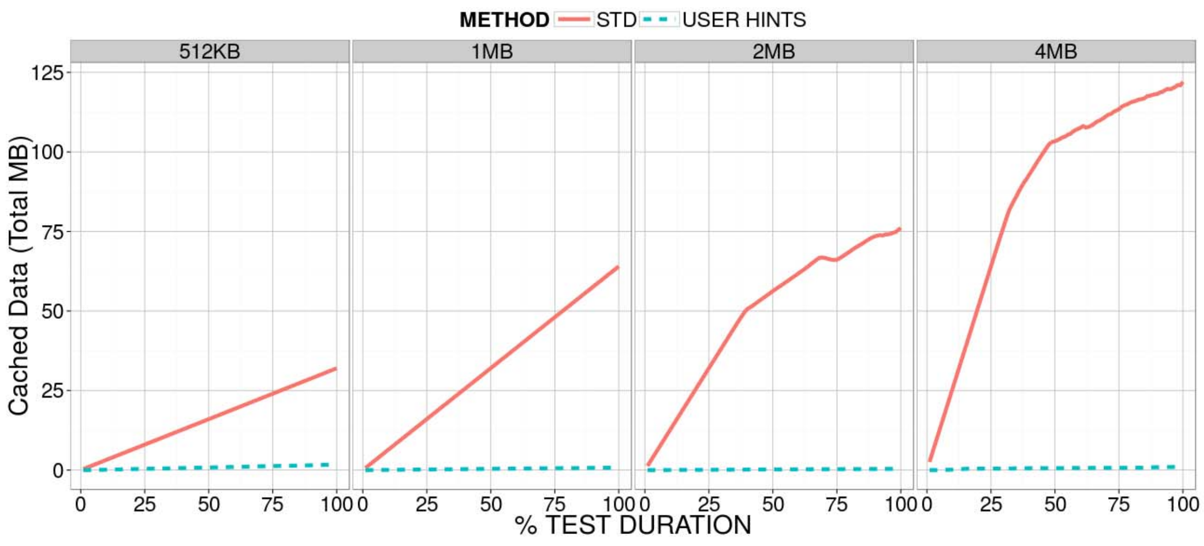


Figure 34: Total memory usage in the page cache (64 processes,no stride, 10 ms process).

<u>Scenario 3 – 64 processes, random informed access</u>

This scenario is similar to the one already described. The main difference is that in this scenario, the 64 processes are issuing “random” requests to each of the 64 files. This random access is not truly random in the sense that the user knows which offsets will be accessed, and how much data needs to be read for each record. Nevertheless, even if this information is predictable, the OS prefetcher is not able to infer a clear access pattern from the offsets accessed, and classifies it as random data accesses, thus deactivating the prefetching mechanism. The user hints mechanism offers a way for applications to inform the prefetching library of these expected access patterns, which can be useful for scenarios such as database queries, indexed data or HDF5/NetCDF structured data.

Figure 35 shows the memory usage for measured in this scenario. Once again, each process performs 100 read operations and we measure the impact of using different record sizes (512KiB, 1MiB, 2MiB, and 4MiB), and different CPU processing times (1ms, 10ms, and 100ms). Since access patterns in this scenario are random, it does not make sense to consider sequential or strided access patterns, which is why they do not appear in the figures. The ECDF curves depicted in the subplots show that, as expected, all the scenarios are favorable to the USER HINTS method. Since the library knows in advance which data blocks are going to be accessed, it can successfully load them in the page cache just before they are needed, offering better performance results than the kernel’s standard prefetching mechanism, which has no basis to work on. The results for the memory usage are similar to those shown in Figure 34.

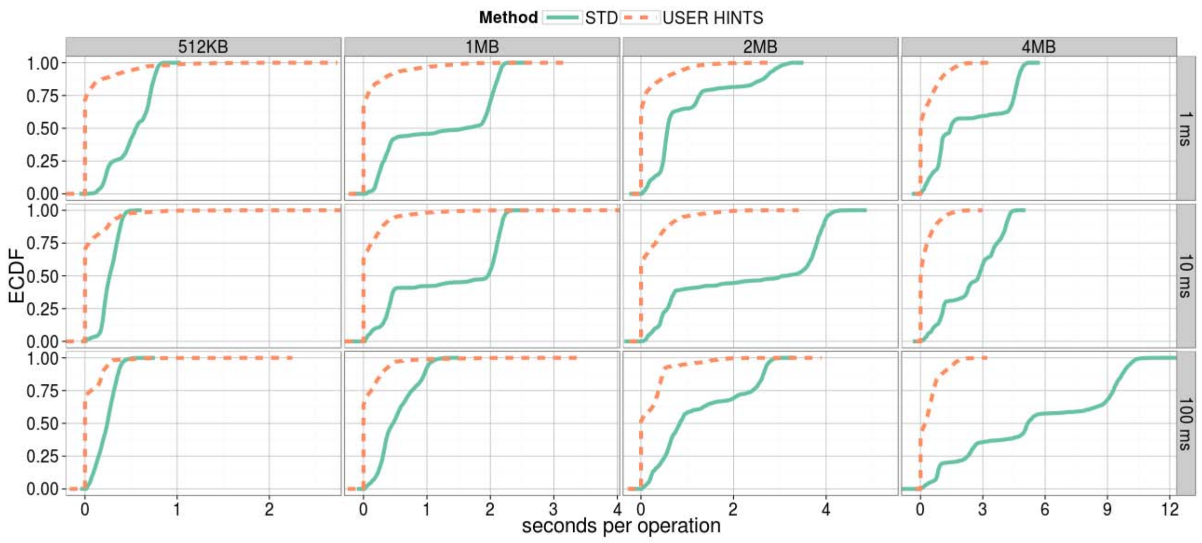


Figure 35: ECDF of the Response time per operation when issuing random requests.

Scenario 4 – Advanced filtering

Finally, and to show how the advanced filtering technique works, in this scenario we show a zoom of the results obtained when applying this technique. The application scans a file reading 4KiB records with a 100ms process time and testing if an integer field is bigger than 100. Figure 36, shows two scenarios: the STD and the ADV FILTER scenario. In the first scenario, a user level application reads each record and decides whether it is usable or not, whereas in the second scenario the filtering is done by the library in the background and the already-filtered records returned to the user. Usable data is shown in blue and with a sequential ID (to allow comparison of the two approaches), while unusable data is shown in red and without a number ID. It can be observed that, if the required information is passed via user hints to the library, the filtering process can be done at the fetcher level, keeping the passed/failed information in an index and storing only the good records. Finally, the user application works as usually but only the correct data is prefetched as the filtered data is returned with a dummy that the user will recognize. With this technique, the library is able to reduce the memory used while being able to prefetch only useful data (for example, one or two requests in advance, depending on the ratio of success rate). A bigger performance improvement can be achieved if the library offers to the application a virtual file, since the user does not need to pass the filter over the dummy data (and we avoid the data copy to the application buffer).

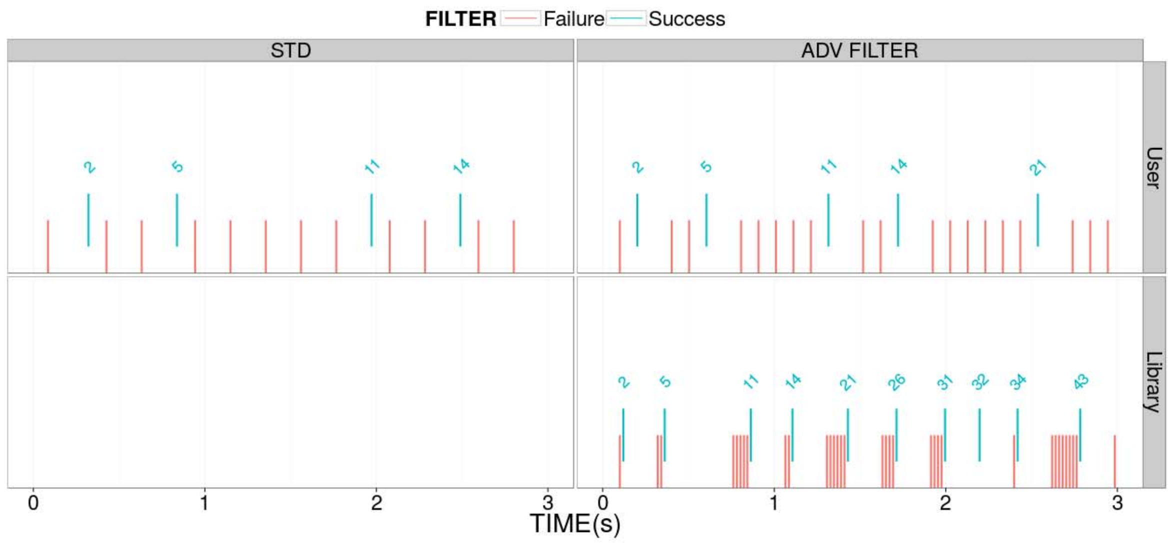


Figure 36: Advanced Filtering with the standard behavior shown on the left, and the user level and fetcher thread on the right.

5.4 Conclusions

The techniques explored in this deliverable can be used on Exascale environments to reduce the memory pressure on the page cache. As we have shown, anticipating user reads, with small information from the user, can produce great benefits. On the one hand, these techniques will always decrease the memory usage but, on the other hand they may increase the response time if the blocks requested are small (<512 KiB) or the access pattern are sequential enough that the OS prefetcher may work in a suitable way.

The second proposal, advanced filtering, will also reduce the memory pressure and will have an effect on performance since the dummy records can be easily removed. Finally, this technique can be moved down the I/O stack, for example to the PFS nodes, to activate also reductions on network congestion.

As a summary, a prefetch library shows benefits in performance and memory usage when:

1) There are consecutive big reads with or without stride and process time involved.
2) The library knows in advance the I/O patterns of the application.
3) The user can describe the records going to be read, and the library is able to apply the filtering during the prefetch process. This allows that only the needed records are cached.

However, there are also situations where it is difficult that library prefetching becomes better than the OS prefetcher:

1) When consecutive reads happen very fast and they are also consecutive on space (i.e. sequential). The OS will prefetch automatically, and the memory used will be minimal as the older pages will be removed automatically (in low memory pressure situations).
2) There is no information about future reads. Without information about the application behavior (user- or automatically-inferred) guided prefetching will not work better than the OS.

Sometimes direct I/O can offer better performance than any prefetching mechanism (OS or library) due to the direct memory transfer allowed by DMA. However, the I/O restrictions should not be underestimated. For example for direct DMA data transfer, need aligned reads/writes and buffers are required, which can prove bothersome to achieve under some environments, and may be a strict requirement to place on the users of a high-performance computing cluster.

6 Summary and Conclusions

This Work Package performed research and development on the programmability of future multipetascale and exascale systems exploring a diverse set of techniques. It was organized into four different tasks, Task 12.1 dealed with auto-tuned runtime environments, Task 12.2 dealt with scalable numerical algorithms, Task 12.3 focused on fault tolerant tools, and Task 12.4 looked at file system optimization for exascale systems. Although, these tasks focused on improving independent areas of an HPC system, there is some relationships between them as well. For instance, there is some relationship between Tasks 12.1 and Task 12.2 where auto-tuning techniques where used to find out the optimal number of threads which can be applied to the OpenMP codes in Task 12.2.

A total of 16 research projects were reporting in this document covering multiple different techniques within these main four areas which are summarized below.

Task 12.1 provided different auto-tuning techniques to improve the performance of codes. Systems and applications are getting so complex that it is imperative to relay on auto-tuning techniques to automatically obtain the desired high performance on the target system without user intervention. This task contributed on different areas such as OpenMP applications, SLURM scheduler, Kepler workflow engine, component based 3D FFT, compiler auto-vectorization, and All-to-all collective communication. A total of seven projects were covering these different areas resulting on 5 whitepapers and 2 additional detailed reports all of them reported on this document.

Task 12.2 has provided new guidelines, algorithmic approaches and adaptive methodologies to improve the performance of numerical algorithms on modern large-scale systems. It demonstrated improvements on diverse numerical algorithms such as HYDRO, PETSc, CP2K and FETI. This task consisted on eigth different projects that resulted on 5 whitepapers and 3 detailed reports. Improvements are achieved in part using aaccelerators with the OpenACC and OpenCL enabling parallel constructs or using shared memory parallelism with OpenMP improving efficiency of the HYDRO application. The approaches for numerical algorithms also include distributed memory parallelism, example applications including PETSc and FETI. One of the major focusses of this task was to show that it is possible to improve the parallel performance of sparse linear iterative solvers. Three projects addressed this issued using a novel scheme that completely avoids the communication latency overhead, or using a pipelined solver to increased computation with communication overlap, or via a strategy to dynamically adjust the local convergence criterion during execution.

Task 12.3 concluded that fault tolerant based on selecting within the application the critical data to save achieves very little overhead at large scale. It was reporting less than 6% on HYDRO at 9,600 cores.

Task 12.4 explored techniques aimed to be used on Exascale environments to reduce the memory pressure on the page cache. It concluded that anticipating user reads, with small information from the user, can produce great benefits. These techniques will always decrease the memory usage which is important at exascale where the amount of memory per core is expected to be small.

In conclusion, in this work package we have shown that applications and system software has to be re-developed in order to deal with the increasing complexity of hardware at the multi-peta scale and exascale computers. Hardware at high scale is going to be complex and thus applications are going to be fully re-developed to deal with that complexity. In this work package we have demonstrated that we have a solid foundation to obtain a very high parallel performance getting close to petascale as demonstrated in all the projects carried out during this last year.

In Task 12.1 we have shown how applications must adapt to the specific characteristics of processor and network topologies as an example. And additionally, in Task 12.2 we have shown that algorithmic changes are needed in order to improve parallel efficiency at large scale. Although this research show promising results, we are still far away to be ready for exascale and thus more research is needed to re-develop codes to this un-precendent parallel machines. From our point of view, it will require integration of technical ideas as well as solid theoretical foundations into its core design to take full advantage of the underlying hybrid architecture of exascale machines. We think that new applications should take these remarks into account from the very beginning in their design process.