SEVENTH FRAMEWORK PROGRAMME
Research Infrastructures

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

PRACE

Partnership for Advanced Computing in Europe

Grant Agreement Number: RI-211528

D6.5
Report on Porting and Optimisation of Applications

Draft

Version:	1.0
Author(s):	Sebastian von Alfthan, CSC
	Giorgos Goumas, GRNET
	Olli-Pekka Lehto, CSC
	Pekka Manninen, CSC
	Mohammad Jowkar, BSC
	Harald Klimach, HLRS
Date:	26.10.2009

Project and Deliverable Information Sheet

<table>
<tr><td rowspan="7">PRACE Project</td><td colspan="2">Project Ref. №: RI-211528</td></tr>
<tr><td colspan="2">Project Title: Partnership for Advanced Computing in Europe</td></tr>
<tr><td colspan="2">Project Web Site: http://www.prace-project.eu</td></tr>
<tr><td colspan="2">Deliverable ID: D6.5</td></tr>
<tr><td colspan="2">Deliverable Nature: Report</td></tr>
<tr><td>Deliverable Level:
PU</td><td>Contractual Date of Delivery:
31 / October / 2009</td></tr>
<tr><td></td><td>Actual Date of Delivery:
30 / October / 2009</td></tr>
<tr><td colspan="2">EC Project Officer: Maria Ramalho-Natario</td></tr>
</table>

Document Control Sheet

<table>
<tr><td rowspan="6">Document</td><td colspan="2">Title: Report on Porting and Optimisation of Applications</td></tr>
<tr><td colspan="2">ID: D6.5</td></tr>
<tr><td>Version: 1.0</td><td>Status: Final</td></tr>
<tr><td colspan="2">Available at: http://www.prace-project.eu</td></tr>
<tr><td colspan="2">Software Tool: Microsoft Word 2007</td></tr>
<tr><td colspan="2">File(s): D6.5.doc</td></tr>
<tr><td>Authorship</td><td>Written by:</td><td>Sebastian von Alfthan (CSC)
Giorgos Goumas (GRNET)
Olli-Pekka Lehto (CSC)
Pekka Manninen (CSC)
Mohammad Jowkar (BSC)
Harald Klimach (HLRS)</td></tr>
</table>

	Contributors:	Raúl de la Cruz (BSC) Bertrand Cirou (CINES) Mauricio Araya Polo (BSC) Andrew Sunderland (STFC) Albert Farres (BSC) Xavier Saez (BSC) Orlando Riviera (LRZ) Jussi Enkovaara (CSC) Xu Guo (EPCC) Joachim Hein (EPCC) Martin Polak (GUP) Paschalis Korosoglu (GRNET) John Donners (SARA) Fernando Nogueira (UC-LCA) Lukas Arnold (FZJ) Maciej Cytowski (PSNC/ICM) Carlo Cavazzoni (CINECA)
	Reviewed by:	Florian Berberich (FZJ) Colin Glass (HLRS)
	Approved by:	Technical Board

Document Status Sheet

Version	Date	Status	Comments
0.1	02/February/2009	Draft	Initial version
0.2	15/September/2009	Draft	Complete document for internal WP6 internal review
0.3	04/October/2009	Draft	Integration of comments from WP6 review, made available to PRACE internal review.
1.0	26/October/2009	Final version	

Document Keywords and Abstract

Keywords:	PRACE, HPC, Porting, Optimisation, PABS
Abstract:	This document reports the optimisation and porting of applications in the PRACE application benchmark suite (PABS) to the PRACE-WP7 prototype machines. Porting encompasses not only compiling functioning programs on target platforms, but also methods aimed at extracting maximum performance through informed choice of compilers, external libraries and platform parameters. Optimisation techniques are techniques for improving the performance of applications on a node-level. This includes techniques aimed at optimising usage of memory-hierarchy, computational parts of processors and algorithmic issues. In this deliverable, we present porting and optimisation reports for each application. We also discuss the best practices extracted from the application reports, from knowledge gathered at the prototype sites and an analysis of synthetic kernels.

Table of Contents

List of Figures

List of Tables

References and Applicable Documents

[1] *NAMD2: Greater Scalability for Parallel Molecular Dynamics*, L. Kalé, et al., Journal of Computational Physics **151**, 283 (1999)

[2] *Scalable Molecular Dynamics with NAMD*, J. Phillips, et al., Journal of Computational Chemistry **26**, 1781 (2005)

[3] *Charm++: Parallel Programming with Message-Driven Objects*, L. Kalé, S.Krishnan, in: *Parallel Programming using C++*, by G.V. Wilson and P. Lu. MIT Press, 175 (1996)

[4] Peter Coveney, Shunzhou Wan, private communication

[5] `http://www.ks.uiuc.edu/Research/namd/wiki/index.cgi?NamdMemoryReduction`

[6] Z. Sroczynski, *Improved performance of QCD code on ALICE*, Nucl. Phys. Proc. Suppl. 119 (**2005**) 1047-1049

[7] Th. Streuer and H. Stüben, *Simulations of QCD in the Era of Sustained Tflop/s computing*, in C. Bischof, M. Brückner, P. Gibbon, G. Goubert, T. Lippert, B. Mohr, F. Peters (eds.), Parallel Computing: Architectures, Algorithms and Applications, IOS Press, Advances in Parallel Computing 15 (**2008**) 535--542,

List of Acronyms and Abbreviations

AOS	Array of Structures; a data layout
BCO	Benchmark code owner. Person responsible for porting, optimizing, petascaling and analysing a specific benchmark application.
BLAS	Basic Linear Algebra Subroutines
BLACS	Basic Linear Algebra Communication Subprograms
BSC	Barcelona Supercomputer Center
CPU	Central Processing Unit
CSC	CSC — IT Center for Science Ltd (Finland)
CSCS	Swiss National Supercomputing Center (Switzerland)
DFT	Density Functional Theory
EPCC	Edinburgh Parallel Computing Centre
FN	Fat-node; describes a cluster with nodes with many processors and/or large amount of memory
FFT	Fast Fourier Transform
GPGPU	General Purpose Graphic Processing Unit
GUP	Institute of Graphics and Parallel Processing, Johannes Kepler University Linz (Austria).
HDF	Hierarchical Data Format.
HLRS	High Performance Computing Center Stuttgart (Germany).
HPC	High Performance Computing; Computing at a high performance level at any given time; often-used synonym with Supercomputing.
IO	Input-Output
JuBE	Jülich Benchmarking Environment
LAPACK	Linear Algebra PACKage
LOC	Lines of code
LRZ	Leibniz-Rechen Zentrum
MD	Molecular Dynamics
MPP	Massive parallel processing
MPI	Message Passing Interface. A library for message-passing programming.
OpenMP	Open Multi-Processing. An API for shared-memory parallel programming.
PABS	PRACE Application Benchmark Suite
PETSc	Portable, Extensible Toolkit for Scientific Computation
PRACE	Partnership for Advanced Computing in Europe; Project Acronym.
PWR6	IBM Power 6 processor
QCD	Quantum Chromo Dynamics
RTM	Reverse time migration
SARA	SARA Computing and Networking Services Amsterdam (the Netherlands).

ScaLAPACK	Scalable Linear Algebra PACKage
SMP	Symmetric multiprocessing
SMT	Symmetric multi threading
SOA	Structure of Arrays; a data layout
SVN	Subversion, a source code repository
TDDFT	Time-Dependent Density Functional Theory
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.
Tier-1	Major national or topical HPC systems.
TN	Thin-node; describes a cluster with nodes with one, or a few, processors per node.
Wiki	Web page or collection of web pages for creating collaborative web sites.
XML	eXtensible Markup Language

Executive Summary

This document reports the work performed in task 6.5. The main objectives of the task were to discover best practices in porting applications to the PRACE WP7 prototype machines, and to study ways of optimizing the serial performance of the PRACE benchmark applications. The parallel performance of applications is studied in task 6.4 and reported in D6.4.

The PRACE WP7 prototype machines represent the current best-of-breed supercomputer architectures, comprising MPP machines, thin- and thick-node clusters, machines based on vector processors and finally machines based on Cell processors. The prototypes are listed in Table 1.

Architecture	Type	Name	Site
Cray XT5	MPP	Louhi	CSC
IBM Blue Gene/P	MPP	Jugene	FZJ
IBM Power6 cluster	Fat-node (FN) cluster	Huygens	SARA
Nehalem cluster	Thin-node(TN) cluster	Juropa	FZJ
NEC SX-9 / Nehalem cluster hybrid	Vector /scalar hybrid	Baku	HLRS
IBM Cell cluster	Cell cluster	Maricel	BSC

Table 1: WP7 prototypes in PRACE

We have looked at the applications in the PRACE application benchmark suite (PABS). They represent the current, and future, computational workload expected on production petascale machines in Europe, and were identified in tasks 6.1 and 6.2. The first version of the suite was chosen based on a survey of the current usage of HPC applications. The final list of codes that emerged after taking into account the coverage of scientific areas, scalability, algorithms and user communities, is the following: Alya, AVBP, BSIT, Code_Saturn, CP2K, CPMD, Elmer, EUTERPE, Gadget, GPAW, Gromacs, HELIUM, NAMD, NEMO, NS3D, Octopus, PEPC, SPECFEM3D, QCD, Quantum_Espresso and WRF.

We report results for these applications, with the exception of Elmer, SPECFEM3D and WRF, which were added to the list in late 2009. Additionally we report results for Siesta, which was on the initial list, but not on the final one. Each application has been ported to an appropriate subset of prototypes, as described in report D6.3.2.

The application reports in Chapter 5 forms the main contribution of this report, detailing the issues and lessons learned for each application. From the reports we have extracted some information on porting and optimisation and present them in Chapters 3 and 4.

In the porting activities we have investigated not only adapting the applications to the prototypes, but also how to get optimal performance without resorting to source level modifications. We discuss porting issues caused by the hardware and software environment of the prototypes. We also look at the requirement for external libraries in the programs and present guidelines on how to leverage numerical libraries efficiently. Finally we discuss, in some depth, compilers and how to choose optimal compiler flags. We have systematically

studied optimal flags using four synthetic computational kernels, in order to discover compiler flags, which potentially enhance the performance of applications.

In the optimisation activities, we have focused on tuning the performance of the programs on the source code level. Here "optimisation" is defined as techniques, which improve the performance of the application on a node level. Here, a node is a shared memory compute element comprising one or more processors. We discuss general optimisation techniques, which are suitable for most platforms. These include techniques that improve the usage of the memory hierarchy, e.g., loop blocking and prefetching. We also look at ways of improving the computational throughput of applications, by using techniques such as SIMD instructions, and strength reduction. We also look at algorithmic optimisations, since in many cases they provide the greatest speedup to an application. Finally we also look at architecture-specific issues for x86 processors, SX-9 vector processors and Cell processors.

1 Introduction

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

The primary goal of PRACE WP6 is to identify and understand the software libraries, tools, benchmarks and skills required by users, to ensure that their applications can use a Pflop/s system productively and efficiently. WP6 is the largest of the technical work packages and involves all of the PRACE partners.

In task 6.5 best practices in porting and optimizing leading HPC applications to PRACE prototype petascale systems (Table 1) are compiled, paving the way for the efficient exploitation of the upcoming Tier-0 systems. The applications under study are the benchmark applications in PABS, which is a benchmark suite being compiled in task 6.3.

The structure of the deliverable is as follows: Chapter 2 describes the methodology by which the work was carried out. In Chapter 3 we present the best practices for porting applications to the prototypes. Chapter 4 discusses optimisation techniques, while Chapter 5 presents application reports that detail the work done on each application.

2 Methodology

The work in this task has been structured around the process of porting and optimising the applications in the PRACE application benchmark suite. For each application a "benchmark code owner" (BCO) has been assigned, being responsible for the respective application. Each BCO has been in charge of an application-centred subproject, which ensures that the application is ported to a subset of prototypes, promising optimisation strategies are investigated and the results are reported. In many application subprojects not only the BCO, but also a number of contributors have been carrying out the work.

The role of the task leader has been to work with the BCOs to ensure that the work progresses smoothly. Additionally, porting and optimisation issues outside of the application centred framework have been looked at.

To organize the work, telephone conferences have been held every second week. The BCOs have attended these telephone conferences and reported on their progress. The work has been coordinated using a collaboration platform (Trac), combining a "Wiki" (collaborative web-publishing system) with a Subversion (SVN) version control server, maintained by CSC. This platform has proven to be very useful in a distributed project such as this, with close to twenty participating organizations.

This report has been compiled in the second year of the project, in 2009. Each BCO has filled in a report template in May 2009 and subsequently updated the report in August 2009. These reports are presented as a part of this report. We have also distilled the lessons learned of the application reports and have presented those results in this report. In addition to the application reports, the results are based on feedback from sites hosting the prototypes and studies on compiler flags for synthetic kernels.

3 Porting

In this document porting is defined as the process of adapting software to work on a new architecture. This includes processes required to compile a functioning binary and activities aimed at producing an optimal binary with respect to its usage of computational resource, e.g., wall clock time or memory requirement. The latter activities are called "optimal porting" in this document, to emphasize the focus on extracting maximal performance out of the target machine. Optimisations requiring source code changes are defined to be in the optimisation category and not in optimal porting.

In general, scientific applications are actively developed on a limited number of platforms. With some effort they are usually portable to other platforms, but the level of difficulty depends on how much the target architecture differs from the supported ones. For example, porting a program designed for general-purpose architectures to an accelerator-based supercomputer requires a significant rewrite in order to fully utilize the accelerators.

Porting of a scientific application to a new architecture involves several aspects, namely hardware, software environment, and compilers. In sections 3.1 to 3.3 we present some general remarks on porting issues. In sections 3.4 to 3.9 we report best practices regarding porting, on a per prototype basis. The information is based on knowledge from the prototype sites, experiences gained in porting the PRACE application benchmarks to the prototypes, as well as results obtained exploring optimal compiler flags for synthetic benchmark kernels.

3.1 Hardware

The hardware in the prototypes range from fairly well understood x86 commodity clusters, to more exotic ones such as the IBM Cell based cluster (section 3.9). In general, applications are easier to port to the more common and widespread hardware platforms. Conversely, programs that have been tuned to an "exotic" architecture, can be challenging to port to more commonly used systems. This typically arises if the program has been programmed in a non-portable language or programming model. In such a case one needs to rewrite the code, which can be a significant effort. Some codes feature specially tuned assembler routines utilizing vector instructions (e.g. SSE2, Altivec), which are non-portable from one architecture to another. An example of this is Gromacs, as described in section 5.10.

Another issue is the memory configuration. In many MPP machines, such as the IBM Blue Gene/P, the amount of memory per core is small, posing problems for programs with high memory consumption.

In addition to compiling a fully functioning program, one must face challenges in extracting full performance from the machine. Cache-sizes, width of vector units and other issues affect the optimal way to utilize a processor. Also, the parallelization strategy can be well suited for one architecture, but not for another.

Finally, disk input/output (I/O) is in some case affected by endianness of the CPU. If one writes out data in a raw format there can be problems when reading the data on another machine, but external I/O libraries are in many cases able to handle this situation.

3.2 Software environment

In clusters the operating system (OS) is normally Linux with complete support for all kernel features. This is not the case for MPP systems, such as Cray XT3/4/5 and IBM Blue Gene/P.

These are running lightweight operating systems on the nodes, giving applications maximum access to hardware. This yields low OS jitter, which enhances applications performance. On the flipside the porting requires cross compilation and the operating system does not support all features. On such platforms one can encounter cross compilation issues, as was the case for GPAW (section 5.9).

Most libraries for HPC are portable and often readily available as pre-compiled modules on the target machine. If the library is vendor specific, commercial, or in limited use in the HPC community, this might not be the case. In some cases the program is also dependent on a specific library version, which may not be available on the machine.

If a pre-compiled version of the library is not available one has to port it to the target machine. Thus, the single task of porting an application to an architecture can actually include more than one individual porting effort.

If the source code of the library on which the program relies is not available, one has to alter the program to use another library. If the library uses standardized interfaces this is trivial; examples of this are the BLAS (Basic Linear Algebra Subroutines) libraries. If the library uses non-standardized interfaces the task is non-trivial but still doable if another library providing similar features is available. For each library call one has to change the calling sequence to match the one of the new library.

Several different implementations of a library may be available on a system. The relative performance of these implementations can vary depending on the individual routine, size of input data, number of parallel processes and even the contents of the input data. It can thus be difficult to predict exactly which implementation is optimal for a certain application and input based on the results of other applications and inputs.

The following sections describe the most important library classes and their performance. The libraries are further described in the D6.6 report, chapter 6 (Assessment of Libraries for Petascale Systems).

3.2.1 *Dense Systems*

Dense matrix operations are very common in HPC. The BLAS (Basic Linear Algebra Subroutines) and LAPACK (Linear Algebra Package) libraries arguably are the de facto standard for these operations. Also many higher-level libraries, such as ScaLAPACK (Scalable LAPACK) and sparse solvers leverage these libraries. Thus, for many applications they are often the most critical routines from a performance perspective. Each PRACE prototype features at least one, often several optimized implementations of these libraries. The relative performance of different implementations can vary depending on the operation and size of the input data. For example, on a Cray XT5, ACML can outperform Cray's libsci in matrix multiplication with small vector sizes, but not with large ones.

PBLAS and ScaLAPACK are the parallel counterparts for the BLAS and LAPACK libraries. The libraries use the BLACS (Basic Linear Algebra Communication Subprograms) library for parallelization. Usually BLACS is linked with MPI as the communication backend. Thus BLACS, as well as PBLAS and ScaLAPACK are specific to a certain MPI library implementation. This in turn means that optimized versions of these calls can usually only be found in the system vendors' own libraries (who also provide their own MPI implementation), such as Cray libsci and IBM PESSL. In addition to the operation type and problem size, the performance of these routines depends on the parallel decomposition of the problem, most importantly the block size and process topology.

3.2.2 *Sparse Systems*

Sparse solvers are not standardized but the PETSc (Portable, Extensible Toolkit for Scientific Computation) framework is often used to provide a common interface for these libraries. Unlike dense matrices, the performance of sparse solvers is also affected by the sparsity pattern of the matrix (how the non-zero values are distributed). The optimal unrolling and prefetching parameters depend on this pattern and thus it is impossible to create general-purpose optimized kernels, which work on all types of patterns.

3.2.3 *Fourier Transforms*

While no standard fast fourier transform (FFT) library interface exists, they generally have a similar interface and many perform operations in two stages: planning and execution phase. In the planning phase a "plan" is created based on the size of the input data. The plan is a description of the combination of codelets (code fragments) that is optimal for solving the FFT for that particular size. In the execution phase the FFT is executed according to the plan.

The plan can be executed any number of times with different input data sets, provided they are of the same size. The thoroughness of the planning stage can usually be adjusted. The fastest algorithms simply estimate the best combination based on the architecture, whereas the more thorough ones run and measure the execution times of different combinations. The most extensive algorithms may take several minutes to complete and might provide only marginal performance improvements. However, their use may be justified if the same plan is reused thousands of times in the course of execution. Some FFT implementations also provide facilities to store measured plans in a file. This can be useful for optimizing applications for specific problem sizes and for avoiding the overhead of executing a thorough planning stage.

To be more specific, we can note that FFTW is perhaps the most common cross-platform FFT library. There are actually two FFTW versions, FFTW2 and FFTW3, with incompatible interfaces. FFTW2 should not be used unless its MPI parallelization facilities are absolutely needed as it is in general slower than FFTW3. An MPI parallelized version of FFTW3 is being developed, but it is not ready for production use.

3.2.4 *Intrinsic Mathematical Operations*

Many of the PRACE platforms feature libraries that can be used to replace a subset of the intrinsic math operations, such as exponential, sin etc., usually provided by the libm library. They tend to be relatively easy to use, often not requiring any changes to the application source code. Usually these functions omit normal error checking and exception handling and may degrade numerical accuracy. Thus it is recommended to carefully check the correctness of results when using them. Some libraries also provide array versions of intrinsic functions, which take arrays as input arguments and perform the specified function on each array member.

3.2.5 *Library support*

In Table 2 we have listed which libraries are available on which prototypes, categorized by their type.

		Louhi	Jugene	Huygens	Maricel	Juropa	Baku
Dense systems	**BLAS**	libsci ACML Cray IRT MKL	Goto ESSL	ESSL ATLAS	SDK	MKL	MathKeisan
	LAPACK	libsci ACML Cray IRT MKL	netlib Goto* ESSL	netlib ESSL	SDK	MKL	MathKeisan
	PBLAS & ScaLAPACK	libsci Cray IRT	netlib PESSL	netlib PESSL			MathKeisan
Sparse systems	**Sequential**	SuperLU MKL	ESSL HSL			MKL	
	Parallel	PETsc (CASK) SuperLU_dist Hypre MUMPS	PETsc PESSL MUMPS Hypre	MUMPS			
Fourier Transforms	**Sequential**	FFTW 2,3 ACML CRAFFT MKL	ESSL FFTW 2,3	ESSL FFTW 2,3	SDK	MKL	MathKeisan
	Parallel	FFTW 2	PESSL FFTW 2	PESSL FFTW 2			MathKeisan
Random Number Generator		ACML	ESSL/PESSL	ESSL/PESSL SPRNG2	SDK		
Intrinsic math		Cray FastMV ACML-MV	MASS	MASS	MASS SDK	MKL	
Other		ParMETIS	IMSL sundials ParMETIS	NAG		MKL, NAG	

Table 2: Available libraries on the PRACE prototypes

In Table 3 we have listed the libraries that the applications require. It is evident that BLAS, LAPACK and FFTs are among the most widely used libraries.

Code	**Libraries**
Alya	Metis, Libspe2, pthread
AVBP	Metis, HDF5
BSIT	Librt, Libnuma, Libspe2, pthread
Code_Saturne	BLAS; optional: CGNS, Metis, HDF5, MED, SCOTCH
CP2K	BLAS, LAPACK, ScaLAPACK, FFT (ACML, FFTW, ESSL)
CPMD	BLAS, LAPACK, FFT (FFTW)
EUTERPE	BLAS, LAPACK, FFT (ESSL, FFTW), PETSc or WSMP
Gadget	FFT(FFTW 2), GSL
GPAW	NumPy, BLAS, LAPACK, ScaLAPACK
Gromacs	FFT(FFTW 3), BLAS, LAPACK

HELIUM	ESSL
NAMD	FFT(FFTW 2), TCL
NEMO	-
NS3D	EAS3, (FFT)
Octopus	FFTW, BLAS, LAPACK, GSL
PEPC	-
QCD	-
Siesta	BLAS, LAPACK, ScaLAPACK
Quantum ESPRESSO	BLAS, LAPACK, ScaLAPACK, FFT (FFTW, ACML, MKL, ESSL)

Table 3: Libraries used by the PABS applications

3.3 Compilers

The foremost requirement for the compiler is that it has to be able to produce a correctly functioning binary. In some cases this requirement is not fulfilled for one of the reasons listed below.

- The source code of the application uses non-standard extensions of a specific compiler. To compile such a program with another compiler often requires extensive modifications to the source code and is thus often cumbersome and error prone.
- The compiler does not support the language features or version that the program uses, e.g., advanced Fortran 95 & 2003 features.
- The source code exposes a bug in the compiler.

There are several ways of attacking these problems. First, if the program supports a cross-platform compiler then one can use it on all supported platforms, e.g., the GNU compiler suite is supported on most platforms but they do not always provide the same performance as vendor provided compilers. In Table 4 we have listed the currently supported compilers on each prototype. Second, if the program only supports vendor or architecture specific compilers the problem has to be approached by solving the underlying incompatibility. The amount of work this entails can be significant. It can also be undesirable to do so in case one looses too much performance on the original platform.

Prototype	GNU	PGI	Path-Scale	Vendor provided compiler
Cray XT5 - Louhi	X	X	X	Cray Compiler Environment (CCE)
IBM Blue Gene/P - Jugene	X			IBM XL
IBM Power6 cluster -Huygens	X			IBM XL
NEC SX-9 - BAKU				NEC compiler
Nehalem cluster - BAKU	X	X		Intel compiler suite
Sun/Bull Nehalem cluster - JuRoPa	X			Intel compiler suite
IBM Cell cluster - MariCel	X			IBM XL (PPU & SPU versions)

Table 4: Compiler support on the PRACE prototype systems.

3.3.1 *Optimisation flags*

After a fully working set of compilers has been identified the next stage is to identify an optimal set of flags for each compiler. After the optimal sets have been identified, one can pick the one providing the best performance.

Considering the large number of compiler flags it is clear that the number of possible combinations is extremely large, making an exhaustive trial of all combinations impossible. Thus, one cannot expect to find the optimal solution, only a very good suboptimal solution. There are essentially two ways of finding such a solution. The first is based on experience, intuition and knowledge, the second on automatic iterative compilation techniques.

Programmers typically have experience and knowledge of a set of compiler flags, which have produced good results for other programs. The compiler manual is also a good source for promising compiler flag candidates. Based on these, and knowledge of the algorithm and its implementation, one can produce a limited number of best guesses. From these candidates one then picks the best performing one. Assuming the optimisation flags are not interdependent, one can also follow a procedure where one adds flags one by one, accepting a flag only if it reduces the runtime. This is the strategy followed in porting the HELIUM code to the Cray XT5 architecture. As the flags are interdependent, this procedure is unlikely to produce the optimal result.

3.3.2 *Iterative compilation*

In this project we have also studied iterative compilation techniques. The goal was to optimize the compiler flags for four synthetic benchmark kernels and to extract a set of compiler flags that is generally useful.

Iterative compilation techniques attempt to tune the compilation by iteratively compiling and running the program. Different algorithms can be used to evolve the compilation procedure during the optimisation. Noteworthy examples are the GCC compiler of the milepost project (www.milepost.eu), and the genetic algorithm (GA) implemented in the Pathscale compiler. Profile guided optimisation in which collected runtime statistics are used to guide the compiler can also be thought of as such a technique, requiring only two compilations.

Here we have developed a simple GPL licensed program named Opfla that implements a genetic algorithm to optimize compiler flags. It's written in Python and comprises 1100 lines of code. It's easy to adapt to new platforms and programs as all the relevant information for compiling and running programs are described in a set of XML files; we successfully ran it on the Cray XT5, IBM Blue Gene/P, IBM Power6 cluster, and the NEC SX-9 prototypes.

Algorithms

Opfla implements two algorithms. The first algorithm is a genetic algorithm that we use to find a sub-optimal set of compiler flags. In the algorithm one first calculates a reference time by compiling the program using a reference flag (usually `"-O2"`) and running the resulting binary. A reference result is extracted from the output to verify subsequent programs. In the next step one creates a population comprising 50 random compiler flag sets. The performance of these are measured, and using a stochastic criteria the best ones are allowed to produce the next population generation by combining the flag sets. The algorithm also introduces some random changes to simulate mutation, this is important to make sure phase space is explored as efficiently as possible. Once one has the new population it's iteratively refined using the proceeding step. We store all solutions and are able to pick the best one.

The second algorithm, flagpruner, is able to remove the least important flags from a set of compiler flags. This is important as GA in practice produces a flag set with tens of flags, but most of these are not important. In fact we find that most flags tend to have no impact on the produced binary, as verified by computing the MD5 sum of each binary. The algorithm iteratively removes the least important flag one after the other until all flags have been removed. The least important flag either has identical MD5 sum, or the least impact on performance.

The current version is mostly suitable for small test kernels as it requires of the order of hundreds to a few thousands compilation-execution cycles. The same compiler flags are also used for all files. To use this procedure for large-scale application tuning one would need to improve the optimisation algorithm to minimize the number of executions, and also enable it to tune optimisation flags on a per file, or per function, basis. Using identical compiler flags for all algorithms will always be a compromise. Additionally, the present scheme is unable to provide insert pragmas in the source code, which are useful for providing the compiler with in-depth information on how to compiler the code for optimal performance.

Test case and analysis procedure

The synthetic benchmark kernels that we studied were from the Euroben set of synthetic kernels: mod2m which performs a matrix multiplication, mod2as which performs a sparse matrix vector multiplication, mod2f which performs a 1D-FFT, and finally mod2h which produces random numbers. These kernels were written in C, hence no careful comparison of Fortran compilers was done. For more information on the Euroben kernels please see the PRACE report D6.3.2.

To give an example of the kind of information we can extract from these runs we take the output of the flagpruner algorithm on Cray XT5 for mod2am and mod2h using PGI 8.0.6 (Table 5 and Table 6). The kernels are quite different, and this is reflected in the optimal flags. For mod2am flags which optimize floating point performance and loop behaviour are important, while mod2h benefits from flags which tune inlining behaviour and unrolling. The relative importance of flags is also evident, as the most important ones are the last to be removed. It's also clear that some flags are dependent on each other in order to produce good performance, e.g., for mod2am `"-Mvect=sse -Mipa=fast"` gives good performance but having either flag separately looses all the speedup compared to the reference `"-fast"`.

Finally we note that for both codes there is a small set of flags, which provides close to optimal performance. One does not need to aim for a large number of flags to get good performance. This is even more the case for large codes with a mix of different algorithms.

Speedup	Flags
0.33	
0.97	-Mvect=sse
1.56	-Mvect=sse -Mipa=fast
1.60	-Mvect=sse -Mipa=fast -Mnoprefetch
1.63	-Mvect=sse -Mipa=fast -Mnoprefetch -Mvect=noaltcode
1.67	-Mvect=sse -Mipa=fast -Mnoprefetch -Mvect=noaltcode -Mnozerotrip
1.70	-Mvect=sse -Mipa=fast -Mnoprefetch -Mvect=noaltcode -Mnozerotrip -Msmart

Table 5: Output of the flagpruner algorithm on Cray XT5 when optimizing the flags for mod2am using PGI 8.0.6. Speedup is in relation to the reference flag, which is "-fast"

Speedup	Flags
0.89	-Minline=levels:11
0.96	-Minline=levels:11 -Munroll=n:16
1.07	-Minline=levels:11 -Munroll=n:16 -Msmart
1.15	-Minline=levels:11 -Munroll=n:16 -Msmart -Mipa=fast
1.19	-Minline=levels:11 -Munroll=n:16 -Msmart -Mipa=fast -O4
1.23	-Minline=levels:11 -Munroll=n:16 -Msmart -Mipa=fast -O4 -alias=traditional
1.25	-Minline=levels:11 -Munroll=n:16 -Msmart -Mipa=fast -O4 -alias=traditional –Mmovnt
1.27	-Minline=levels:11 -Munroll=n:16 -Msmart -Mipa=fast -O4 -alias=traditional –Mmovnt -Msmartalloc=huge

Table 6: Output of the flagpruner algorithm on Cray XT5 when optimizing the flags for mod2h using PGI 8.0.6. Speedup is in relation to the reference flag which is "-fast"

Cray XT5 - Louhi

On the Cray XT5 architecture there are four compilers which are relevant: The Portland Group's PGI Fortran/C/C++ Compiler, the PathScale Compiler Suite, the GNU compiler collection, and the Cray Compiler Environment (CCE). We have analyzed the flags of all of these compilers using the procedure described above.

For the PGI (8.0.6) compiler we used "-fast" as the reference. It provides in general good performance while still being fairly safe to use. Our experience is that it does break some programs. The results for the four synthetic kernels under study are presented in Table 7.

Kernel	Optimal flags	Speedup
Mod2am	-Mnoprefetch -Mipa=fast -Msmart -Mvect=sse -Mvect=noaltcode –Mnozerotrip	1.70
Mod2as	-Mipa=fast -Msmart -Mvect=sse -Mvect=noaltcode -Munroll=c:4 –Mnomovnt	1.28
Mod2f	-Mprefetch=plain -Minline=size:10 -Mipa=fast -Mvect=sse -Mvect=fuse -Mvect=uniform -Mnozerotrip -Munroll=c:8	1.27
Mod2h	-O4 -alias=traditional -Minline=levels:11 -Mipa=fast -Msmart -Msmartalloc=huge -Mvect=prefetch -Mnozerotrip -Munroll=n:16 -Mmovnt	1.28

Table 7: Optimal flags obtained using Opfla for the PGI compiler. The speedup is the increase in performance when compiling using these flags, compared with compiling with "-fast"

We also studied how general these flags were, by computing the speedup of each kernel when using the optimal flags of all the other kernels. This matrix is presented in Table 8. It's evident that mod2h differs a great deal from the other three; mod2h computes random numbers and is sensitive to loop optimisations and inlining. The other three are more dependent on good floating point performance. Loop optimisations and inlining do play a role also for these kernels.

	Mod2am-flags	Mod2as-flags	Mod2f-flags	Mod2h-flags
Mod2am	1.70	1.51	1.56	1.06
Mod2as	1.18	1.28	0.94	0.90
Mod2f	1.18	1.17	1.27	0.99
Mod2h	0.92	0.92	0.91	1.28

Table 8: Speedup of each kernel compiled with the flags presented in Table 7

For the PathScale (3.2.0) compiler we used "-O3" as the reference. The results for the four synthetic kernels under study are presented in Table 9. The results are not as good as for the PGI compiler, only marginal speedup was obtained for mod2am, mod2as and mod2f. Only for mod2h did we achieve significant speedup, underlining its different character. There are two possible reasons for the poor speedup: the reference "-O3" already provides close to optimal performance, or the optimizer is not able to find a good solution. The latter is a distinct possibility as PathScale has a much larger number of possible flags, and thus its phase space of solutions is much greater.

Kernel	Optimal flags	Speedup
Mod2am	`-OPT:ro=3 -OPT:treeheight=on -OPT:unroll_size=256 -IPA:linear=ON -CG:local_fwd_sched=on`	1.06
Mod2as	`-OPT:ro=2 -OPT:alias=typed -OPT:treeheight=on -OPT:unroll_times_max=8 -IPA:multi_clone=1 -CG:local_fwd_sched=on`	1.08
Mod2f	`-O3 -WOPT:mem_opnds=on -OPT:alias=typed -OPT:unroll_times_max=8 -OPT:unroll_size=256 -TENV:X=4 -IPA:multi_clone=2 -CG:prefetch=OFF -CG:cflow=off -LNO:prefetch=1`	1.10
Mod2h	`-OPT:malloc_alg=1 -OPT:unroll_times_max=16 -OPT:unroll_size=512 -IPA:linear=ON -CG:cflow=off`	1.64

Table 9: Optimal flags obtained using Opfla for the PathScale compiler. The speedup is the increase in performance when compiling using these flags, compared with compiling with "`-O3`"

We also studied how general these flags were by computing the speedup of each kernel when using the optimal flags of all the other kernels. This matrix is presented in Table 10. It is evident that the optimal flags for mod2am, mod2as and mod2f are not transferrable. They reduce performance in many cases compared with the reference flag.

	Mod2am-flags	Mod2as-flags	Mod2f-flags	Mod2h-flags
Mod2am	1.06	0.92	1.01	1.01
Mod2as	0.96	1.08	0.89	1.00
Mod2f	1.04	1.05	1.10	1.00
Mod2h	1.38	1.24	1.49	1.64

Table 10: Speedup of each kernel compiled with the flags presented in Table 9

For the GNU (4.2.0) compiler we used "`-O3`" as the reference. The results for the four synthetic kernels under study are presented in Table 11. Only marginal speedup was obtained for mod2as and mod2f. Only for mod2h and mod2am did we achieve a speedup greater than 10%.

Kernel	Optimal flags	Speedup
Mod2am	`-Os -fsched-stalled-insns=6 -fsched2-use-traces -ftree-ch -ftree-vectorize -funroll-loops -freorder-blocks -fno-cprop-registers`	1.58
Mod2as	`-O3 -fmodulo-sched -fsched-stalled-insns=3 -fsched2-use-superblocks -fprefetch-loop-arrays -fno-guess-branch-probability -ffast-math -frename-registers`	1.07
Mod2f	`-O2 -funsafe-loop-optimizations -falign-labels=8 -funsafe-math-optimizations -mfpmath=sse,387`	1.02
Mod2h	`-O2 -fforce-addr -fsched-stalled-insns=2 -ftracer -fprefetch-loop-arrays -fno-peephole2 -falign-functions=64 -falign-labels=8 -falign-jumps=64 -ffast-math -funroll-all-loops`	1.15

Table 11: Optimal flags obtained using Opfla for the GNU compiler. The speedup is the increase in performance when compiling using these flags, compared with compiling with "-O3"

We also studied how general these flags were, by computing the speedup of each kernel when using the optimal flags of all the other kernels. This matrix is presented in Table 12. It's evident that none of the flag sets are transferrable.

	Mod2am-flags	Mod2as-flags	Mod2f-flags	Mod2h-flags
Mod2am	1.58	1.20	1.00	1.16
Mod2as	0.73	1.05	0.95	1.02
Mod2f	0.99	0.91	1.02	0.96
Mod2h	1.12	1.14	1.00	1.15

Table 12: Speedup when computing the speedup of each kernel compiled with the flags presented Table 11

For the Cray compiler we used `"-O3"` as the reference. The results for the four synthetic kernels under study are presented in Table 13. Only marginal, or none, speedup was obtained. Our conclusion is based on this evidence that in addition to the normal `"-O3"` or `"-O2"` flags there is little speedup to be obtained from tweaking compiler flags.

Kernel	Optimal flags	Speedup
Mod2am	`-h fp3`	1.00
Mod2as	`-h nopattern -h ipa3 -h scalar1 -h zeroinc -h fp0`	1.08
Mod2f	`-h scalar3`	1.00
Mod2h	`-h zeroinc -h fp1`	1.00

Table 13: Optimal flags obtained using Opfla for the Cray compiler. The speedup is the increase in performance when compiling using these flags, compared with compiling with "-O3"

IBM Blue Gene/P - Jugene

On the IBM Blue Gene/P architecture only the IBM XLC 9.0 compiler was studied. We used `"-O2"` as the reference. The results for three synthetic kernels under study are presented in Table 14. Impressive speedups were obtained for all kernels, but one has to keep in mind that this flag does not do any aggressive optimisations. Comparing with more generally used more aggressive flags would yield less impressive speedup.

Kernel	Optimal flags	Speedup
Mod2am	`-O5 -qalias=noansi -qdirectstorage -qignerrno -qlibansi -qnoprefetch -qinline -qmaxmem=8192 -qhot -qipa=missing=isolated -qipa=inline –qdatalocal`	2.38
Mod2as	`-qipa=level=2`	1.3
Mod2f	`-O2 -qarch=450 -qignerrno -qnoprefetch -qhot=arraypad -qipa=level=2 -qipa=missing=pure –qnostrict`	1.55

Table 14: Optimal flags obtained using Opfla on Blue Gene/P with the IBM XLC 9.0 compiler. The speedup is the increase in performance when compiling using these flags, compared with compiling with `"-O2"`

IBM Power6 cluster - Huygens

On this architecture only the IBM XLC 9.0 compiler was studied. We used `"-O2"` as the reference. The results for the four synthetic kernels under study are presented in Table 15. Very good speedups were obtained for all kernels, especially mod2am and mod2h, but one has to keep in mind that the `"-O2"` flag does not do any aggressive optimisations.

Kernel	Optimal flags	Speedup
Mod2am	`-O5 –qinline`	3.00
Mod2as	`-qipa=level=2 -qipa=inline -qnostrict -qunroll=yes`	1.42
Mod2f	`-O3 -qtune=auto -qalias=noansi -qlibansi -qhot -qipa=level=2 -qipa=inline -qunroll=no`	1.40
Mod2h	`-qalias=noansi -qipa -qunroll=yes`	2.65

Table 15: Optimal flags obtained using Opfla for the IBM XLC 9.0 compiler. The speedup is the increase in performance when compiling using these flags, compared with compiling with `"-O2"`

NEC SX-9 - Baku

The mod2am and mod2a Euroben kernels have been run through Opfla on the SX for the Fortran and the C compiler, respectively. What turns out to be essential is the identification of data dependencies. In Fortran it is much easier for the compiler to recognize data independencies than in C. For the mod2am Kernel of Euroben a speedup of a factor of roughly 20 can be gained by switching on the `"-pvctl nodep"` option. This flag tells the compiler to assume no data dependencies in any loop, and thus allows a maximal vectorization. In the special case of this kernel this is fine, and results are still valid. But in a real code you should provide the dependency information to the C-compiler via pragmas (see section 3.7).

For the Fortran compiler an added global `nodep` compiler flag does not affect the mod2a performance, as the compiler can already recognize the independence in the crucial loops, thus it is sufficient to compile the code with `"-c hopt"`, which is 33% faster than the reference which was no flags in this case.

3.4 Porting to Cray XT5 - Louhi

Louhi is a Cray XT4/XT5 MPP supercomputer. Louhi has a theoretical peak performance of 102.2 Tflops/s, not counting the service nodes handling logins, I/O, etc., and an estimated Linpack performance of 76.5 Tflop/s.

The Cray XT4 partition encompasses 11 cabinets with 1012 compute nodes, each containing one 2.3 GHz AMD Barcelona quad core processor, giving 4048 cores in total. The Cray XT5 partition encompasses 9 cabinets with 852 compute nodes with two quad core processor each; 7 cabinets have 2.3 GHz AMD Barcelona quad core processors, while 2 cabinets have newer 2.7 GHz AMD Shanghai quad core processors. Each XT5 node is equipped with 1 to 2 GB/core 800 MHz DDR2 memory. The nodes in the XT4/XT5 system are connected via a fast proprietary SeaStar2+ network providing a high-bandwidth, low-latency, network.

The latter mentioned two cabinets are the official PRACE prototype system. In practice the whole XT5 partition has been used for the work described in this deliverable, the XT4 partition has not been utilized as its performance profile differs from the XT5. As the Shanghai processors were installed in the summer of 2009, the work described in this document has been carried out on Barcelona processors if the processor type hasn't been specified.

The operating system for the Cray XT line of computers was originally Catamount but has since been replaced by Cray Linux Environment (CLE). CLE is a stripped down version of Linux providing low overhead, and thus good performance. In practice porting programs to CLE is often easier than to Catamount but there are still pitfalls. One major missing feature is support for dynamic libraries; only static linking is supported. Support for this will be available in future versions of the operating system.

As regards libraries and environment variables, one should note that as mentioned above different libraries have different performance profiles, thus no general recommendation can be given. The supported (major) libraries are listed in Table 2.

From the porting effort done in PRACE we can note that Libsci was the clear favourite library for BLAS/LAPACK routines, Code_Saturne, CP2K and GPAW all utilized it. For GPAW it can be noted that the performance for ACML and Libsci was roughly equal. For enhancing ScaLAPACK factorization, one can also try to use Cray's Iterative Refinement Toolkit (IRT) by setting the environment variable `IRT_USE_SOLVERS=1`. It's a library that uses mixed precision, a rough answer is computed using single precision and the answer is then refined using double precision. For more information, see the intro_irt manpage.

For the CRAFFT FFT library one should ensure that the fftw_wisdom files from `/opt/xt-libsci/[vernum]/` directory are in the directory where you execute the program from.

For enhancing math intrinsic speed, one can try the Cray fast math library. This can be done by loading the libfast module (`'module load libfast'`), and recompiling and linking with the library (`'-lfast_mv'`). See the intro_fast_mv manpage for more information.

As to the MPI library there are several environment variables, described below, that can be tuned to improve its reliability and performance. All available variables are listed on the intro_mpi manpage.

One can improve the performance of MPI_Alltoall by changing the cut-off point below which the store and forward alltoall algorithm is used for short messages. The default cut-off is 1024 bytes, independent of the number of processes in the MPI job. In our experience this number is tuned for very large jobs using of the order of ten thousand cores. For smaller jobs our measurements show that this value should be increased. The optimal values on a XT5 under load are listed in Table 16. Alternatively one can use the CSCs tuned version of the MPI_Alltoall function. This version aggregates messages on-node before doing the alltoall communication between nodes. It also implements similar optimisations for MPI_Alltoallv, MPI_Allgather and MPI_Allgatherv. To use it, simply load the module ("module load testing/csc-mpi") and recompile. More information can be found in the Gromacs section of the PRACE D6.4 report.

Enabling an optimized memory copy that has been tuned for large messages and/or large number of processes by the variable `MPICH_FAST_MEMCPY`, can be advantageous if you run jobs using more than 256 processes.

The optimum message size limit for using the eager protocol (`MPICH_MAX_SHORT_MSG_SIZE`, default 128000 bytes) depends on the message profile of your application. Therefore either increasing or decreasing it may give better performance. In most cases, performance is increased when increasing the size of the eager limit. The penalty is that the buffers that are used to receive unexpected messages may run out.

On the Cray XT5 architecture one can run out of unexpected message buffers, which are 60 megabytes by default. This turns up if a large-scale program does not pre-post receives, or if one has increased the eager limit as described above. The unexpected message buffer size can be controlled via the variable `MPICH_UNEX_BUFFER_SIZE`. One can also activate `MPICH_PTL_SEND_CREDIT`, which enables a flow control algorithm that should avoid all buffer overruns. This was used by the NAMD application, see section 5.12.

MPI processes	Optimal MPICH_ALLTOALL_SHORT_MSG (bytes) value
16	8800
32	8000
64	4500
128	3800
256	2500
512	1600
1024	1400
2048	1300

Table 16: Optimal cut-off for short- and long-message AlltoAll algorithm as a function of the number of MPI processes.

As mentioned above, there are four relevant compilers on the Cray XT architecture: The Portland Group's PGI Fortran/C/C++ Compiler, the PathScale Compiler Suite, the GNU compiler collection, and the Cray Compiler Environment (CCE). PGI is the default compiler, but GNU and CCE are also officially supported by Cray. It can be noted that none of the compilers is faster in general. This was seen in the study in section 3.3.2, where all compilers, with the exception of CCE, were fastest for at least one kernel. Also the semi-optimal flags of

different applications in Table 17 show that three different compilers have been used. PGI is here the favourite, partly because it's the default compiler.

Each compiler is invoked using "`cc`" or "`ftn`". These commands use the currently loaded compiler environment (module) and implicitly handle the cross-compiling issues. It also links the program to the libraries of the currently loaded modules. It should also be noted that compiler flags specifying processor architecture are always active. This is the reason they haven't been explicitly used by PABS applications.

Code	Complier	Flags
Code_Saturne	PGI	`-O2 -fast`
CP2K	PathScale	`-O3 -OPT:Ofast -OPT:early_intrinsics=ON -LNO:simd=2 -intrinsic=PGI`
GPAW	PGI	`-fast`
GROMACS		`-fast`
HELIUM	PathScale	`-O4 -OPT:Ofast:unroll_analysis=ON -LNO:fusion=2:full_unroll_size=2000:simd=2 -LIST:all_options=ON`
Namd	GNU	`-O3 –funroll-loops`
Nemo	PGI	`-O3`
NS3D	PGI	`-fast`
PEPC	PGI	`-O3 -fast`
QCD	PGI	`-O3 -fast -Mipa`
Quantum Espresso	PGI	`-O3 -fast -r8`

Table 17: PABS compiler flags in Cray XT5

In Table 17 one can note that rather aggressive compiler optimisation seems to be feasible throughout PABS. A common feature in all the flag sets is the enabling of SSE vectorization. Enabling SSE vectorization does not always work, for NEMO activating it breaks the code. Normally this only happens in a particular routine. In that case one can get better performance by pinpointing the particular routine that doesn't work and disabling vectorization for it with a pragma.

Based on Table 7 and Table 17 we have extracted a list of compiler flags found to be meaningful for the PGI compiler:

- "`-fast`" is a flag that turns on a set of flags that generally provide good performance. In practice it often provides good, but not the best performance. -fast turns on: "`-O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align -Mflushz`"
- "`-O1`" to "`-O4`" Ever higher levels of optimisations.
- "`-Mipa=fast`" Enable InterProcedural Analysis (IPA). Also activates "`-O2`", at a minimum. fast chooses generally optimal "`-Mipa`" flags. May increase compile time significantly.

- "-Mvect=sse" Enables the usage of SSE instructions (included in -fast)
- -Msmart Enable AMD64-specific post-pass instruction scheduling.
- "-Msmartalloc=huge" Add a call to the routine mallopt in the main routine. Link in the huge page runtime library, so dynamic memory will be allocated in huge pages.
- "-Mnoprefetch" In some cases it can be beneficial to turn prefetching off, to avoid evicting data from cache too early.
- "-Mprefetch=plain" Turn on prefetching. In addition to plain there are also other options.
- "-Mvect=noaltcode" Do not generate alternative SSE code.
- "-Minline=levels:n" Inline n levels. Default n is one, it can be beneficial to attempt higher levels (e.g. 3,5 or even 10)
- "-Munroll=c:u" Try with u=4 or higher
- "-Munroll=n:u" Try with u=4,8 or 16.
- "-Munroll=m:u" Try with u=4,8 or 16.
- "-Mmovnt" Force generation of nontemporal moves.
- "-Mnozerotrip" Don't include a zero-trip test for loops. Use only when all loops are known to execute at least once.

For the PathScale compiler we have based on Table 9 and Table 17 extracted a list of compiler flags found to be meaningful. The conclusion is that "-O3" seems to work well for many cases. In the case of HELIUM, flags changing unrolling, loop fusion and vectorization improved performance by up to 30% compared with "-O3". Adding vectorization was also important for CP2K.

- "-O3" Turn on aggressive optimisation (-O2 is on by default).
- "-IPA" Inter-procedural optimisation
- "-LNO:simd={0|1|2}" Level of vectorization (SSE2), 0 is none and 2 is the most aggressive level
- "-LNO:fusion={0|1|2}" Level of loop fusion, 0 is none and 2 is the most aggressive one.
- "-OPT:ro=n" Specify the level of acceptable departure from our language floating-point, round-off, and overflow semantics. Values of n from 0 to 3 valid; 0 means no change in accuracy, while 3 means any mathematically valid transformation is allowed.
- "-OPT:treeheight=ON" Can be very important. Enables re-association in expressions to reduce the expressions tree height.
- "-OPT:unroll_size=n" Set the ceiling of maximum number of instructions for an unrolled inner loop, e.g. n=256
- "-OPT:alias=typed" Two pointers of different types cannot point to the same location in memory.

- `"-OPT:alias=restrict"` Distinct pointers are assumed to point to distinct, non-overlapping objects.
- `"-CG:local_fwd_sched=on"` Instruction scheduling
- `"-LNO:prefetch={0|1|2|3}"` Prefetching, 0 is none, 3 most aggressive.

For the GNU compiler we have based on Table 11 and Table 17 extracted a list of meaningful compiler flags. The extracted interesting flags are:

- `"-O2"` GCC performs nearly all supported optimisations that do not involve a space-speed tradeoff
- `"-O3"` Optimize yet more.
- `"-Os"` Optimize for size. `-Os` enables all `-O2` optimisations that do not typically increase code size. May be faster than `-O2` & `-O3`
- `"-ftree-ch"` Perform loop header copying on trees. On by default, except when Os is active in which case this often should be activated
- `"-funroll-loops"` Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop.
- `"-frename-register"` Avoid false dependencies by making use of registers left over after register allocation. Enabled by default with `"-funroll-loops"`.
- `"-fsched-stalled-insns=2"` 2 instructions can be moved prematurely from the queue of stalled instructions into the ready list, during the second scheduling pass. Can also try 4,6,... instructions.
- `"-fno-guess-branch-probability"` In some cases it can be useful to turn of branch guessing.
- `"-ftree=vectorize"` Perform loop vectorization on trees.
- `"-fno-cprop-registers"` After register allocation and post-register allocation instruction splitting, we perform a copy-propagation pass to try to reduce scheduling dependencies and occasionally eliminate the copy.

3.5 Porting to IBM Blue Gene/P - Jugene

Jugene is a MPP Blue Gene/P machine comprising 72 Racks with 73728 compute nodes. The peak performance of the whole machine is 1 petaflop and the Linpack performance is 825.5 teraflops. Each individual processor is quite slow, but the fast network allows programs to scale very well on the hardware, thus achieving extremely good performance. If a program is unable to scale, then the performance extracted from a BG/P machine might be less than from more traditional clusters or MPP machines with faster nodes. This is the case for Gromacs, among others. See report D6.4 for more details.

The nodes run a minimalistic operating system allowing programs to get maximum performance from each node. This unfortunately makes it more difficult to port applications, as one has to cross-compile for the compute nodes. Some of these issues were encountered when porting Gpaw, see section 5.9 for details.

Looking at the hardware in more detail it can be seen that each SMP node contains four PowerPC 450 processors running at 850 MHz. Each core has two floating point units, called "double hummer". Compiling for double hummer can improve performance in most cases ("`-qarch=450d`"), although special requirements, like memory alignment, have to be considered. Each node has 2 GB of memory, giving only 512 MB per core. This can be an issue for memory intensive programs. One can avoid it to some degree by choosing the correct execution mode

- *Virtual Node*: Each node has 4 MPI tasks, 1 thread per task
- *Dual*: Each node has 2 MPI tasks, 1-2 threads per task
- *SMP*: Each node has 1 MPI task, 1-4 threads per task

It can also be noted that there are three interconnect networks, which are specialized in different tasks. Firstly, there is a three-dimensional torus for the compute nodes. Secondly, a global tree network for collectives encompassing the compute nodes. Finally, there is a 10 Gigabit ethernet network for I/O Nodes. One can choose the topology for the program to achieve maximum performance using the `BG_CONNECTION` variable. The options are: `MESH` (default), `TORUS` or `PREFER_TORUS`. This choice has a big effect on the performance of the application. If the choice is unclear then TORUS is the safest and recommended option. It can be activated by adding "`# @ bg_connection = TORUS`" to the job script. The variable `BG_MAPING` determinates the MPI rank distribution over the partition, by default (`BG_MAPING=XYZT`) the ranks are distributed first over the nodes and after each node has got one rank, the next #nodes ranks are distributed. Many applications might show better performance with `BG_MAPING=TXYZ`, i.e. the first node is filled first, then the next one, and so on and thus neighbouring ranks might share one node.

The available libraries are listed in Table 2. It can be noted that Jugene has a broad support of the basic HPC libraries.

Code	Compiler	Flags
AVBP	IBM XL	`-qstrict –qfixed -O3 -qhot -qmaxmem=-1 -qarch=440 -qtune=440`
Code_Saturne	IBM XLC	`-O3 -qtune=450-qarch=450`
	IBM XLF	`-O3 –qhot –qarch=450d -qtune=450`
CP2K	IBM XL	`-O2 -qarch=450 -qcache=auto -qmaxmem=-1 -qtune=450`
CPMD	IBM XL	`-O -w -qsmp=omp -qnosave -qarch=450 -c -I/bgsys/drivers/ppcfloor/comm/include`
Euterpe	IBM XL	`-qtune=450 -qarch=450 -O3 -qautodbl=dbl4 -qmaxmem=-1 -qflag=I:I -I$(OBJDIR) -qmoddir=$(OBJDIR) -qsuffix=cpp=F90 -qfixed`
Gadget	IBM XL	`-qtune=450 -qarch=450d -O5`
GPAW	IBM XL	`-O5 -qhot=nosimd -qlanglvl=extc99 -qnostaticlink -qsmp -qarch=450d -qtune=450 -qflag=e:e`
GROMACS	IBM XL	`-O3 -qarch=450d -qtune=450`
HELIUM	IBM XL	`-O4 -qarch=450d -qtune=450 -qlanglvl=extended -qfree=f90 -qrealsize=8 -qsuffix=f=f90 -qessl`
NAMD	IBM XL	`-Q3 -Q -qhot -qarch=450d -qtune=450 -DFFTW_ENABLE_FLOAT`
PEPC	IBM XL	`-qtune=450 -qarch=450d –O5`
QCD	IBM XL	`-qtune=450 -qarch=450d -O3 -qhot`
Quantum Espresso	IBM XL	`-O3 -qstrict -qsuffix=cpp=f90 -qdpc=e -qtune=450 -qarch=450 -qalias=noaryovrlp:nointptr -q32`

Table 18: PABS compiler flags for IBM Blue Gene/P - Jugene.

In Table 18 one can note that for all codes the IBM compiler was used, not like on the Cray XT5 system, where a broader range of compilers was employed. In the case of GPAW the GNU compiler was also tested, but the performance was 10-15% worse (see section 5.9 for details). A common feature in all the flag sets is the explicit setting of the architecture through `"-qtune"` and `"-qarch"`. Aggressive optimisation levels were also used by several codes, e.g., PEPC gained 25% in performance going from `"-O2"` to `"-qtune=450 -qarch=450d –O5"`.

There is a simple sequence of compiler flags for the IBM XL compiler that one can use to increase an application's performance. The results should be verified after every step.

First, one should always use one of the following two sets of flags:

1) `-qtune=450 -qarch=450`
2) `-qtune=450 -qarch=450d`

The latter one generates "double hummer" instructions.

Second, one should test the following flags in combination with the preceding ones and pick the fastest one with correct output. The options are:

```
1) -O2
2) -O3 -qstrict
3) -O3
4) -O3 -qhot
5) -O4
6) -O5
```

Based on the list above, Table 14 and Table 18 we have extracted a list of compiler flags found to be meaningful for the IBM XL compiler:

- `"-O2"` Basic optimisation level (default)
- `"-O3"` Intermediate level of optimisation
- `"-O4"` Enable the IPA (interprocedural optimizer) at compile time, limited scope analysis, SIMD instructions
- `"-O5"` enable the IPA (interprocedural optimizer) at link time, full program analysis, SIMD instructions
- `"-qtune=450"` Tune code for PowerPC 450 processors
- `"-qarch=450"` Target PowerPC 450 processor architecture
- `"-qarch=450d"` Target PowerPC 450 processor architecture and generate, where possible, double hummer instructions
- `"-qstrict"` No accuracy should be lost
- `"-qhot"` allow high order transformation, SIMD instructions.
- `"-qinline"` Enable inlining with default parameters (The compiler attempts to inline all appropriate functions with 20 executable source statements or fewer.)
- `"-qipa"` Enable IPA with default level=1: limited alias analysis, and limited call-site tailoring.
- `"-qipa=level=2"` Enable IPA with full interprocedural data flow and alias analysis.
- `"-qunroll=yes"` Instructs the compiler to search for more opportunities for loop unrolling than that performed with `"-qunroll=auto"` (default setting). In general, this suboption is more likely to increase compile time or program size than auto processing.
- `"-qnostrict"` Allow optimisations which may alter the semantics of a program.

3.6 Porting to IBM Power6 cluster - Huygens

Huygens is an IBM Power6 cluster clustered SMP (Symmetric Multiprocessing) system, with 104 nodes giving a peak performance of 60 Teraflops. The machine can be characterized as a fat-node cluster with each node containing 16 dual core processors (IBM Power6, 4.7 GHz).

Memory access is uniform in each node. The processors support symmetric multi threading (SMT) enabling one to run up to 64 threads or processes on each node.

The Power6 core is running at a higher frequency than other comparable processors, 4.7 GHZ. For floating point calculations there are two independent FPUs and a VMX unit. The VMX unit is a derivative of the Altivec unit of older architectures and thus only supports single precision. Each core has its own 128 KB L1 cache. L2 is 8 MB in total; 4MB are assigned to each core and both cores have fast access to each others cache. The L3 cache has a size of 43 MB per processor. The network between the nodes is a regular Infiniband.

Enabling SMT is a key technique one can use to improve performance of a program. This enables one to utilize idle resources on the processor in case one of the threads is stalling. One example of its potential is shown in the NAMD report (section 5.12), especially in Figure 10.

The available libraries are listed in Table 2. Huygens has a broad support of the basic HPC libraries.

Code	Compiler	Flags
AVBP	IBM XL	`-q64 -qsuffix=f=f -O3 -qstrict -qtune=auto -qarch=auto`
Code_Saturne	IBM XLC	`-O4`
	IBM XLF	`-q64 -qfixed -I. -qextname -O3 -qtune=pwr6 -qarch=pwr6`
CP2K	IBM XL	`-O2 -q64 -qarch=pwr6 -qcache=auto -qmaxmem=-1 –qtune=pwr6`
CPMD	IBM XL	`-O -q64 -qmaxmem=32768 -qtune=pwr6 -qarch=pwr6 -qsmp=omp -qnosave -qextname=printmemsize`
Euterpe	IBM XL	`-q64 -qtune=pwr6 -qarch=pwr6 -qcache=auto` `-qextname=flush -O3 -qstrict -qautodbl=dbl4 -qmaxmem=-1 -qflag=I:I -I$(OBJDIR) -qmoddir=$(OBJDIR)-qfree=f90 -qsuffix=cpp=F90 -qfixed`
Gadget	IBM XL Double precision	`-q64 -O5 -qipa=exits=endrun,MPI_Finalize:inline=auto`
	IBM XL Single precision	`-q64 -qtune=pwr6 -qarch=pwr6 -O5 -qipa=exits=endrun,MPI_Finalize:inline=auto -qhot=vector -qenablevmx -qnostrict -qunroll=yes`
GPAW	IBM XL	`-O3`
GROMACS	IBM XL	`-O3 -qstrict -qarch=pwr6 -qtune=pwr6`
HELIUM	IBM XL	`-qfree=f90 -O4 -qessl -qarch=auto -qtune=auto –qhot`
NAMD	IBM XL	`-q64 -arch=auto -qtune=auto -qcache=auto` `-O3 –qnohot -qunroll`

NEMO	IBM XL	`-O3`
NS3D	IBM XL	`-O5`
PEPC	IBM XL	`-qtune=pwr6 –qarch=pwr6 –O4`
QCD	IBM XL	`-qtune=pwr6 –qarch=pwr6 –O4`
Quantum Espresso	IBM XL	`-q64 -qarch=auto -qtune=auto -O2 -qsuffix=cpp=f90 -qdpc -qalias=nointptr -Q -qautodbl`

Table 19: PABS compiler flags for IBM Power6 cluster - Huygens

In Table 19 we have listed the compiler flags used by the codes. It can be noted that most have used the `"-qarch"` and `"-qtune"` options, as one should. Many codes have used aggressive optimisation levels (`"-O4"` and `"-O5"`), but for example CP2K did not work above `"-O2"` and NEMO was unstable above `"-O3"`. The optimal compiler flags can also be dependent on issues such as SMT, the ones listed here for NAMD are optimal when using it (see section 5.12 for more information).

As for the Blue Gene/P there is a simple sequence of compiler flags for the IBM XL compiler that one can use to increase an application's performance. The results should be verified after every step.

First, one should use the following options at all times:

```
-qtune=auto -qarch=auto -qmaxmem=-1
```

Second, try the following options in sequence and pick the fastest one with correct output. The options are:

```
1) -O2
2) -O3 -qstrict
3) -O3
4) -O3 -qhot
5) -O3 -qhot=nosimd
6) -O4
7) -O5
```

If the program in question computes in single precision on can also try to automatically tune for VMX, the following options are recommended by IBM for automatic generation of VMX instructions for suitable code segments. Note that appropriate alignment and pointer conditions should be met.

For C:

```
–qarch=pwr6 –qtune=pwr6 –qhot=simd –qnablevmx –qaltivec
-qipa=malloc16 -qalias=noaryovrlp,nopteovrlp
```

For Fortran:

```
–qarch=pwr6 –qtune=pwr6 qhot=simd –qnablevmx
-qalias=noaryovrlp,nopteovrlp
```

It can be noted that for GADGET two alternative compiler flag sets were used, with one of them supporting automatic generation of VMX instructions. This gave a speedup of 18% compared with the regular double precision version.

Based on the list above, Table 15 and Table 19 we note that essentially the same flags are interesting on this platform, as on the Blue Gene/P platform. Please look at section 3.5 for more information.

3.7 Porting to NEC SX-9 / Nehalem cluster - Baku

Baku is a hybrid prototype available at HLRS, providing a SX-9 vector system alongside an Intel Nehalem cluster with 64 Nvidia Tesla GPUs. The peak performance of the SX-9 partition is 19.7 TFlops, with 192 processors on 12 nodes. The cluster partition comprises 700 dual socket nodes with Intel Xeon (X5560) Nehalem running at 2.8 GHz. Additionally 32 nodes provide 2 Nvidia Tesla GPGPUs each (physically located in 16 Nvidia S1070 enclosures).

Most of the work within PRACE has been concentrated on the vector SX-9 partition, as the Nehalem cluster part is a recent addition to the machine. Some interesting work on utilizing the GPGPUs using the PGI compiler has been detailed in the NS3D report (section 5.14).

Usually getting fairly standard conforming code running on the NEC-SX platform is not a big issue, however many programs achieve only very poor performance without some tuning. The most essential thing one needs to do, to gain any advantage from the vector machine is to identify the vectorizable loops within the code, and allow the compiler to recognize them. A common task, which needs to be done for this, is the changing of loop orders in nested loops. This might be necessary even across subroutine boundaries. If the code is written in C one should also provide the C-compiler the information, that there are no dependencies in a given loop via compiler directives: `#pragma cdir nodep`.

Code	Compiler	Flags
NS3D	NEC	`-Chopt`
Quantum Espresso	NEC	`-ftrace -f2003 nocbind -float0 -Cvopt -eab -R2 -P openmp -Wf,-Ncont,-A dbl4,-P nh,-ptr byte,-pvctl noifopt loopcnt=9999999 expand=12 fullmsg vwork=stack,-fusion,-O noif,-init stack=nan heap=nan`

Table 20: Compiler flags for NEC SX-9 part of Baku

You should ensure, that your code can be compiled with `"-C hopt"`, any issues this produces should be resolved by source code transformations, as these are good hints, where there might be performance issues. For application written in C you might also try to compile your code at least partially with `"-C aopt"`, the most aggressive optimisation flag. This might reveal even more spots, where tunings might be necessary.

For efficient I/O it might be necessary to change the default buffer size. This can be done for Fortran by the environment Variable `'F_SETBUF'` and for C by the environment Variable `'C_SETBUF'.`

3.8 Porting to Sun/Bull Nehalem cluster - JuRoPA

As this prototype was available at a late stage of the project, there are no best practices to report.

3.9 Porting to IBM Cell cluster - MariCel

MariCell is a IBM cluster comprising 72 QS22 computational nodes with 2 PowerXCell 8i processors each, and 12 JS22 service nodes, each containing 2 Power6 processors. The peak performance of the QS22 (Cell) partition is 15.6 Teraflops (double precision). The interconnect is an InfiniBand 4 x DDR. For detailed information on the Cell processor please read section 4.6.

Porting to the Cell is quite different from porting to the other prototypes. It's fairly straightforward to port an application if one only uses the PPU. The PPU is quite slow and thus one needs to use the SPUs to get competitive performance. To use the SPUs one typically has to rewrite the computationally heavy parts using the Cell SDK, or some other programming model. Issues involved in porting codes to the Cell using low-level programming methods are described in section 4.6. There are also other programming models one can use on the Cell, for more details see the PRACE report D6.6.

Several applications were ported to the prototype within this project, but only Alya, BSIT and Siesta were ported to utilize the SPUs. In both codes the effort involved was significant. Please read sections 5.3 and 5.18 for more information on this effort.

Code	Compiler	Flags
ALYA	IBM XL PPU	`-qflag=i -qnoescape -qsuffix=f=f90 -qextname=flush,etime -qfree=f90 -qmoddir=$O -I $O -c -q64 -qarch=cellppu -qtune=cellppu -O3 -Wl,-q`
BSIT	IBM XL PPU	`-qflag=i -qnoescape -qsuffix=f=f90 -qextname=flush,etime -qfree=f90 -qmoddir=$O -I $O -c -q64 -qarch=cellppu -qtune=cellppu -O5 -Wl,-q`
CPMD	IBM XL	`-O3 -q64 -qarch=cellppu -qtune=cellppu`
Euterpe	IBM XL	`-q64 -qtune=cellppu -qarch=cellppu -qcache=auto -qfree=f90 -qextname=flush -O3 -qstrict -qautodbl=dbl4 -qmaxmem=-1 -qflag=I:I -I$(OBJDIR) -qmoddir=$(OBJDIR) -qsuffix=cpp=F90 -qfixed`
Siesta	IBM XL PPU Fortran	`-qsuffix=cpp=F -c -qfixed -O3 -qstrict -qtune=cellppu -qarch=cellppu -qhot=simd -q64 -qipa=malloc16 -qextname -qalign=struct=natural`
	IBM XL PPU C	`-O5 -qstrict -qtune=cellppu -qarch=cellppu -qhot=simd -q64 -qinline`
	IBM XL SPU C	`-O5 -qhot=simd –qarch=cellspu -qtune=cellspu –qcpluscmt`

Table 21: Optimal compiler flags for the PRACE benchmark applications on the Cell prototype

In Table 21 we have listed the compiler flags used by the applications. Again, we see similar flags as for the other two IBM platforms. One should note that there are two compiler targets on the Cell processor, the SPU and the PPU, and one should target the compilation accordingly.

4 Optimisation

Software optimisation is a very intricate task. It requires deep understanding of the execution behaviour of the candidate application and very good knowledge of the architectural characteristics of the underlying execution platform. Although the optimisation suite includes a large number of techniques, in several cases only a specific combination of them under careful tuning is able to provide meaningful performance improvements. This is due to the fact that each optimisation technique attacks distinct performance bottlenecks that need to be dominant in the software under consideration, for the optimisation to be effective. More importantly, some code optimisations, or combinations of them, may even be harmful to the performance of the application.

There is no standard optimisation methodology that can be applied to any pair of application and execution platform. Nevertheless, one can provide a set of relevant guidelines that can reasonably assist in the code optimisations process: At first, the user needs to clearly measure and assess the initial performance of the application. The peak performance of the application is heavily dependent on the implemented algorithm and the peak performance of the system. A comparison of the achieved performance with the one provided by *performance modelling* of the algorithm and the machine's peak performance can provide a solid basis for the performance bounds and the potential of optimisation.

The second step is to discover the hot spots in the execution of the application (i.e. the code parts in which most of the execution time is consumed) and to characterize the performance bottlenecks. For single-threaded applications, major bottlenecks are *disk I/O*, *memory* (latency or bandwidth) and *CPU*. For multi-threaded applications *synchronization* may consume a large part of the execution time, and in several cases can kill performance. *Performance analysis* tools can greatly assist in locating hot spots and characterizing performance bottlenecks.

Heavy access on the disk can be a severe impediment to high performance, since the disk is the slowest in the memory hierarchy, and should definitely be given the highest priority before proceeding to optimisations attacking other bottlenecks. In general, the programmer needs to ensure that the disk is accessed in the most coarse-grained way and promote data reuse. Beyond that, the operating system and disk hardware play the most important role in disk I/O optimisation. Since, beyond the aforementioned general guidelines, the involvement of the developer in the optimisation of disk I/O bottlenecks is rather limited, we focus our attention on techniques to alleviate the bottlenecks in memory, CPU and synchronization that will be discussed in detail in the forthcoming paragraphs. Thus, the third step in a rough optimisation process is to select candidate optimisations for the specific performance problem and proceed in a trial-error-tune cycle.

Modern optimizing compilers incorporate a rich set of automatic code optimisations. The selection of a successful combination of them requires at least an understanding of their impact on the code. However, since the compiler's primary criterion is to preserve program correctness, in several cases it is not able to perform a specific code optimisation, being conservative in the code changes it applies. In these cases the developer needs to implement the optimisation himself, develop the code in a way that the compiler can safely apply the optimisation, or, if this option is supported by the compiler, instruct the compiler that the candidate optimisation is legal.

Overall, code optimisation seems to be more of an art than a science. This means that, although there are a number of steps that one can follow to achieve higher performance, there is no absolutely safe path to performance improvement. The famous quote by Donald Knuth

"premature optimisation is the root of all evil" designates that optimisation should be performed wisely, step-by-step and always supported by solid experimental data of the achieved performance in each step.

4.1 Memory hierarchy optimisation techniques

The programmer can apply a number of memory hierarchy optimisation techniques. Some of the memory-related problems these techniques try to address are memory latency, memory bandwidth, TLB misses, cache misses, false sharing. Memory latency becomes a problem when data are accessed irregularly (non-sequentially), for example when accessing linked lists, graphs represented by adjacency lists, etc. Memory bandwidth, on the other hand, becomes a limitation when large amounts of consecutive data are accessed in a streaming way, with light computation performed on them. False sharing arises in systems with distributed, coherent caches, when a system participant attempts to periodically access data that is not altered by another party, but that data shares a cache line with data that is altered. In that case, the coherence protocol may force the first participant to reload the whole cache line. In the following paragraphs we discuss techniques that are able to improve the access of data to main memory and significantly increase the performance of programs that suffer from issues related to bottlenecks in the memory hierarchy.

A general guideline for improving memory performance is enhancing the locality of reference. There are two basic types of locality: spatial and temporal locality. Spatial locality refers to the use of data that are stored relatively closely in memory, whereas temporal locality refers to the reuse of the same data within a relatively small amount of time. It is critical to enhance both types of locality, as hierarchical memory is designed with the locality principle in mind. Therefore, programs can benefit by using data that are already cached in the upper levels of the memory hierarchy and avoiding bringing other data into the upper levels of the hierarchy, displacing data that will be used shortly in the future.

4.1.1 *Blocking*

Blocking (or loop tiling) is an optimisation that partitions the iteration space of a loop into smaller blocks, in order to maximize accesses to the data loaded into the cache before the data are replaced. The technique leads to partitioning a large array into smaller blocks, which fit into the cache size, thus mitigating cache misses. Consider, for example, the matrix multiplication loop:

```
for (i=0; i<N; i++)
     for (j=0; j<N; j++)
          for (k=0; k<N; k++)
               x[i][j] = x[i][j]+y[i][k]*z[k][j];
```

By applying blocking, the loop becomes:

```
for (jj=0; jj<N; jj+=B)
     for (kk=0; kk<N; kk+=B)
          for (i=0; i<N; i++)
               for (j=jj; j<min(jj+B,N); j++)
                    for (k=kk; k<min(kk+B,N); k++)
                         x[i][j] = x[i][j]+y[i][k]*z[k][j];
```

With tuning of the block size (B), the cache misses can be greatly reduced. In the previous example, blocking exploits a combination of spatial locality and temporal locality, as *y* benefits from spatial locality and *z* benefits from temporal locality. Blocking can also be used

to help register allocation. When block sizes are chosen to be small enough so that the whole block can be stored in registers, memory accesses are avoided (register blocking). Blocking is a major loop optimisation heavily applied by vendors in their implementations of scientific software (BLAS, LAPACK, FFT, etc).

4.1.2 *Loop permutation*

Loop permutation (loop interchange) is the process of exchanging the order of two iteration variables, in the case of nested loops. Loop permutation is useful when the initial loop accesses data in non-sequential order, while the permutated loop accesses data in the order they are stored. Therefore, this technique aims to reduce misses by improving spatial locality. For example, consider the following loop:

```
for (j=0;j<100;++j)
      for (i=0;i<100;++i)
            a[i][j] = i + j;
```

Assuming the elements of the matrix are stored row-major, the memory is accessed in strides of 100 words. The spatial locality can be improved by interchanging the two loops:

```
for (i=0;i<100;++i)
      for (j=0;j<100;++j)
            a[i][j] = i + j;
```

Another advantage of loop permutation is that it does not affect the code size. However, the programmer must keep in mind that there are certain constraints regarding loop permutation. It is not always safe to exchange the iteration variables due to dependencies between statements for the order in which they must execute. To determine whether loops can be interchanged safely, dependence analysis is required.

4.1.3 *Loop fusion*

Loop fusion is a loop transformation that replaces multiple loops by combining them into a single one. Suppose the original code has the form:

```
for (i = 0; i < 100; i++) {
      a[i] = 1;
}
for (i = 0; i < 100; i++) {
      b[i] = 2;
}
```

The two loops can be fused so that the loop looks like:

```
for (i = 0; i < 100; i++) {
      a[i] = 1;
      b[i] = 2;
}
```

Using a single loop, the cost of testing and branching is mitigated. Loop fusion can also benefit register reuse. However, there are cases where loop fusion can hurt performance instead of improving it, for example if the data locality within each separate loop is affected by their merging.

4.1.4 *Loop distribution*

Loop distribution (loop fission) is the opposite of loop fusion. With this technique, a loop is broken into multiple loops over the same index range, the body of which consists of a part of the initial loop's body. Consider the following loop:

```
for (i = 0; i < 100; i++) {
     a[i] = 1;
     b[i] = 2;
}
```

After loop distribution, the code looks like:

```
for (i = 0; i < 100; i++) {
     a[i] = 1;
}
for (i = 0; i < 100; i++) {
     b[i] = 2;
}
```

Loop distribution is useful when the data locality of the memory references inside the loop body needs to be improved. It is also helpful when a loops needs to be split into multiple tasks in order to be executed by a multiprocessor, as it can convert a sequential loop to multiple parallel loops.

4.1.5 *Prefetching*

Data prefetching is a technique to hide cache miss latency by bringing data closer to the processor before the processor requests it. One way to initiate prefetching is by inserting prefetch instructions into the program. This insertion is typically performed statically by the compiler and is called software prefetching. Prefetching can also be initiated dynamically by observing the program's behaviour. This dynamic prefetch generation is typically performed by hardware (hardware prefetching). In the case of software prefetching, the programmer or the compiler needs to insert explicit prefetch instructions, provided by the architecture's instruction set. These instructions are essentially non-blocking memory loads, which bring the data into the processor's cache prior to its use, but have no other side effects. In this way, the cost of memory access is overlapped with useful processor work. Prefetch instructions are NOPs from the processor's standpoint and thus are merely hints to the memory system, which can ignore them when resources are scarce. Historically, software prefetching has been quite effective in reducing memory stalls for scientific programs that reference array-based data structures (regular memory access patterns). More recently, it has been applied to programs that reference pointer-based data structures as well (pointer-chasing codes).

Hardware prefetching techniques have two main advantages over software prefetching techniques: first, they have better dynamic information, and therefore can recognize unexpected cache conflicts that are difficult to predict in the compiler, and second, they do not add any instruction overhead to issue prefetches. On the other hand, the primary difficulty related to hardware prefetching is detecting the memory access patterns. However, hardware data prefetching techniques have been shown to successfully reduce the memory stall times associated with sequential, stride and some pointer-chasing access patterns. Sequential prefetching detects and prefetches for accesses to contiguous locations. Stride prefetching detects and prefetches accesses that are *s* cache blocks apart between consecutive accesses, where *s* is the amount of the stride. For more complex access patterns, resulting either from accesses to linked data structures or from indirect array references, there are more

complicated schemes such as correlation prefetching and content-directed prefetching. Correlation prefetching relies on the fact that the sequence of the accesses is often repeated during program execution. Content-directed prefetching tries to identify pointers in linked data structures and initiate prefetching of data before the actual pointer dereferencing. Although hardware prefetching undertakes the work automatically, programmers can consider the existence of such architectural mechanisms, so that their programs effectively trigger the prefetching units.

4.1.6 *Overlap of memory transfer and computations (double-buffering)*

When a code segment consists of repeatedly operating calculations on some buffer and then waiting for new data to be transferred into the same buffer, in order to be operated on, a considerable amount of time might be spent in waiting for the transfer to complete. When the total time needed to complete the transfer is not negligible with respect to the total execution time, the code segment can benefit significantly from double buffering. Double buffering is a form of multibuffering, the method of using multiple buffers in a circular queue to overlap computation and data transfer. Specifically, it is possible to speed up the process by allocating two buffers and overlapping computation on one buffer with data transfer in the other. The method is better explained by the following pseudocode:

```
int next_idx, buf_idx = 0;
// Initiate transfer
init_xfer(B[buf_idx]);
while (--buffers) {
     next_idx = buf_idx ^ 1;
     // Initiate next transfer
     init_xfer(B[next_idx]);
     // Wait for previous transfer to complete
     wait_xfer(buf_idx);
     // Use the data from the previous transfer
     use_data(B[buf_idx]);
     buf_idx = next_idx;
}
// Wait for last transfer to complete
wait_xfer(buf_idx);
// Use the data from the last transfer
use_data(B[buf_idx]);
```

4.1.7 *Cache-line issues*

There are a number of issues related to cache lines that can result in performance degradation. As a first example, it is in most cases beneficial to store data aligned with respect to cache line boundaries. Accessing unaligned data with size less or equal than the cache line size, can lead to more than just one cache misses. Moreover, it is a good practice to try to store data that are used together in the same cache line, in order to improve the spatial locality of the program.

Cache thrashing occurs when a program accesses memory in a pattern that leads to multiple main memory locations mapping to the same cache address. Each time one of these items is brought into the cache, it overwrites another needed item, resulting in excessive cache misses. All benefits from locality of reference are, therefore, lost, since data reuse is no longer possible. The problem is more intense for caches with low associativity, and also for unified caches. A common case that leads to cache thrashing is the allocation of arrays in memory locations with a distance that is a multiple of the cache line size. The problem is easily fixed

by increasing the distance between the arrays. This can be accomplished by either increasing the array sizes or inserting a padding array.

Additionally threaded programs may run into false-sharing problems, as described in section 4.4.

4.2 Computational optimisation techniques

In the cases that the performance bottleneck resides inside the processor, i.e. the application under consideration is computationally bound, a different suite of optimisation techniques has been proposed to reduce the execution time. The targets in this case are to reduce the instructions executed by the processor, to legally change their order (so as to better utilize the processor's units) or perform them in bulk, provided the architecture supports vector parallelism. The following paragraphs describe approaches that achieve the above targets.

4.2.1 *SIMD instructions*

SIMD (Single Instruction, Multiple Data) is processing in which a single instruction operates on multiple data elements that make up a vector data-type. SIMD instructions are also called vector instructions. This style of programming implements data-level parallelism. Most processor architectures available today have been augmented, at some point, in their design evolution with short vector, SIMD extensions. Examples include Streaming SIMD Extensions (SSE) for x86 processors, and PowerPC's Velocity Engine with AltiVec and VMX. The different architectures exhibit large differences in their capabilities. The vector size is either 64 bits or, more commonly, 128 bits. The register file size ranges from just a few to as many as 256 registers. Some extensions only support integer types while others operate on single precision, floating point numbers, and yet others process double precision values. Today, the Synergistic Processing Element (SPE) of the CELL processor can probably be considered the state of the art in short vector, SIMD processing. Possessing 128-byte long registers and a fully pipelined fused, multiply-add instruction, it is capable of completing as many as eight single precision, floating point operations each clock cycle.

An application that may take advantage of SIMD instructions is one where the same operation is performed on a large number of data points, for example adding the same value to a large number of floating point or integer numbers. Such operations are very common in many multimedia applications, but scientific computing can also benefit greatly from SIMD instructions.

One can program directly in SIMD instruction sets, but it involves a number of challenges. For example, data alignment restrictions are very common, scatter/gather operations can be tricky, and instruction sets are highly processor-specific. Moreover, many algorithms cannot be vectorized, for instance algorithms with many branches. Most compilers on the other hand are able to generate SIMD instructions automatically, but one needs to carefully construct the loops in such a way that the compiler is able to generate them.

4.2.2 *Better instruction scheduling*

Instruction scheduling is an optimisation used to improve instruction-level parallelism, which improves performance on machines with instruction pipelines. State-of-the-art compilers usually manage to achieve near-optimal instruction scheduling for most modern platforms, therefore avoiding pipeline stalls. However, there are cases when the compiler fails to rearrange instructions in an optimal way, for example for the dual-issue pipeline of the

Synergistic Processor Elements (SPEs) of the Cell processor. In such cases it is meaningful to reorder instructions either manually, or by using some tool for this purpose.

4.2.3 *Loop unrolling*

Loop unrolling (or loop unwinding) is another loop transformation, aiming at reducing the cost of testing and branching at the end (or the beginning) of each loop iteration. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code. Loop unrolling can also be used to improve instruction scheduling, allowing instructions from different iterations to be scheduled together.

A typical loop to optimize is as follows:

```
for (x = 0; x < 100; x++) {
     do_work(x);
}
```

The optimized code, unrolled by a factor of 4, becomes:

```
for (x = 0; x < 100; x+=4) {
     do_work(x);
     do_work(x+1);
     do_work(x+2);
     do_work(x+3);
}
```

Loop unrolling has two major side effects. First, the optimized code typically is much larger than the original loop code (depending on the unroll factor). This can result in higher instruction cache miss rates, which may affect the performance negatively. Second, the technique increases the register usage in a single iteration, possibly decreasing performance. Both of these side-effects may also render the usage of loop unrolling limited or even impossible for architectures with a limited amount of registers (for example, x86 processors) or limited memory (for example, the SPEs of the Cell processor). Loop unrolling may also be necessary in order to achieve vectorization of some code segment.

4.2.4 *Software pipelining*

Software pipelining is a technique used to reorganize loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. The purpose of software pipelining is to separate the dependent instructions that occur within one single loop iteration. There is some start-up and finish-up code that needs to be added before and after the loop.

Software pipelining is better understood by looking at a simple example. Consider the following loop:

```
for(i = 0; i<100; ++i) {
     read(i);
     compute(i);
     write(i);
}
```

After software pipelining, the loop may look like this:

```
/* start-up code */
read(0);
read(1);
```

```
compute(0);
/* loop body */
for(i = 0; i<98; ++i) {
     read(i+2);
     compute(i+1);
     write(i);
}
/* finish-up code */
compute(99);
write(98);
write(99);
```

Software pipelining can be thought of as symbolic loop unrolling. Its advantage over loop unrolling is that the transformed loop consumes less code space. These two techniques address a different kind of overhead. While loop unrolling aims at decreasing the cost of the loop overhead - testing and branching at each loop iteration, software pipelining reduces the time when the loop is not running at peak speed to once per loop - at the start-up and finish-up code. Therefore, these techniques can be used in combination, yielding the best results.

4.2.5 *Strength reduction*

Strength reduction is an optimisation where slow math operations are replaced with equivalent but faster operations, according to several mathematical identities. The cost and benefits depend highly on the target CPU and sometimes on the surrounding code (depending on availability of other functional units within the CPU). Some typical applications of this method are replacing integer division or multiplication by a power of 2 with an arithmetic shift or logical shift operation, and replacing integer multiplication by a constant with a combination of shifts, adds or subtracts. Some simple examples are shown in Table 22.

Original operation	**Optimized operations**
`y = x / 16`	`y = x >> 4`
`y = x * 8`	`y = x << 3`
`y = x * 2`	`y = x + x`
`y = x * 15`	`y = (x << 4) - x`

Table 22: Examples of strength reduction.

4.2.6 *Inlining*

Inline expansion (inlining) is an optimisation that replaces a function call with the body of the function that is called. Inlining can improve performance at the cost of increasing the code size. A normal function call transfers control to the function through a branch or call instruction. When the function is inlined, control flows directly to the function body, thus removing the cost of the function call and return functions. However, this is not the only benefit from function inlining. Inlining also makes additional optimisations possible, such as constant propagation for a parameter.

The main disadvantage of inlining is the increase of code size. As in the case of loop unrolling, this limits the usage of inlining for architectures that have a limited amount of memory. Inlining may make the generated code slower as well: for instance, by decreasing locality of reference. Inlining can be done either automatically, by the compiler, or manually,

by copying and pasting the function body. In the latter case, every change to the function body must be copied to its duplicated versions, so there is the risk of bugs arising by overlooking one copy of the function body. Therefore, it is preferable to let the compiler do the inlining. For this purpose, some languages (for example, C and C++) support the inline keyword in function definitions. This keyword serves as a hint to the compiler that it should try to inline the function.

4.3 Algorithmic optimisation

In the large majority of cases, loop and compiler transformations are able to provide a significant performance boost and lead to satisfactory overall performance. This strategy for optimisation can be considered as *code optimisation* in the sense that it focuses on specific parts of the code, without considering the overall algorithm at a higher level. The advantage of code optimisation is that it can be even applied automatically by compilers or with the guidance of the user, and even if this is not possible, it can be rather easily implemented by the developer, without necessarily having a complete view of the entire application. However, in several cases code optimisation is not able to ultimately attack severe bottlenecks of the application that are due to inherent deficiencies of the implemented algorithm. For example, an algorithm may be conceptually serial, irregular or may require a large amount of communication and synchronization.

When code optimisation reaches the limit, one needs to seriously consider complete reengineering of the application by employing an alternative algorithmic approach. Although the asymptotic behaviour of algorithms (worst-case execution steps for large input sizes, expressed using the O notation) is an important approach to guide the selection of the most efficient algorithm, it is not the only criterion especially when parallelism comes into play. For example, when calculating single-source shortest paths for a directed graph, Dijkstra's algorithm terminates asymptotically in significantly less number of steps for real-life graphs compared to Bellman-Ford's algorithm. However, Dijkstra's algorithm is inherently serial, while Bellman-Ford's algorithm can be easily parallelized. Yet, in the case of the parallel Bellman-Ford, frequent synchronization may greatly degrade performance and prevent performance speedups if the architecture does not support efficient synchronization. Thus, careful selection of the algorithm that well adapts to the characteristics of the underlying platform is of vital importance for the final performance of the application.

4.3.1 *Precision issues*

In numerical computing, there is a fundamental performance advantage in using the single precision, floating point data format over the double precision one. Due to the more compact representation, twice the number of single precision data elements can be stored at each level of the memory hierarchy including the register file, the set of caches, and the main memory. Handling single precision values consumes less bandwidth between different memory levels and decreases the number of cache and TLB misses. Also the throughput of SIMD instructions is effectively doubled when using single precision.

The performance advantages of single precision, that can reach up to a speedup of 2 compared to double precision, greatly favour implementations that are based on this precision mode. For cases where the result needs to be in double precision, mixed precision implementations can be considered. For example, iterative system solvers can perform an initial set of iteration in single precision and when single precision does no longer suffice for the tentative solution, the algorithm can shift to double precision. If this algorithmic and data change does not impose significant overhead and provided that the convergence properties of the solver are not

changed, the mixed mode precision is expected to provide substantial performance improvement.

4.3.2 *Data structures*

The selection of data structures greatly affects the performance behaviour of an algorithmic implementation. In theory, data structures also affect the overall execution steps of an algorithm. For example, a priority queue can be implemented in various ways, e.g. as a linked list, as a binary tree, as a binary heap and others. Each of these data structures has different costs for the various operations applied on the data structure (e.g. extract minimum element, insert, delete, find). The selection of a proper data structure needs to be carried out considering the number of the various operations performed on the data. In practice, the implementation of a data structure is also of vital importance of the final performance, even when the asymptotical behaviour of the algorithm does not change.

4.4 Optimisations for threaded applications on multicore platforms

4.4.1 *NUMA-aware data placement*

Non-Uniform Memory Access (NUMA) characteristics are now becoming common in modern multiprocessor and multicore platforms, as an attempt to minimize memory contention and improve scalability for shared-memory parallel applications. AMD's HyperTransport and Intel's QuickPath technologies are two examples that demonstrate the shift from shared-bus designs to NUMA architectures. NUMA systems incorporate multiple memory controllers and buses, so that each processor (or cluster of processors) has a separate path to main memory in its proximity. As such, contention with other processors' memory requests can be significantly reduced. However, the memory access latency is now asymmetric, because a processor can access faster its local memory than a distant memory. For this reason, in order to establish high performance through multiple uncontended and fast paths to main memory, application threads and their corresponding data need to be placed as close to each other as possible.

Operating systems now provide application programmer interfaces allowing the user to perform thread and memory placement as desired. For example, the Linux OS provides a set of system calls that handle the binding of threads onto specific processors. Another set of system calls enables the allocation and migration of memory pages to a specified set of memory banks. When the programmer is aware of the access patterns of his parallel application, it is possible to benefit from proper thread and data placement by using in concert system calls from both families. For example, a parallel application with high per-thread spatial locality could see notable performance improvement if its threads are distributed evenly across the nodes of the platform (i.e., neighbourhoods of cores and their adjacent memory bank(s)), so that a thread and the data it most frequently accesses reside in the same node. When the memory access pattern is difficult to be drawn, heuristics can be employed in order to dynamically migrate data and/or threads at runtime (e.g. using feedback from processor counters).

4.4.2 *Shared Caches in multicore processors*

Modern multicore platforms usually share some level of cache hierarchy. This sharing can

sometimes be destructive and other times beneficial.

When application threads work on disjoint data under a common cache, it might be possible that one thread's data are being mapped to cache locations where another thread's data already reside, thus causing the eviction of the latter from the cache. Such inter-thread conflict misses are common when the cache associativity and/or capacity are small. There are two main approaches to deal with this issue. If there are available processors on the system that share none or few levels of cache hierarchy, the application could utilize them to migrate a conflicting thread there. If thread migration is not possible (e.g. because of full system utilization), software techniques could be employed in order to effectively partition the total space of the shared cache. Such techniques include tile size adjustment for tiled codes, block data layouts and copying. Sequential tiled codes tend to use tiles equivalent to the total available cache space, in order to best exploit temporal locality without much tiling overhead. When such codes are parallelized to multiple threads, however, the tile sizes have to be re-adjusted so that they altogether fit in the shared cache. Besides that, care must be taken so that tiles from different threads do not conflict with each other. This can be controlled more easily when the elements of each block are stored in contiguous memory locations. Block data layouts and copying are two techniques to achieve that.

At the other extreme, many parallel applications exhibit opportunities for constructive cache sharing. In this case, threads sharing a portion of their working set should be scheduled on processors that share some level of cache hierarchy. Threads working on the same or adjacent data are good candidates for this, since the corresponding blocks need to be fetched only by the first requesting thread, and the other sharers will be able to reuse them afterwards.

In all of the aforementioned cases a key issue is how to identify contention or sharing in cache and how to take actions to avoid or exploit it, respectively. Depending on the program's complexity, one could use feedback from performance counters, profiling methods or even simple inspection in order to identify how the threads interact with each other on their shared data. Then, depending on the program's runtime behaviour, the application of the appropriate techniques could be done statically at compile time by the user or the compiler, or dynamically by the operating system or a runtime system.

4.4.3 *False-sharing and variable placement*

False sharing issues may turn up if an application uses thread-based parallelization techniques. If two or more processors work on independent data stored in the same cache block, the cache coherency mechanisms of the system may force the whole block to move across the bus with every data write, causing memory stalls and increasing the bus traffic. Such “ping-pong” scenarios can introduce large penalties and thus must be avoided in programs. A naive approach would be to use a separate cache line for each different variable in the program. In general, this practice is not acceptable since it would increase the memory footprint of the program and we would not be able to take advantage of the spatial locality.

What actually is needed is a more judicious classification of shared variables by the programmer. At first, global variables that are read-only or rarely written should be grouped together, hoping that the compiler will place them in a separate memory region in the data segment. Global read-write variables that are often used together in time proximity could also be grouped (e.g. by placing them in a structure), in order to reduce their memory footprint and increase the spatial locality. Global read-write variables that are often written by different threads can be moved onto their own cache line. This could be done by properly aligning such a variable to the cache line boundary and then adding padding at the end to fill the remainder of the cache line. Finally, variables that are defined global, not for sharing purposes but

because they must have a global scope with respect to other program functions, should be specified as thread-local and moved to a per-thread separate area (sometimes known as Thread-Local Storage). Modern compilers such as gcc provide extensions that enable the programmer to specify a global variable as thread-local.

4.5 Optimisation for NEC SX-9 vector processors

The NEC SX-9 is the latest model of the SX series. The numerical performance of the system is provided by large vector processing units, operated at clock rate of 3.2 GHz, which are supported by a small scalar part operated at 1.6 GHz. Each vector processor is equipped with four eight-way vector pipes, resulting in a theoretical peak performance of 102.4 GFLOPs per processors. The computational throughput is supported by a high bandwidth memory connection of 256 GB/s.

The new generation provides support for up to 16 processors in a single shared memory node with up to 1 TB of main memory. The nodes are connected by the NEC IXS crossbar.

Any application which is to gain high performance on the SX system needs to take advantage of the vector facilities. The most important factor for vectorization is data independence, just as for general parallelization. It might therefore be best suited to think about the vectorization features as an additional level of parallelism. From this point of view, vectorization is the fine grained parallelism which has to be at the bottom level of a parallel hierarchy. The NEC-SX9 uses vector registers with a length of 256 8-byte words, thus vector lengths ideally would be multiplies of this quantity in order to fully utilize the vector units. The loop iterations need to be independent of each other in order to allow vectorization. If the data independence is not obvious to the compiler, the programmer has to provide a hint by the “nodep” directive directly in front of the loop in question. In C such directives are indicated by `“#pragma cdir <directive>”` and in Fortran by `“!cdir <directive>”`.

Many applications can take advantage of this kind of vectorising functionality but are written in a layout that prohibits vectorization, as the independent data loops are not the innermost ones. In these cases it is necessary to draw those loops down to the innermost level, at least in chunks of reasonable size. In some cases it might be necessary to create temporary arrays to store results of innermost loops across different loop levels of outer algorithmic logic, but often a simple loop swapping can be applied. A stride one access to the vectorising arrays is very beneficial and should be deployed wherever possible. This might result in changes to the overall data structures, as array indices might need to be changed in order to allow sequential access along the dimension, which is to be vectorized. In the case of indirect addressing it might be helpful to preload the index vectors into vector data registers and ease thereby the access to the payload data.

Fortran supports a convenient array notation, allowing modifications of complete arrays just with the same notation as simple scalar values. This way of programming vectorises very well, in many cases even for multi dimensional arrays.

Even though the system provides a high byte per FLOP of about 2.5, this throughput is attached to very high latencies. Thus long vectors are needed to utilize the high memory bandwidth. Also, the compiler needs to be able to decide the independence of all chunks. This means for example, that the inner loop lengths must not dependent on the outer loops. In order to improve the performance even more it is necessary to take advantage of the memory hierarchy, reducing the amount of necessary data transfer to the main memory, to the bare minimum. The SX vector processors do not provide the functionality of associative caches to overcome memory bottlenecks. Instead there are vector data registers available, alongside the newly introduced Assigned Data Buffer (ADB).

Vector data registers are very fast stores for vector data. They are only slightly slower than the actual arithmetic vector registers and therefore yield the largest prospects for performance improvement. Obviously they are limited in size to 256 words, and there is only a limited amount of them available. The programmer can explicitly put arrays into the vector data registers by applying the "`vreg(var1, var2, var3)`" directive for the array variables in question. Deployment of vector data registers should only be used for more complicated scenarios, e.g., when temporary data has to be maintained across different loop levels. For plain simple long loops with only scalar temporary values, the described mechanisms are completely taken care of by the compiler.

ADB is an intermediate fast storage, with a size of 256 KB and bandwidth of 4 Bytes/FLOP. The functionality of this storage resembles that of a cache, by transparently mapping to the main memory, however it is not automatically used for any data, but rather has to be filled manually, by specifying which vector data should be put onto the ADB with the "`on_adb(a1, a2, a3)`" directive right before a vectorised loop. The compiler uses the ADB for example to swap vector data registers away, if they are overfilled.

Additionally one should avoid using the weak scalar units as much as possible. An important example is the use of routines with a very high number of calls. The calling overhead of lightweight routines easily dominates the execution time. Such parts of the code should be either inlined or made more fat by pulling enclosing loops inside the routines. Another point here is program logic, which should be kept outside the actual computational kernel whenever possible.

In order to find the most important parts of an application, that badly need vectorization, there is a tool provided by NEC, called ftrace, which provides very detailed information about each routine. Beside the fraction of computational time and the number of calls, it also details the average vector length, the time spent in the vector unit and the time spent in memory accesses due to conflicts. Additionally the compiler can produce very convenient listings, listing the modifications it made and which loops are handled how (loop exchange, collapse, split and so on). It is especially useful to see the vectorization status of any loop in the application, and to see where there is some room for improvements.

4.6 Optimisation for Cell processors

Cell is a heterogeneous single chip multiprocessor, with a total of 9 cores, and a unique architecture to support the bandwidth needs of the cores. There are two specialized types of cores, namely one Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). On current generation Cell processors, the PPE and SPE both run at 3.2 GHz. The combined theoretical peak performance of the Cell processor is over 200 Gflops for single precision math and over 100 Gflops for double precision.

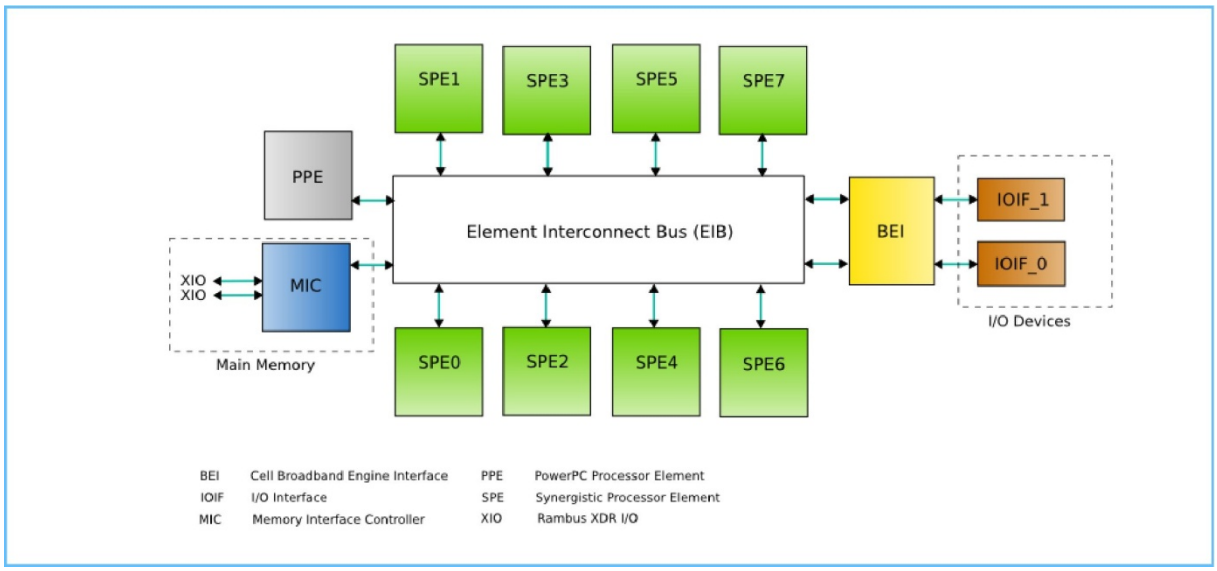

Figure 1: Cell processor elements

All Cell elements are connected to each other via a powerful cyclic store-and-forward bus, called the Element Interconnect Bus (EIB). Besides connecting the nine processor elements, the EIB also connects a high-speed Memory Interface Controller (MIC), and a Broadband Engine Interface (BEI). The BEI manages data transfers between the EIB and I/O devices. The MIC is a high-bandwidth on-chip interface between the EIB and main memory and can provide 25.6 GB/s of bandwidth to main memory.

4.6.1 *Power Processor Element*

The PPE is Cell's main processor and has a traditional architecture such as Intel or PowerPC processors. The PPE is a 64-bit RISC general-purpose processor with SIMD extension. The PPE is dual threaded but does not implement out of order execution. It has a conventional cache hierarchy with a 512-KB L2 cache and 32-KB L1 data and instruction caches.

4.6.2 *Synergistic Processor Elements*

The SPEs provide most of the raw performance for the Cell processor. The SPEs are meant to be used as specialized accelerator co-processors, which the PPE can offload work to. The SPEs are all identical 128-bit RISC processors and are specialized for compute-intensive SIMD applications. Scalar computations are very inefficient on the SPEs and should be avoided.

To remain simple the SPEs are not multi-threaded like the PPE. They can, however, dual-issue instructions. One pipeline supports fixed-point, floating-point instructions etc. whereas the other pipeline supports load/store, shuffle instructions etc. Furthermore, the SPEs don't have a branch prediction unit. Instead the SPE instruction set has special branch hint instructions that can be used to tell the hardware if a branch is likely to be taken or not. Providing correct hints is essential for achieving good performance on the SPEs.

The most notable difference between the PPE and SPEs is that the SPEs don't have caches, and they cannot access main memory directly. Instead they only have direct access to what's called a Local Store (LS). The LS is a very fast on-chip memory with a size of 256KB. To fill the LS with data, one needs to use a SPE's dedicated DMA controller. DMA commands are

asynchronous instructions that fetch data from main memory to a SPE's LS and vice versa. Since a SPE can only execute instructions in its own LS, a SPE application plus its static and dynamic data must be able to fit in the LS. However it is often the case that not all data can fit in the LS. In this case data has to be continuously flushed to main memory and replaced with new data as the SPE program executes. This is usually done using multi-buffering techniques, which will be covered later.

The DMA controller on each SPE acts as a specialized co-processor for its associated SPE. It can hide memory latency very effectively, by overlapping data transfers with instruction execution. There are several restrictions on the size of the transferred data, and its alignment. If these are not fulfilled an exception is thrown and the DMA transfer cancelled. A DMA transfer always takes up 128-bytes (a cache-line) of bandwidth, one should therefore always seek to transfer data in multiples of 128-bytes. The theoretical peak bandwidth is 51.2 GB/s.

The SPE DMA transfers to the LS are either fetched from the L2 cache, main memory or other local stores. DMA transfers from main storage have high bandwidth and moderate latency, whereas transfers from the L2 have moderate bandwidth and low latency. If the SPEs and the PPE work on the same data, the SPEs will not be able to transfer data that resides in the L2 cache at full bandwidth, although they can achieve lower latency.

4.6.3 *Inter-processor communication*

There are three different ways for a SPE to communicate with other units in the Cell processor:

- Mailboxes
- Signals
- Direct DMA transfers

Mailboxes and signals are low latency methods of sending and receiving small messages. For larger data transfers, one should always use DMA transfers. A typical way of doing this would be to initiate a DMA transfer to a LS and then notify of completion by sending a mailbox or signal message.

One thing to be aware of is that all mailbox and signalling operations go through the EIB, which all devices are connected to. A common design pattern in Cell is for the PPE to busy wait for messages from the SPEs by checking their outbound mailboxes. However, this is done via the EIB, which means that the EIB will be flooded with data to check the mailboxes and thus reducing the bus bandwidth. A better way to achieve the above is to have the PPE busy wait on the cache and instead let the SPEs send their messages to the cache via DMA transfers.

Element Interconnect Bus

A multi-processor with nine cores, requires a lot of sustained internal bandwidth to perform well without stalling. This bandwidth is delivered by the Element Interconnect Bus (EIB). The EIB is comprised of a data arbitration unit, and four circular data rings, running at half the frequency of the processor cores. Each ring is 16 bytes wide and can handle up to 3 concurrent transfers. There are several restrictions on the EIB that may reduce the number of concurrent transfers.

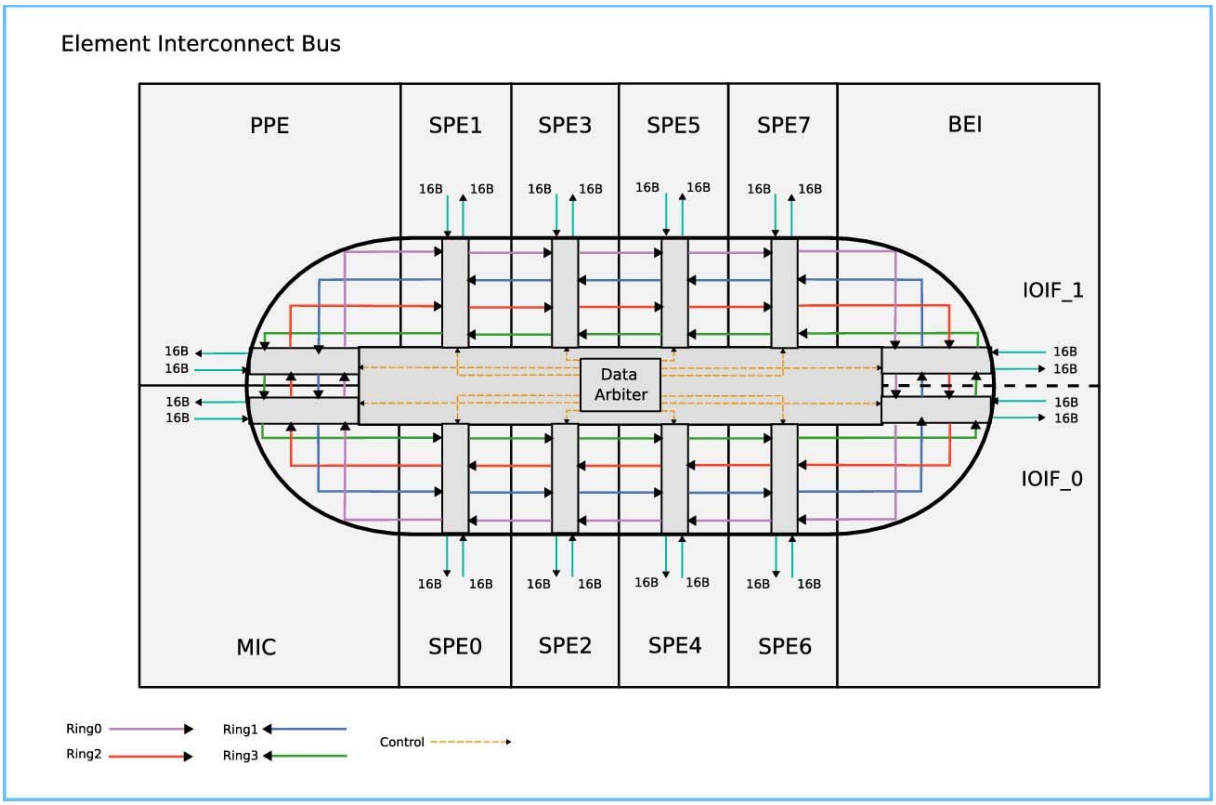

Figure 2: Element interconnect bus

Data moves in hops around the EIB. As Figure 2 shows, there are a total of 12 hops, one for each connected element. The shorter the distance between two communicating elements, the less latency. The choice of which ring to use is transparent to the programmer. So as long as transfers don't overlap, they can occur concurrently. It is therefore important to work out the EIB traffic patterns beforehand, so one can arrange communication to minimize blocking and latency.

4.6.4 *Multi-Buffering*

To be able to hide SPE DMA transfer latencies, a technique called multi-buffering must be employed. When using multi-buffering, memory transfers can be hidden by distinguishing between buffers currently being used for data transfers and buffers being used for computation. By using two buffers, B0 and B1, DMA transfer latencies can be hidden by switching between B0 and B1, when transferring and using data.

4.6.5 *Levels of Parallelism*

Cell's unconventional architecture, forces programmers to deal with four levels of parallelism, to achieve it's peak performance. Since Cell has eight SPE cores and a dual-threaded PPE core, it can run up to ten tasks in parallel. This is known as task-level parallelism. Each core in turn supports SIMD instructions, which enables them to operate on multiple data points in parallel. This is called data level parallelism and can boost performance considerably. Furthermore, all Cell cores support dual-issuing of instructions, which adds yet another level of parallelism, called instruction-level parallelism. These three levels of parallelisms are not

unique to the Cell processor, and are supported by most current generation of mainstream processors.

Where the Cell processor differs from mainstream processors, however, is in the support for memory-level parallelism. Since the DMA controllers are dedicated, instructions and memory transfers are independent of each other and one can therefore hide memory latencies by running instructions and DMA transfers in parallel. Hiding memory latencies can increase performance considerably, but it also complicates things, since it forces programmers to manually fetch data themselves, before its needed.

4.6.6 *PPE and SPE Workloads*

Cell's heterogeneous nature makes its cores specialized for different workloads. For compute intensive applications that require frequent predictable memory access, a SPE is preferred because of its DMA controller and SIMD optimized engine. Although the PPE can also perform SIMD instructions, its traditional architecture and cache will limit its performance for such workloads, compared to a SPE.

Since the SPEs are specialized for compute intensive task, there are inevitably other tasks, which they are not suitable for. If memory access patterns are erratic and not predictable at run-time, the SPEs do not perform very well. Another unsuitable task for the SPEs, is code which contains many unpredictable branches. Branch misspredictions are relatively expensive on the SPEs and since they don't have branch prediction units, the responsibility falls on the programmer to predict branches.

The PPE and SPEs nicely complement one another and one has to take advantage of this to achieve the Cell processors peak performance.

4.6.7 *Translation Look-Aside Buffer (TLB)*

In the Cell processor the TLB is shared by all SPE cores, which can lead to severe trashing of the TLB as one SPE overwrites entries needed by another one. Thrashing can be limited by using larger page sizes. The larger the page sizes, the more physical memory the TLB can cover, which is known as the TLB reach. This is true for all multi-core processors, however, it is something Cell programmers in particular have to be aware of, since Cell has relatively many cores.

4.6.8 *Summary*

On paper the Cell processor offers an impressive performance, compared to other commodity processors, however, this comes at the cost of complexity. To write efficient code, Cell requires several levels of awareness from a programmer. In a sense, Cell gives control back to the programmer. The most obvious example of this are the SPE cores with no cache. This forces the programmer to act as the cache instead, manually anticipating and fetching data before its needed. To hide memory latency one has to use multi-buffering and to get efficient memory transfers, programmers have to align and pad data. In turn to achieve high transfer bandwidths, one has to calculate communication patters on the EIB beforehand, and distribute threads to the SPEs accordingly. Also, because of the heterogeneous nature of the PPE and SPE cores, one has to be aware of the type of workloads they are specialized for. This means that when optimizing applications for the Cell processor, one has to have the whole architecture in mind, since optimizing each part alone will not result in peak performance.

5 Application reports

5.1 Alya

Raúl de la Cruz (BSC)

The Alya System is a Computational Mechanics (CM) code with two main features. Firstly, it is *specially designed* for running with the highest efficiency standards in large scale supercomputing facilities. Secondly, it is capable of solving different physics, each one with its own model characteristics, in a coupled way. Both main features are intimately related, meaning that all complex coupled problems solved by Alya must retain the efficiency. Among the problems it solves are: Convection-Diffusion-Reaction, Incompressible Flows, Compressible Flows, Turbulence, Bi-Phase Flows and free surface, Excitable Media, Acoustics, Thermal Flow, Quantum Mechanics (TDFT) and Solid Mechanics (Large strain). By *specially designed* we mean that Alya is a parallel program designed from scratch, that solves in a flexible yet clear way every kind of CM model.

5.1.1 *Application description*

Alya is based on Finite Element Methods and this kind of application can be divided in two main computational tasks, executed inside a time-step loop: the element loop and the solver. These two tasks are executed for each time-step of the simulation.

In the first task, the element loop, the element matrices and the RHS vectors are computed for each element of the domain. All the terms in the set of PDEs describing the physical problem are computed at the gauss points and added into the matrices. At the same time the boundary conditions are applied to them. Finally all the matrices and vectors are assembled into the global system depending on the element connectivity.

The second task is the solver, which takes the A matrix and the RHS vector assembled in the element loop and solves the system for the current time-step. The solver can be iterative (GMRES, CG) or direct (LU/Gauss-siedel).

Alya consists of 3 different entities: kernel, services and modules. The kernel is involved in mesh generation, coupling physical models and I/O. Services perform tasks like parallelization, domain decomposition or optimisation support. Finally the modules solve different sets of PDE's describing the physical problem.

The original code, which is executed in a PPE unit, is written in Fortran90. The SPE code (element loop at this moment and SpMV in the future) is written in C ANSI C99. The original code (running in the PPE) is quite readable, well commented and well organized. This is not the case for the newly written PPE/SPE management code. This code is specific for the Cell/B.E. architecture and it requires good knowledge of the architecture. The SPE code has been written in ANSI C99 using Vector/SIMD instructions and DMA specific instructions for Cell/B.E. architecture. Due to the use of this kind of instructions, and other optimisations required (alignment issues, structure of arrays data layout), a trade-off has to be made between readability and performance.

Alya uses a number of libraries, namely:

- Libmetis: Metis is a partitioner used for the domain decomposition among the MPI tasks.

- Libspe2: PPE code uses libspe2 to manage SPEs (creation, destroy, signaling, etc...)
- Libspe2f: This is an internal library to manage libspe2 in Fortran90 code (wrapper)
- Libpthread: library used to create threads, which control SPE context.
- Libthreadmanage: internal library used to manage SPE threads (errors, control).

5.1.2 *Porting*

The porting task to the PPU part of the CELL chip was straightforward. No big effort was required and was completed within less than 1 pm. The code was ported to the Cell/B.E. architecture, not to the other prototypes.

This was the first time that the code was ported to the Cell/B.E. architecture. The code was moved to the Maricel cluster and we adapted the configuration file (set PPE compiler and flags). For the MPI version we had to tune the OpenMPI configuration to set IBM XLF90 as the compiler by default (export OMPI_* variables). The compilers used are ppuxlf90 and ppuxlc. Some specific flags are:

```
-qflag=i -qnoescape -qsuffix=f=f90 -qextname=flush,etime
-qfree=f90 -qmoddir=$O -I $O -c -q64 -qarch=cellppu
-qtune=cellppu -O3 -Wl,-q
```

MPI compilation require some tuning of the OMPI_* environment variables to be able to compile the source code with IBM XL compilers.

5.1.3 *Optimisation techniques*

On the PPE side two inter-linked optimisations were implemented. First, we reordered the nodes of the mesh in an intermediate sequential buffer (Gather/Scatter task) to enable easy DMA transfers by the SPEs. Thus, this optimisation let us use DMA transfers instead of DMA-list transfers. Second, we overlapped memory transfer and computation using triple-buffering. These optimisations are described below in detail as technique 2.

On the SPE side further optimisations were implemented. The computationally most intense parts in the element loop of the SPE have been unrolled and software pipelined to increase the throughput (IPC) and avoid stalls in the SPE pipeline (see technique 3). Implicitly we have also implemented a blocking algorithm due to Local Store size constrains. To do so, we package as many data elements into the buffers as they fit in the SPE's Local Store (see technique 1). Additionally the SPE code has been vectorized to get maximum profit of the SPE capabilities (see technique 4). Finally instructions have been reordered in intensive computing loops to avoid starving and stalling of the pipeline.

Technique 1: SPE-wise element loop computation

As it was described before there are two main computational parts in Alya. One of them is the element loop computation, which numerically is a good fit for the SPE units. It was thus important to port this task to the SPEs, to exploit all the capabilities of the Cell/B.E. processor.

The elements of a node domain are processed by chunks (depending on the free space in the LS). The input data required is packaged in main memory and processed by the SPEs. Each SPE fetches one chunk of input data (set of data required for element computation) and it performs the element loop computation over that data set. When the SPE has finished the computation, it writes back the output (Element matrices and RHS vectors) to the main

memory. After this, the SPE waits for a signal from PPE (mailbox) to know if there are more chunks to process or if the thread execution can finish.

The main difficulties involved in implementing the optimisation, was the need to synchronize the 8 SPEs with the PPE and implementing the Fortran90 code on the PPE-side (the PPE code is written in Fortran90, the SPE code is written in ANSI C99).

Not considering the data transfers (gather/scatter/DMA) the theoretical speed-up that should be achieved is 8x (1 PPE vs 8 SPEs at the same clock rate) because no vector version is implemented.

Technique 2: PPE driven (Gather/Scatter tasks)

There are two options to solve the problem with the sparse distribution of data in main memory and the DMA transfers. The easy way is what we call PPE driven. In a PPE driven implementation, the PPE gathers the data from the global vectors into sequential vectors where straightforward DMA transfers can be done by the SPEs. On the other hand, the SPE driven, where the SPEs fetch the sparse data directly from the global vectors, is non-trivial and requires much more research.

The PPE get the values that each SPE requires for the computation and write them in a temporal buffer. In this way the required data is sequential in those buffers (at least 1 buffer for each SPE), and it can be fetched directly using simple DMA transfers. A very similar operation is done for the SPE results (scattering). Each SPE writes the results into a buffer, where the PPE gets the data (element matrices and RHS vector) and it scatters the results into the global system. To hide SPE latencies while they are computing the element loop, we have implemented a triple-buffering code for the input/output data. Thus, we are able to overlap element computation by the SPEs and Gather/Scatter by the PPE. Using triple-buffering for Gather/Scatter we can revamp the performance removing all the PPE idle times due to SPEs computation. This is not the case for the SPEs, where they are only removed partially due to the time spent in Gather/Scatter tasks. Time required for gather and scatter on the PPE is longer than the kernel execution time on the SPEs. The main challenge in implementing this optimisation lay in improving the performance in the gather and scatter routines to reduce as much as possible the SPE idle times.

If we consider the DMA transfers (Memory-LS communication) and the computation, the theoretical speed-up should be the same as when not considering the communication. But this is not the case as the PPE cannot feed all the SPEs with the required input and write back the output results in the main memory global system fast enough.

Technique 3: Vectorize the code (SIMD-SPE instruction set)

Utilizing SIMD instructions in the SPEs is a must if we want to achieve the maximum performance. The vectorization of the code was done using SOA data layout (Struct of Arrays). This data layout permits a faster vectorization of the code compared with other techniques like AOS (Array of Structs). On the other hand SOA data layout requires some reordering of the structures at the Local Store. Even if this was implemented for the CELL/B.E. processor, the optimisation should work for all SIMD implementations.

The main challenge in implementing this optimisation lay in reordering those structures in the Local Store that needed changes.

Considering only SPE computation (element loop) the theoretical speed-up should be 2x at each SPE compared with the scalar version on the unit. The main reason is because we are computing 2 floating points (vector double) per instruction instead of 1.

Technique 4: Reducing stalls in SPEs

We profiled the SPE routines and saw that most time was spent in two subroutines. The Asmvis tool was used to analyze the SPE pipeline in those subroutines and we realized that a lot of stalls were being produced. We found the spots and we applied loop unrolling, software prefetching and software pipelining. This kind of optimisations can be done in any super-scalar processor.

The main challenge lay in finding the optimal parameters for loop unrolling, software pipelining and software prefetching.

After the optimisation the pipeline execution was quasi-optimal (less than 10% of stalls in the internal loops). We were able to improve the performance in those subroutines by a factor of 1.5.

5.1.4 *Results of optimisation effort*

The impact of the employed optimisation techniques on the application performance is quite good. For the SPE-wise element loop computation we have got a speedup of 11.5 compared to the original version running only in the PPE. However, this performance can still be improved due to the PPE performance bottleneck of the gather/scatter task. A SPE driven implementation of the gather/scatter may lead to an improvement.

The expected gain in performance was twice the actual that we are getting. The reason is due to the idle time during which the SPEs are waiting for data from the PPE. The bottleneck is reordering data for the gather/scatter task. The reasons could be the low peak performance of the PPE (IPC throughput) and the number of L1/L2 cache misses (memory bound).

5.1.5 *Conclusions*

The major constraint is the PPE performance. The PPE is an in-order execution processor with very few functional units. This property makes it difficult to get good performance on the critical task that is executed on it. We cannot rely on executing critical code on such a processing unit. We should port it (if possible or feasible) to the SPEs.

Data that we have to use for element loop computation is sparse in main memory and non-trivial to access from the SPEs. To help on that task the PPE gathers/scatters the required data into a sequential buffer. The main unsolved problem is that the PPE driven gather/scatter approach is not efficient and the SPEs are only working 60% of the time in the element loop computation; they are idle waiting for data from the PPE in the remaining time. If we would have started a new code from scratch we would have changed noticeable the data layout structure. The data layout can be an important handicap for DMA transfers to the SPEs.

The solution could be to include a better PPE unit (Power6 or Power7 processor) in the Cell/B.E. chip or implement the gather/scatter task through SPEs using DMA list (SPE driven). In fact, the SPE driven approach for gather/scatter task can be implemented with the actual hardware. However it requires DMA list research and maybe some reordering of the mesh data at the beginning. It should be interesting if future Cell/B.E. implementations would not have constraints for DMA list transfers of less than 16bytes.

We have SPE-wised the element loop computation, and the second remaining computational part is the solver. Most of the time the solver executes SpMV (sparse matrix vector operation). To further optimize the program we could provide an internal/external library with such linear algebra operations on the SPEs.

5.2 AVBP

Bertrand Cirou (CINES)

AVBP is one of the very few codes that can simulate turbulent combustion taking place in turbulent flows within complex geometries. It has been jointly developed in France by CERFACS and IFP to perform Large Eddy Simulation (LES) of reacting flows, in gas turbines, piston engines or industrial furnaces. This compressible LES solver on unstructured and hybrid grids is employed in multiple configurations for industrial gas turbines (Alstom, Siemens, Turbomeca), aero gas turbines (SNECMA, Turbomeca), rocket engines (SNECMA DMF Vernon), laboratory burners used to study unsteady combustion (Cambridge, École Centrale Paris, Coria Rouen, DLR, Karlsruhe University, Munich University).

5.2.1 *Application description*

AVBP is based on Finite Element Methods, and these kind of applications can be divided in two main computational tasks executed inside a time-step loop: the element loop and the solver.

In the element loop, the element matrices and the RHS vectors are computed for each element of the domain. All the terms in the set of PDEs describing the physical problem are computed at the gauss points and added into the matrices. At the same time the boundary conditions are applied to them. Finally all the matrices and vectors are assembled into the global system depending on the element connectivity

The second task is the solver, which takes the A matrix and the RHS vector assembled in the element loop and it solves the system for the current time-step. The solver can be iterative (GMRES, CG) or direct (LU/Gauss-siedel).

AVBP is written in Fortran90. The code is readable, commented, and well organized. It uses two libraries: metis which is a partitioner used for the domain decomposition among the MPI tasks, and HDF5 which is a portable library used for parallel file I/O

5.2.2 *Porting*

The porting was easy as AVBP was already ported on BG/L, power5, XT4, Itanium

Xeon Linux SGI ICE 8200EX

AVBP needs a number of libraries: libmetis, hdf5, and a MPI library. The platform supports all, except libmetis which was installed and compiled with the AVBP source code. The MPI library used here was SGI MPI MPT. We chose Intel's compiler suite and the following flags:

```
-O3 -convert big_endian -fno-alias
```

The "`-convert big_endian`" compiler flag is needed to convert file I/O to big endian format. To get optimal performance the placement was chosen so that each node has 8 cores so the ranks of MPI process are {0,1,...,7} on the first node {8,9,...,15} on the second node and so on. Also, the following SGI MPI MPT parameters were set:

```
MPI_CONNECTIONS_THRESHOLD=8192
MPI_LAUNCH_TIMEOUT=120
```

IBM Power6 cluster - Huygens

On this platform libmetis was installed and compiled with the AVBP source code. We also used the vendor provided MPI library. As compilers we chose IBMs compilers. The optimal flags for the program were:

```
-q64 -qsuffix=f=f -O3 -qstrict -qtune=auto -qarch=auto
```

Each node has 16 cores so the ranks of MPI process were placed so that ranks {0,1,...,15} are on the first node {16,17,...,31} on the second node and so on.

IBM Blue Gene/P - Jugene

On this platform libmetis was installed and compiled with the AVBP source code. We also used the vendor provided MPI library. As compilers we chose IBMs compilers (blrts_xlf90 and blrts_xlc -qlanglvl=stdc89). The optimal flags for the program were:

```
-qstrict -qfixed -O3 -qhot -qmaxmem=-1 -qarch=440 -qtune=440 -
I$(BGL_SYS)/include
```

5.2.3 *Optimisation techniques*

Regarding optimisation one major optimisation has been done to improve serial performance where we replaced some computation code by calls to BLAS library. The effort for implementing this optimisation amounted to 1 pm.

About 15 % of the time is spent in some self written computation loops. It is well known that hardware vendors provide optimized computation library that perform better than self written code. These computation loops have been replaced by equivalent calls to BLAS library. The optimisation is applicable to essentially all platforms as BLAS is widely available.

The main difficulty in implementing the optimisation was due to indirect access in arrays, the self written code did not match to exactly one BLAS. We had to copy the data and use two BLAS calls. In fact, this negated the performance benefits of the BLAS calls.

5.2.4 *Results of optimisation effort*

The use of BLAS seemed to be a good idea but as the match was not perfect we had to restructure and copy the data, which lead to performance degradation

5.3 BSIT

Mauricio Araya Polo (BSC)

BSIT is an application that is used in geophysics to delineate sub-surface structures using the reverse time migration (RTM) technique. RTM simulates the acoustic propagation in a given medium. In the simulation, first the medium is excited by introducing a wavelet (a shot), expressed as a function of frequency and time. Then, wave propagation (called forward propagation) is mathematically simulated by using an acoustic wave equation. Then, RTM repeats the task in a backward fashion: starting from the data recorded by the receivers, it propagates the wave field back in time (backward propagation). Finally, when both fields representing the forward and backward propagation are available, a cross-correlation between them is performed to generate the output image.

The acoustic wave propagation equation is a Partial Differential Equation (PDE). We assume an isotropic, non-elastic medium, where density is not variable. This equation is solved with a Finite Difference method (FD). The PDE solver involves a 3D stencil computation, followed by its corresponding integration in time.

The source code has 21895 lines of code (LOC) of C programming language, 400 LOC of bash scripts and some LOC of Makefiles. The C LOC are divided among the three main parts of the system: kernel PPE, kernel SPE and master. The kernel PPE code has 3007 LOC, the kernel SPE code has 3388 LOC and the master code has 15500 C LOC and bash scripts.

The application uses: librt for asynchronous IO, libnuma for enable the NUMA capabilities, libspe2 for SPE intrinsics, libpthread for the thread management intrinsics, libm for the mathematics function intrinsics and MPI as a message passing library.

The readability of the code is quite high for the master and PPE code, well commented although not so well documented. The readability of the SPE code is low, this is due to the poor documentation and because the trade-off between readability and full optimisation.

From the high level point of view the application follows a master-worker scheme. In our case the master distributes the work, coordinates the task and assembles the results. Every worker executes the RTM kernel, where some subtasks are executed in the PPE and other in the SPE. The master preprocesses the data in order to distribute the work among the workers. When the workers (RTM kernel) send back their results the master postprocesses them in order to generate the final image result. Master accounts for 10% of the workload, and the workers for the remaining 90%.

5.3.1 *Porting*

The porting concentrated on the CELL architecture and proved to be very difficult; reaching high-performance took even more time. Mostly we rode the learning curve from the very beginning with this novel architecture. No less than 10 pm was used for this porting effort.

It was the first time that the code was ported to the Cell/B.E. architecture. Although the platform is constantly under software updates, we were able to do the port. For the master part of our application extra libraries were required. We needed a completely new design for the workers part, because of the architecture particularities.

For the PPE-side of the code the compiler was IBMs ppuxlc. The flags used for compilation were:

```
-qflag=i -qnoescape -qsuffix=f=f90 -qextname=flush,etime
-qfree=f90 -qmoddir=$O -I $O -c -q64 -qarch=cellppu
-qtune=cellppu -O5 -Wl,-q
```

5.3.2 *Optimisation techniques*

On the PPE-side of the code resides the master and parts of the worker tasks. It's roughly the same algorithm as in the original code. At the level of communication, I/O and file system management some minor adaptations where made.

One the SPE-side of the code resides the worker computational kernel. Here we used load balancing, domain decomposition, code vectorization and several different loop optimisations.

Utilizing SIMD in the SPEs is fundamental if we want to achieve the maximum performance. As for Alya, the vectorization of the code was done by adopting a SOA data layout. This data layout permits a faster vectorization of the code compared with other techniques like AOS.

On the other hand SOA data layout requires some reordering of the structures at the Local Store. This change required complete reorganization of the kernel code, including boundary conditions. The SIMD achieves expected linear scalability with speedup of the order of 8x.

5.3.3 *Results of optimisation effort*

The impact of the employed optimisation techniques on the application performance is quite good. For the SPE-wise element loop computation we got an improvement of 8x times faster than the original version running only in the PPE. Against the MareNostrum platforms the speedup are of the order of 10.

The expected gain in performance was the one we expected; however, there is room for improvement, for instance IPC throughput can be increased.

5.3.4 *Conclusions*

In porting to the Cell/B.E. we achieved the target performance. Two main issues were solved. First, an efficient data structure mapping. Second, optimal use of the hardware, in particular bandwidth. Further possibilities for optimisations may lie in implementing stream flow of data among the SPEs.

The major constraint is the PPE performance. The PPE is an in-order execution processor with very few functional units. This property makes it difficult to get good performance on the critical task that must be executed on it. As much code as possible must be ported to the SPEs. In our case the PPE is not fast enough to produce the buffers that the 8 SPEs requires for each element loop computation. Also, the data layout can be a handicap for DMA transfers to the SPEs.

A solution would be to include a better PPE unit (Power6 or Power7 processor) in the Cell/B.E. chip. It should be interesting if future Cell/B.E. implementations have fewer constraints for DMA transfers.

5.4 Code_Saturne

Andrew Sunderland (STFC)

5.4.1 *Application description*

Code_Saturne® is a multi-purpose Computational Fluid Dynamics (CFD) software, which has been developed by EDF-R&D (France) since 1997. The code was originally designed for industrial applications and research activities in several fields related to energy production; typical examples include nuclear power thermal-hydraulics, gas and coal combustion, turbo-machinery, heating, ventilation, and air conditioning. The code is based upon a co-located finite volume approach that can handle three-dimensional meshes built with any type of cell (tetrahedral, hexahedral, prismatic, pyramidal, polyhedral) and with any type of grid structure (unstructured, block structured, hybrid). The code is able to simulate either incompressible or compressible flows, with or without heat transfer and has a variety of models to account for turbulence.

The initial version of Code_Saturne for the PRACE project is v1.3.2. For ease of porting, the GUI has been excluded from the benchmarks. The code base is very large, consisting of

500,000 lines written in Fortran77, C and Python. The breakdown of programming languages in v1.3.2 is as follows:

- 49% Fortran77
- 41% C
- 10% Python

Install scripts for a range of architectures are provided with the code. In version 2.0 of the code, to be released in March 2010, the Fortran77 will be replaced by Fortran95 structures. A beta version of 2.0 has been made available for PRACE benchmarking very recently. This new version of the code includes many of the optimisations described in this section of the document.

No external libraries are required for compilation of the code. Basic Linear Algebra Routines (BLAS) are provided as source files within the code suite. However there are directives that can be set to specify use of vendor BLAS libraries. As these should be highly tuned to the underlying architecture therefore using these may improve performance.

Users have the option to use the following libraries (these options are not invoked for the purposes of the PRACE benchmarking exercise):

- CGNS (CFD General Notation System)
- HDF5 (Hierarchical Data Format)
- MED (Model for Exchange of Data)
- METIS (Serial Graph/Mesh Partitioner)
- SCOTCH (Serial Graph/Mesh Partitioner)

The code is well structured, written in a clear style and is commented throughout in French and English. Installation is somewhat involved, but installation scripts for the PRACE prototypes have reduced porting times. A document describing the installation process and directory structure can be obtained from the authors.

Sparse Iterative Linear Solver for both Pressure and Velocity

Full simulations generally involve many hundreds or even thousands of time steps. Overall compute time is usually dominated by time spent in the Sparse Iterative Linear Solver. Code_Saturne includes two different (but related) methods to solve the sparse linear system of equations *(Ax=b)*:

Iterative methods for solving sparse matrix equations are based on a sequence of approximations to the solution. Each step of the sequence usually involves a series of matrix and vector operations. Finally the scheme converges to a solution within a given tolerance value, usually specified by the program and/or user. Iterative methods are well suited to large-scale applications as they do not suffer from the fill-in associated with direct methods and therefore storage requirements are much less.

Several iterative solvers are available, but for sparse symmetric systems Conjugate Gradient is the most predominant. Evidently the performance of the solver is dependent upon (i) the efficiency of the vector-vector and matrix-vector operations and (ii) the rate of convergence of the solver. The efficiency of (i) can be guaranteed by the use of level 1 BLAS for vector-vector operations and the implementation of an efficient sparse matrix-vector multiplication routine. For (ii) the spectral properties of the sparse system determine the rate of convergence of the solver. These properties can be improved via the introduction of preconditioning steps

into the CG algorithm. The equation to be solved then becomes $PAQQ^{-1}x = Pb$ where P and Q are left and right preconditioners respectively. In Code_Saturne a Jacobi preconditioner, based on an exact solution of the diagonal in the matrix, is implemented.

The multigrid algorithm is a fast and efficient method for solving systems involving partial differential equations. The algorithm solves a series of problems on a hierarchy of grids with different mesh resolutions. The main advantage of multigrid is that it accelerates the convergence of a base iterative method by correcting, from time to time, the solution globally by solving a coarse problem. This has the effect of removing errors of lower frequencies from the iterative solution and therefore accelerating the rate of convergence. The transition through the hierarchy of grids usually takes place via a V- or W-cycle of computations. A Conjugate Gradient solve takes place at each grid. In the Code_Saturne multigrid solver each set of coarser mesh points is a subset of finer grid mesh points. In the parallel implementation the minimum number of grid points that a process can hold is one. On very large sets of processors this can result in limitations for the coarsest allowable grid and therefore convergence rates on very large core counts could be somewhat slower than on smaller core counts.

5.4.2 *Porting*

Included in the Code_Saturne code bundle is an install script for automatic configuration and installation of the code on Unix/Linux-based HPC systems, including several high-end Cray and IBM systems. The code has been ported to the following Prace prototypes:

- CSC Cray XT4/XT5
- FZJ IBM Jugene
- SARA IBM-Power6
- HLRS Ontare

All the prototype platforms provide mature Fortran and C programming and compilation environments. This meant that the porting exercise was generally very straightforward. One point of note is that on the HLRS SX9 system the C99 compiler was required rather than C89.

Cray XT5 - Louhi

The code had been ported to the Cray XT4 (Hector) as part of a previous project so no specific porting issues were encountered. No specific source code modifications were required.

For numerical libraries we choose the Cray Scientific Library Libsci: This includes highly optimized Blas routines for use in the preconditioned sparse linear solver stage

The optimal compiler was found to be PGI. The compiler flags were set on a file by file basis and are set to `"-O1", "-O2",` and `"-O3 –fast"` for computationally critical routines.

IBM Blue Gene/P - Jugene

The code has been developed on an IBM BlueGene/L system so there were no porting issues

We used two libraries provided by IBM. The IBM Engineering and Scientific Library (ESSL) for highly optimized BLAS routines for use in the sparse linear solver stage, and the IBM Mathematical Acceleration Subsystem Library MASS, which is a set of subroutines for mathematical functions. They are linked as follows:

```
-L/opt/ibmmath/essl/4.3/lib –lesslbg
```

```
-L/opt/ibmcmp/xlmass/bg/4.4/bglib -lmass -lmassv
```

The optimal compiler for the program appeared to be the IBM xlf compiler. The optimal flags were `"-O1"`, `"-O2"`, and `"-O3 –qtune=450 –qarch=450"` for specific routines in C and `"-O3 -qarch=450 -qtune=450"` in Fortran.

The default stack memory limit on the BG/P is relatively low. This value needs increasing for the large datasets tested here. This can be achieved by setting the following keyword in the loadlever batch submission script.

```
#@ stack_limit = 300MB
```

IBM Power6 cluster - Huygens

The code had already been ported to an IBM Power5 system so the step to a Power6 based system was minor.

The same libraries as on the IBM Blue Gene/P were chosen. Namely, The IBM Engineering and Scientific Library Essl, and the IBM Mathematical Acceleration Subsystem Library Mass.

```
–lesslbg
```

```
-lmass -lmassv
```

The optimal compiler for the program appeared to be the IBM xlf compiler. The compiler flags were for C code `"-q64 -O3 -qtune=pwr6 -qarch=pwr6"` , and `"-q64 -qfixed -I. -qextname -O3 -qtune=pwr6 -qarch=pwr6"` for the Fortran code.

The default stack memory limit on the IBM Pwr6 system is relatively low. This value needs increasing for the large datasets tested here. This can be achieved by setting the following keyword in the loadlever batch submission script.

```
#@ stack_limit = 300MB
```

5.4.3 *Optimisation techniques*

Version 2.0 Beta of Code_Saturne includes the following optimisations:

Pre/Post Processing:

- partitioning for parallel computing is done separately; hence, when changing the number of processors from one calculation to another, it is no longer necessary to redo the full preprocessing (mesh pasting especially)
- optional parallel algorithm for mesh pasting (not yet compatible with periodicity)
- gradient tests (comparison of the gradient of a test function with the analytical solution) are done with all the gradient calculation algorithms in a single run of the check_mesh utility

Parallel Solver:

- new algebric multigrid algorithm for solving purely diffusive linear systems (pressure

  equation, vector potential for electric arcs, radiative transfer equations, velocity correction for Lagrangian tracking and ALE mesh velocity equation). This algorithm decreases the time for solving the pressure equation.

Multigrid

Experiments have shown that the integration of a multigrid-based conjugate gradient solver benefits performance by

1. accelerating convergence rates in the iterative solver, i.e. reducing the number of iterations per timestep.
2. reducing the average number of floating-point operations per iteration (due to fewer mesh points in the coarser grids

In addition to improving performance, this optimisation improves the robustness of the code by making convergence to the solution more likely. This optimisation is an integral part of the solution algorithm and therefore this optimisation is independent of the underlying computer architecture.

Figure 3 compares the solver performance of Multigrid/Conjugate Gradient vs Conjugate Gradient on a Cray XT4 platform. It shows how Multigrid improves performance by around a factor of three on lower processor counts, though the performance gains somewhat decrease on higher processor counts.

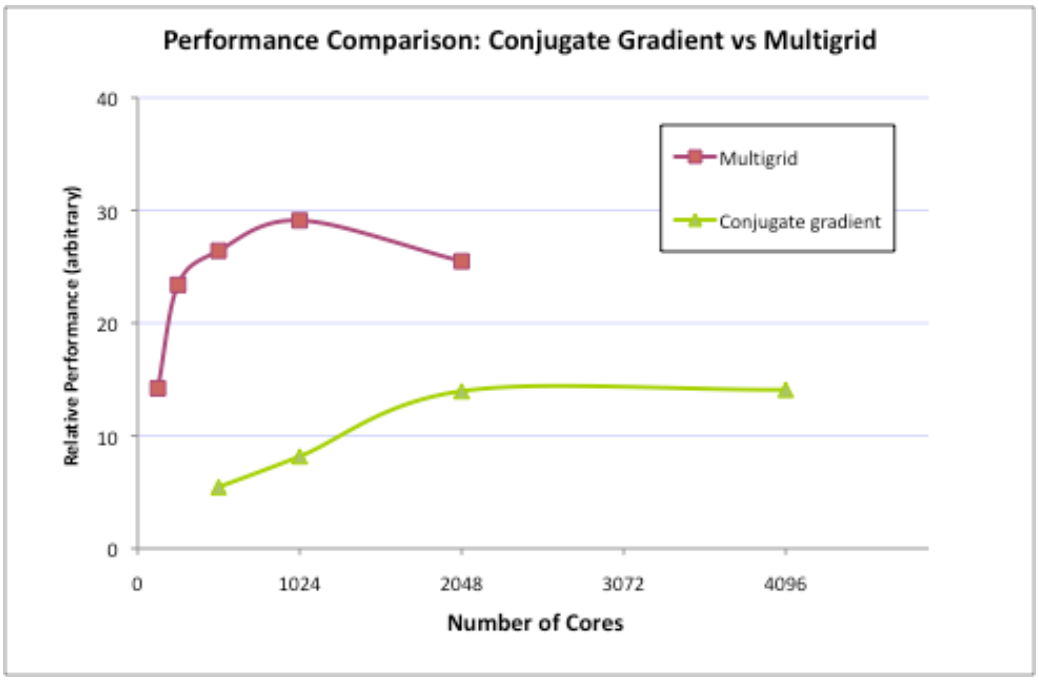

Figure 3: Multigrid vs. Conjugate Gradient performance on a Cray XT4

The multigrid solver is now the default solver in Code_Saturne. Users can choose to revert to the original Conjugate Gradient solver by applying a switch in the input file. Our analysis has shown that the multigrid method is the faster method, often reducing solver times by over 50% on lower processor counts (as shown in Figure 3). However the figure demonstrates that the parallel scalability of multigrid reduces more quickly than with Conjugate Gradient. The reasons behind this behaviour, along with possible remedies are explained in document 6.4, which discusses petascaling techniques

Machine-specific directives for scalar numerical operations

Cray-Pat profiles at 128 cores on the Cray XT5 shows 65% of the runtime in scalar numerical routines:

```
100.0% | 76111 | -- | -- | Total
|-----------------------------------------------
| 64.6% | 49177 |  --  |  -- | USER
||----------------------------------------------
|| 21.7% | 16544 | 2988.72 | 15.4% |_mat_vec_p_l_native
|| 10.9% | 8284 | 1515.80 | 15.6% |_conjugate_gradient_mp
||  7.9% | 6034 | 2619.52 | 30.5% |_alpha_a_x_p_beta_y_native
||  7.5% | 5686 | 1357.38 | 19.4% | cblas_daxpy
||  4.8% | 3620 | 1044.47 | 22.6% |_polynomial_preconditioning
||  4.6% | 3477 | 2077.88 | 37.7% | gradrc_
||  2.5% | 1936 | 323.73 | 14.4% |_jacobi
```

All of these are targets for optimisation. Most have been optimised for IBM & Bull platforms using pragmas helping the compiler produce optimal code. Some optimisation also exists for AMD processors, such as the one on the Cray XT5, which could potentially provide a speed boost.

An attempted optimisation strategy was to replace matrix vector calls by calls to highly optimised BLAS routines. This substitution was implemented on the Cray XT platform to take advantage of libsci, but yielded no substantial improvements. This work was ongoing at the time of writing and the analysis will be revisited for optimisations of the newly-released version 2.0 of Code_Saturne.

Compiler Flag Optimisations

The effect on performance of optimized compiler and linker settings has been analyzed for several of the PRACE prototype systems. The dataset investigated is the 10 Million Cell T-Junction case. The target processor count varies between platforms in order to achieve good throughput for multiple tests, therefore cross-platform comparisons should not be made between the runs detailed below. The results are summarized in the tables below.

CSC Cray XT5:

Optimisation Flags	Comments	Time (s)
`-O2`	equivalent to -O	1383.21
`-O2 -fast`	Chooses generally optimal flags for the target platform (see below)	1278.83
`-O3 –fast`	Level 2 optimisations plus more aggressive code hoisting and scalar replacement optimisations.	1280.0
`-O4 –fast`	Level 3 optimisations plus hoisting of guarded invariant floating point expressions.	1277.54
`-O4 –fast –Mipa=fast`	Inter procedural optimisations	1282.45

Table 23: Compilation flag optimisation analysis on Cray XT5 with 32 processors, with fully occupied compute nodes.

For the Cray XT5 platform we found that adding –fast (equivalent to –fastsse) to –O2 resulted in around an 8% speed-up on 32 processors. Adding higher levels of optimisation made little change: differences of around 1% in runtime as shown are within usual deviations for repeated runs on this platform.

FZJ IBM Jugene (IBM BlueGene/P):

Optimisation Flags	Comments	Time (s)
`-O3 -qtune=450-qarch=450`	Generates standard powerpc floating-point code, uses a single FPU	635.54
`-O3 -qtune=450-qarch=450d`	Attempts to generate double FPU code	642.74
`-O4`	Linker error occurred during the linking of an object produced by the IPA step	-

Table 24: Compilation flag test runs on Jugene with 1024 processes in VN mode.

Code_Saturne has been developed on the BlueGene platform. Recommended starting settings from the developers are `"-O3 -qtune=450 -qarch=450".`

All the timing runs were undertaken in virtual node (VN) mode. This mode utilises all four cores for computation on a Blugene/P node. Alternative modes are DUAL mode (2 tasks per node) and SMP mode (1 task per node). Tasks in jobs run in DUAL or SMP mode have access to greater, or even exclusive, shares of the memory hierarchy per node. However it is usually the case that more effective use is made of the overall resource by using all available cores in the nodes (i.e. VN mode) and this is the mode used in the tests.

The BlueGene/P processor has 2 floating point units, known as the *double hummer*. The XL compilers have options to create floating-point code to take advantage of these special units. The default setting for the Jugene compiler is double hummer optimisation **on** (*-qarch=450d*). This can be turned **off** by setting *-qarch=450* during compilation. Both options are tested in Table 24. Setting *-qarch=450* results in marginally faster run times compared to *–qarch=450d*.

At -O4 the level of Inter Procedural Analysis is increased. Although compilation of source files completed, the linker stage failed.

SARA IBM Power6:

Optimisation Flags	Comments	Time (s)
`-O2`	Equivalent to -O	1167.92
`-O3`	Aggressive optimisation, allows re-association, replaces division by multiplication with the inverse. Implies `–qhot=level=0`	1088.14
`-O3 –qtune=pwr6 –qarch=pwr6`	generates standard PowerPC floating-point code, uses a single FPU	911.04
`-O4`	`-O4 = -O3 -qtune=pwr6 -qarch=pwr6 -qcache=pwr6 -qhot=level=1 -qipa=level=1`	817.49

Table 25: Compilation flag analysis on IBM Power6 cluster- Huygens. The runs used 128 tasks, with 32 tasks per node.

Running with Simultaneous Multi-Threading (64 tasks per 32 core node) reduced the runtime by a further 8%. This is therefore the recommended mode for prototype benchmarking.

Results of optimisation effort

The results of the optimisations described above are shown in Figure 4 as *time per iteration* of the 10M cell dataset. This stage of the calculation generally dominates overall execution time in benchmark runs. The broken lines represent original timings from v1.3 on the prototype platforms and solid lines represent new timings from the optimized v2.0 beta. Levels of compiler optimisations in the two sets of runs are identical, so differences in performance are due to source code optimisations. Due to the recent release of v2.0, at the time of writing testing on all the PRACE prototypes have not been completed therefore timings from a similar XT platform to Louhi (HeCTOR) are provided for comparison.

Figure 4 shows that the optimisations in v2.0 (primarily the switch to a multigrid solver) have resulted in substantial improvements in performance. By comparing equivalent runs on the same or similar platforms it is shown that v2.0 is generally outperforming v1.3 by a factor of around 2 to 3. The parallel scalability of v2.0 is discussed in PRACE deliverable report 6.4.

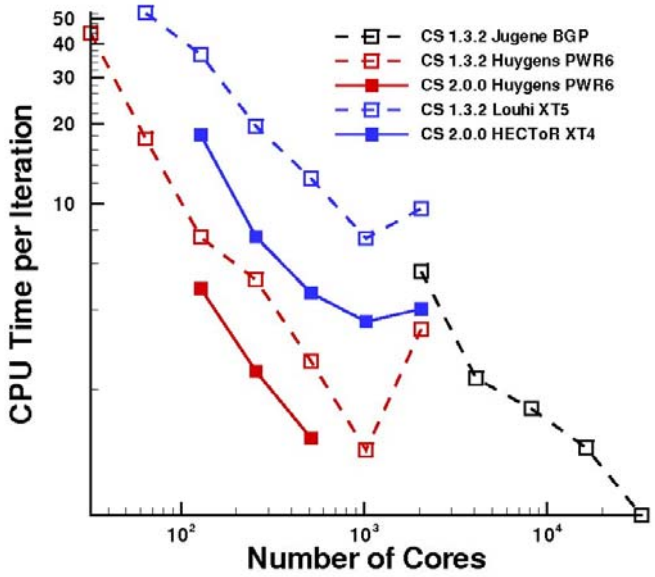

Figure 4: Optimisation effort results, including the switch to a newer version.

5.4.4 *Conclusions*

Optimisation techniques implemented during the course of the project have produced significant improvements to performance in Code_Saturne. The introduction of a multigrid-based solver has had a major impact, both by reducing execution times significantly and by providing more robust solvers which near-guarantee convergence. The iterative solver is composed of multiple vector-vector and matrix-vector operations and our investigations to date suggest that these algorithms are efficiently implemented on the PRACE platforms. However future work, particularly with Cray experts, will focus on attempting to optimize these operations further.

Compiler optimisation experiments yielded varying results on the prototype platforms. Performance increases from higher levels of compiler optimisations were significant on the

IBM Pwr6 architecture. However on the Cray XT5 and BlueGene/P architectures, introducing higher levels of compiler optimisation had little effect. Simultaneous Multi Threading (SMT) on the IBM Pwr5, where multiple threads can issue instructions to the multiple functional units on a single cycle, improved performance significantly. This suggests that memory latency (as opposed to memory bandwidth) is critical to the serial node performance of Code_Saturne.

5.5 CP2K

Pekka Manninen (CSC)

CP2K is a freely available (GPL) program to perform atomistic and molecular simulations of solid state, liquid, molecular and biological systems. It provides a general framework for different methods such as e.g. density functional theory (DFT) using a mixed Gaussian and plane waves approach (GPW), and classical pair and many-body potentials.

5.5.1 *Application description*

CP2K is written in Fortran 95. The code is well-structured, polished and easy to maintain. Furthermore, it is easy to read, and its clarity poses no problems to optimisation efforts.

CP2K needs fast Fourier transform and linear algebra (Lapack) libraries. Many commercial and non-commercial libraries are supported, such as FFTW, ACML and ESSL. The relative computational load depends very much on the problem (system studied, level of theory etc.). In a quite representative test case the DGEMM routine of Lapack was the most intense. Tuned libraries are employed by the code whenever possible.

5.5.2 *Porting*

In general, porting efforts were average, makefiles needed some tuning but no or little changes to source code were necessary on any platform. The story is different for the experimental OpenMP+MPI parallelized Hartree-Fock code, that is very hard to get to work anywhere. It needs lots of external code (e.g. the LibInt library), user-compiled versions of the basic libraries (BLAS, LAPACK, ScaLAPACK), compiles only with one version of the gfortran compiler, etc. Therefore this experimental part – with which the BCO was involved to at least to some extent – is left mostly out of the discussion and we will concentrate on "optimal porting" of the conventional quick-step DFT part of the code. The code was ported to Cray XT5 (Louhi), IBM Blue Gene/P (Jugene) and IBM Power6 cluster (Huygens) prototypes. Estimated effort for porting the code to platforms is around 1.5-2 pm.

As mentioned earlier, numerical libraries play a key role in the performance of CP2K, so the best implementation for them (especially for the DGEMM routine) should be found. As regards DGEMM: in Cray, it was found in the LibSci library; and from ESSL in the IBM prototypes. The built-in FFT library in fact gave better performance than the other options! The others were FFTW2 or FFTW3; ACML's FFT was buggy, giving erroneous results.

Cray XT5 - Louhi

This was the first time the code was ported to a XT5 and there was no makefile available in the CVS (there was a makefile for the XT3 platform), so the BCO made one that is now part of the CVS as well as the tar.gz version of the code. Practically no source code modifications were needed.

The prototype had very good versions of all the necessary libraries. After a bit of experimenting, LibSci was chosen for Lapack routines (that has the GotoBLAS assembler routines); CP2K's built-in library for FFT turned out to perform best.

PGI, Pathscale and GNU compilers were tested, of which Pathscale provided the fastest binary. Mostly the latest version of each compiler was tested (as moving down into an earlier version did not give immediate performance benefit, it was expected that the latest would provide the best performance).

Quite aggressive optimisation seemed to be ok and the more the compiler could vectorize the code, the better the performance was. Interprocedural optimisation broke the code and could thus not be used. The final set of flags were chosen to be for Pathscale:

```
-O3 -OPT:Ofast -OPT:early_intrinsics=ON
-LNO:simd=2 -intrinsic=PGI
```

All the different rank placements (MPICH_RANK_REORDER) were tried out, the default one (SMP-like) being the best.

IBM Blue Gene/P - Jugene

There was a makefile for a BG/L machine that was rather easy to adapt for BG/P. No source code modifications were needed. I was however unable to utilize the additional FP unit in each core; the code broke with the flags that enable it.

There were some problems with the FFT in the ESSL library, therefore (and backed up with the experiences in Cray XT5) the built-in library was again used in the performance analysis. FFTW3 seems to work too. ESSL was used for Lapack, and the IBM's MASS (Mathematical Acceleration Subsystem) was employed. The prototype featured top-end libraries.

The only available compiler is the XL Fortran that is usually regarded as one of the best available. Optimisation levels over O2 broke the code. The final set of flags was chosen to be:

```
-O2 -qarch=450 -qcache=auto -qmaxmem=-1 -qtune=450
```

No experimenting with the environment variables was performed.

IBM Power6 cluster - Huygens

The porting and performance analysis in this machine was kindly carried out by Vegard Eide (Sigma Norway). There was a makefile for some Power machine that could be adopted for Power-6. No source code modifications were needed.

ESSL's FFT and FFTW3 work fine. ESSL was used for Lapack. This prototype also featured top-end libraries.

The only available compiler is the XL Fortran that is usually regarded as one of the best available. Optimisation levels over O2 broke the code here too. The final set of flags was chosen to be:

```
-O2 -q64 -qarch=pwr6 -qcache=auto -qmaxmem=-1 -qtune=pwr6
```

No experimenting with the environment variables was performed.

5.5.3 *Optimisation techniques*

As most of the time in a medium and parallel run was spent in MPI routines, and most of the remaining time in library routines (mostly DGEMM), there was no place at all for “traditional” loop level serial optimisation. All the user functions contributed less than 1% of the wall time, and even that was optimally written.

The role of I/O bottlenecks in form of superfluous output was pointed out to the developers, who subsequently adjusted the default output levels. E.g. omitting the printing out the SCF energy in each SCF step yields 20% speedup in the code!

5.5.4 *Results of optimisation effort*

If the recommendations for the superfluous default output are not considered; the Prace impact is none. There are large gains available with compiler optimisation. The present flags should provide significantly faster binary than a "first guess port".

The profile and HW counters were basically unaltered since the present set of complier flags was fixed.

5.5.5 *Conclusions*

At least one lesson learned: I/O is always an issue, no matter how innocent it looks.

Code was already library-dominant, so there was not much room for hands-on involvement. Finding the truly optimal set of optimisation flags with a sampling algorithm will be the topic of further investigation. The utilization of the other FP unit in BG/P is a potential source of speedup. Larger caches and faster CPU frequencies would a priori speed up this DGEMM-dominant code.

5.6 CPMD

Albert Farres (BSC)

The CPMD code is a parallelized plane wave/pseudopotential implementation of Density Functional Theory, particularly designed for ab-initio molecular dynamics. It was originally developed by R. Car from the International School for Advanced Studies, Trieste, Italy and M. Parrinello from the Dipartimento di Fisica Teorica, Università di Trieste, Trieste, Italy, and the International School for Advanced Studies, Trieste, Italy. Nowadays it is maintained by the CPMD consortium, coordinated by Prof. Michele Parrinello (Chair of Computational Science, ETH Zurich) and Dr. Wanda Andreoni (Program Manager of Deep Computing Applications at IBM Zurich Research Laboratory).

The code is completely written in FORTRAN 77 except some architectural dependent parts written in C. Currently, there are about 20 000 lines of code. External libraries used by the program are BLAS, LAPACK and optionally FFTW. It also uses hybrid parallelization (MPI/OpenMP).

We have used two datasets. The first contains 32 water molecules in the liquid phase at 100 Ry and MT pseudopotentials, and the second one 64 Ionic Liquids.

5.6.1 *Porting*

All the porting efforts have been concentrated in porting dependent libraries, i.e., recompiling theses libraries or searching recompiled binaries and tuning compiler options for each architecture.

IBM Blue Gene/P - Jugene

The code had already been ported to the architecture. We did not need to do any source code modifications. The prototype has all required libraries preinstalled, so we use the implementation provided by the prototype manufacturer. For FFT we used the FFTW library.

The optimal compiler choice has been bgxlf_r. This is the IBM XL Fortran compiler specific for the BlueGene architecture. Optimal Flags for object generation were:

```
FFLAGS='-O -w -qsmp=omp -qnosave -qarch=450 -c \
        -I/bgsys/drivers/ppcfloor/comm/include'
```

Optimal flags for object linkage were:

```
LFLAGS=' -O -w -qnosave -qsmp=omp -qarch=450d  \
       -L/bgsys/drivers/ppcfloor/comm/lib \
       -lmpich.cnk -ldcmfcoll.cnk -ldcmf.cnk \
       -L/bgsys/drivers/ppcfloor/runtime/SPI      -lSPI.cna      -lrt
        -lpthread -L/bgsys/local/lib -llapack -lesslsmpbg '
```

CPMD is a hybrid (MPI+OpenMP) application. Thus optimal system environment variables for running the code are:

```
OMP_NUM_THREADS="4"
```

IBM Power6 cluster - Huygens

This porting had no specific porting issues. This was the first time the code has been ported to this architecture, but we did not need to do any source code modifications.

The prototype had all required libraries preinstalled, so we use the implementation provided by the prototype manufacturer. For FFT we used the FFT-ESSL library.

The optimal compiler choice was xlf_r. This is the IBM XL Fortran compiler. Optimal flags for object generation were:

```
FFLAGS = -O -q64 -qmaxmem=32768 -qtune=pwr6 -qarch=pwr6 \
                -qsmp=omp -qnosave -qextname=printmemsize
```

Optimal flags for object linkage were:

```
LFLAGS = -O -q64 -L/sara/sw/lapack/3.1.1/lib -llapack -lesslsmp \
            -qsmp=omp -qarch=pwr6 -qnosave
```

IBM Cell cluster - Maricel

This porting had no specific porting issues. This was the first time the code has been ported to this architecture and some source code modifications were necessary. Specifically, minor changes were introduced to make it compatible to the FFTW3 library, without using threads.

The prototype did not have all required libraries preinstalled, so we had to compile and install some of them. Specifically, the BLAS library had no support for 64 bits architecture and FFTW3 was not installed.

The optimal compiler choice was ppuf77. Optimal flags for object generation were:

```
FFLAGS = -O3 -q64 -qarch=cellppu -qtune=cellppu
```

Optimal flags for object linkage were:

```
LFLAGS = -O3 -q64 -qarch=cellppu -qtune=cellppu \
```

```
-L/opt/openmpi/ppc64/lib -llapack -lblas_xlf -lblas \
-lmpi_f77 -lmpi -L/home/Applications/afarres/soft/lib \
-lfftw3_gcc -lspe2
```

Optimal system environment variables for running the code were:

```
OMP_NUM_THREADS=1
PATH=$PATH:/opt/openmpi/ppc64/bin/:/opt/perf/bin/
LD_LIBRARY_PATH=/opt/openmpi/ppc64/lib/
BLAS_USE_HUGEPAGE=0
```

5.6.2 *Optimisation*

No source level optimisation was performed.

5.6.3 *Conclusions*

We have ported the code to three PRACE prototypes.

5.7 EUTERPE

Xavier Saez (BSC)

EUTERPE is a gyro kinetic PIC (particle-in-cell) code in a three dimensional domain in coordinates and two in velocities plus time. Its main target is to simulate the micro-turbulences in the Plasma core. The code uses the Vlasov approximation over an electrostatic fixed equilibrium, so the collision term of the Boltzmann equation is negligible.

5.7.1 *Application description*

EUTERPE is written in Fortran90 (about 47 files) and C (2 files). The application includes a tool to generate the electrostatic fixed equilibrium. The code is well documented, which makes it easy to read and follow. The current release is 2.54.

EUTERPE requires the following libraries in order to compile: a library with MPI implementation (MPICH), a FFT library (ESSL, FFTW ...), and a library for solving sparse linear systems of equation (PETSc or WSMP).

EUTERPE can be divided in two computational parts, executed inside a time-step loop. In one part, particles interact with the electrostatic field. This interaction is described by the equation of motions. The corresponding set of coupled differential equations is integrated in time using and explicit fourth-order Runge-Kutta method. In the other part of the code, the fields created by the particles are calculated. This is done by solving the quasi-neutrality equation. Its source term is the gyro-averaged ion density, which is calculated from the particle positions in a process called charge-assignment. Finite elements are used to represent both the electrostatic potential and the particle shape function in the charge-assignment. The finite elements used are a tensor product of cubic B-splines.

Most of the computational load falls on the routines that compute the motion of the particles (push), the solution of the lineal solver (poisson_solver) and the noise reduction with fast fourier transformation (ppcfft2).

5.7.2 *Porting*

The greatest efforts for porting EUTERPE were focused on the checking of library availability, searching for unavailable libraries and the tuning of the compiler flags for each architecture.

One of the lessons learned from porting the code to different platforms was the need to be organized in the management of different configurations. I think maintaining a different file with the compiler flags and libraries for each platform is a good practice. Likewise, developing a makefile that auto-detects the platform and selects the suitable configuration file.

Finally, the effort involved in this task was about 4 pm.

IBM Blue Gene/P - Jugene

EUTERPE had already been ported to this architecture. There were no specific porting issues on this platform, nor were any modifications to the source code necessary.

The selected libraries were: ESSL (libesslbg), PETSc and MPI2. All the libraries were installed previously.

The compiler used was mpixlf90_r (bgxlf_r). The optimal flags were:

```
-qtune=450 -qarch=450 -O3 -qautodbl=dbl4 -qmaxmem=-1
-qflag=I:I -I$(OBJDIR) -qmoddir=$(OBJDIR) -qsuffix=cpp=F90
-qfixed
```

Finally, if WSMP was the selected library for solving sparse linear systems instead of PETSc, the environment variables `MALLOC_TRIM_THRESHOLD_` and `MALLOC_MMAP_MAX_` should be set to -1 and 0, respectively. In all other cases, no specific variable has to be set.

IBM Power6 cluster - Huygens

This is the first time that EUTERPE has been ported to this platform. There were no specific porting issues on this platform, nor were any modifications to the source code necessary.

The selected libraries were: ESSL, PETSc and MPI2. ESSL and MPI2 were installed previously. PETSc was not pre-installed so its installation was requested from the support team.

The compilers used were mpcc (xlc_r) and mpfort (xlf_r). The optimal flags were:

```
-q64 -qtune=pwr6 -qarch=pwr6 -qcache=auto
-qextname=flush -O3 -qstrict -qautodbl=dbl4
-qmaxmem=-1 -qflag=I:I -I$(OBJDIR) -qmoddir=$(OBJDIR)
-qfree=f90 -qsuffix=cpp=F90 -qfixed
```

Finally, if WSMP was the selected library for solving sparse linear systems instead of PETSc, the environment variables `MALLOC_TRIM_THRESHOLD_` and `MALLOC_MMAP_MAX_` should be set to -1 and 0, respectively. In all other cases, no specific variable has to be set.

IBM Cell cluster - Maricel

This is the first time that EUTERPE has been ported to this platform, but no modifications to the source code were required. Although it was necessary to set the following environment variables for compiling the source code with MPI:

```
export OMPI_F77=/opt/ibmcmp/xlf/cbe/11.1/bin/ppuf77
export OMPI_F90=/opt/ibmcmp/xlf/cbe/11.1/bin/ppuxlf90
export OMPI_FC=/opt/ibmcmp/xlf/cbe/11.1/bin/ppuxlf90
export OMPI_CC=/opt/ibmcmp/xlc/cbe/10.1/bin/ppucc_r
```

The selected libraries were: BLAS, LAPACK, FFTW3 and OPENMPI. BLAS had not support for 64bit architecture, FFTW3 was not pre-installed and LAPACK and OPENMPI were available. On the other hand, the PETSc library is not yet installed, so EUTERPE is using our hybrid version of the solver.

The compilers used were mpicc (ppuxlc) and mpif90 (ppuxlf90). The optimal flags were:

```
-q64      -qtune=cellppu      -qarch=cellppu      -qcache=auto
-qfree=f90   -qextname=flush   -O3   -qstrict   -qautodbl=dbl4
-qmaxmem=-1    -qflag=I:I    -I$(OBJDIR)    -qmoddir=$(OBJDIR)
-qsuffix=cpp=F90 -qfixed
```

Equally important, to link it was necessary to add:

```
-L/opt/openmpi/ppc64/lib -lmpi_f77 -lmpi
```

Finally, the following environment variables have to be set for running EUTERPE:

```
export
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/openmpi/ppc64/lib
export PATH=$PATH:/opt/openmpi/ppc64/bin/:/opt/perf/bin
```

5.7.3 *Optimisation techniques*

The code has a good performance, so our priority has been to improve the scalability of the application. Up to now, we have not applied any optimisation technique in the original code, but we have plans to improve the performance of the solver and the FFT in the near future.

The implementation of the Deflated Conjugate Gradient will allow us to go faster, because the required number of iterations for finding the solution will be reduced.

About FFT, we are studying the scheduling of the communication because we have the feeling that it can get better.

Finally, we need to consider, that the code is memory bound, because the cost of memory operations is linear with respect to the cost of computation. For example, the equation of the movement of the particles has the same order of float operations as memory accesses. Likewise, in the multiplication of a sparse matrix by a vector, the data are not reused and there are not enough float operations to take advantage of the hardware. Therefore, the increase in the memory bandwidth would improve the performance of the application.

5.8 Gadget

Orlando Riviera (LRZ)

GADGET computes gravitational forces with a hierarchical tree algorithm (optionally in combination with a particle-mesh scheme for long-range gravitational forces) and represents

fluids by means of smoothed particle hydrodynamics (SPH). The code can be used for studies of isolated systems, or for simulations that include the cosmological expansion of space, both with or without periodic boundary conditions. In all these types of simulations, GADGET follows the evolution of a self-gravitating, collisionless N-body system, and allows gas dynamics to be optionally included.

5.8.1 *Application description*

The current public version is known as version 2.0.3. However, in PRACE a non-open version (3.0) was used. This new version includes mainly improvements in the memory management.

The source is written in Ansi C (C99) and contains more then 92000 lines of code. GADGET is an MPP-based program. The parallelization is achieved by means of the MPI-interface and any MPI implementation, which follows the 1.1 standard, should compile successfully. Apart from the MPI library, it is necessary to link the application against the GNU Scientific Library (GSL), for these tests GSL version 1.09 until 1.12 have been used, but newer and future versions should also work. Another library necessary for compilation is the Fastest Fourier Transformation in the West (FFTW). The only version compatible is version 2.1.5, which at the moment of writing this report is fully MPI-implemented. Newer versions, although offering better performance, have a different interface and the parallel MPI implementation is still at the beta stage.

Several options are available at compile time through pre-processor flags. This strategy reduces significantly the number of branches and improves performance. Thus, for different calculation types different binaries are generated. For this benchmark, a cosmological, commoving box with periodic boundary conditions was used.

The input used in this benchmark consists of a set of 4096 particles. This set is tiled according to a given factor (GlassTileFac) in each direction forming a cube-like structure. The size of the input and the number of particles follow a power-of-3 law:

$$\text{Number of Particles} = 4096 * \text{TileFac}^3$$

The size of the input is platform dependent but (using double precision) can be calculated with:

$$\text{Data Size(MB)} = 0.236 * \text{TileFac}^3$$

This is roughly 57 bytes of information for each particle.

5.8.2 *Porting*

Porting is normally straightforward. It requires some care, especially when selecting the FFTW library. Some platform maintainers offer the last version available (3.0) and GADGET has to be compiled against an older version (2.1.5).

It is desirable to link against dynamic libraries (MPI, FFTW, GSL, etc.) using some optimisation options, which requires a larger compilation time (*ipo, fast, qipa*, etc.) or you can use newer versions of the libraries without recompiling your code.

Altix 4700

Small modifications are required to adapt the source to the convention used at LRZ/HLRB2. HLRB2 is a dual-core Itanium2-based system with 9728 cores and 39 TB of memory distributed into 19 partitions. Each partition has 512 cores. The memory available per core is

4GB. Since each partition can be used as a shared system, a lot of memory can be assigned to a MPI-Task, pthread or OpenMP thread in an asymmetric fashion.This system has a NUMAlink 4 as interconnect. 600 TB of storage is attached directly to the computer. The peak performance is 62.3 Tflops, while the Linpack Performance is 56.5 Tflops.

For compiling GADGET, the standard GSL (version1.09 – 1.12) is required. This library ought to be compiled with the Intel compiler, at least *-O3* optimisation flag and linked with the Intel's math kernel library (mkl 9.0) for BLAS and LAPACK support.

Another library is FFTW, which is compiled also with the Intel compiler and the SGI's MPI implementation (MPT). The libraries installed are double or single precision with no prefix, forcing a subsequent modification in the code of GADGET.

For building GADGET and its dependencies the Intel C/C++ compiler version 10.1 was used, that generates highly optimized code for the Itanium2 core family. The options used to gain the best results were:

```
-O3 -ipo -ftz -mtune=itanium2 -mcpu=itanium2
```

Since in several places some functions are called many times, the `-ipo` option ensures inlining and allowing more inter procedural optimisation.

Finally, if the code is linked with the libraries already installed on the system, the module system hast to be used for a proper environment settings. If the libraries are built by the user, it is advisable to hardcode the path into the executable (linker -R option) to avoid errors by calling the wrong library version. The load manager used at LRZ is PBS-Pro 10

IBM Power6 cluster - Huygens

GADGET was ported on this architecture for the first time but no modifications in the source code are required and as in the previous case you need to call the right FFTW version. FFTW and GSL are needed, but only FFTW double precision is installed on this system and only static libraries are available. To perform calculations in single precision and take advantage of the vector unit, it is necessary to rebuild the FFTW library for single precision. GSL has also to be built by the user in all cases.

The IBM C/C++ compiler family has to be used (version 10.1 in this case) but by default the compiler will generate Power5 and Power6 compatible code. Optimized code generated specifically for Power6 must use at least 3 options:

```
-q64 -qtune=pwr6 -qarch=pwr6
```

The following table summarizes the run times (only the execution part with no pre-processing), by running a 32MPI-Task problem on one node, reducing the internodes communication overhead, with an adequate input data set of $2x10^6$ particles (TileFac=8):

Flags	Runtime (s)
`-q64 -qtune=pwr6 -qarch=pwr6 -O3`	229.32
`-q64 -qtune=pwr6 -qarch=pwr6 -O4`	208.09
`-q64 -qtune=pwr6 -qarch=pwr6 -O5`	206.1
`-q64 -O5 -qipa=exits=endrun,MPI_Finalize:inline=auto`	204.12
`-q64 -qtune=pwr6 -qarch=pwr6 -O5 -qipa=exits=endrun,MPI_Finalize:inline=auto -qhot=vector -qenablevmx -qnostrict -qunroll=yes`	172.88 (single precision)

Table 26: Compiler flags and execution times for Gadget on Huygens

For this case *-O5* optimisation implies *-qtune* and *-qarch* for the specific target architecture, *-qhot=level=1* and *-qipa=level=2* are also set. If the exit functions are known you can tell the compiler that at these branches the procedures will not return (avoiding some save/restore sequences). Also inlining is recommended by giving the appropriate *-qipa* option and specifying the functions names that should be inlined or using auto inlining.

It has to be noted that, if the code can use single precision or integer arithmetic, you can obtain higher performance. For this architecture the compiler can generate code that will profit from the AltiVec units by taking advantage of VMX instructions and automatic SIMD vectorization and vectorizing segments of the code. A step further would be to include AltiVec intrinsics to replace general purpose subroutines.

IBM Blue Gene/P - Jugene

One can face some minor problems when porting GADGET onto Jugene. Specifically by using the GSL or FFTW libraries, since the front end and the compute node have different architectures.

GSL has to be built with the proper compiler and compiler options. In this case the configure script has to called with the following options:

```
configure CC=bgxlc_r CFLAGS=-qtune=450 -qarch=450 -O3.
```

FFTW with MPI support has to be used. However, configure will always find `mpcc` instead of `mpixlc_r` as the MPI compiler. In order to solve this problem, configure has to be slightly modified and the following options have to be also given:

```
CC=bgxlc_r CFLAGS=-qtune=450 -qarch=450 -O3 --enable-mpi
```

Finally, in many platforms the GMP library (GNU **M**ultiple **P**recision Arithmetic Library) is found by default inside of one of the known directories. In Jugene, the path has to be explicitly specified and is: /bgsys/local/gmp.

It is advisable to use the IBM's C/C++ compiler and their MPI scripts (*bgxlc_r* and *mpixlc_r*). This test was conducted using a 32-node card in order to eliminate inter node communication. Each card has 32 cpus and 4 cores/cpu. In the VNM mode, 128 MPI-tasks can be launched with a maximum of 1 GB of RAM per task. The data was generated with GlassTileFac=10 and contained 4.096 million particles.

Flags	Runtime
`-qtune=450 -qarch=450 -O3`	60.14
`-qtune=450 -qarch=450 -O4`	58.62
`-qtune=450 -qarch=450 -O5`	58.05
`-qtune=450 -qarch=450 -O5 -qhot -qipa=level=2`	58.05
`-qtune=450 -qarch=450d -O5 -qhot -qipa=level=2`	59.29

Table 27: Compiler flags and execution times for Gadget on Jugene

The best set of options was with `-O5`, which implies `-qhot=level=0`. Interprocedural optimisation (`-qipa`) yields no visible improvements.

The expansion done to the BG/P has reduced the amount of memory available to 1GB per MPI-Task in Virtual Node Mode (VNM). GADGET is a memory extensive application and we are approaching the memory limit for useful large scale problems. Eventually an OpenMP implementation might reduce the execution time for a given problem.

On Jugene the maximum number of OpenMP threads is 4. Since Petascale systems use 1 or even 2 orders of magnitude more cores than current systems, larger share memory systems are more suitable for feasible OpenMP implementations of GADGET.

Nehalem Cluster - Baku

The cluster consist of 700 nodes, each with two quad-core Intel Xeon (X5560) Nehalem at 2.8 GHz with 8MB cache and 12GB of memory per node. The peak performance per node is 11.2 GFlops. The system has a total of 62TFlops. The nodes are interconnected via InfiniBand. I/O is preferably (as for this benchmark) done on a lustre file system connected via InfiniBand.

Intel compiler V11.0 and OpenMPI V1.3 installed by HLRS were used. FFTW and GSL have to be built by the user. FFTW 2.1.5 with double precision and the GSL library as provided in the JuBE framework. GADGET and GSL where compiled with *CFLAGS=-O3*

5.8.3 *Other optimisation techniques*

Load Balancing is an important issue for massively parallel computers, especially for problems where the domain decomposition will vary as the simulation evolves, for example particle based problems, grid refinement, etc. GADGET uses a Hilbert-Peano curve to redistribute the load as optimal as possible.

There are two functions: force_treeevaluate and force_treeevalueate_ewald_correction, which account for 48% and 33% of the total time. Unfortunately these functions do not depend on any particular kernel. These functions (file: forcetree.c) consist of several loops (for and whiles) and branches. Some branches are evaluated for every cell within the domain and contain many sub-branches, up to 4 levels. Some sections inside these branches are large sections of code. Thus, the compiler cannot take advantage of branch predication. In general, an overview of these sections will indicate a higher ratio of branch miss prediction and probably cache misses.

Some optimisation techniques, which do not require extensive modification of the code are, for example, using a reverse counter in for loops. Several sections of the code are also well suited for code hoisting, in which some calculations are moved out of the loops.

Some hard coded constants can be replaced for variables in the register. This modification has shown improvement up to 6% on the Altix system. On Power6 there was no improvement.

5.8.4 *Conclusions*

Before applying any optimisation techniques it is best to try to know in more detail the nature of the code and algorithms. This helps selecting the appropriate set of compiler flags and compiler optimisation instructions.

GADGET, as well as other programs, presents a memory limitation. A solution in a multi-core environment is to deploy one single MPI-Task in a blade to get enough memory resources but keeping the remaining cores unused. In this case using OpenMP or threading will be advantageous, especially in systems with a low amount of memory per node. On the Jugene system, the maximum expected speed-up with a very efficient OpenMP implementation is 4.

Optimisations for specific architectures will always pay off, as in the case of the Power6 architecture, where making use of the AltiVect units, single precision arithmetic (with no code modifications) gives up to 18% performance improvement. For platforms with accelerators the increment in performance can be even higher but it is necessary to make the effort required to port the code.

For GADGET larger memory is an advantage and depending on the characteristics of problems to be solved, it might be the crucial point in determining, whether the architecture is suitable. Since for now it is a pure MPI application, faster node interconnects are also a desirable feature.

Since GADGET spends around 15-20% of the run time in integer operations (sorting), the use of vector units specifically designed for these operations in combination with a compiler capable of making good use of these features is highly recommended.

5.9 GPAW

Jussi Enkovaara (CSC)

GPAW is a software package for electronic structure calculations of nanostructures. The software can work within density-functional theory for ground state calculations as well as within time-dependent density-functional theory for excited state calculations. GPAW is GPL-licensed open-source software and it is developed in several universities and research institutes.

5.9.1 *Application description*

The program is written in Python and C. Currently, there are ~50000 lines of Python and ~13000 lines of C. External libraries used by the program are Numpy (fast array interface to Python), BLAS, LAPACK, SCALAPACK, and MPI. Most of the Python-code is well documented and readable, parts of the C-code on the other hand are harder to read.

The program contains extensive test suites, which helps in confirming the correct behaviour of the program in porting and optimisation efforts.

The main high-level parts of the ground state calculation within density-functional theory (DFT) and their scaling with system size (N), are:

1. Construct Hamiltonian O(N)
    a. Solve Poisson equation
2. Subspace diagonalization $O(N^3)$
    a. Calculate Hamiltonian matrix from wave functions
    b. Diagonalize Hamiltonian matrix
    c. Rotate wave functions
3. Iterative refinement of wave functions $O(N^2)$
4. Orthonormalization $O(N^3)$
    a. Calculate overlap matrix
    b. Cholesky decomposition
    c. Rotate wave functions

In the above parts of the algorithm, constructing the Hamiltonian and iterative refinement of wave functions involve user functions while subspace diagonalization and orthonormalization utilize mostly library functions.

In the real-time time-dependent density-functional theory (TDDFT) calculation the high level algorithm is:

1. Construct Hamiltonian O(N)
2. Time-propagate wave functions $O(N^2)$

Time-dependent calculation relies more on user functions than the standard DFT calculation.

The physical quantities (wave functions, densities, potentials) are represented on uniform real-space grids. The most important user functions in both calculation modes are the finite-difference derivatives on the real-space grid.

5.9.2 *Porting*

GPAW depends on following external packages/libraries:

- Python
- NumPy
- BLAS, LAPACK (SCALAPACK)
- MPI

On standard Unix-platforms all the packages expect NumPy are normally available, and the installation of NumPy is relatively straightforward. As GPAW relies on Python's distutils system for compiling the C-parts of code, the same compiler and options that were used when building Python are used by default. It is possible to specify additional compiler options or even use a different compiler, but one has to make sure that all the libraries are compatible with the used compiler.

Generally, supercomputers do not offer standard Unix-environment at least in the compute nodes, which together with cross-compilation issues can make porting of GPAW a non-trivial task. There were previous ports to Cray XT5 and to IBM Blue Gene/P, which made the porting effort within PRACE relatively small.

Cray XT5 - Louhi

Currently, Python is not supported on the operating system of compute nodes, the Compute Node Linux (CNL). The main problem being the lack of shared libraries which Python uses extensively for extension modules. Porting of Python to CNL is a non-trivial task, which requires modifications to the Python source code in order to enable static linking of the required Python modules. These porting issues were already solved earlier.

Once Python runs on the compute nodes, porting of NumPy and GPAW is relatively straightforward and requires only specification of correct BLAS, LAPACK and SCALAPACK libraries. Tests indicated that in this application, there were no large differences in the performance of ACML and Libsci libraries.

The port on Cray XT5 was done with PGI C-compiler. Only the '-fastsse' compiler flag was used. As the DFT calculation relies largely on library functions (whose importance scales as $O(N^3)$ with the system size), the used compiler and compiler options have relatively minor effects on the performance on Cray XT5 which has also been confirmed in earlier tests. In TDDFT calculations the proper choice of compiler and options could have larger effects, however, this requires further work.

The detailed porting instructions for Cray XT5 can be found at *https://wiki.fysik.dtu.dk/gpaw/install/Cray/louhi.html*

IBM Blue Gene/P - Jugene

Currently, Python as well as shared libraries are supported on BG/P. The code has previously been ported to other BG/P systems. The largest problem in the current porting effort was the accidental use of wrong system libraries. As the front-end and the compute nodes on BG/P are different, cross-compilation is required. Even though the processor architectures on front-end and compute nodes are different, code built for front-end nodes can in some cases still be executed on compute nodes. However, the behaviour might be ill-defined. In the current porting effort, code built accidentally with front-end libraries passed over 90 % of the test suite, and it took some time to figure out the problem for the remaining failures. Once the correct libraries were used, the whole test suite could be executed correctly.

Python on BG/P is built with the GCC C-compiler and in the first phase it was also used for GPAW. For BLAS, the ESSL library was used. In order to use the xlc compiler, technical issues with Python's distutils build system require a special compiler wrapper script. Preliminary tests indicate that on BG/P the xlc compiler can increase the performance by 10-15 % compared to GCC. The compiler options used with xlc are:

```
-O5 -qhot=nosimd -qlanglvl=extc99 -qnostaticlink -qsmp
-qarch=450d -qtune=450 -qflag=e:e
```

More detailed porting instructions for different BG/P platforms can be found at https://wiki.fysik.dtu.dk/gpaw/install/platforms_and_architectures.html

IBM Power6 cluster - Huygens

IBM Power6 cluster has a standard Linux operating system, which makes porting of GPAW relatively simple task. Python is supported on Power6 and also NumPy is preinstalled on the system. At this stage only the GCC compiler with `'-O3'` optimisation was used. The ESSL library was used for BLAS.

The detailed porting instructions for IBM Power6 cluster can be found at https://wiki.fysik.dtu.dk/gpaw/install/Linux/huygens.html

5.9.3 *Optimisation techniques*

Already for medium size system DFT calculations most time is spent in library routines mainly BLAS and MPI. The most important BLAS routines used are dgemm, dsyrk2k, and dsyrk. These operations scale $O(N^3)$ with the system size, so when going to petascale calculations with larger systems to simulate, their contribution increases even more. Thus, in DFT calculations there is very little room for optimisation of the serial performance of user functions, and most of the effort has been focused on optimisation of parallel performance. Serial performance has not been improved during the PRACE work.

The most important optimisation aspect in the code is the use of system-optimized libraries as much as possible. With the current state of the code, the best possibilities for performance increases for DFT are in algorithmic developments and not in the tuning of the program code.

5.9.4 *Conclusions*

The perfomance of DFT calculation is largely determined by the performance of underlying libraries, especially BLAS. An increasingly important aspect of GPAW is the combined use of Python and C. Due to its interpreted nature, Python code is naturally relatively slow, but by implementing the most critical parts in C and using libraries, the overhead from Python is in current test systems typically less than 10 %.

During this work only a little work has been done for optimisation of TDDFT calculation. TDDFT calculation relies more on user functions, and currently the Python overhead is also larger than in DFT calculation. Thus, there could be more room for optimisations in the TDDFT part of the code. TDDFT calculation relies largely on matrix-vector and vector-vector operations and the memory bandwidth is an important aspect for the performance.

5.10 Gromacs

Sebastian von Alfthan (CSC)

Gromacs is a molecular dynamics package primarily designed for biomolecular systems such as proteins and lipids. It is actively developed by an international team of contributors, mostly originating from central and northern Europe.

5.10.1 *Application description*

The current version of the package is 4.0.5 and it consists of 370555 lines of C code. In addition to the simulation program there are almost 1 million lines of assembler code for non-bonded interactions. The large number of lines in the assembler part is due to the fact that there are several versions supporting different instruction sets, some of which only differ by a small degree. There are also separate versions for Intel compilers and other compilers. The non-bonded assembler kernels support 3DNow!, SSE, SSE2, IA-64 and AltiVec. Gromacs is a well written and documented code, but the number of comments are quite limited, as in many other scientific codes.

The parallelization is done using MPI. As for libraries it supports both FFTW2 and FFTW3 but it is recommended to use FFTW3 due to its superior performance. Gromacs relies heavily on FFT computation for efficient computation of long-ranged forces. Gromacs also utilizes BLAS routines, but not in performance sensitive areas.

In a classical MD simulation one simulates material on an atomistic level. The particles represent atoms, or coarse grained particles representing a group of atoms. To simulate a system of particles one iteratively updates the particle positions forward in time. At each time-step one computes the forces affecting each particle, based on force-fields describing the interaction between the particles. These force-fields describe Coulombic (long ranged) forces, van der Wals interaction, bonds in polymers, etc. Additionally a real MD program implements constraints, e.g., to describe hydrogen bonds, and thermostats and barostats to simulate different ensembles, e.g., NVT, NPT. The main computational load in a typical MD program is the computation of forces.

Short-ranged forces in Gromacs are computed using assembler routines, and are thus computationally very efficient. Long ranged forces can be computed in several ways in Gromacs; the two relevant ways are reaction field (RF) and particle mesh Ewald (PME). In RF the computation is done in the same manner as with short-ranged forces, using assembler routines, while PME is in practices completely limited by the performance of the FFT library and the MPI_Alltoall performance. Thus we haven't done any low-level optimisation in Gromacs as there isn't any low hanging fruit. We did spend some time improving the scalability as presented in PRACE report D6.4.

In PRACE we have several test cases but the one which is most relevant comprises two vesicles in water, with 1752 POPC lipids and 334489 water molecules giving in total 1094681 atoms. When using PME the number of grid points in the parallelized x-direction is 176. This test case was kindly contributed by Erik Lindahl, one of the Gromacs developers.

5.10.2 *Porting*

It was fairly straightforward to port Gromacs to the three studied prototypes, IBM Blue Gene/P, Cray XT5 and IBM Power6 cluster. At the time of writing, Gromacs is compiled using the GNU autotools (autoconf & make), but future versions will use cmake. The required libraries, FFTW 3.x and BLAS, are available on practically all architectures. The only thing that has to be ensured, is that one is able to utilize assembler routines suitable for the processor, if such are available. Gromacs had assembler routines for all machines to which we ported the code.

Cray XT5 - Louhi

On the Cray XT4/XT5 prototype, Louhi, we used both the PGI and the GNU compiler. No major differences in performance where noted. The compiler flags that we used for the PGI compiler where:

```
-fast
```

IBM Power6 cluster - Huygens

Gromacs was ported successfully to the Power6 prototype, Huygens. On this platform we encountered a problem with the autoconf. Configuration failed as it was unable to execute a MPI binary and thus failed when checking the size of an integer. A work-around for the configure failure is to allow configure to execute the MPI binary produced on the login node. This can be done as follows:

```
export MP_HOSTFILE=$HOME/myhostfile
export MP_PROCS=1
> $MP_HOSTFILE
for i in `seq $MP_PROCS` ; do hostname >> $MP_HOSTFILE; done
```

The compiler flags that we used for the IBM XL compiler were:

```
-O3 -qstrict -qarch=pwr6 -qtune=pwr6
```

IBM Blue Gene/P - Jugene

Gromacs was ported successfully to the Blue Gene/P prototype, Jugene. The compiler flags were:

```
-O3 -qarch=450d -qtune=450
```

5.10.3 *Optimisation techniques*

No optimisation has been performed on the code.

5.10.4 *Conclusions*

We have successfully ported Gromacs to three prototype systems. No optimisations have been performed, as the chance of making any major breakthrough is very limited. We have focused our resources on improving the scalability of Gromacs as detailed in PRACE report D6.4.

5.11 HELIUM

Xu Guo (EPCC)

The application HELIUM uses time-dependent solutions of the full-dimensional Schrödinger equation to simulate the behaviour of HELIUM atoms. The source code was developed at Queen's University Belfast and has access restrictions.

5.11.1 *Application description*

The HELIUM source code is written in a single Fortran 90 file with 14569 lines. Sparse linear algebra is used in the HELIUM application. It is relative straightforward to port HELIUM on different architectures as no specific libraries or environments are pre-required for compiling and execution on most systems. However, due to the code being only one lengthy source file, it could be difficult when trying to optimise HELIUM via manual code modifications.

5.11.2 *Porting*

Porting HELIUM to the prototypes is relative straightforward as the source code is only one single file. Most compilers can be used for compiling HELIUM, but a few code modifications may be needed to adjust to the compilers. When using the PGI compiler on the Cray XT5 for the cases with large parameters, it may encounter a reallocation limit compiling problem.

It is important to select the proper number of cores for different problem sizes. The number of cores and memory size required for a specific problem size are related to the parameter values in the source code. Not every number of cores can be used for benchmarking the test cases, especially those with large problem size.

The memory limit is usually a porting issue for the large problem size test cases on most of the platforms, as the application will consume a lot of memory. Even with a successful build, the execution may be failed due to the system memory size limitation. For some executions,

half populated or quad populated task allocation on the nodes may help to solve the memory limitation issue, but it may also deliver a poor performance. For the large problem size test cases, it may take around 1~2 pm to port to a given architecture, including finding out the proper number of cores and data local volume, etc.

HELIUM has been ported successfully to Cray XT4/XT5, Power 6 and BlueGene/P. It can scale up to 2485 cores when using the 1540-block test case on the Cray XT5 and Power6. A larger problem size of 3060 blocks was also benchmarked successfully on BG/P.

Cray XT5 - Louhi

On the Cray XT4/XT5 prototype, Louhi, a small HELIUM test case can be compiled directly using either the PGI Fortran90 or PathScale Fortran90 compiler. However, using the PGI compiler will encounter a reallocation limit compiling problem when some parameters in the source code were assigned large values, i.e. the PGI compiler is not suitable for a medium or large HELIUM test case running on the Cray XT5. Therefore all benchmarking on Cray XT5 was done with the PathScale Fortran90 compiler.

Using the PathScale Fortran90 compiler for the HELIUM build on the Cray XT5 (Louhi):

- The original flags used for porting was: `-O3`.
- The final optimal flags was: `-O4 -OPT:Ofast:unroll_analysis=ON -LNO: fusion=2:full_unroll_size=2000:simd=2 -LIST:all_options=ON.`

HELIUM was ported to a Cray XT4 with dual-core processors before the PRACE project and in the PRACE WP6, it was the first time to port HELIUM to a Cray XT4 with quad-core processors and a Cray XT5, which was quite similar with the porting to the dual-core Cray XT4 system.

Some modifications of the code were necessary to adjust to the PathScale compiler. The developer has already merged most of the modifications to the latest version code.

The memory limitation was a common issue for the HELIUM porting. Not all the test cases with different problem sizes could run with a specific number of cores on Cray XT5. With a proper number of cores used, the executions of HELIUM test cases will be successful on fully populated nodes. To satisfy the memory requirement, half-populated or quad-populated might be helpful to run successfully but usually the performance will be quite low. Usually when the memory limitation is reached, the execution will freeze up and fail.

IBM Power6 cluster - Huygens

HELIUM was ported successfully to the Power6 prototype, Huygens. To compile the Fortran 90 source code on Huygens, the flag `-qfree=f90` must be used for the `mpfort`, or the build will fail. By default, the `mpfort` will invoke the IBM FORTRAN MPI compiler.

Using the IBM FORTRAN MPI compiler for a HELIUM build on the Power6 (Huygens):

- The original flags used for porting were: `-qfree=f90 –O3`.
- The final optimal flags were: `-qfree=f90 -O4 -qessl -qarch=auto -qtune=auto -qhot`

Not every number of cores can be selected for benchmarking. There are big possibilities of not fully allocating tasks on each node, therefore it is very important to define the total number of tasks and the total number of nodes clearly in the job script, or the execution will be unsuccessful. For example:

```
#@ total_tasks = 630
#@ node = 20
```

Which means that there was a total of 630 MPI tasks allocated on 20 nodes. 19 of the nodes were fully populated, i.e. 32 tasks, and 22 tasks were assigned to the remaining node.

It was the first time to port HELIUM to a Power6 system with Linux OS, but the code was ported to a Power5 system with AIX OS before. The basic portings (without optimisation) were quite similar.

Some minor code modifications were necessary to adjust to the compiler. The developer has already merged most of the modifications to the latest version code.

IBM Blue Gene/P - Jugene

The porting of HELIUM to the BG/P prototype, Jugene, was successful. The source code can be compiled using the IBM XL MPI Fortran compiler with `-qlanglvl=extended` specified.

- The final optimal flags were: `-O4 -qarch=450d -qtune=450 -qlanglvl=extended -qfree=f90 -qrealsize=8 -qsuffix=f=f90 -qessl`

All the timing runs were undertaken in virtual node (VN) mode. This mode utilises all four cores for computation on a Blugene/P node. Alternative modes are DUAL mode (2 tasks per node) and SMP mode (1 task per node). Tasks in jobs run in DUAL or SMP mode have access to greater, or even exclusive, shares of the memory hierarchy per node. However it is usually the case that more effective use is made of the overall resources by using all available cores in the nodes (i.e. VN mode) and this is the mode used in the tests.

The Loadleveler batch jobs require the setting below:

```
#@ stack_limit = 200MB
```

5.11.3 *Optimisation techniques*

Based on the profiling results, some routines in the HELIUM code with large calculation loops were very expensive. Also, when scaling to larger numbers of cores, the MPI communications will have an increasing impact on the code performance. Therefore, optimising HELIUM was started by selecting sets of the compiler flags to adjust to the prototype architectures and improve the CPU, memory and cache usage for the loop calculation performance. Some reduction of the MPI cost was tried as well to improve the scaling performance.

In task 6.5, HELIUM was optimised on performance for the Power6 (Huygens @ SARA) and Cray XT5 (Louhi @ CSC) using the problem size 1540 blocks on 630 cores and 1540 cores. On BG/P (Jugene), a 176-block test case was optimised with 66 cores running. It took around 6 pm to implement and benchmark all the optimisations. Further manual coding optimisations could be done, e.g. hybridize the MPI code as mixed mode code, which will take another 4 pms at least.

The profiling toolkits were used during the optimisations. On Cray XT5, Craypat was used. On the Power6 system, both the IBM HPC toolkit and Scalasca were used.

On Cray XT5 (Louhi@CSC):

- Compiler flags were selected to improve HELIUM performance: `-O4 -OPT:Ofast:unroll_analysis=ON`

```
-LNO:                    fusion=2:full_unroll_size=2000:simd=2
-LIST:all_options=ON.
```

- The loop optimisation techniques used included loop fusion, loop fission, loop unrolling, loop vectorisation for SIMD, and array padding.
- Redundant MPI communications were removed.
- Some loops were merged to reduce the total iteration times.

On Power6 (Huygens@SARA):

- Compiler flags were selected to improve HELIUM performance:

```
-qfree=f90 -O4 -qessl -qarch=auto -qtune=auto -qhot
```

- Redundant MPI communications were removed.
- Some loops were merged to reduce the total iteration times.
- Temporary variables were introduced to reduce floating point operations.

On BG/P (Jugene@FTZ):

- Compiler flags were selected to improve HELIUM performance:

```
-O4 -qarch=450d -qtune=450 -qlanglvl=extended -qfree=f90
-qrealsize=8 -qsuffix=f=f90 –qessl
```

Technique 1: Compiling Optimisation on Cray XT5 (Louhi@CSC)

HELIUM performance improvement on Cray XT5, Louhi, was started with compiling optimisation. Using the compiler options for optimisation can be quite effective without too much manual code modifications.

Using higher optimisation level compiler options:

The original porting just used the common performance tuning compiler flag `–O3`. The higher level optimisations were tested for the HELIUM performance; `–O4` delivered better performance while `–O5` gave worse performance.

`-OPT:Ofast` is a commonly used performance tuning option for the PathScale compiler on the Cray XT5. This option resulted in performance improvement.

Loop optimisations:

The user routine profiling shows that the most expensive calculation routines all have large calculation loops. The Pathscale compiler has a specific nested loop optimiser flag: `-LNO:<suboptions>`, which is able to apply multiple loop optimisation techniques by using different suboptions.

Loop fusion may change the temporal locality, while loop fission, the inverse of loop fusion, can reduce conflict/capacity misses. Only one of these two techniques should be selected at the same time. Both loop fusion and loop fission can be applied via the compiler flag suboptions, with different optimisation levels, such as:

```
-LNO:fusion=1:fission=0;
-LNO:fusion=2:fission=0;
-LNO:fission=1:fusion=0;
-LNO:fission=1:fusion=0.
```

The best performance was given by `-LNO:fusion=2:fission=0` and the test run passed the correctness check.

Loop unrolling is a technique for increasing the size of the loop body, which may give more scope for better schedules and may reduce branch frequency. Using the compiler to do the loop unrolling optimisation can keep the code readable as well as improve performance effectively. The loop unrolling options used were:

`-OPT:unroll_analysis=ON -LNO:full_unroll_size=2000`,

To turn on the unroll analysis and set the fully unroll loop size to 2000 (the default value). The unrolling optimisation level and unrolled loop size may affect the optimisation effect. This should be controlled carefully when the Pathscale compiler options are used.

Loop vectorisation and making use of the SIMD instructions can be helpful to improve the calculation performance. On the Cray XT5 (Louhi), the loop vectorisation for SIMD can be set on different levels:

```
–LNO:simd=1;
-LNO:simd=2.
```

The best performance was given by `-LNO:simd=2` and the test run passed the correctness check.

Array padding is an optimisation technique to reduce conflict cache misses. From the HW counter profiling results on Cray XT5 (Louhi), it can be seen that plenty of cache misses occurred on level 1 and level 2 caches during the code execution which could be caused by different reasons. The option `–OPT:pad_common=ON` was tested but the optimised result with array padding option was not as good as expected. The possible reason is that "conflict" was not the main reason for the cache misses.

Table 28 shows the execution time comparison between the performance of the original porting, the performance of higher optimisation level flags used and the performance of loop optimisation options used.

Cores	-O3 (s)	-O4 –OPT:Ofast(s)	-O4 -OPT:Ofast:unroll_analysis=ON -LNO:full_unroll_size=2000 -LIST:all_options=ON (s)
630	936	731	729
1540	338	316	312

Table 28: HELIUM performance comparison using compiling options

Technique 2: Compiling Optimisation on Power6 (Huygens@SARA)

HELIUM performance improvement on Huygens was started with compiling optimisation, which was quite effective without too much manual code modifications.

Using higher optimisation level compiler options:

The original porting just used the common performance tuning compiler flag `–O3`. The higher level optimisations were tested for the HELIUM performance; `–O4` delivered better performance while `–O5` gave worse performance.

Fit for the architecture:

Using the architecture tuning compiling option will be helpful to fit the code to the given architecture. Options used were `–qarch=auto –qtune=auto` to detect the architecture automatically. If not set, the default compiling will fit to the Power3 architecture.

High-order transformations:

The compiler option `-qhot` performs a full high-order transformation during optimisation, where possible, including the auto-arraypad.

Link with essl library:

Enabling the Engineering and Scientific Subroutine Library (ESSL) routines to be used in place of some Fortran 90 intrinsic procedures when there is a safe opportunity to do so. This improved the HELIUM performance effectively.

Table 29 shows the comparison between the original porting HELIUM performance and the compiling optimised code performance.

Cores	`-qfree=f90 -O3` (s)	`-qfree=f90 -O4 -qessl -qarch=auto -qtune=auto -qhot` (s)
630	726	714
1540	324	319

Table 29: Using compiling optimisation for HELIUM on Power6(Huygens)

Technique 3: Compiling Optimisation on BG/P (Jugene@FZJ)

HELIUM takes advantage of highly optimised numerical library routines from IBM's Engineering Scientific Subroutine Library (ESSL). Faster runs are generated when specifying `-qessl` in the compiler and linker options than when specifying the link to ESSL by hand (-L/opt/ibmmath/essl/4.4/lib –lesslbg).

On Jugene the environment variable `DCMF_INTERRUPT` can be set in Loadlever scripts in order to enable the overlapping of communication and computation. The code was timed with and without the setting and no significant difference to run-time was observed.

The BlueGene/P processor has 2 floating point units, known as the double hummer. The XL compilers have options to create floating-point code to take advantage of these special units. The default setting for the Jugene compiler is double hummer optimisation on (`-qarch=450d`). This can be turned off by setting `-qarch=450` during compilation. Both options are tested in the table above. The setting makes little difference to run times at the `-O3` level of optimisation, but double hummer on (default & `-qarch=450d`) produces a significant speed-up when compiled at `-O4`.

The highest level of optimisation, `-O5` produced slower code on Jugene than `-O4`, though the physical results appeared correct. The program developers strongly advise against using `-O5` in their program notes.

Table 30 shows the compiling optimisation results step by step on Jugene.

Run No.	Optimisation Flags	Comments	Time (s)
1	No optimisation	Default Optimisation: None. Exceeded Time limit for job class	1800+
2	`-O1`	Exceeded Time limit for job class	1800+
3	`-O2`	equivalent to `-O`	833.42
4	`-O3`	Aggressive optimisation, allows re-association, replaces division by multiplication with the inverse	803.44
5	`-O3 -qtune=450 -qarch=450`	generates standard powerpc floating-point code, uses a single FPU	796.98
6	`-O3 -qtune=450 -qarch=450d`	Attempts to generate double FPU code	803.44
7	`-O4`	`-O4 = -O3 –qhot –qipa=level=1`	590.87
8	`-O4 -qtune=450 -qarch=450`		762.94
9	`-O4 -qtune=450 -qarch=450d`		591.07

Table 30: Compiling flag optimisations on BG/P (Jugene).

Where in the flag option `–O4` and `–O5`:

- `-qhot`: invokes high-order transformation module and adds vector routines unless `–qhot=novector` specified
- `-qipa`: inter-procedural analysis (IPA). The level specified determines the amount of ipa performed:

  Level 0: Performs only minimal interprocedural analysis and optimisation.

  Level 1: Turns on inlining, limited alias analysis, and limited call-site tailoring.

  Level 2: Full interprocedural data flow and alias analysis.

Technique 4: Removing redundant MPI communications on Cray XT5 and Power6

The MPI communications have more and more impact on the code performance when scaling to large numbers of cores used. However, some of the MPI communications were not necessary which were removed to reduce the communication cost.

The routine `Test_MPI` is called with every code execution, which was only to test whether the MPI works well on the given environment. This is only for the development debugging and not necessary for a real HELIUM application run. In the routine `Test_MPI`, `MPI_barrier`, `MPI_Allreduce` and `MPI_sendrecv` were called multiple times. The call for `Test_MPI` at the code initialisation stage is therefore removed to reduce the communication cost.

Table 31 shows the performance comparison on Louhi with and without the `Test_MPI` routine. Table 32 shows the performance comparison on Huygens with and without the `Test_MPI` routine. All the tests used the compiling optimised flags.

Cores	Original (s)	Without **`Test_MPI`** (s)
630	729	729
1540	312	311

Table 31: HELIUM performance comparison with and without the `Test_MPI` on Cray Xt5 (Louhi)

Cores	Original (s)	Without **`Test_MPI`** (s)
630	714	712
1540	319	309

Table 32: HELIUM performance comparison with and without the Test_MPI on Power6 (Huygens)

Technique 5: Merging loops on Cray XT5

Merging loops having the same boundaries can reduce the total number of loop iterations. In the HELIUM code, some of the large loops with the same boundaries were separated. Some of the loops did not go though the loop index in order, which could cause more cache misses. Therefore, some of the loops were merged together based on the profiling results, including the `Incr_with_1st_Deriv_op_in_R1`, `Incr_with_1st_Deriv_op_in_R2`, `Incr_with_2nd_Deriv_in_R1` and `Incr_with_2nd_Deriv_in_R2`.

As the results of this optimisation, HELIUM performance on Louhi was improved based on the no Test_MPI version code and the cache misses were reduced.

Cores	Without **`Test_MPI`**(s)	**Merging loops (s)**
630	729	723
1540	311	302

Table 33: Merging loops on Louhi reduce the execution time.

Cores	Without **`Test_MPI`** **(total cache misses)**		**Merging loops (total cache misses)**	
	L1 cache misses	L2 cache misses	L1 cache misses	L2 cache misses
630	6616603138	2967997049	2300447361	705369051
1540	169407136	163015818	150040558	90228266

Table 34: The total cache misses on Louhi before and after merging loops

Technique 6: Merging loops and using temporary variables on Power6

On Huygens, the merging loops optimisation was implemented in the same way as on Louhi and also improved HELIUM performance.

Also, on Huygens, based on the profiling results, some temporary variables were introduced for representing intermediate results during long calculations to reduce the floating point operations. This was not effective on Louhi but worked well on Huygens.

Cores	Without **`Test_MPI`**(s)	**Merging loops and using temporary variables(s)**
630	712	670
1540	309	298

Table 35: Merging loops and using temporary viables on Huygens reduce the execution time.

Cores	Without **`Test_MPI`** **(total cache misses)**		**Merging loops** **(total cache misses)**	
	L1 cache misses	L2 cache misses	L1 cache misses	L2 cache misses
630	4.36E+13	9.57E+11	4.55E+13	9.38E+11
1540	1.18E+14	8.44E+11	1.13E+14	8.43E+11

Table 36: The total cache misses on Huygens before and after merging loops and using temporary variables

5.11.4 *Results of optimisation effort*

In general, the optimisations discussed above improved HELIUM performance on the Cray XT5 (Louhi@CSC), Power6 (Huygens@SARA) and BG/P (Jugene@FZJ).

The compiler options optimisation was quite effective without changing the HELIUM code manually. The specific flags on Louhi for the nested loops optimisation have more options and therefore are easier to control to improve the large loop calculations in the original HELIUM. The Power6 and BG/P have a better architecture tuning option and default compiler optimisation effect (e.g. with the default `-qhot` used).

One of HELIUM's performance bottlenecks when scaling to large numbers of cores is the high communication expense. Although not too much can be done to improve it, removing the MPI communication `Test_MPI` was helpful to reduce the unnecessary communication costs on the Cray XT5 and Power6 prototypes.

From the manual code modification aspect, although merging loops and introducing temporary variables may reduce the readability of the original code, it can also help with a better memory cache usage, i.e. to reduce the cache misses. However, this may depend on the system architecture. An improvement was achieved on the Cray XT5 and Huygens but cause not significantly on BG/P.

Figure 5 below shows the performance comparison between the original code porting and the optimized code on the Cray XT5 (Louhi) and Figure 6 is the cost figure. Note, the cost is execution time multiplied by the number of cores, i.e. a horizontal curve means a linear scaling. Figure 7 and Figure 8 are the plots based on Power6 (Huygens).

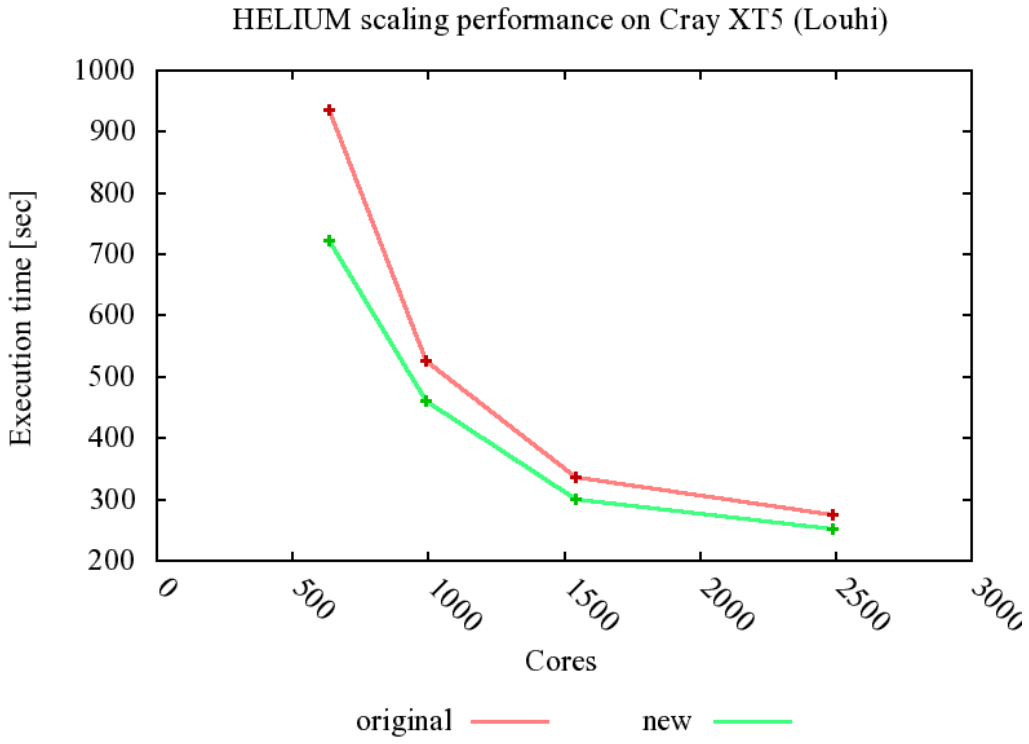

Figure 5: HELIUM scaling performance on Cray XT5 (Louhi)

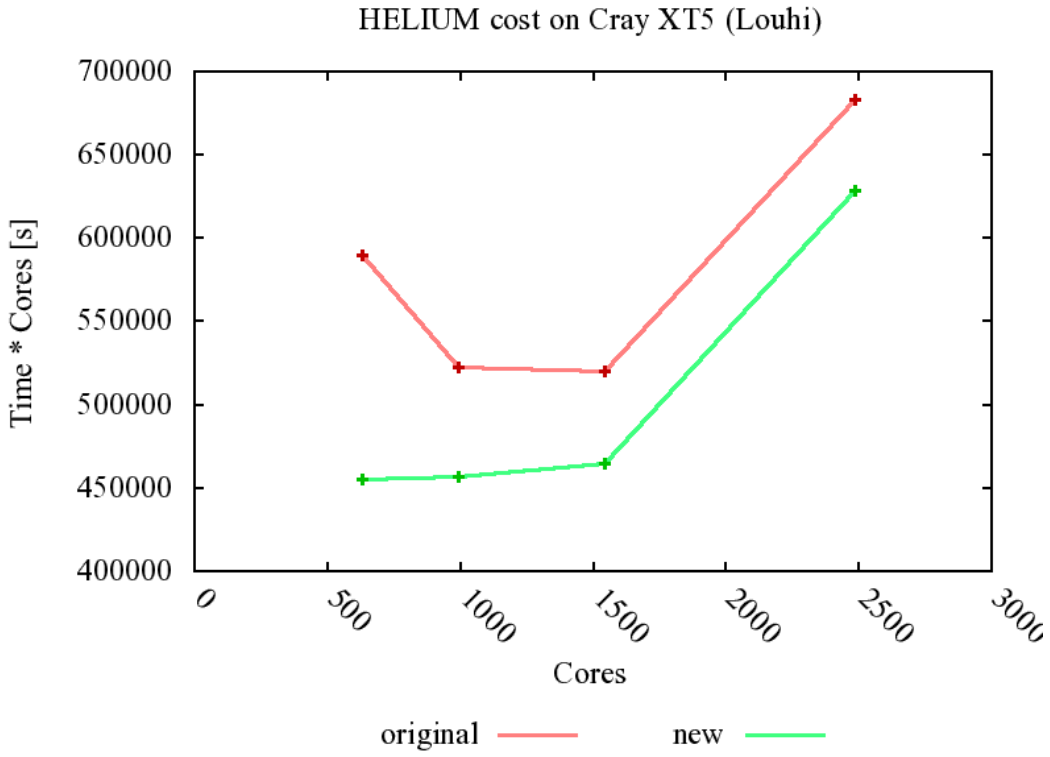

Figure 6: HELIUM cost on Cray XT5 (Louhi)

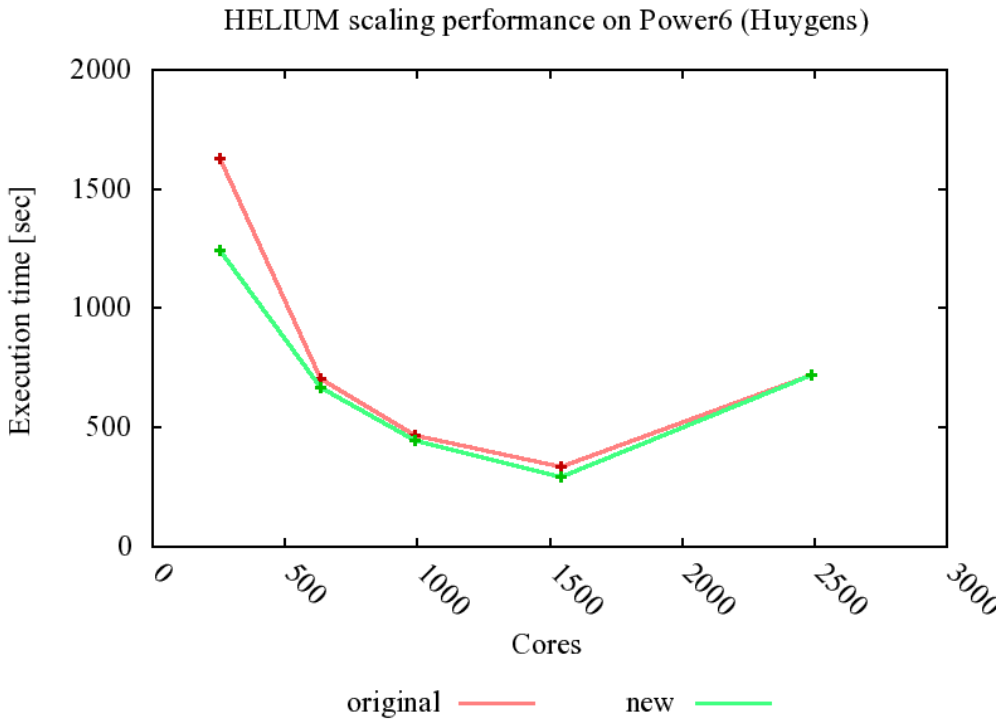

Figure 7: HELIUM scaling performance on Power6 (Huygens)

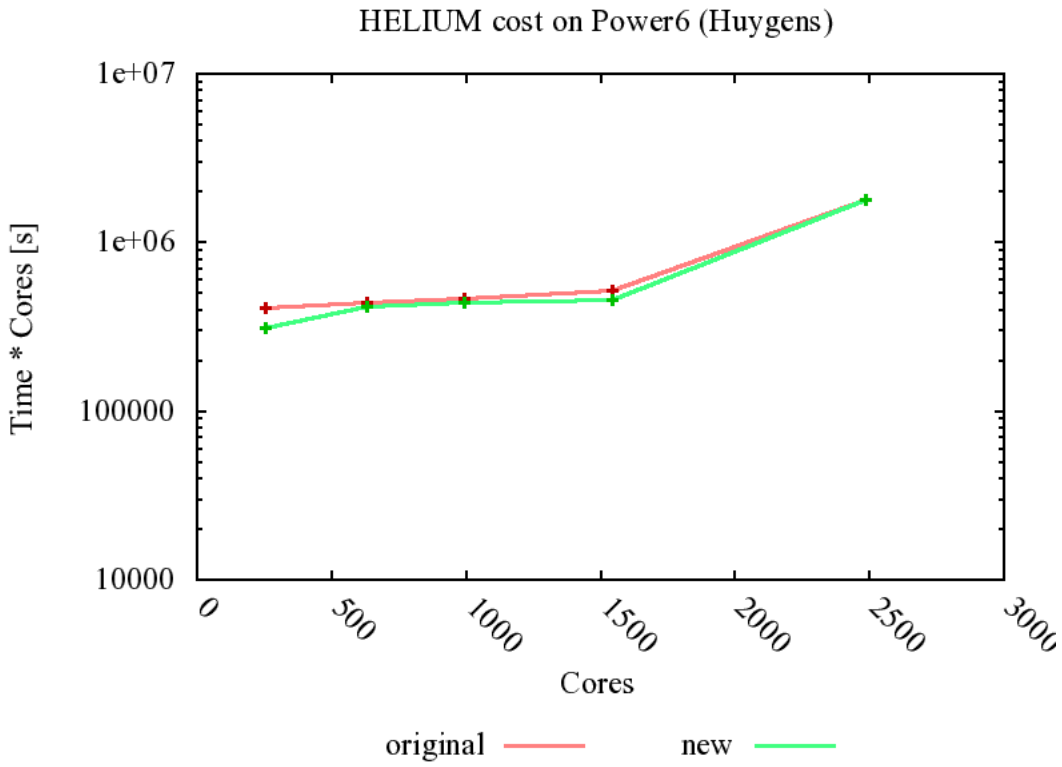

Figure 8: HELIUM cost on Power6(Huygens)

5.11.5 *Other optimisation techniques*

A basic mix-mode hybrid optimisation was also implemented on Cray XT5 and Power6. However, the results were not as good as expected. Due to the time limit, there was not enough time for further investigations. However, further profiling and implementation of the

hybrid optimisation is quite interesting, which may be helpful for future performance improvements.

5.11.6 *Conclusions*

Most of the optimisations were effective, based on the profiling results analysis. Performance profiling is very helpful to find out the performance bottlenecks. It should be noticed that the profiling results could be different with changing problem size and number of cores used. Therefore profiling based on a large enough problem size and number of cores could be more helpful for this PRACE task. Also, the profiling results may vary depending on the system situation, so averaging over multiple runs is necessary to have reliable results.

Compiler optimisation is useful as a good starting point. It can improve performance well while keeping the code correctness without too much manual code modifications. However, the compiler optimisation depends a lot on the compilers and system environments. The compiling optimisations specific for the calculation loops on Cray XT5 (Louhi) were very useful and effectively. This may be used for other application optimisations as well.

Correctness check is important. No matter what optimisation is used, the code correctness should be the first thing to be ensured.

Further profiling and manual code modifications might be helpful for future optimisations, e.g. the hybrid mix-mode programming for large calculation loops. However, this may require a lot more investigation as the basic hybrid tests did not gain the performance improvement expected.

5.12 NAMD

Joachim Hein (EPCC)

Martin Polak (GUP)

Paschalis Korosoglu (GRNET)

NAMD is a widely used molecular dynamics application designed to simulate bio-molecular systems on a wide variety of compute platforms[1][2]. NAMD is developed by the “Theoretical and Computational Biophysics Group” at the University of Illinois at Urbana Champaign. In the design of NAMD particular emphasis has been placed on scalability when utilising a large number of processors. The application can read a wide variety of different file formats for e.g. force fields, protein structure etc., which are commonly used in bio-molecular science.

5.12.1 *Application description*

When the Prace project started, NAMD version 2.6 was the most up-to-date release. In March 2009 a beta release of NAMD 2.7, commonly called 2.7b1, became publically available. Early work in Prace was based on NAMD 2.6. Since NAMD 2.7b1 shows significantly more scope for petascaling, as shown in PRACE report D6.4, we have now switched the work to the new version 2.7b1.

The application source is written in C++ using Charm++ parallel objects[3] for the data exchange between the compute tasks. The actual NAMD source consists of 157 *.C files and 196 *.h files. These files contain a total of about 120000 lines of code.

The required Charm++ can be built on a wide variety of communication protocols. Charm++ is typically not installed on a computing platform, hence building Charm++ is typically the first step when installing NAMD. The source of Charm++ is distributed with the NAMD source. In case of NAMD 2.7b1 Charm++ version 6.1 is included. For this investigation we have build Charm++ on top of the MPI library provided on the prototype architectures. Building a production version of NAMD requires the following libraries

- TCL
- Single precision version of FFTW 2.1.5
- Charm++

NAMD uses a cut-off distance, which is specified in the input files. The forces between atoms separated by less than the cut-off distance are calculated directly in positions space. For atoms separated by more than this distance the long range electro-static forces are calculated in Fourier space using the particle mesh Ewald (PME) method.

For its parallelisation NAMD uses a spatial decomposition. The simulation volume is divided into orthorhombic regions called patches[1]. The diameter of these patches is larger than the cut-off distance. Hence for the calculation of the direct forces, knowledge of atom positions on the home patch and the 26 neighbouring patches is all that is required.

For the time integration NAMD uses a velocity Verlet algorithm in case of a constant energy simulation while a Brünger-Brooks-Karplus method is used for a canonical ensemble of constant temperature. In case of constant pressure and temperature a combination of Langevin-piston and Hoover methods is used.

NAMD automatically adjusts the load balance during the first part of the simulation. The computational load is measured for each patch and patches are moved between the processors to balance the load. Most of the code required for the load balancing features is part of Charm++ instead of the actual NAMD source. The load balancing takes the first 300 time steps of simulation. The performance improvement due to the dynamical loadbalancing is largest when using a very large number of processors. The slower initial steps need to be taken into account when estimating the performance of a large production run from a test simulation lasting only a few hundred steps.

5.12.2 *Testcases*

For the performance investigation of this study we have obtained input data sets containing TCR-pMHC-CD complexes in a membrane environment[4]. These systems are relevant for the investigation of immune response reactions triggered by transient calcium signalling.

The basic data set contains four complexes and has a total size of about 1 million atoms. Larger input data sets of two and nine million atoms have been obtained by placing two or nine copies of the original system in a single simulation volume. The configurations use a step size of 2 fs.

5.12.3 *Porting*

For a large number of platforms, binary executables can be downloaded from the NAMD website. However these will only work if everything required by the executables is installed in the standard place. If these executables do not work on a given system or no binary is supplied for your architecture you have to build NAMD from source. To support this, the NAMD source distribution contains a number of architecture specific files for NAMD and

Charm++ to tailor the make procedure to specific architectures. If your architecture is supported in this way and the required FFTW and TCL libraries are in place, building Charm++ and NAMD can be accomplished in a few hours.

The Charm++ distribution comes with its own test suite. We strongly recommend using these tests to ensure the Charm++ has been build correctly before starting with the actual NAMD executable.

Cray XT5 - Louhi

For NAMD 2.7b1 this prototype is very well supported by the developers of NAMD and Charm++. The NAMD 2.7b1 distribution contains the complete source for Charm++ 6.1. To build the latter we used the architecture description file `mpi-crayxt,` which is part of the Charm++ 6.1 distribution. This file contains build options for the gnu compiler suite. If one were to try either of the PGI or Pathscale compilers on the prototype separate architecture description files would need to be developed.

The FFTW 2.1.5 on the prototype has been build for the PGI compiler. To avoid any potential conflict, we recompile this from source using the gnu compiler. Since NAMD requires only the single precision version and uses the C bindings, a simplified build of FFTW is all that is needed.

The TCL scripting library is also not provided on the prototype. We build tcl version 8.4.19 using the gnu compiler.

Once this is in place, NAMD can be built using the CRAY-XT-g++ architecture files of the distribution. Building NAMD 2.6 was not as easy, since the Cray XT5 prototype was not available when this version of NAMD got released.

When running NAMD on this prototype one frequently encounters failing runs due to the MPI-library running out of resources. This typically happens during the load balancing step, when all processors need to share their performance data with rank 0. In this situation rank 0 quite often runs out of space to buffer the large number of unexpected message. In our experience setting the environment variables

```
export MPICH_UNEX_BUFFER_SIZE=100M
export MPICH_PTL_SEND_CREDITS=-1
```

inside the job submission script overcomes these problems. If the library still runs out of buffer space, one typically receives a clear error message explaining the problem and suggesting the relevant environment settings, which need modifying.

IBM Power6 cluster - Huygens

For this architecture there is no obvious architecture file supplied with either Charm++ 6.1 or NAMD 2.7b1, which builds successfully. To build Charm++ 6.1 we started with the architecture file developed by the SARA support staff to build Charm++ 5.9, which had been given to us earlier in the PRACE project. To build the newer version 6.1, a line

```
#define CMK_64BIT 1
```

needs adding to the `conv-mach.h` file. With this modification we managed to build a Charm++ 6.1 library, which passes the tests.

We used the TCL 8.4 and FFTW 2.1.5 library installed on the prototype.

To build NAMD 2.7b1 we could use the compiler options and pre-processing flags from the `Linux-PPC-MX64-xlc64.arch` file provided with the distribution.

When running on this prototype we observe best performance when we use the SMT feature provided by the hardware. This will be discussed below in more detail.

In its present form this prototype suffers from large performance variations between runs. These are under investigation by the service provider and the vendor of the system. Here we report on the best results we have obtained so far, as an indication of the capabilities of the hardware once the problems are overcome.

IBM Blue Gene/P - Jugene

NAMD 2.6 had already been ported and installed to the IBM Blue Gene/P prototype by FZ-Juelich and the compile steps could easily be reproduced using the provided loadleveler scripts for patching config files of TCL 8.3.3 (where a special port for IBM Blue Gene/P exists), FFTW, Charm++ 6.0 and NAMD 2.6 and running the build processes on the compute nodes. BG/P is an officially supported platform starting from this specific versions. During the tests with the memory optimized version NAMD 2.7b1 it turned out to be easier / more efficient to use the system installed version FFTW 2.1.5 since this is site maintained and already compiled for optimal performance. We build the required CHARM++6.1 and the actual NAMD on the front-end nodes, using the same switches and patches as done for the previous versions. This step is even necessary since the compute nodes are quite less performing and therefore compilation would exceed the maximum wall time for small partitions. The patching needed on IBM Blue Gene/P mainly involves setting correct paths of the libraries needed and using the IBM xlC compiler with their optimum settings provided by FZJ. Similar to our experience on the Cray XT5 prototype, we need to increase the buffer space assigned to the MPI library at run time. Adding `-env DCMF_RECFIFO=64554432` to the arguments of `mpirun` increases the buffer space to 64 MB, which was typically sufficient.

5.12.4 *Optimisation*

Performance of version 2.7b1

When the new NAMD version 2.7b1 got released, it is an obvious question to ask, how does it perform in comparison to the previous version 2.6. For this assessment we used an input configuration with 1 Million atoms as used in research for immune reactions[4].

The new NAMD 2.7b1 offers to compile the application with a reduced memory footprint. This feature is extremely useful when using a system which offers limited memory per processing core (e.g. IBM Blue Gene/P) or if extremely large benchmarks with several million atoms need studying, see PRACE report D6.4. It is interesting to see whether this feature also affects performance.

In Figure 9 and Figure 10 we compare the performance of NAMD 2.6 and 2.7b1 with and without reduced memory footprint on the prototypes Cray XT5 and IBM Power6 cluster. This is done for the 1 million atom benchmark system. Figure 10 also shows the effect of SMT, which is discussed in the next section. The figures shows a substantially improved performance of version 2.7b1 over version 2.6. Using an executable with reduced memory footprint leads to a small additional performance improvement.

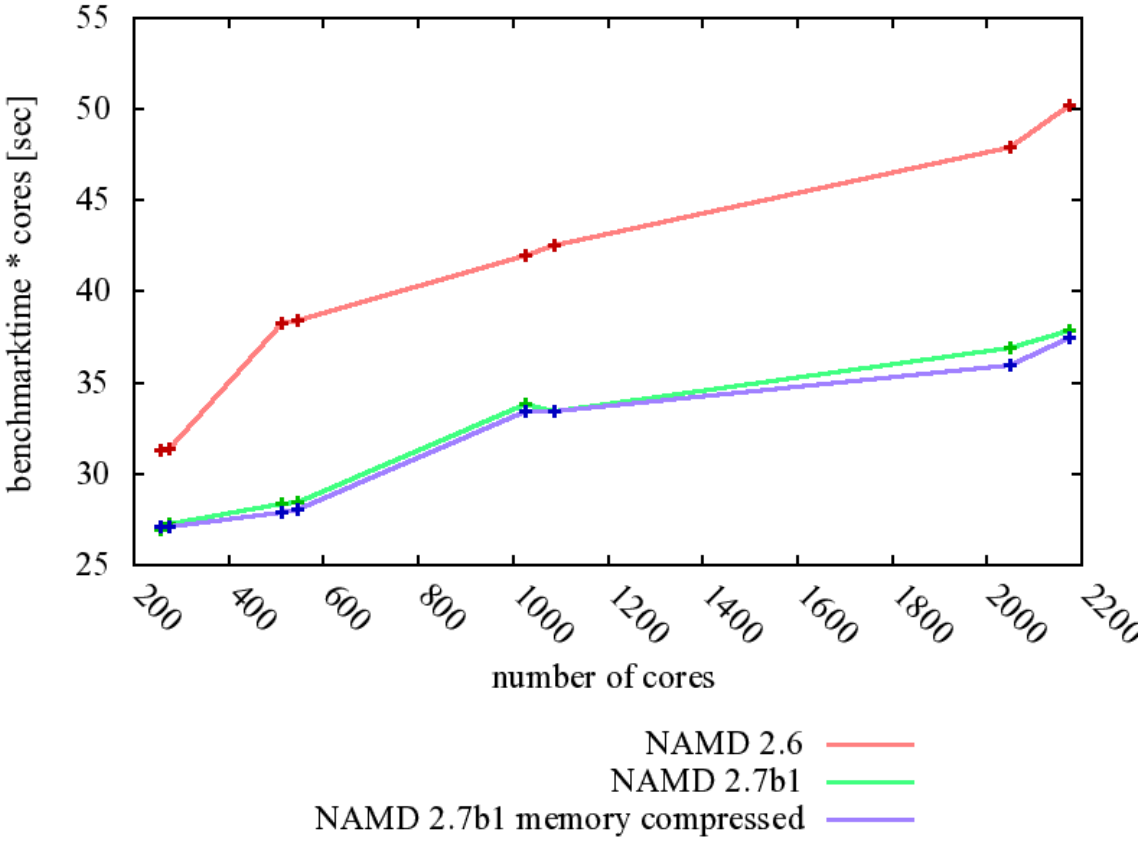

Figure 9 Performance of different versions of NAMD on the Cray XT5 prototype. On the vertical axis, we plot the NAMD benchmark time multiplied by the number of physical processors used for simulation, which is a measure for the total computational cost of a single NAMD step.

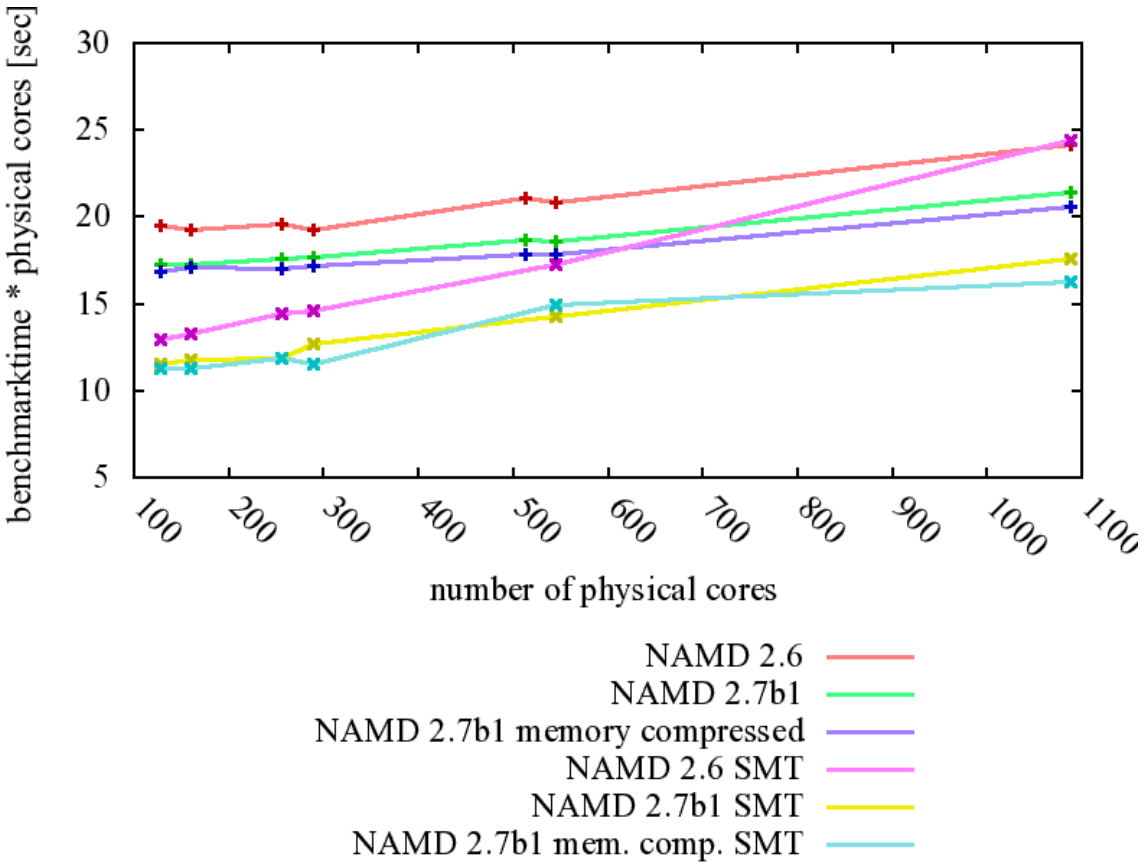

Figure 10 Performance of different versions of NAMD on the IBM Power6 cluster prototype. The figure also shows the effect of using SMT. On the vertical axis, we plot the NAMD benchmark time multiplied by the number of physical processors used for simulation, which is a measure for the total computational cost of a single NAMD step.

Simultaneous Multi Threading

The hardware of the IBM Power6 cluster prototype offers Simultaneous Multi Threading (SMT). When using SMT, in the case of NAMD two computational tasks are placed on a single physical compute core. Using SMT is advantageous if a single compute task cannot fully exploit all the functional units of the compute core, leaving resources idle, which could be exploited by another task.

The above figure on the performance results of the IBM Power6 cluster system also contains the results when using SMT. Results are shown against the physical processors used. E.g. for a calculation using 128 physical processor 256 tasks get placed on them.

One observes a substantial performance improvement from SMT for all three versions of NAMD. The ordering of the different versions is not affected by SMT. NAMD 2.7b1 with a reduced memory footprint shows the best performance. It is interesting to note that when using SMT, the new NAMD 2.7b1 shows better scalability over NAMD 2.6.

Compiler optimisation

On the Cray XT5 prototype an investigations of the effect of the compiler on the performance was carried out. For this investigation we used version 4.3.3 of the gcc compiler. The performance has been measured for using 2.3 GHz Opteron “Barcelona” and 2.7 GHz Opteron “Shanghai” processors. The architecture specific files for the Cray XT supplied with the NAMD 2.7b1 source suggest the following compiler options, which we call “default”:

```
-O3 -ffast-math -static -fexpensive-optimizations -fomit-frame-pointer
```

It is important to node that for gcc 4.3.3 the option `-fexpensive-optimizations` is included at level `–O2` or higher and is therefore redundant. The following table lists our experiments and lists the best observed "Benchmark time" for a number of re-trials. This was investigated for NAMD 2.7b1 when using a reduced memory footprint. We used the 2 million atom benchmark on 256 compute tasks:

No	Compiler options	2.3GHz processor	2.7GHz processor
1	`-O2`	0.204 s	0.165 s
2	`-O3`	0.203 s	0.163 s
3	`-O3 –funroll-loops`	0.196 s	0.157 s
4	`-O3 -ffast-math -static -fexpensive-optimizations -fomit-frame-pointer`	0.203 s	0.160 s
5	`-O3 -ffast-math -static -fexpensive-optimizations -fomit-frame-pointer –funroll-loops`	0.203 s	0.160 s
6	`-O3 -funroll-loops -ffast-math`	0.202 s	0.160 s
7	`-O3 -static -fexpensive-optimizations -fomit-frame-pointer -funroll-loops`	0.196 s	0.158 s

Table 37: NAMD performance with different compiler flags on Cray XT5

For both processor types we obtain a similar picture. The performance shows only a small dependency on the comiler options. The default option No 5 delivers already a good performance. We notice that `–funroll-loops` gives a slight performance advantage while the option `–ffast-math` hinders performance. The options "`-O3 –funroll-loops`" give the best performance and a slight improvement over the default No 5.

We have performed a similar investigation on the IBM Power6 cluster prototype. For this investigation we have used the IBM XL C/C++ compiler family version 10.1. With respect to this prototype we always used the following set of compiler options, which we call "base options":

```
-q64 -arch=auto -qtune=auto -qcache=auto
```

The following table lists our experiments and lists the best observed "Benchmark time". This was investigated for NAMD 2.7b1 when using the reduced memory footprint mechanism. We used the 2 million atom benchmark on 256 processors without and with the SMT (Simultaneous Multi-Threading) feature. The column "Additional Compiler Options" list the options used in addition to the above base options. One point to mention is that usage of `–O3` without the `–qnohot` flag breaks the code.

No	Additional Compiler Options	Without SMT (256 tasks)	With SMT (512 tasks)
1	`-O2`	0.161 s	0.105 s
2	`-O2 -qfloat=relax`	0.160 s	0.104 s
3	`-O2 -qunroll`	0.151 s	0.099 s
4	`-O2 –qunroll -qfloat=relax`	0.151 s	0.100 s
5	`-O3 -qnohot`	0.126 s	0.086 s
6	`-O3 –qnohot -qfloat=norngchk`	0.125 s	0.085 s
7	`-O3 –qnohot -qinline`	0.126 s	0.085 s
8	`-O3 –qnohot -qunroll`	0.124 s	0.083 s
9	`-O3 –qnohot -qunroll -qfloat=relax`	0.123 s	0.084 s
10	`-O3 –qnohot –qunroll -qfloat=relax:norngchk`	0.123 s	0.085 s

Table 38: NAMD performance with different compiler flags on IBM Power6 cluster - Huygens

In both cases (with and without SMT) the results are similar. We see that the usage of the `–O3` compiler option boosts the performance of the code. In addition usage of the `–qunroll` switch improves the overall performance. Without the SMT feature we have optimal results with option No 10 whereas with SMT the option No 8 gives the best results.

Build options for IBM Blue Gene/P

For the prototype IBM Blue Gene/P version 6.1 of charm++, which is part of the NAMD 2.7b1 distribution, offers two sets of architecture files. One set is named `mpi-bluegenep` and the other one just `bluegenep`. While the `mpi-bluegenep` file set links against the mpi library on the system, the `bluegenep` file set does not. Our understanding is that the `bluegenep` file set bypasses the MPI layer and links directly to the communications layer underneath the MPI library[7].

The actual NAMD 2.7b1 source has a `BlueGeneP-MPI-xlC.arch` architecture description file to go with the charm++ `mpi-bluegenep` file set and a `BlueGeneP-xlC.arch` file to match the charm++ `bluegenep` file set.

Using our 1 million atom dataset on 2048 cores, placing four computational tasks on each quadcore node, we studied the benefit bypassing the MPI layer has on the performance. We also investigated the effect on the performance of a number of compiler options in the charm++ and the actual NAMD compilation. The results are summarised in the following table.

No	Task mapping	MPI used	Charm++ compiler options	NAMD compiler options	Time (s)
1	default	no	-O3 -qstrict -qhot -qarch=450d -DCMK_OPTIMIZE=1	-O3 -Q -qhot -qarch=450d -qtune=450 -DFFTW_ENABLE_FLOAT	0.0501
2	TXYZ	no	-O3 -qstrict -qhot -qarch=450d -DCMK_OPTIMIZE=1	-O3 -Q -qhot -qarch=450d -qtune=450 -DFFTW_ENABLE_FLOAT	0.0504
3	default	yes	-O3 -qstrict -qhot -qarch=450d -DCMK_OPTIMIZE=1	-O3 -Q -qhot -qarch=450d -qtune=450 -DFFTW_ENABLE_FLOAT	0.0505
4	default	no	-O3 -qhot -qstrict -qarch=450d -qtune=450 -DCMK_OPTIMIZE	-O3 -Q -qhot -qstrict -qarch=450 -qtune=450 -DFFTW_ENABLE_FLOAT	0.0508
5	default	no	-O3 -qhot -qstrict -qarch=450 -qtune=450 -DCMK_OPTIMIZE	-O3 -Q -qhot -qarch=450 -qtune=450 -DFFTW_ENABLE_FLOAT	0.0532
6	default	no	-O5 -qarch=450 -qtune=450 -qipa=level=2 -DCMK_OPTIMIZE	-O3 -Q -qhot -qarch=450d -qtune=450 -DFFTW_ENABLE_FLOAT	0.0500

Table 39: NAMD performance with different compiler flags on IBM Blue Gene/P

The performance measurements show the obvious, that bypassing the MPI layer slightly outperforms the MPI version, but not by very much. Changing the task mapping or the compiler options does not have a significant effect on the performance either. Trying

higher optimisation (-O4 and greater) flags for the NAMD breaks the compilation during the linking stage. Using higher optimisation levels for charm++ does not affect the performance. The conclusion for this platform is similar to the other platforms. Changing the compiler settings leads only to small changes in the code's performance.

5.12.5 *Conclusions*

The beta release of the forthcoming NAMD 2.7 shows substantially improved performance on the Prace prototype Cray XT5 and IBM Power6 cluster. Using a reduced memory footprint leads to a further performance gain. On the prototype IBM Power6 cluster using SMT improves the performance per physical processor. On the prototypes Cray XT5, IBM Blue Gene/P and IBM Power6 cluster NAMD performance depends only weakly on the compiler options. We have been able to obtain a small performance improvement by changing the options for the Cray XT5 and IBM Power6 cluster platform.

5.13 NEMO

John Donners (SARA)

NEMO (Nucleus for European Modelling of the Ocean) is a state-of-the-art modelling framework for oceanographic research, operational oceanography seasonal forecast and climate studies. It includes ocean dynamics, sea-ice, biogeochemistry and adaptive mesh refinement software.

5.13.1 *Application description*

The NEMO code is completely written in Fortran 90 and consists of many modules. It is well written and structured, with a description of the purpose and development history of every routine. The code only partially describes the arguments and local variables, which is impractical, especially in combination with the (too) concise variable names.

The code is parallelized using pure MPI. It uses a regular 2-dimensional domain decomposition, which is static, so the model needs to be recompiled for every change of the decomposition. ECMWF mentioned that it worked on a version of NEMO that does not require recompilation, but this is still in development. The model can remove domains, which consist only of land-points; a utility gives the minimum nr. of tasks required to run a certain domain decomposition. Because for the PRACE benchmark the number of cores was fixed, the utility was used in reverse, i.e. to determine the domain decomposition that would have the most 'empty' domains.

It uses a separate library for I/O (IO-IPSL), which is built on top of NetCDF. Note that it could also use another file format, called dimg. This library is not developed by the NEMO team and is therefore not considered for modification. However, it is clear from benchmarks that the I/O is a prominent factor above 1000 MPI tasks and I will comment on the possible changes that can be made to the I/O.

NEMO uses finite differences method on a regular, 3-dimensional grid. It has advanced advection-diffusion schemes and can incorporate many tracers in the ocean. The seaice component uses about 10% of the computational load. The surface boundary condition can be calculated using two different algorithms: a preconditioned conjugate gradient (pcg) algorithm or a successive over-relaxation (sor) method. Both methods have a low computation to communication ratio, but different communication patterns. Both methods require a periodic Allreduce call (not necessarily every timestep) to determine the convergence. The pcg algorithm needs two further Allreduce calls every timestep to determine the direction of the next iteration. Its relative computational load (however, mostly due to communication) is about 20% and quickly increases with numbers of tasks.

5.13.2 *Porting*

The application has been run on several different platforms, so porting caused little problems in general. I spent most of my time to get to know the JuBE benchmarking environment and integrate the PRACE setup. The porting to all platforms took 1-2 weeks, the integration process took several weeks.

IBM Power6 cluster - Huygens

The code has not been ported to the Power6 architecture before, but has been ported to other IBM architectures. No source code modifications were needed. One particular issue is that precompiler flags are passed using '-WF,-D' instead of '-D'. NEMO uses the NetCDF format

for its I/O; version 3.6.3 is used on huygens, which is installed as a module. The model is run using fully subscribed nodes, since there is no benefit from undersubscribing nodes.

The NEMO code is by default compiled using the XL Fortran compiler (version 12.1) using -O3. The IBM Power6 compiler has the option -qhot (hot='higher-order transformations') to increase performance through advanced optimisations. However, NEMO becomes unstable (non-converging) when all code is compiled using this option. IBM released an optimized version of NEMO for the Power6 architecture to the developers. Unfortunately, the accompanying IBM report that describes the changes to the NEMO code was not available at the passing of the deadline of this report, so we could only glimpse at the code to look for source modifications. These included the explicit, compiler-specific directive '@PROCESS HOT' in the following routines: dynldf, dynnxt, eosbn2, solsor, traldf_iso, trazdf, zdftke, zdftke_jki. IBM used a different NEMO configuration than is used in PRACE, but because this set of routines takes up more than 26% of the wall clock time in PRACE runs, it is assumed that the optimisations are transferable to the PRACE setup. Unfortunately, there is no noticeable change in performance with these optimisations. IBM also introduced the extra compiler flags '-lmass_64 -lmassvp6_64 -qnoprefetch', without positive (or negative) impact on the performance. Please note that these conclusions are found within the PRACE project, not by IBM.

Jesus Labarta has made a performance analysis of NEMO with the Paraver tool on Power6, which showed a low ipc (=instructions per cycle) of around 0.5 in the communications routines. It is thought that this is caused by the copying of the halo regions of the 3D arrays that are not contiguous in memory. In the original code, the halo exchange is done in two steps: first the north-south halos are communicated, after which the east-west halos are exchanged. This has the advantage that also the halo regions from 'diagonal' neighbours in all four corners are available, which is found to be essential for NEMO to function properly. It is planned to use derived datatypes and persistent communication channels to speedup the performance of the cpu-intensive part in the communication subroutine.

Cray XT5 - Louhi

On the Cray platform are several compilers available: Cray, PathScale and PGI. Jean-Guillaume Piccinali compared the performance of different compilers for the NEMO code on the Cray system. The NEMO model uses -r8 (automatically promote REALs to REAL*8s), which is incompatible with the way the NetCDF library is compiled using the Cray compiler. This prevents us from using the Cray compiler for NEMO at the moment. The PathScale compiler is 60% slower than the PGI compiler, even when compiled with -Ofast, so we recommend using the PGI compiler for NEMO. The PGI compiler gives a 15-20% improvement when used with the -O3 flag over the unoptimized NEMO binary. The equivalent of the -qhot IBM compiler flag for the PGI compiler is -fast, however when the -fastsse or -fast flags are applied to all routines, the runs are segfaulting from the start. The model is run using fully subscribed nodes on the Cray, since there is no benefit from undersubscribing nodes.

NEC SX-9 - Baku

The NEMO code has been used mostly on the NEC platform and as such runs efficiently on that architecture. NEMO developers mention that on the NEC vector platform, trying to use loop combining or autovectorization results in crashes of the code.

5.13.3 *Results of optimisation effort*

The optimisations that were tried so far did not have a desired performance increase.

5.13.4 *Conclusions*

Apparently, compiler flags are already optimised by the NEMO developers, since on all platforms it was noted that extra or more aggressive flags would not result in faster execution or would let the model crash.

Furthermore, from the fact NEMO has quite a flat profile, none of the routines uses more than 15% of the cpu time for its calculations, we concluded that hand optimisation would most probably not give significant speedups, but may negatively impact the readability of the code. The code owners are quite concerned about the readability, since less readable code makes it more costly to maintain. Therefore, our effort concentrated on the scaling aspect of the code, also because the communication routines dominate the profile on 512 cores and above.

5.14 NS3D

Harald Klimach (HLRS)

NS3D is a code for direct numerical simulation (DNS) of the compressible 3D Navier-Stokes equations and is used for the simulation of sub-, trans- and supersonic flows. It uses compact finite differences in two dimensions and a spectral ansatz in the third. Integration in time is done using a fourth order Runge-Kutta scheme.

5.14.1 *Application description*

The code is completely written in Fortran 95 and neatly separated into modules, especially there are no old inherited Fortran 77 remnants. There are also some useful comments throughout the code, however they are mostly in German. The code authors try to maintain a uniform source layout, which is very helpful for non-authors, when reading the code. The main computationally intensive algorithms are the solving of tridiagonal equation systems, Fast Fourier Transformations (FFT) and some complicated derivative term computations.

The tridiagonal equation systems are necessary for the compact finite differences. For their solution the Thomas algorithm is used and thus involves a very tight recursion and data dependency. The simulations can also be performed using less efficient plain finite differences instead, which changes the computational requirements dramatically, especially breaking up tight recursion schemes.

The complicated derivative term codes were generated by Maple, and involve many computing steps on relatively few numbers, resulting in high performance without much tuning effort.

The used FFT implementation requires the number of points to be a power of two and thus can be implemented very effectively.

For the FFT and for I/O the EAS3 library is used. This is the only library used by the application, however it might be possible to replace the FFT calls by system specific implementations provided by the vendor. However so far only a specially tailored implementation of the EAS3 FFT for NS3D has been investigated and promised to be a good path for high performance.

The datasets used for optimisation and porting are covering the complete code. They deploy compact finite differences, which need a tridiagonal equation system to be solved. Even so the

datasets are simplified, to allow easier deployment on varying sets of processor numbers. They are derived from a real world example and represent actual production runs very well.

5.14.2 *Porting*

The code was developed for the NEC SX series and makes use of a compiler flag to promote reals and integers to 8 byte quantities instead of the default 4 byte. This has an impact on the used libraries, as they need to be compiled the same way. Especially for the MPI library this is an issue, as it cannot be assumed to be available on all systems with 8 byte integers. A first general porting step therefore was the conversion of the code to consistently use double precision real numbers but 4 byte integers.

These changes also affected the used EAS3 library. The changes made, to support the new

behaviour of NS3D have been fed back to the library developers and have been made publicly available from version 1.6.7 on.

After resolving this, mostly just minor issues arose on each platform. The effort needed to port NS3D to IBM Power6 cluster, Cray XT5 and SMP-ThinNode+Vector roughly was around 2 – 3 PMs in total. One issue for example, was the fact, that the naming scheme of files didn't allow more than 1000 processors and changing this required modifications in several parts of the code.

Lessons learnt from the porting are already common knowledge, however not always obeyed by developers. Most issues arose from non-standard-conform code, which relied on special behaviour of the compiler. This is usually easy to fix but can be quite burdensome. For portability reasons, the code should therefore adhere to the language specification as close as possible. Another issue was the usage of MPI-1 interfaces, which involved address references and needed to be replaced by the corresponding MPI-2 interfaces. So the lesson here is to use implementations, which can handle varying data representations of the hardware, MPI-1 shouldn't be used anymore.

A good practice that became clear during the porting phase was the separation of the preprocessor step from the compiling step. As the preprocessing step provided by the compiler may vary from platform to platform, it is more reliable to use a separate preprocessing step, which can provide the same behaviour independent from the compiler. Preprocessing is a necessity, especially when optimised code parts are to be used for different platforms respectively.

When writing Fortran 90 code it is a very good idea to explicitly state the kind of all numerical entities, including literal constants, at least for floating point numbers. The kind should be specified using the `selected_real_kind` function provided by the language.

NEC SX-9 - Baku

As the code has been developed on and for the NEC SX series, no porting was required for the NEC SX9. With only one vectorizing compiler and MPI library being available, these were used. The default optimisation level was used since the most aggressive level “hopt” did not provide significant improvements for the original version and conflicted with the optimisations done by hand (see below). Being a vector computer the optimal length of the inner blocks is the vector length of the machine. The EAS3 library is the only library besides MPI that was not preinstalled on the prototype and needed to be installed. When shared memory parallelization is not used, the corresponding compiler flag should be removed to increase the performance as done here.

Porting to the Intel Nehalem cluster required only adaptations of the compilers and its options in the Makefile of the main program and of the EAS3 library. The Intel compiler was used with the optimisation level -O3. Other installed compilers like GNU or PGI have not been analysed yet. The optimal block length (see optimisation techniques) was found to be 128.

IBM Power6 cluster - Huygens

The process of porting the code to the IBM Power6 cluster yielded no major problems. The main issues arose from compiler incompatibilities, where the IBM compiler was stricter than the NEC compiler, like for example missing spaces between stop statements and attached strings. For Fortran-C interaction, some symbolnames and variable sizes had to be additionally specified in the code, as this is the first time that the code has been executed on an IBM Power system. The only required library EAS3 had also to be ported to the machine. It might be beneficial to replace the EAS3 derived FFT by the ESSL implementation, however, as the currently used FFT is specially tailored to the specific application, there is only slight a improvement expected from such a replacement. Up to now, the value for the length of the innermost loop has not been investigated.

The code was compiled using the most aggressive general optimisation flag “-O5”. No deeper analysis of possible compiler flags was done so far. However, accidentally in the beginning some runs were performed without any compiler flags at all, so with actual Power3 execution code. This showed a performance of roughly 400 times slower than the Power6 executable with the “-O5” option. The application can benefit from the SMT feature of the processors.

Cray XT5 - Louhi

There were no platform specific issues in porting to this architecture, however, mainly for this platform a new naming scheme had to be introduced for the data files in order to allow more than 1000 processes. Initial analysis showed that, using the default PGI compiler suite of the system, it is possible to compile the code using the -fast compiler flag, deploying heavy optimisations.

IBM Blue Gene/P - Jugene

Porting of the application and its EAS3 library was relatively straightforward. Yet one should keep in mind that the compute nodes are 32 bit. The code was compiled with the highest optimisation level -O5 as done for the IBM Power6 cluster – Huygens. Omitting the double-hummer option (-qarch=450) decreased the computational speed and thus the default setting (-qarch=450d) was used. The introduced strip-mining (see optimisation techniques) allows a variation of the innermost block length. The best performance was reached for a length of 256. With up to some 20000 processors being used, the efficiency of the pipelined Thomas algorithm decreases. So the discretization was switched from compact to explicit finite differences. This requires less computational time per grid point and provides a better speedup. However, one should note that the accuracy of the spatial discretization decreases, e.g. for an error of 1% the explicit stencil of 4^{th} order requires double the grid points in x- and y-direction than the compact 6^{th}-order scheme.

5.14.3 *Optimisation techniques*

Due to the fact, that the code was already running and developed on the NEC-SX8, most parts of it yielded a very good vectorization. Obviously this is a prerequisite for good performance on the SX architecture, but it is also beneficial on other platforms with pipelining.

Another computational optimisation technique deployed mainly for the SX architecture is the reduction of subroutine calls, which is again good for any machine, but especially important on the vector architecture. The benefits are twofold. First there is a reduction of overhead, as less calls have to be performed. Second there is usually a larger computational part with more options for optimisation within the same scope. In NS3D this had to be done for the FFT routines. The FFT routines where called within a loop, resulting in a large number of calls. By putting this outer loop into the routines themselves, the number of calls was dramatically reduced and it became possible to greatly enhance the possible vector length within the FFT.

The next step to gain more performance involves blocking of the long vectorizable loops, to provide better usage of the hierarchical memory. On the SX systems there are vector data registers available, which can be explicitly assigned to variables. In order to use them, a blocking with the vector data register length of 256 is necessary. This technique was for example deployed in the tridiagonal solver. The use of blocking mechanisms in crucial parts also allows fine tuning on caching systems by modifying the length of the innermost blocks.

Furthermore, analysis showed some unnecessary operations, which could be removed without affecting the functionality of the code.

Specially tailored FFT and reduced number of subroutine calls

After profiling the application it quickly turned out, that the computation of FFTs is a major part of the overall computing time. It also turned out, that some routines were called relatively often, as they resided within a loop.

Most of those routines were easily changed to include the loop, previously surrounding them, reducing the number of calls and allowing better vectorization within them. As this also provides better cache usage and does not destroy optimisation contexts for the compiler, this is beneficial for all platforms. Especially the FFT computation benefits from these changes, as it was the major part in the code, where this technique has not already been deployed. The effect of the optimized FFT is shown for the NEC-SX9 where 100 time steps were computed by 16 MPI processes on a domain with 50 million grid points in total. The execution time is reduced by 10% and the FLOP rate is increased accordingly. This is mainly due to a decrease of bank conflicts, which yields a growth of the overall vector operation ration to 99.56%. The effect of this optimisation is even stronger if shared-memory parallelization is used. In this case the computational speed is increased by up to 25% compared to the original FFT. However pure MPI without shared-memory parallelization yields a better scaling up to moderate numbers of processors. Thus the effective benefit must be rated as 10%.

	original FFT	optimized FFT
execution time [s]	275.00	250.24
GFLOPS	177.14	195.29
bank conflict (network) [s]	160.2 – 168.4	139.6 – 148.4
bank conflict (CPU) [s]	54.8 – 55.2	40.1 – 40.4
vector operation ratio [%]	99.45	99.56
memory [GB]	72.70	79.87

Table 40: Comparison of the original FFT with the optimized one (50 Mio. grid points, 100 time steps, 16 MPI processes).

Blocking loops, especially in the tridiagonal solver

After investigating the code structure it turned out, that there are some spots, where temporary data is reused several times. For those occasions it is beneficial to put this data as close to the CPU as possible, thus it does not need to be reloaded more often than necessary. On the NEC-SX vector system there are vector data registers available, which can explicitly be used by the programmer to store arrays, fitting into a vector close to the computing units. On caching machines there are associative caches available, where often-used data can reside close to the CPU.

As the storage close to the computing units is very limited, a mechanism to limit the data size there is needed. Blocking complete loop structures in a way that the innermost loops are always limited to a given iteration count provides this.

This technique requires a relatively large effort, as usually many loops need to be reordered and data structures need to be partially restructured. However, it yields increased performance for all platforms. An issue with this technique might be the increased memory footprint, if data is to be propagated from one loop block to the next one. However, as the blocks are naturally limited in size, the additionally needed memory in is minor, and can even be reduced by adjusting the block length.

A very important part of the code, the tridiagonal solver, which also limits the scalability of the application, was reduced in execution time for the x-direction by a factor of 2 and in y-direction by a factor of 3 on the NEC-SX9. On the other systems a more detailed analysis with varying block lengths to make best use of the available cache was partially done.

Using the Nvidia GPU as hardware accelerator via compiler directives

The use of hardware accelerators (GPU) was checked on the Nehalem part of the SMP-ThinNode+Vector prototype for a selected part of the code, deemed to be most suitable. By restricting this technique only to a small part it was possible to separate it easily into a small module on its own, easing the deployment.

Due to their high vector performance of up to 50 GFLOP/s per processor on the SX9, the computation of the time derivatives was chosen as being an ideal benchmark for evaluating the graphic-board computing. The code was compiled using the Portland compiler version 11, which is the only compiler supporting Fortran code on graphic boards in this fashion up to now. Running portions of the code on the accelerator hardware was achieved by simply adding compiler directives which specify the region of the code to be executed on the GPU

(!acc region). Thus the modifications are quite easy to implement. However, subroutine calls inside such a region are not supported and thus it is not possible to run the program completely on the accelerators. The drawback of this is that data is copied to and from the graphic board for each GPU region each time such a region is executed.

In order to minimize data transfer between memory and accelerator board, the originally five subroutines were merged into one single routine.

The computational performance of the Nvidia Tesla S1070 was tested for different grid sizes up to 2GB memory and compared with pure CPU computations on the Intel Nehalem (2.8 GHz). The resulting computing time of the subroutine is shown in Figure 11 for single (kind=4) and double precision (kind=8). In all cases, the computing time is linearly dependent on the grid size. On the CPU, computing time is proportional to the grid size almost independently from data precision. The increase of computing time is substantially reduced on the accelerator boards, especially in the case of single precision computations. For double precision, the GPU increases the performance of this part of the code by a factor of 1.7.

By adding an additional internal loop inside the acceleration region, the number of operations was increased while keeping the same amount of data to be transfered. This allowed an estimation of the time spent on computation in relation to the time used for copying data to and from the accelerator. For all considered cases (0.22 to 3.54 million grid points, double precision), data transfer required more time than the actual computation with the ratio being in the range of 2 to 4. This reduces the sustained performance from 19 GFLOPs based on the actual computing time down to 5.5 GFLOPs based on the complete executing time. The data transfer rate derived from the number of arrays is found to be between 2.25 and 3.8 GB/s. With up to almost 50% of the maximum rate of the PCI-Express bus being reached, the PCI-Express bus is a bottleneck for double-precision computations.

With GPU computing being a relatively new field, future compilers may allow larger continuous portions of the code to be automatically cast off to the accelerator, thus reducing the time required for data transfer.

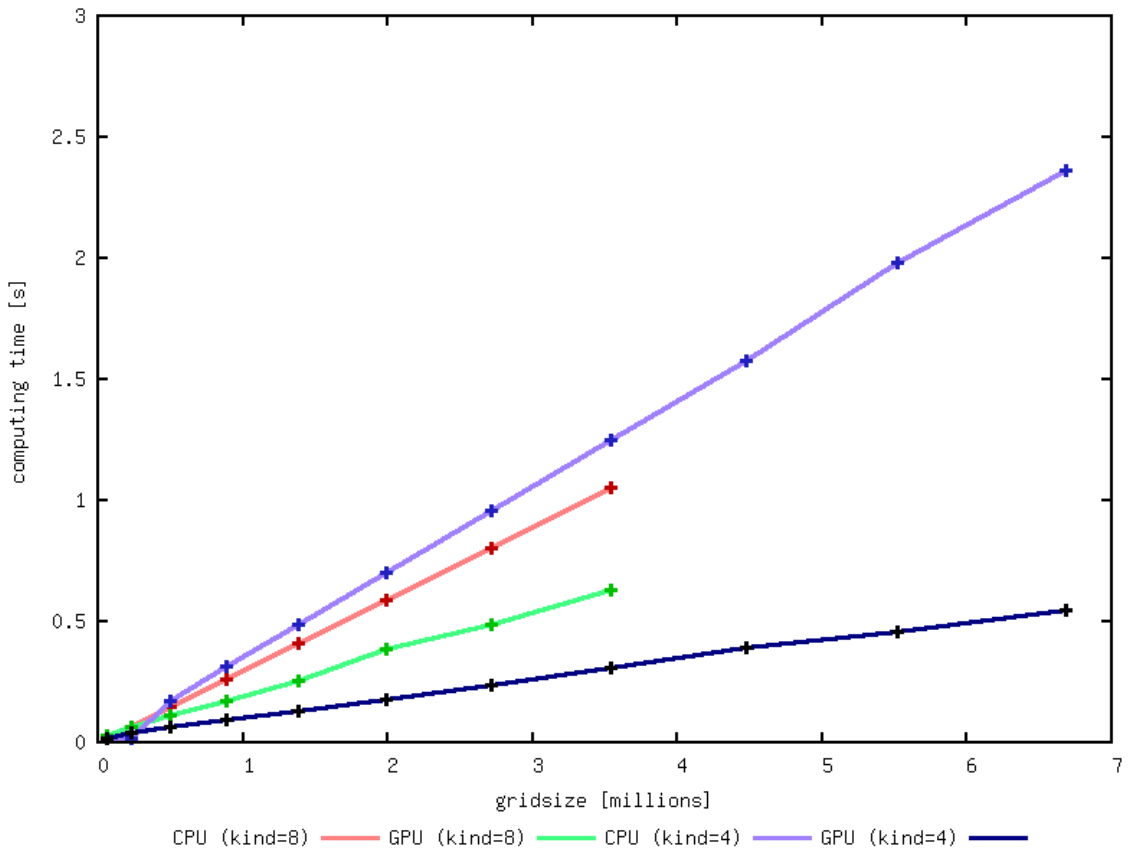

Figure 11: Computing time of merged subroutine ddt for CPU (Nehalem 2.8GHz) and hardware accelerator (Nvidia Tesla S1070) as a function of grid size.

5.14.4 *Results of optimisation effort*

Overall a performance improvement of roughly 35 % on a single NEC SX-9 node was achieved by the optimisation techniques described above. Also the time fraction spent within the tridiagonal solver was reduced from 38 % to 21 %, thus increasing the scalability of the code.

5.14.5 *Other optimisation techniques*

The main part of the code's parallelization concept is domain decomposition in two of the three directions where spatial derivatives are computed by compact finite differences. The computation of their right-hand sides is straightforward using non-blocking data exchange between neighboring domains. The tridiagonal equation system due to the left-hand side of the compact scheme is solved by the Thomas algorithm, which is made of three recursive loops. Since the first loop contains only coefficients and no flow variables, it is done only once during initialization, thus not affecting the parallel performance. The serialization of an ad-hoc implementation is avoided by pipelining of the occurring derivatives. Figure 12 shows the principle concept of this algorithm for 4 derivatives (#1 - #4) on 3 processors (MPI proc 1 - 3): the first domain starts with derivative number one. After its completion, the intermediate values are sent to the next domain. While the second domain continues with derivative number one, the first domain continues with the recursive loop for derivative number two. With a sufficient number of derivatives (here up to 25), the relevance of dead times is reduced significantly. One may increase this further by splitting each derivative along the other directions as long as communication times do not outbalance the actual computation.

However, this was omitted in the code since this would prevent vectorization and/or shared-memory parallelization. Thus the parallel efficiency of the solver decreases for large numbers of domains in one direction. However, all other computations are local for each MPI process and typically domain decomposition is done in two directions for larger cases.

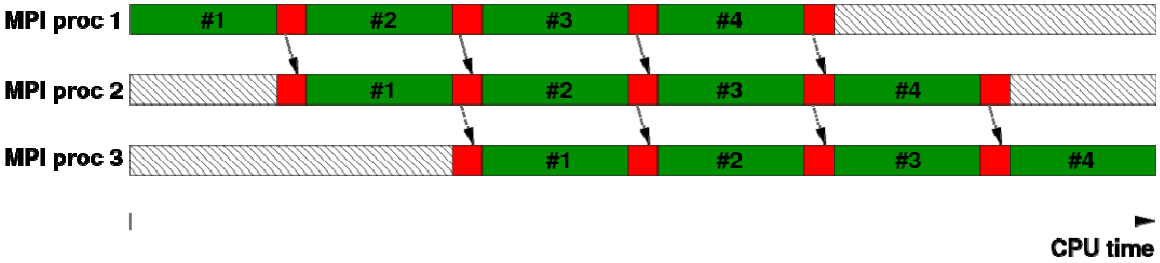

Figure 12: Illustration of the pipelined Thomas algorithm solving four equation systems on three domains. Green denotes the actual computation and red MPI communication. Grey areas correspond to dead times.

Due to the spectral discretization along the third axis shared-memory parallelization is employed in this direction to avoid the exchange of large data sets. Thus the compute-expensive FFTs are local for each MPI process. A high vectorization rate is achieved by data operation along large arrays and loop fusion when possible. Due to the structured grid, indirect addressing is almost completely avoided. With the array sizes being fixed at compilation, the compiler has further possibilities for optimisation, which also may provide some performance increase. The drawback of specially-compiled executables is reduced by a starter script which automatically compiles the code with the appropriate array sizes and submits the job to the queuing system

5.14.6 *Conclusions*

An important fact, that turned up during the work on this code, was that necessary optimisations were mostly platform agnostic, though with different impact. In this special case it also turned out, that the single core optimisation of a crucial part was also essentially important for scaling improvements.

The most important feature implemented in the course of this work was the deployment of blocking, to allow efficient use of the memory hierarchy in the most important computational parts. There are still some important routines, which do not run with the expected performance yet. Their optimisation seems to require more platform specific changes and deeper analysis.

On the caching platforms an analysis of the performance in dependence of the block length needs to be performed in more detail. With a single parameter it was possible to fine-tune the application to a given architecture. A further improvement could be gained by the usage of vendor provided FFT implementations.

The application is mainly memory bandwidth bounded, as it is acting on huge amounts of data. The main constraining factor thus is the Byte per FLOP rate of the system. There are some important parts, where many operations are performed on relatively few data sets. These may still gain some speedup from increased FLOP rates. But overall the main stress is on the memory system. The optimisations done in PRACE aim to reduce this bandwidth dependence, however these are only limited.

As can be very nicely seen in the analysis of the simple deployment of GPUs for heavy computing loops, the code should be able to take advantage of hardware accelerators, if their connection is fast enough, e.g. directly connecting them to the CPU interconnect system.

5.15 Octopus

Fernando Nogueira (UC-LCA)

Octopus is a computer code to calculate excitations of electronic systems. The code relies on Density Functional Theory (DFT) to accurately describe the electronic structure of finite 1-, 2- and 3-dimensional systems, like e.g. quantum dots, molecules and clusters. Although further code development is still needed, there is also the possibility of describing infinite systems. The code is released under the GNU Public License and is freely available at: http://www.tddft.org/.

5.15.1 *Application description*

The code is mainly written in FORTRAN 90, but it contains some parts written in C and it also relies on several Perl scripts. The code consists of approximately 120k lines of FORTRAN 90 code, and 20k lines of C code.

Beside a MPI library, the code requires some standard external libraries: FFTW, BLAS, LAPACK, and GSL. Other libraries are also required, but they are currently bundled with the code: METIS and/or ZOLTAN for parallel partitioning, NEWUOA for derivative-free optimisations, POISSON_ISF and SLATEC for the solution of Poisson's equation, QSHEP for interpolations, and LIBNBC for non-blocking communications.

Octopus uses auto-tools to ease the compilation process and is thus very easy to compile for different architectures/systems.

Octopus is a tool for the calculation of electronic excitations, using the Time-Dependent formulation of DFT (TDDFT). TDDFT calculations start from a converged DFT ground-state. Due to this, octopus must also compute the ground-state of the system of interest, and this can be viewed as a pre-processing step.

In Octopus the functions are represented in a real space grid. The differential operators are approximated by high-order finite differences. The propagation of the time-dependent Kohn-Sham equation is done by approximating the exponential of the Hamiltonian operator by a Taylor expansion.

Octopus has a multilevel parallelization. First, the processors are divided in groups, and each one gets assigned a number of orbitals. This is a very efficient scheme, since the propagation of each orbital is almost independent. The limitation to scalability is given by the number of available states, but this number increases linearly with the system size. Then the real space is divided in domains assigned to different processors. For the application of differential operators the boundary regions have to be communicated. This is done asynchronously, overlapping computation and communication. The scaling of this strategy is limited by the number of points in the grid that, as in the previous case, also increases linearly with the system size. Finally each process can run several OpenMP threads. The scalability is limited by regions of the code that do not scale due to limited memory bandwidth.

The ASCII input file is parsed by an engine that allows for the use of variables. It also automatically assumes default values for all input parameters that are not explicitly assigned a value in the input file. The output is plain text for summary information, and platform-independent binary for wave functions.

5.15.2 *Optimisation techniques*

Octopus was compiled with either the default compiler flags or a set of flags that was already known to be almost optimal and was therefore included in the configure script. The only common feature to all these flags was the optimisation level (-O3), that albeit aggressive is known to lead to correct results with octopus (extensive care has been taken by the developers ensuring that different compilers and/or compiler flags do not lead to different results).

IBM Blue Gene/P - Jugene

The IBM XLC compiler was used for C with the following options:

```
-g -O3 -qarch=450d -I/bgsys/local/gsl/include
```

The IBM XLF compiler was used for the fortran files with the following flags:

```
-g -O3 -qarch=450d -qxlf90=autodealloc -qessl -qsmp=omp
```

IBM Power6 cluster - Huygens

The IBM XLC compiler was used for C with the following options:

```
-g -O3 -qarch=auto -I/sara/sw/gsl/1.11/include
```

The IBM XLF compiler was used for the fortran files with the following flags:

```
-g -O3 -qfree=f90 -qnosave -qarch=auto -qxlf90=autodealloc -qessl
```

Thin-node Nehalem cluster - Juropa

The Intel compiler suite was used for C with the following options:

```
-u -fpp1 -pc80 -align -unroll -O3 -ip -no-fp-port
-mno-ieee-fp -vec-report0 -no-prec-div
```

The Intel compiler suite was used for fortran with the following options:

```
-nbs -p -u -fpp1 -pc80 -align -unroll -O3 -ip -no-fp-port
-mno-ieee-fp -vec-report0 -no-prec-div
```

IBM Cell cluster - Maricel

On this architecture the PPU versions of the IBM XLC and XLF compilers were used. For C the compiler flags were:

```
-q64 -O3 -qarch=auto -qcache=auto
-I/home/Extern/fnogueira/gsl/include
```

For fortran the compiler flags were:

```
-q64 -O3 -qarch=auto -qcache=auto -qextname
-qxlf90=autodealloc
```

5.15.3 *Conclusions*

Due to the late inclusion of octopus in PABS it was extremely difficult to test different compiler flags and improve on the application performance. However the tests that were actually run did point to some possible ways of improving octopus' performance. Octopus relies on several standard external libraries, like FFTW, BLAS and GSL. Although the libraries used in the test runs were, in principle, optimized for the prototype being used, this

optimisation did not lead to improved performance. It is true that octopus does not spend much time on these libraries but, at least in the Cell prototype, for which no specific code optimisations were made, these libraries should have had a bigger impact, as the PPU's were used solely through them. This possibly indicates that these libraries (FFTW in particular) can be further optimized for this architecture, bringing improved performance to octopus and all other codes that depend on them. Another clue for future improvement of the code comes from the profiling of the runs. Octopus has an internal profiler that keeps track of the time spent in each "code block" and actually computes the performance of each of these blocks in MFlop/s (this calculation is based on a reasonably accurate estimate of the real number of floating point operations done in each block). The code blocks are not necessarily single subroutines, but also possibly groups of subroutines. The analysis of the profiler output from the largest run made in each prototype showed that two blocks account for 40%-50% of the time spent in the benchmark. These two blocks are NL_OPERATOR_BATCH and VNLPSI_REDUCE_BAT. Although they were not the most often called blocks, each of these blocks was called several times during the benchmark, NL_OPERATOR_BATCH being called twice for each VNLPSI_REDUCE_BAT call. The total time spent in each block was roughly the same for the IBM Power6 cluster, Nehalem cluster and Cray XT5 prototypes. However, in the Blue Gene/P prototype the code spent twice as much on NL_OPERATOR_BATCH as on VNLPSI_REDUCE_BAT, while in the IBM Cell cluster prototype it was the VNLPSI_REDUCE_BAT that took almost three times longer to complete than NL_OPERATOR_BATCH. As these two prototypes don't have much memory available per computing core, these results point to a possible memory-binding problem of octopus.

5.16 PEPC

Lukas Arnold (FZJ)

PEPC is a parallel tree-code for rapid computation of long-range (1/r) Coulomb forces for large ensembles of charged particles. The heart of the code is a Barnes-Hut style algorithm employing multipole expansions to accelerate the potential and force sums, leading to a computational effort O(NlogN) instead of the $O(N^2)$ which would be incurred by direct summation. Parallelism is achieved via a `Hashed Oct Tree' scheme, which uses a space-filling curve to map the particle coordinates onto processors.

5.16.1 *Application description*

The source code is divided into two parts: the general PEPC-library, which provides the kernel of the benchmark, and an application frontend. In the beginning of the benchmarking progress, in the PRACE context, the PEPC-B frontend was used. This frontend is used to calculate laser-plasma interaction. As the benchmark is focusing only on the electromagnetic interaction, the benchmark framework has switched to PEPC-E. This frontend provides an optimized, mostly in the sense of memory management, way to solve the benchmark problem, as it deals only with electromagnetic interaction.

The full code is written in Fortran 90 and does not use any external library.

The code is well written, but hardly documented. In the current state the maximum number of mpi tasks is limited by the memory requirements. The main problem is the memory size of the oct tree, which grows nonlinearly with the number of mpi tasks. As long as this limitation is not solved, PEPC will not be capable of utilizing a petaflop machine. To approach this limitation, each mpi task must not store global information about the oct tree, as it is done

now, but only partial information. However, this approach will result in a fundamental restructure of the communication scheme.

The main algorithm is the following: at first one has to generate all particle-interaction lists, than all forces are communicated and summed and finally the equation of motion for each particle is evaluated. The main computational load, in terms of FLOP/s, is the function 'sum_forces', which calculates all acting forces; it contains about 90% of the total floating point operations.

The PEPC-B frontend needs some pre-processing steps, which are divided in two steps: an application external and an application internal step. The external step is mainly the build up of directory structures and the creation of particle lists. The application internal pre-processing step is the initialization of the initial values for the simulation. There are no post-processing steps. The PEPC-E frontend has eliminated the external steps.

The parallelization strategy in PEPC is to distribute all particles equally on the processes, with respect to their physical positions. Each process collects all information needed to integrate the equation of motion for its particles from all other processors. Obviously each process provides information, particle positions or multipole moments for the others. Thus the communication pattern is not structured or only next neighbor dominated but global, due to the long range electromagnetic interaction. The parallelization is realized within the MPI framework.

The current I/O strategy is that each process creates a single file and writes its particle properties in text mode to this file. This works fine for small processor numbers but will not work for large partitions. This is a scaling task, i.e. the output will not create many small files, which are accessed only serially in the worst case, but one single file in parallel and binary mode.

A dataset for PEPC is a particle distribution, i.e. the prescription of the (initial) position and velocities as well as other properties. This means that it is easy to generate any problem size, just by modifying the particle number. Petascale and real world datasets are in the order of 50 to 100 million particles, or even higher.

5.16.2 *Porting*

As PEPC does not depend on any external library, but the mpi library, and is programmed in a general manner, the porting is straightforward.

There have been no porting issues on the target platforms, thus the effort is less than one person month.

IBM Blue Gene/P - Jugene

The PEPC code did not have to be modified and did not show any porting issues as it has already been run on this architecture.

The best performance was reached with the IBM XL compiler and the maximal optimisation level as well as the architecture specific arguments

```
-qtune=450 -qarch=450d –O5.
```

The achieved speedup is in the order of 25%, compared to the standard optimisation level (`-O2`).

Cray XT5 - Louhi

The PEPC code did not have to be modified and did not show any porting issues. It was not run on this architecture before.

The best performance was reached with the PGI compiler and the maximal optimisation level

`-O3 –fast .`

The achieved speedup is in the order of 25%, compared to the standard optimisation level (`-O2`).

IBM Power6 cluster - Huygens

The PEPC code did not have to be modified and did not show any porting issues as it has already been run on this architecture.

The best performance was reached with the IBM XL compiler and a high optimisation level as well as the architecture specific arguments

`-qtune=pwr6 -qarch=pwr6 –O4 .`

The achieved speedup is in the order of 15%, compared to the standard optimisation level (`-O2`). The highest optimisation level (`-O5`) failed, as the application could not be executed correctly. The global IPA (Inter Procedural Analysis) changes the code's behaviour in an inconsistent way. This is just an issue for this architecture, as the same compiler generates a valid code for the Blue Gene/P architecture.

5.16.3 *Floating point performance*

The floating point performance of PEPC-E is acceptable, within the computation intensive parts, i.e. 'sum_force', it reaches up to 20% peak performance. The global values range from 20% to 5%, see Figure 13. These numbers indicate, that PEPC-E's performance is acceptable and motivates to focus the developer's effort on scaling and not on optimisation. The scaling work is reported in report D6.4.

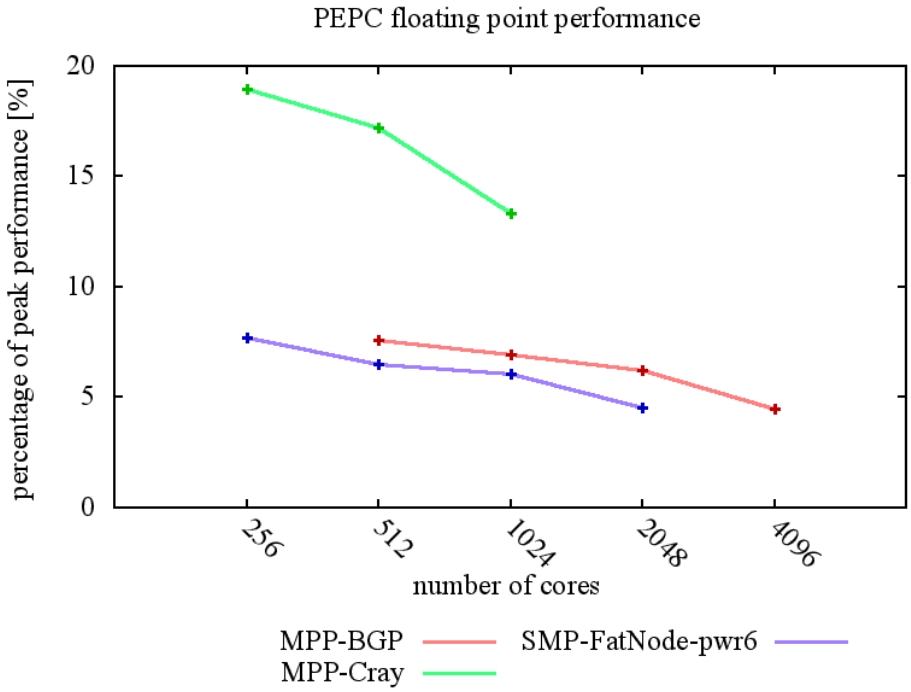

Figure 13: Performance of PEPC-E on target architectures.

The performance on the IBM Power6 cluster can be easily increased by up to 20% by using the SMT mode. However, this mode is not available on this prototype, yet. This speedup can be shown on non-PRACE prototype IBM Power6 machines, like jump at FZJ.

5.16.4 *Conclusions*

PEPC's performance is in an acceptable range on the target architectures. As stated before, no optimisation has been done, as the main focus lies on scaling, see D6.4. However, one can clearly identify the Cray XT5 architecture as the architecture on which PEPC-E shows the best floating point performance.

The next optimisation steps should not focus on the floating point intensive parts, as these are already quite efficient, but on the remaining ones to increase the global floating point performance. These efforts might focus on the 'tree_walk' and 'key2addr'routines.

5.17 QCD

Lukas Arnold (FZJ)

The quantum chromodynamics (QCD) benchmark consists of five kernels, each representing different implementations of solvers for the lattice QCD. All kernels are packed into one executable resulting in a single benchmark.

5.17.1 *Application description*

The source code is divided into the wrapper part (written in C) and the kernels called within it. The kernels are written in C as well as in FORTRAN. The whole benchmark does not depend on any external library.

The code is well structured and thus readable. This is due to the fact that these kernels are not the production codes used by the corresponding scientific groups, but rather portable and stand-alone implementations of some well known lattice QCD solvers.

As only benchmarking versions of the kernels are used, there is no significant pre- or post-processing. This is also true for file I/O.

The parallelization strategy is to split the computational domain, a 4D or 3D grid with periodic boundary conditions, into regular domains and distribute one domain to each computing units. Each of these sub-domains needs some information from its neighbours for the solver in the domain interior. This information corresponds to boundary hyper-surfaces which must be exchanged every iteration. This exchange is the main communication task and it is implemented within the MPI framework, at least for the portable kernel implementations. Three of the kernels (A, C and E) require regular calls to global sums of scalar values which are performed a few times every iteration.

The problem size for the QCD benchmark is defined by the grid size. The initial conditions for the solver are analytically available so no datasets are needed for the benchmarks. Thus any problem size can be generated including datasets for petascaling. The benchmark is set up with respect to current scientific problem sizes as well as beyond them to utilize the future petascale machines.

5.17.2 *Porting*

The porting of the QCD benchmark to any general purpose architecture does not cause any fundamental issues. The main problems are generated thru the fact, that five complex applications must be linked into one executable. These results on the one hand in a language mixture (C/FORTRAN), where the interfaces between the wrapper part and the kernel part must be threaten differently on each target prototype. On the other hand, the kernels are all using similar solver, thus the function naming is also similar, forcing a systematic renaming of the kernel functions.

The effort needed for the setup of this benchmark, does not only include the porting, but it also must take into account the development of the benchmark itself, as this is a compilation of five independent applications. The development of the wrapper, porting, testing, integration into the JuBE framework and the instrumentation effort lies in the order of 2 pm.

As this benchmark contains five full lattice QCD applications and they all should be combined into one executable, there have been some naming issues. Thus the renaming was the only code modification which was needed.

No external libraries are used.

IBM Blue Gene/P - Jugene

All of the involved kernels have been already ported to the architecture, so that there are no porting issues to report. However, the virtual memory limit had to be increased to 3 GB to be able to link all five kernels into one executable. This limit has been increased for all user.

The optimal choice of the compiler flags is

```
-qtune=450 –qarch=450d –O3 -qhot
```

At the environment level, one variable is of particular interest for the QCD benchmark, the BG_MAPPING. It determinates the placement of the MPI tasks on the partition. It is recommended to set this variable as an argument for the mpirun command to TXYZ, i.e.

```
mpirun -env 'BG_MAPPING=TXYZ' .
```

This ensures, that during the MPI rank distribution, nodes are filled first, before the next node gets its ranks. By doing so, neighboring computational sub-domains are more likely sharing a node, which reduces the communication time.

Cray XT5 - Louhi

The optimal choice of the compiler flags is

```
-O3 –fast -Mipa
```

IBM Power6 cluster - Huygens

The optimal choice of the compiler and its flags is

```
-qtune=pwr6 –qarch=pwr6 –O4
```

5.17.3 *Optimisation techniques*

All of the kernels are trivial implementations of highly optimized versions, thus no optimisation techniques have been applied in the PRACE project. However, some information on the optimized versions is available and will be presented here.

The following information is within the kernel A context, but it represents the general situation.

A typical optimisation of lattice QCD codes is programming the kernel, which is the so called hopping matrix multiplication taking about 80% of the compute time, in assembler. At the single core level this leads to a speed-up of up to three for that part of the code [, 2]. The speed-up of the overall parallel program varies roughly between 20% and 100% [2].

The assembler is not programmed manually but rather by writing a code generator. The development of such a generator is of the order of six months. Adapting and optimizing an existing code generator has taken about two months for a new platform in the case of [2].

The code authors provide some explicit information and speed-up numbers on the Altix 4700 machine [2].

Because the memory bandwidth is so much lower than the L3 cache bandwidth, it is important to partition the problem in such a way that fields are kept which are needed during the conjugate gradient iterations in the L3 cache, so that in principle no access to local memory is required.

From Table 41 one can see that lattices up to about 8^4 sites fit into the L3 cache. When staying inside the L3 cache assembler code is roughly a factor of 1.3 faster. Outside the L3 cache the assembler is faster up to a factor of 2.7. The reason for this speed-up is prefetching. Prefetching is important in the parallel version even if the local lattice would fit into the cache, because data that stems from remote processes will not be in the cache but rather in main memory.

lattice	#cores	Fortran [Mflop/s]	assembler [Mflop/s]
4^4	1	3529	4784
6^4	1	3653	4813
8^4	1	3245	4465
10^4	1	1434	3256
12^4	1	1329	2878
14^4	1	1103	2766
16^4	1	1063	2879

Table 41: Performance of the hopping matrix multiplication on a single core on the Altix 4700 [7].

To improve inter-node communication one can choose a good data layout [1] and/or overlap communication and computation [7]. In the case of [7] this is also done in the assembler. These are high level optimisations that require changing of larger parts of the code.

However, even with the portable version of the kernels a reasonable floating point performance is achieved. Figure 14 shows the percentage of the peak performance on the target architectures for various partition sizes.

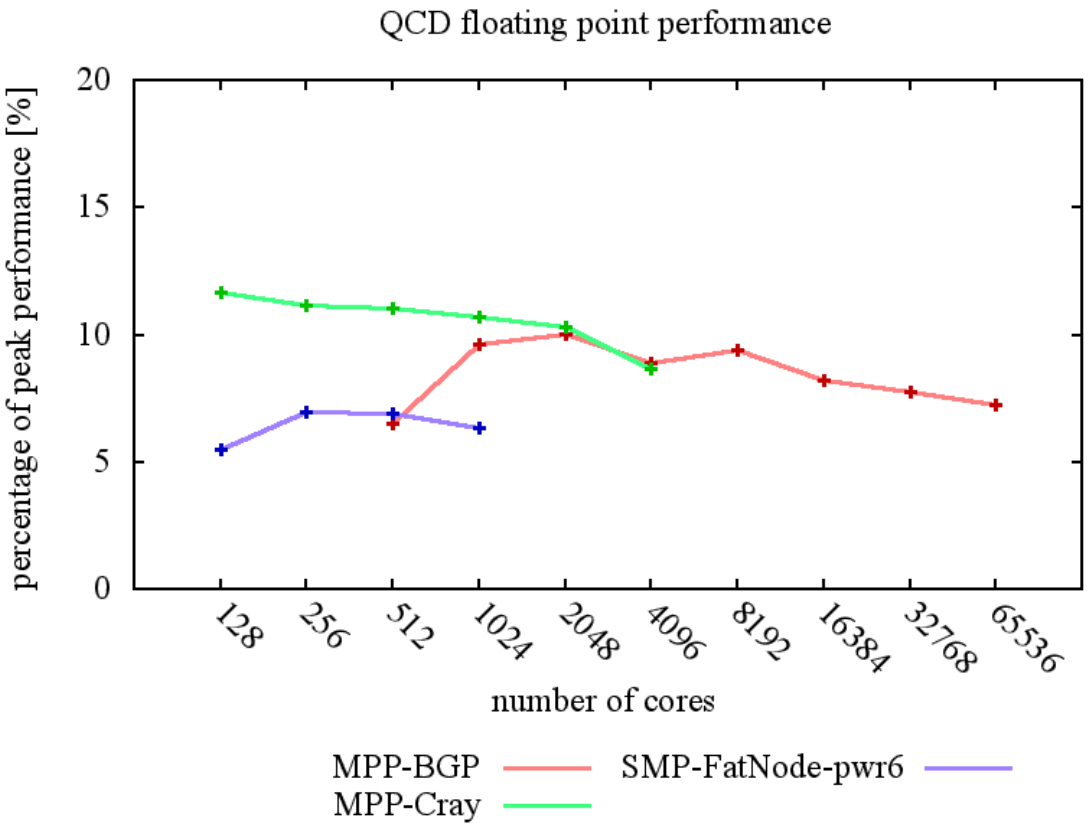

Figure 14: Combined lattice QCD benchmark performance.

5.17.4 *Torus network topology*

The four dimensional computational domain in the lattice QCD benchmark is, in most cases, split in all four directions. This splitting has to be explicitly set for each kernel. On the IBM

Blue Gene/P architecture this distribution should be reflected by the requested torus geometry. In the case of kernel E, it has been shown that a matching of the application and network geometry results in a speedup in the order of 10%. The LoadLeveler variable for setting the network geometry is `bg_shape`.

5.17.5 *Conclusions*

As there exist highly optimized versions of the lattice QCD kernels, no effort has been put into optimisation. A few optimisation techniques, mainly the usage of a self-defined meta language which is compiled to a assembler code, has been presented. However, it should be pointed out that the effort to get to high performance, for example up to 40% of the peak performance on 16 racks IBM BG/P, is enormous and in most cases not portable.

5.18 Siesta

Maciej Cytowski (PSNC/ICM)

Siesta is an ab initio molecular dynamics simulations code for molecules and solids implementing N-order DFT techniques. It is widely used for molecular dynamics computations in HPC projects. Siesta was initially an academic code. Although it is not free software, it is distributed free of charge to all academic users. At the same time commercial licenses are sold to private companies. The parallel MPI version developed by code authors and the Barcelona Supercomputing Center is now in use.

5.18.1 *Application description*

Siesta is written in Fortran90. It uses dynamic memory allocation and module mechanisms. Main libraries used by Siesta are BLAS, LAPACK and METIS for I/O formatting. The readability of the code is very good. Programmers should know the Fortran90 programming techniques.

5.18.2 *Porting*

A Cell processor is a 9 core chip with hybrid design. It consists of one Power Processing Unit (PPU) and eight vector Synergistic Processing Units (SPUs). Compiling the SIESTA code with available compilers produces a PPU binary whose computational performance is poor. This is due to the fact that the PPU is not designed for computations and some special optimisations and porting techniques are required in order to achieve a good performance with the use of SPU cores.

Cell API which enables execution of highly optimized code on SPUs is designed and implemented only for the C programming language. Hence we decided to use a porting and optimisation scheme where time consuming Fortran routines are first rewritten in C and then ported and optimized on SPU with the use of the API. The effort involved in the results presented here is around 2 pm's. The total effort to achieve a 2 times speedup could be, in our opinion, estimated at 3-4 pm's.

We have learned once again that data alignment is very important for Cell optimisation. Not only the SIMD operations but also DMA transfers depend heavily on the alignment. We were using many different techniques to align our data in Fortran. We have achieved a more than

20% speedup involving cache/memory optimisations on the PPU side and SPU parallel implementations. Moreover we programmed some important optimisations that could be used further to achieve additional speedups.

IBM Cell cluster - Maricel

The Siesta code was compiled with the use of CELL architecture specific compilers. We decided to use the IBM XL Fortran for Multicore Acceleration for Linux, V11.1 compiler (ppuxlf). It was the first time the code was ported to this architecture.

We encountered problems related to a compiler bug during the porting phase. In write_subs.F and struct_init.F Siesta call the `cmlAddMolecule` subroutine which is a public member of the `flib_wcml` module defined in `Src/wxml/flib_wcml.f90`. The `cmlAddMolecule` subroutine was called without defining some optional arguments, which triggered a bug in the compiler. We solved it by giving dummy values to the optional arguments when calling the function.

Source code modifications are crucial in order to achieve good performance on Cell chips. The porting phase ended up with a working PPU version of the code. We checked with a benchmark run prepared by Siesta users in our HPC center that the PPU version is 2 to 3 times slower than the implementation on current general purpose architectures (x86). This kind of performance is typical for the compuationally weak PPU. We have decided to perform an optimisation effort on some of the intensive subroutines from the execution time profile.

Siesta uses BLAS and LAPACK. Both of these libraries have their optimized versions on the Cell architecture. Unfortunately both of these optimized versions could be used in a very limited way by Siesta as only a few of them are optimized for the Cell processor.

BLAS and LAPACK functions used by Siesta and optimized on Cell:

```
ddot, dscal, dgetrf, dgetri, dcopy, dsymm, sgemm, dscal, dgetrs
```

BLAS and LAPACK functions used by Siesta and not optimized on Cell:

```
zscal, zaxpy, zswap, zscal, ilaenv, zcopy, zpotrf, zhegst, zcopy, ztrsm,
zhegv,zhegvx, zhemm, zgemm, zheevx, ilaenv, dpotrf, dsygst, dtrs, dsygv,
dsygvx, dsyevx, lsame, xerbla, dlamch, dlansy, dlascl, dsytrd, dsterf,
dstedc, dormtr, zlanhe, zlascl, zhetrd, zunmtr, dspev
```

We decided to use the XLF compilers with following flags. Our choice was consulted with Siesta users from our HPC centre, since Siesta is very sensitive for compiler optimisations.

PPU Fortran code:

```
-qsuffix=cpp=F -c -qfixed -O3 -qstrict -qtune=cellppu
-qarch=cellppu -qhot=simd -q64 -qipa=malloc16 -qextname
-qalign=struct=natural
```

PPU C code:

```
-O5 -qstrict -qtune=cellppu -qarch=cellppu -qhot=simd -q64 -qinline
```

SPU C code: -c

We decided to use the original Siesta testing environment to benchmark our optimisation effort.

5.18.3 *Profiling*

We have performed a profiling process of Siesta on Cell architecture. Our diagnosis consisted of two important profile tools: Siesta internal time measurements and the Visual Performance

Analyzer (VPA) coupled with the Oprofile tool. The first was used to measure the performance of optimized computational kernels. The second was used in the initial phase to give a deep view of Siesta computational profile. The results obtained with VPA are presented below:

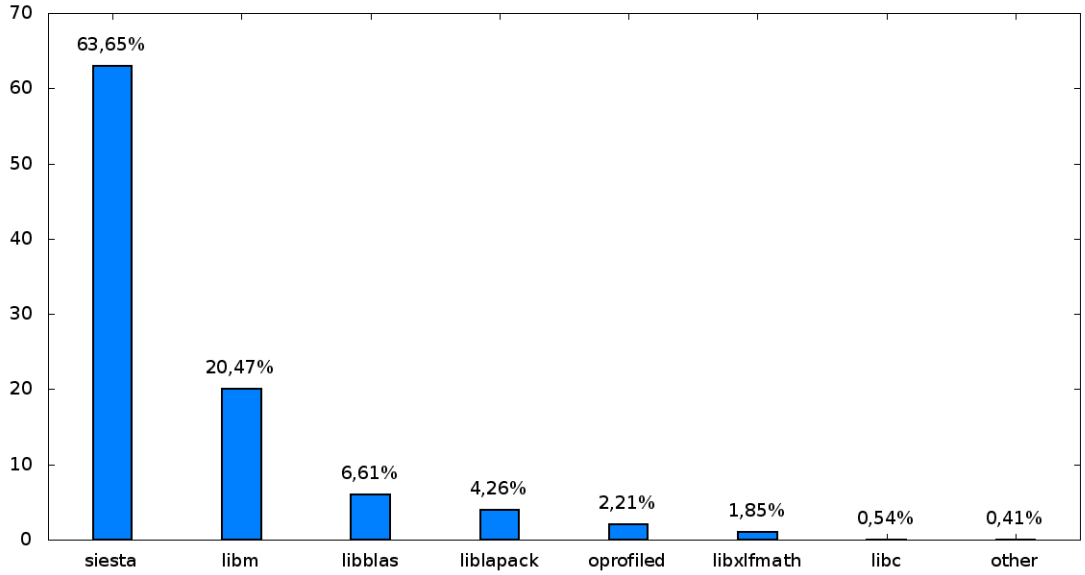

Figure 15: System profile

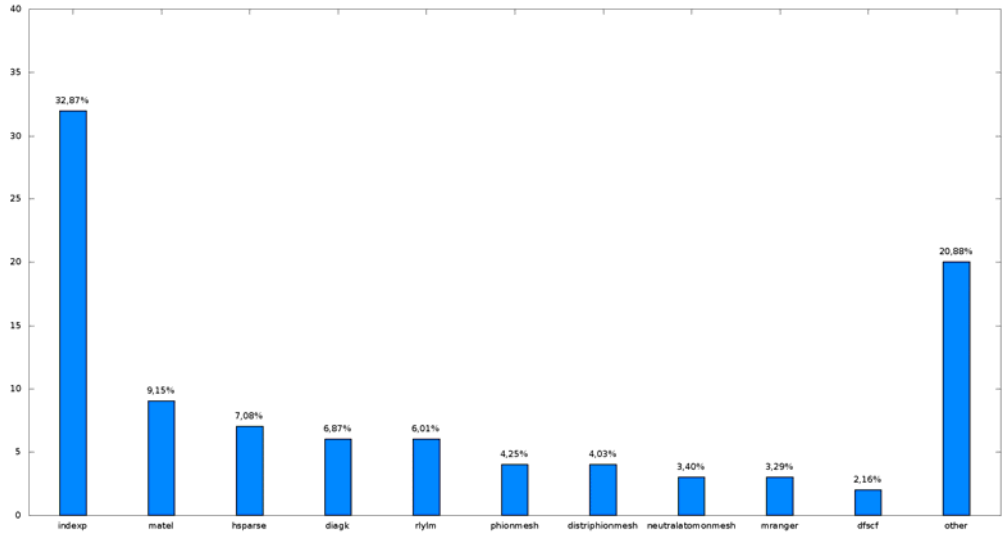

Figure 16: Siesta profile

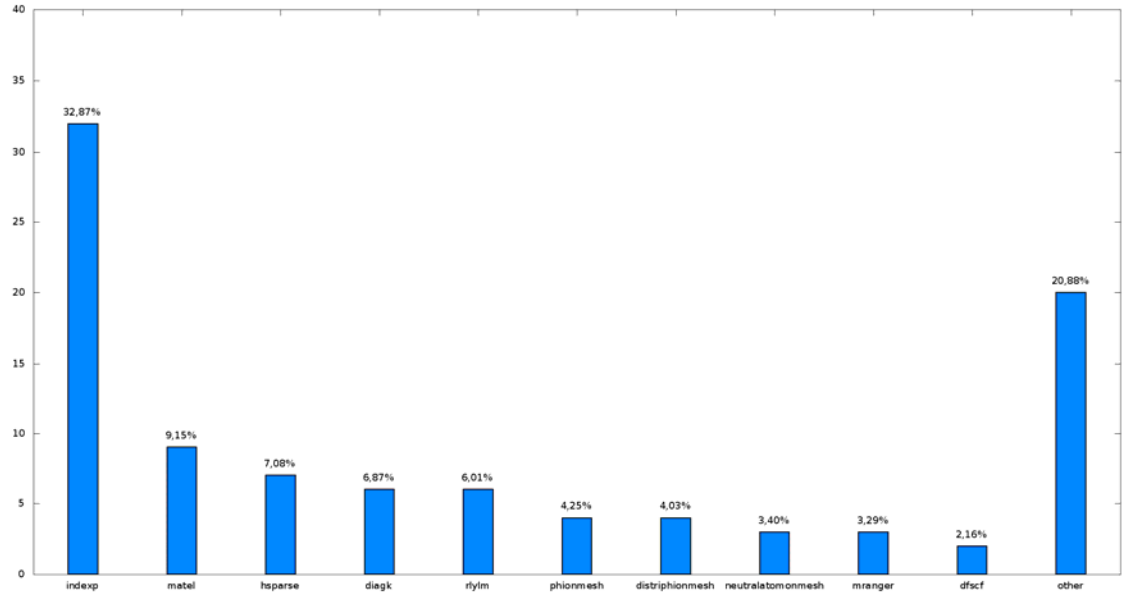

Figure 17: Lapack profile

The very important observation that we have made is that many time intensive functions that could be found in Siestas profile are not-inlined Fortran module member functions that are used multiple times in loops inside some of the Siesta subroutines. For instance `indexp` and `rlylm` module functions are massively used by the `matel`, `phionmesh` and `neutralatomonmesh` subroutines. We have created a specific SPU version for some of these functions. This opens a very wide perspective for further optimisation of Siesta on Cell.

We also see a possibility for further optimisation with some of the LAPACK functions involved. One could consider their CLAPACK versions for SPU optimisation.

5.18.4 *Optimisation techniques*

The optimisation effort consisted of the following tasks:

- Code Profiling - ¼ PM
- Implementation of C SPU functions – 1 PM
- Parallel SPU implementation of computational kernels – 1 PM
- SIMDization of the SPU code – ¼ PM
- Cache/memory optimisations - ¼ PM

Some of the optimisations made and results achieved are described in details below.

Implementation of C SPU functions

We have implemented and optimized following functions on SPU:

`rlylm`

`indexp`

The implementation of the above functions on SPU enables us to create a multicore version of many time consuming subroutines of Siesta.

Parallel SPU implementation of computational kernels

We have seen in the profile results that the `distriphionmesh` subroutine is one of the most computationally expensive functions. On the other hand it seemed not to be very complex and therefore easy to port onto the SPU. The porting process involved the step of identification of the computational core with the use of profiling results. The next step was a Fortran to C translation, which was made by hand rather than with the use of automatic translators like F2C. The code consisted of two main loops. The SPU version was implemented by dividing the first loop count over the available SPU threads. Since the memory usage inside the `distriphionmesh` subroutine was much higher than 256kB (SPU Local Store size) we have designed a two stage double buffering scheme for SPU parallel computations. Additionally, we have also SIMDized the `distriphionmesh` routine on the SPU. The results of the two SPU versions are shown in Table 42.

	Original		SPU		SPU-SIMD	
Routine	**Total time (s)**	**Total time (%)**	**Total time (s)**	**Total time (%)**	**Total time (s)**	**Total time (%)**
`distriphionmesh`	47.920	14.64	7.880	2.75	5.350	1.88

Table 42: Speed of different versions of the distriphionmesh function: original, SPU version and SIMD optimized SPU version

Cache/memory optimisation – diagk.F

We have also performed a set of cache/memory optimisations inside the `diagk.F` file. The resulting code consumes more memory but gives a 8% speedup. The results are presented in Table 43.

	Original version		Optimized version	
Routine	**Total time (s)**	**Total time(%)**	**Total time (s)**	**Total time(%)**
c-buildHS	27.060	7.69	24.080	8.48
c-eigval	42.200	11.99	16.900	5.95
c-eigvec	49.000	13.93	24.120	8.49
c-buildD	27.880	7.92	28.430	10.01

Table 43: Speedup obtained for routines in diagk.F

5.18.5 *Results of optimisation effort*

The results achieved during a 2 month porting are presented in Table 44. The code speedup is of about 20% but as we can clearly see it can be much higher since the optimisation made (for instance in `distriphionmesh` subroutine) can be easily adopted to work in other specific code fragments (like `phionmesh` or `neutralatomonmesh`).

Version	Execution time (s)	Speedup
Original	352.4	
Diagk optimisation	323.3	1.081
Diagk optimisation SPU distriphionmesh	276.4	1.176
Diagk optimisation & SPU distriphionmesh (SIMD)	273.9	1.204

Table 44: Total speedup obtained for Siesta, using the VarCell Siesta internat test. Timings are measured using the Linux "time" command.

5.18.6 *Conclusions*

We have used our knowledge on Cell code porting and optimisation to achieve better performance of some specific Siesta computational kernels. We have learned how to port Fortran codes on Cell. To achieve this we had to design a specific Fortran to C to SPU code translation.

Our profiling and optimisation effort was efficient. We achieved more than 20% speedup in little more than 1 month of work (2-3 persons involved). The work can be easily adopted for another code optimisations. Our work could be also considered in a much more interesting configuration, namely a Hybrid x86 – Cell mode. For that we should make an additional comparison between the SPU optimized code and its x86 version. The SPU optimized code could then be implemented with the use of Hybrid techniques like DaCS or ALF.

5.19 Quantum ESPRESSO

Carlo Cavazzoni (CINECA)

QUANTUM ESPRESSO (QE) is an integrated suite of computer codes for electronic-structure calculations and materials modeling at the nanoscale, based on density-functional theory, plane waves, and pseudopotentials (norm conserving, ultrasoft, and PAW). QUANTUM ESPRESSO stands for opEn Source Package for Research in Electronic Structure, Simulation, and Optimisation. It is freely available to researchers around the world under the terms of the GNU General Public License

5.19.1 *Application description*

QE is mainly written in Fortran90, but it contains some auxiliary libraries written in C and Fortran77. The whole distribution is approximately 500K lines of code, even though the core computational kernels (called CP and PWscf) are roughly 50K lines each. The QE distribution is by default self-contained, even if it can be linked with most external libraries, such as FFTW, MKL, ACML, ESSL, ScalaPACK and many others. Note that the use of external, vendor specific libraries for FFT and linear algebra kernels is necessary to obtain maximal performance. For this reasons QE contains dedicated drivers for FFTW, ACML, MKL, ESSL, SCSL and SUNPERF FFT specific subroutines.

QE has been developed over time by many researchers not necessarily experts in code developing and design; but in the last years a lot of effort has been dedicated to making the

code more readable and more easily extensible. One of the results of this effort is that in the current version (QE 4.1) core numerical algorithms with the highest impact on performance are well separated from the rest of the code. This fact together with the modular structure of the code should make optimisations and petascaling techniques easier to implement.

Main algorithms in the code are FFT, Iterative diagonalization of Hermitian matrix, matrix multiplications. On average the code spends roughly half of the time in linear algebra subroutines and one half in FFT subroutines. This proportion varies strongly with the simulated system. The time spent for linear algebra is mainly for matrix multiplications.

5.19.2 *Porting*

QE installation is assisted by a configure autoconf script, already set up for most HPC systems and clusters on the market. Nevertheless, since the target machines of this work are prototypes, they are not all covered by the present version of the configure script. In these cases the configure script has to be fed with the proper parameters for the compilers, compiler flags and libraries. Sometimes it is possible to instruct configure to produce the makefile parameters for a similar HPC system, and then modify the makefiles by hand. This procedure is made easier by the presence of a single file (make.sys) containing all building parameters and included by all makefiles, and eventually one has to change only this specific file.

The code has been ported to four prototypes (baku, huygens, jugene, louhi) with an estimated effort of 2 pm.

IBM Power6 cluster - Huygens

The production version of QE had been already ported to Power6 linux architecture but, since for benchmarks and petascaling in PRACE we have planned to use a hybrid programming model MPI+OpenMP, which is still under development. The configure script was used only to setup makefile parameters for a plain MPI build. Then it was necessary to modify makefiles by hands in order to setup flags and library for a hybrid build.

A few lines of code have been modified to implement a couple of workarounds for compiler bugs (non automatic variable in recursive function, MATMUL intrinsic).

QE on Huygens has been linked with esslsmp, massvp6, ScaLAPACK and blacs and it has been compiled with the IBM compiler XL compiler suite (xlf90 and xlc). Compiler flags used to build the executable were:

```
-q64 -qarch=auto -qtune=auto -O2 -qsuffix=cpp=f90 -qdpc
-qalias=nointptr -Q -qautodbl
```

To reduce the memory allocated by MPI for its own housekeeping, leaving more memory for the application itself, the following environment variables have been used when running the code:

```
MP_CSS_INTERRUPT=yes
MP_SHARED_MEMORY=no
MP_RC_MAX_QP=1000000
export MP_BUFFER_MEM=2M
```

IBM Blue Gene/P - Jugene

One of the previous version of QE had already been ported to Blue Gene architecture, so no major problem has been found. The only noteworthy issue was that the machine specific configuration used had not been included in the configure script. To circumvent this issue we

configured for a generic powerpc machine with linux, and then modified makefiles by hand in order to setup flags and library for BG/P.

QE has been linked with ScaLAPACK, BLACS, esslbgsmp and mass, and it has been compiled with the IBM compiler XL xompiler suite (mpixlf90_r, bgxlc_r, and bgxlf_r). The optimal flags for QE were:

```
-O3 -qstrict -qsuffix=cpp=f90 -qdpc=e -qtune=450 -qarch=450
-qalias=noaryovrlp:nointptr -q32
```

We also found that when running QE it is better to use the torus network.

Cray XT5 - Louhi

To install QE on louhi it is sufficient to run the configure with proper compilers:

```
./configure MPIF90=ftn F90=ftn F77=ftn CC=cc
```

QE was already running on Cray XT machines, but it was the first time it was ported to a Cray XT5. It has been linked with ScaLAPACK, BLACS, acml and compiled with ftn an cc Cray compiler wrapper to the PGI 8.0.6 compiler suite. The optimal compiler flags were:

```
-O3 -fast -r8
```

NEC SX-9 - Baku

QE has previously been ported to the SX8 architecture, but not to the latest SX9 architecture. To compile it configure has to be run in a hybrid shell as follows:

```
>SXEXEC_NODE=v900 hybsh
>./configure --enable-parallel ARCH=necsx
```

We also had to manually change the make.sys file containing all the parameters required by the different Makefiles. In particular to finalize the configuration of QE on this machine the following line had to be changed:

```
DFLAGS =     -D__SX6
             -Dfinite_size_cell_volume_has_been_set=finite_size_cell_vol_set
             -D__USE_3D_FFT -D__MPI -D__PARA -D__SCALAPACK -D__OPENMP
```

Additionally, we also had to modify a few lines in the FFT driver

QE has been linked with Mathkesian FFT library, BLAS, LAPACK, BLACS, SCALAPACK, and it has been compiled with the NEC fortran compiler. The optimal flags were:

```
-ftrace -f2003 nocbind -float0 -Cvopt -eab -R2 -P openmp
-Wf,-Ncont,-A dbl4,-P nh,-ptr byte,-pvctl noifopt
loopcnt=9999999 expand=12 fullmsg
vwork=stack,-fusion,-O noif,-init stack=nan heap=nan
```

5.19.3 *Optimisation techniques*

Technique 1: use of machine specific library (blas, lapack and fft)

A simple profiler run has shown that most of the time is spend in the BLAS subroutine, FFT and parallel data transposition required by the 3D FFT. QE was already arranged to be linked with the most optimized library available on different platforms like: essl, acml, mkl, ScaLAPACK, fftw, scilib, mass, etc. So that on each platform we choose the most appropriate library in substitution of explicit code. This requires setting makefiles with the appropriate parameter for path and preprocessor macro. Only on the SX9 we had to modify the driver for the FFT in order to compile with the mathkesian vector library available on that platform.

Linking the code with different vendor LAPACK libraries one has to pay attention that some library subroutines have the same name but different arguments, like ZHPEV in ESSL versus ZHPEV in ACML or MKL.

For QE linking with optimized libraries is mandatory to obtain the maximum performance. As an example the difference between the use of FFTW driver and ESSL driver on BG/P is reported below.

QE comes with an FFT interface to most common FFT libraries. On BG/P we tested the FFTW and ESSL FFT subroutines. Here are the results for test case GRIR443 running on 4096 cores, with arguments "-ntg 8 -input grir443_test":

timing	FFTW	essl
walltime	9m19.97s	8m53.30s
init_run	133.99s	127.67s
electrons	399.88s	380.50s
cft3s	84.63s	68.24s

Table 45: Speedup when replacing FFTW with the essl library on the Blue Gene/P system

Technique 2: vectorization

This is a QE optimisation specific for SX9 (Baku). A simple profile has shown that there was a lot of function calls (at least one call for each point of the grid) taking a lot of time. This is specific for NEC SX architecture because the calls destroy vectorization. The following routines are called a huge number of times:

- `funct.xc_spin`
- `pw_spin`
- `funct.gcx_spin`
- `hpsort_eps.hslt`
- `slater_spin`
- `pbex`
- `funct.xc_spin`

Optimisation has been implemented for the following routines: `funct.xc_spin, pw_spin, hpsort_eps.hslt, slater_spin.` Vectorization was achieved by inlining the routines in their calling functions, this enabled the compiler to vectorize the calling routines. This optimisation is highly specific for SX9 even if some improvement has been observed also on other platforms. It has required some changes in the semantics of the code, and rewriting a few hundred lines of code.

In Table 46 we have listed the code timing before and after the optimisation on NEC SX-9.

timing	Original	Optimised
walltime	22m20.61s	21m 1.63s
init_run	428.89s	399.56s
electrons	896.97s	829.65s

Table 46: Speedup when inlining frequently called routines on the SX-9. Test case GRIR443. Number of cores 1024.

Note that vectorization, reducing the number of subroutine calls, has a positive effect on other platform too. In Table 47 we report the performance gain obtained on BG/P for test case GRIR443 test using 4096 cores.

Timing	Original	Optimised
walltime	8m53.30s	7m56.76s
init_run	127.67s	125.95s
electrons	380.50s	325.43s

Table 47: Speedup when inlining frequently called routines on IBM Blue Gene/P. Test case GRIR443. Number of cores 4096.

Technique3: more shared memory parallelization

On BG/P, in order to fit into node memory, the code needs to be run in SMP mode, then it is of fundamental importance to use the Hybrid MPI+OpenMP programming model in order not to waste available cores. But OpenMP support is still new in QE (it has been implemented within PRACE in order to make QE petascaling) and many code parts do not yet execute in multithreading.

Multithreaded section of the code have been implemented looking at the code profile on a relatively low number of cores (few hundreds - few thousands), but on BG/P using many thousands of cores, new code sections, not yet ported to OpenMP, start to be relevant for performances. Then we have parallelized explicitly with OpenMP these parts. Files affected: flib/functionals.f90 Module/functionals.f90 PW/paw_onecenter.f90 PW/v_of_rho.f90

This optimisation is completely platform agnostic or, more precisely, it is relevant for all architectures with SMP nodes.

Vectorization together with the OpenMP parallelization of new sections of the code allow us to get 890,466 GFLOPs on a single SX9 node in pure OpenMP, which translates to roughly 54 % of overall peak

5.19.4 *Results of optimisation effort*

The performance gain with the optimisations applied is quite relevant on all platforms and range from a factor of 2 on BG/P to 10% on other platform like XT5. In this number we also include the petascaling improvements described in D6.4.

The optimisation of more general interest that has been implemented within the porting effort is the substitution of the FFTW driver with the ESSL driver on BG/P. In particular with an FFT grid of (168,168,192) the gain obtained in the 3DFFT high level driver of QE (cft3s) using ESSL is quite significant (see Table 45).

5.19.5 *Conclusions*

Porting and testing QE to PRACE prototypes allow us to significantly improve the overall performance and scalability of the code. In fact the different characteristics of the prototypes have pointed out new bottlenecks that never showed up before. This has forced us to look in very different directions regarding the optimisations. Nevertheless most, if not all, the optimisations give performance improvement on all platforms. At the end, we can say, that the resulting code is in general more efficient.

Further optimisation should be carried out in the parallel 3DFFT driver, especially on the data transposition performed between the FFT along z and the FFT in the xy plane. This is the part of the code whose scalability saturates first. It would be interesting to try a fully non-blocking solution, where MPI tasks send data segments to their destination as soon as they have been computed.

Looking at the platform characteristics, what could still limit the scalability of QE is the performance of the MPI alltoall and alltoallv, OS jitter, MPI latency and thread fork/join overhead.

6 Conclusions

In this document we report the results of the optimisation and porting effort of WP6 task 5. In this task the PRACE benchmark applications were ported and optimised for the WP7 prototype systems. In addition, investigations into optimal porting techniques were carried out.

This report covers the optimisation and porting of 19 applications. The applications were ported, on average, to three prototype systems.

As was to be expected, the difficulty of porting an application to a prototype depended to a large degree on how "exotic" the prototype was.

The cluster based systems, IBM Power6 cluster (Huygens) and Nehalem clusters (Juropa and Baku), posed perhaps the least problems in porting. The usual problems were encountered, such as library support and compilers producing non-working code at higher optimisation levels. For the two MPP prototypes, Cray XT5 (Louhi) and IBM Blue Gene/P (Jugene), additional problems such as cross compilation issues were encountered. The systems with 1 GB, or less, memory per core turned out to be problematic for some applications tuned for platforms with more memory.

The NEC SX-9 vector prototype (Baku) proved to be a mature platform with no major porting problems, although only a limited number of applications were ported to it. To get good performance one did need to perform source code level tuning, to help the compiler vectorise the code.

The Cell based system (Maricel) was the most challenging platform, as expected. To support the specialized SIMD processors, which provide the majority of the Cell processors computational power, significant portions of the applications had to be rewritten. Some codes were run on the Cell system by only compiling them for the slow general purpose processor in the Cell, which unfortunately gives very poor performance.

When porting an application, compiling a functioning binary is only the first step; producing an "optimal port", where the execution time is minimized through careful selection of compiler flags, external libraries, and runtime parameters, is a non-trivial problem. This problem can be addressed by collecting "best practices" and by carefully applying them to the porting process. This requires trial-and-error based experimentation, guided by intuition and experience. Alternatively the optimal porting problem could be treated as a general optimisation problem. Opfla, an iterative compilation framework developed in this task, worked well on small synthetic kernels. More advanced machine-learning techniques suitable for large-scale applications appear to be a promising research topic to enable optimal use of future peta- and exa-scale supercomputers.

Optimisation on a source code level was mostly focused on tuning the performance for the vector and Cell based systems; but work was also carried out on the other prototypes. Source level optimisation was not carried out for all applications. One reason is that, in most cases, the factor limiting performance is scalability; such work has been carried out and is reported in D6.4. Also, many codes spend the majority of their execution time in external numerical libraries. In such cases the most reasonable way to improve their performance is through "optimal porting" techniques. Finally we can also note that the applications considered here are established applications, and have thus already been optimised over the years, i.e., with regards to node-level optimisation most low hanging fruit have already been picked.

The work carried out in this task has benefitted a number of European HPC applications. The applications have been ported and optimised to the new PRACE prototype petascale systems,

and best practices on this work have been collected. This information has been disseminated in this report, but has also been communicated to the original developers, thus benefitting the HPC application community in Europe. To conclude we note that this work has prepared the way for the efficient exploitation of the upcoming Tier-0 systems.