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

D6.4
Report on Approaches to Petascaling

Final

Version: 1.0
Author(s): Mohammad Jowkar, BSC
Carlo Cavazzoni, CINECA
Georgios Goumas, GRNET
Xu Guo, EPPC

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: < D6.4></td></tr>
<tr><td colspan="2">Deliverable Nature: <DOC_TYPE: Report / Other></td></tr>
<tr><td rowspan="2">Deliverable Level:
PU*</td><td>Contractual Date of Delivery:
31 / 10 / 2009</td></tr>
<tr><td>Actual Date of Delivery:
30 / 10 / 2009</td></tr>
<tr><td></td><td colspan="2">EC Project Officer: Maria Ramalho-Natario</td></tr>
</table>

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

Document Control Sheet

<table>
<tr><td rowspan="6">Document</td><td colspan="2">Title: <Report on Approaches to Petascaling ></td></tr>
<tr><td colspan="2">ID: <D6.4></td></tr>
<tr><td>Version: <1. ></td><td>Status: Draft</td></tr>
<tr><td colspan="2">Available at: http://www.prace-project.eu</td></tr>
<tr><td colspan="2">Software Tool: Microsoft Word 2003</td></tr>
<tr><td colspan="2">File(s): D6.4.doc</td></tr>
<tr><td rowspan="4">Authorship</td><td>Written by:</td><td>Mohammad Jowkar (BSC),
Carlo Cavazzoni (CINECA),
Xu Guo (EPCC),
Giorgos Goumas (GRNET),</td></tr>
<tr><td>Contributors:</td><td>Sebastian von Alfthan (CSC),
Mauricio Araya (BSC),
Lukas Arnold (FZJ),
Raul de la Cruz (BSC),
John Donners (SARA),
Jussi Enkovaara (CSC),
Albert Farres (BSC),
Rogeli Grima (BSC),
Joachim Hein (EPCC),
Guillaume Houzeaux (BSC),
Harald Klimach (HLRS),
Paschalis Korosoglu (GRNET),
Pekka Manninen (CSC),
Martin Polak (GUP),
Orlando Rivera (LRZ),
Xavi Saez (BSC),
Sami Saarinen (CSC),
Andy Sunderland (STFC),</td></tr>
<tr><td>Reviewed by:</td><td>Aad van der Steen (Uitrecht); Dietmar Erwin, FZJ</td></tr>
<tr><td>Approved by:</td><td>Technical Board</td></tr>
</table>

Document Status Sheet

Version	Date	Status	Comments
0.1	01/April/2009	Draft	Initial version
0.2	13/October/2009	Draft	Complete document for internal WP6 internal review
1.0	26/October/2009	Final version	

Document Keywords and Abstract

Keywords:	PRACE, HPC, Research Infrastructure
Abstract:	This deliverable reports on approaches to petascaling and evaluates promising petascaling techniques and optimizations. The approach taken to achieve this goal was to port and optimize several important and highly used applications to different HPC prototypes, so as to achieve as much scalability as possible in the given time of WP6. The best practices and lessons learned by scaling each application have been documented in this deliverable. The idea is that others can draw from these experiences when scaling their own applications. The above applications are from the scientific community and were chosen in tasks 6.1 and 6.2 so as to cover a broad range of scientific areas, and are considered to be representative of the European HPC usage. Furthermore each application has been ported and optimized to several different architectures to get a better understanding of the suitability of the applications on different architectures and vice versa. The work done in task 6.4 will together with task 6.5 be used in task 6.3 to create a benchmark set. Task 6.3 in return, will be used by task 5.4 for evaluating and comparing potential future petaflop/s systems.

Copyright notices

© 2008 PRACE Consortium Partners. All rights reserved. This document is a project document of the PRACE project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the PRACE partners, except as mandated by the European Commission contract RI-211528 for reviewing and dissemination purposes.
All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.

Table of Contents

Project and Deliverable Information Sheet i
Document Control Sheet ii
Document Status Sheet ii
Document Keywords and Abstract iii
Table of Contents iv
List of Figures vii
List of Tables viii
References viii
List of Acronyms and Abbreviations x
Executive Summary 1
1 Introduction 2
1.1 Structure of the report 2
2 Approaches to Petascaling 3
2.1 Load Balancing 3
2.1.1 *Static* 4
2.1.2 *Dynamic* 6
2.2 Parallel I/O 8
2.2.1 *Parallel NetCDF* 9
2.2.2 *MPI-IO* 11
2.2.3 *GPFS* 12
2.3 Checkpointing 13
2.3.1 *Diskless Checkpointing* 13
2.3.2 *User-Directed Checkpointing* 14
2.3.3 *Compiler-Assisted Checkpointing* 14
2.3.4 *Fast Checkpoint Compression* 14
2.4 Hybrid Parallelization 14
2.4.1 *Main Rules for Mixed Programming* 16
2.4.2 *Analysis of Hybrid Parallelization in QuantumEspresso* 16
2.4.3 *Limits of MPI* 18
2.4.4 *Hybridization strategy* 18
2.4.5 *Implementation design* 21
2.4.6 *Other explicit OpenMP parallelization.* 23
2.4.7 *Performance of the hybrid code* 24
2.4.8 *Conclusion* 24
2.5 Minimizing of Communication Overheads 25
2.5.1 *Point-to-Point Communications* 26
2.5.2 *Collective Communications* 28
2.5.3 *Logical Communication Mapping in the Physical Communication Layer* 28
3 Applications 30
3.1 Alya 30
3.1.1 *Application description* 30
3.1.2 *Petascaling Techniques* 32
3.1.3 *Results* 35
3.1.4 *Conclusions* 35
3.2 AVBP 36

3.2.1 *Application description* 36
3.2.2 *Petascaling techniques* 36
3.2.3 *Results* 37
3.2.4 *Conclusions* 38
3.3 BSIT **39**
3.3.1 *Application description* 39
3.3.2 *Petascaling techniques* 41
3.3.3 *Results* 43
3.3.4 *Conclusions* 43
3.4 Code_Saturne **44**
3.4.1 *Application description* 44
3.4.2 *Petascaling techniques* 48
3.4.3 *Results* 51
3.4.4 *Conclusions* 55
3.5 CP2K **56**
3.5.1 *Application description* 56
3.5.2 *Petascaling techniques* 57
3.5.3 *Results* 57
3.5.4 *Conclusions* 58
3.6 CPMD **59**
3.6.1 *Application description* 59
3.6.2 *Petascaling techniques* 59
3.6.3 *Results* 59
3.6.4 *Conclusions* 60
3.7 ECHAM5 .... **61**
3.7.1 *Application description* 61
3.7.2 *Petascaling techniques* 61
3.7.3 *Results* 62
3.7.4 *Conclusions* 62
3.8 EUTERPE **63**
3.8.1 *Application description* 63
3.8.2 *Petascaling techniques* 64
3.8.3 *Results* 65
3.8.4 *Conclusions* 65
3.9 GADGET **66**
3.9.1 *Application description* 66
3.9.2 *Petascaling techniques* 67
3.9.3 *Results* 67
3.9.4 *Conclusions* 71
3.10 GPAW **72**
3.10.1 *Application description* 72
3.10.2 *Petascaling techniques* 74
3.10.3 *Results* 75
3.10.4 *Conclusions* 76
3.11 GROMACS **76**
3.11.1 *Application description* 76
3.11.2 *Petascaling techniques* 77
3.11.3 *Results* 78
3.11.4 *Conclusions* 81
3.12 HELIUM **81**
3.12.1 *Application description* 81
3.12.2 *Petascaling techniques* 82

3.12.3 *Results* ... 84
3.12.4 *Conclusion* ... 89
3.13 NAMD ... 90
3.13.1 *Application description* ... 90
3.13.2 *Petascaling techniques* ... 91
3.13.3 *Results* ... 92
3.13.4 *Conclusions* ... 95
3.14 NEMO ... 96
3.14.1 *Application description* ... 96
3.14.2 *Petascaling techniques* ... 97
3.14.3 *Results* ... 98
3.14.4 *Conclusions* ... 98
3.15 NS3D ... 98
3.15.1 *Application description* ... 98
3.15.2 *Petascaling techniques* ... 99
3.15.3 *Results* ... 100
3.15.4 *Conclusions* ... 103
3.16 Octopus ... 104
3.16.1 *Application description* ... 104
3.16.2 *Petascaling techniques* ... 105
3.16.3 *Results* ... 105
3.16.4 *Conclusions* ... 107
3.17 PEPC ... 108
3.17.1 *Application description* ... 108
3.17.2 *Petascaling techniques* ... 109
3.17.3 *Results* ... 111
3.17.4 *Conclusions* ... 112
3.18 SIESTA ... 112
3.18.1 *Application description* ... 112
3.18.2 *Petascaling techniques* ... 113
3.18.3 *Results* ... 114
3.18.4 *Conclusions* ... 115
3.19 QCD ... 115
3.19.1 *Application description* ... 115
3.19.2 *Petascaling techniques* ... 116
3.19.3 *Results* ... 117
3.19.4 *Conclusions* ... 117
3.20 Quantum_Espresso ... 118
3.20.1 *Application description* ... 118
3.20.2 *Petascaling techniques* ... 119
3.20.3 *Results* ... 121
3.20.4 *Conclusions* ... 124
4 Summary ... 125

List of Figures

Figure 1 I/O software stacks provide the connection between applications and I/O hardware.............. 8
Figure 2 NetCDF File Structure 10
Figure 3 Partitioning a file among parallel processes 11
Figure 4 Speed-up of the Linear Algebra(subtask of a Car-Parrinello simulation of 256 water molecules.. 20
Figure 5 Speed-up of the ad-hoc 3D FFT subtask of a Car-Parrinello simulation of 256 water molecules. Grid size is 200x200x200. See text for comment. 22
Figure 6 Speed-up relative to 64 cores of a Car-Parrinello simulation of 256 water molecules 24
Figure 7 Comparisons of classical CG with Deflated CG. (Left): 2D thermal and turbulent tall cavity. 34
Figure 8 Comparisons of AllReduce and AllGather for the coarse space vector in the deflated CG.... 34
Figure 9 Performance of AVBP 37
Figure 10 Master-Worker Scheme 39
Figure 11 RTM two-way wave propagation 40
Figure 12 RTM time breakdown 40
Figure 13 Double-buffering technique 42
Figure 14 Ghost Cell Method 46
Figure 15 T-junction Dataset 47
Figure 16 Mixing Grid Dataset 47
Figure 17 I/O Overheads of Serial I/O on the Cray XT4 50
Figure 18 Parallel MPI/IO implementation in Code_Saturne v.2.0 50
Figure 19 Relative Performance of Conjugate Gradient and Multigrid on the Cray XT4 52
Figure 20 Parallel Performance of Mesh Partitioning software on the IBM BG/P 53
Figure 21 Parallel Scaling of Code_Saturn on the PRACE Prototype Systems for the 10M cell T-Junction dataset 54
Figure 22 Scalability of CPMD 60
Figure 23 Gadget Scaling Behaviour on Huygens 69
Figure 24 Gadget Scaling Behaviour on Jugene 70
Figure 25 Speedup of shared memory all-to-all routine compared to the original MPI routine. 78
Figure 26 Speedup of shared memory all-to-all vector routine compared to the original MPI routine. 79
Figure 27 Speedup of shared memory all-gatherl routine compared to the original MPI routine......... 79
Figure 28 Speedup of shared memory all-gather vector routine compared to the original MPI routine. 80
Figure 29 HELIUM scaling performance on the prototype Cray XT5 (Louhi) 85
Figure 30 HELIUM scaling cost on the prototype Cray XT 5 (Louhi) 85
Figure 31 HELIUM scaling performance on the prototype Power6 (Huygens) 86
Figure 32 HELIUM scaling cost on the prototype Power6 (Huygens) 86
Figure 33 HELIUM strong scaling performance on the prototype BG/P (Jugene) 88
Figure 34 HELIUM strong scaling cost on the prototype BG/P (Jugene) 88
Figure 35 HELIUM weak scaling performance on the prototype BG/P (Jugene) 89
Figure 36 HELIUM weak scaling cost on the prototype BG/P (Jugene) 89
Figure 37 Computational cost per step for the MPP-Cray prototype forNAMD 2.7b1 with memory reduction 94
Figure 38 Computational cost per step for the SMP-FatNode-pwr6 prototype when using NAMD 2.7b1 94
Figure 39 Wall clock time required for a 10ns simulation for NAMD 2.7b1 with memory reduction. 95
Figure 40 Effect of shared memory parallelism before and after tunings on a single NEC-SX9 node101
Figure 41 Analysis of several task per process combinations on 8 and 12 NEC-SX9 nodes............. 102
Figure 42 Scaling before and after optimisation on the NEC-SX9 processors in comparison to ideal scaling of the original code 103
Figure 43: Scaling of PEPC-B on target architectures 110

Figure 44: Scaling of PEPC-E on MPP-BG architecture 110
Figure 45: QCD kernel scaling on MPP-BGP. All kernels, but KC, are run for strong scaling. 116
Figure 46: QCD kernel scaling on SMP-FatNode-pwr6. All kernels, but KC, are run for strong scaling. 116
Figure 47 QCD kernel scaling on MPP-Cray. All kernels, but KC, are run for strong scaling..... 117

List of Tables

Table 1: Data Access Routines 12
Table 2 Total computation duration in seconds for helicopter turbine benchmark 37
Table 3 Total computation duration in seconds for the helicopter turbine benchmark (low mem) 38
Table 4 Experimental Speed on BAKU(HLRS) 512 x106 particles 71
Table 5 Experimental Speed on BAKU(HLRS) 64 x106 particles 71
Table 6 Speedup of Gromacs using the shared-memory MPI_Alltoal on the Cray XT5 prototype..... 80
Table 7 Speedup of shared-memory MPI_Alltoall. NP is the total number of cores 81
Table 8 MPI profiling comparison on Cray XT5 before and after removing Test_MPI. 84
Table 9 Routine profiling comparison on Cray XT5 before and after merging loops. 84
Table 10 MPI profiling comparison on Power6 before and after removing Test_MPI. 86
Table 11 NAMD memory footprint 93
Table 12 Results for C240. 106
Table 13 Results for 650-atom light-harvesting complex. 107
Table 14 GRIR443 test run using hybrid and pure MPI code 121
Table 15 AUSURF112 test run using hybrid and pure MPI code 122
Table 16 Test case GRIR686 122

References

1 *NAMD2: Greater Scalability for Parallel Molecular Dynamics*, L. Kalé, et al., Jounal 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 Prace D6.3.1: *Report on available Performance Analysis and Benchmark Tools, Representative Benchmarks.*
6 http://www.ks.uiuc.edu/Research/namd/wiki/index.cgi?NamdMemoryReduction
7 Prace D6.5: Optimisation report
8 J.S. Plank and K. Li, Faster Checkpointing with N+1 Parity, 24th International Symposium on Fault-Tolerant Computing, Austin, TX, June, 1994, pp 288--297
9 J.S Plank et al., Libckpt: Transparent Checkpointing under Unix, Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995, pp. 213—223
10 J.S Plank et al., Compiler-Assisted Memory Exclusion for Fast Checkpointing, IEEE Technical Committee on Operating Systems and Application Environments, 7(4), Winter 1995, pp 10-14
11 J.S Plank et al., Compressed Differences: An Algorithm for Fast Incremental Checkpointing, Technical Report CS-95-302, University of Tennessee, August, 1995

12 T. Sterling, P. Messina, and P. H. Smith. Enabling technologies for petaops computing. MIT Press, 1995
13 F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing, page 12. IEEE Computer Society, 2000. L. L. Ewing and A. Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In R. Eigenmann and B. R. de Supinski, editors, International Workshop on OpenMP, volume 5004 of Lecture Notes in Computer Science, pages 36{47. Springer, 2008
14 R. Rabenseifner and G. Wellein. Communication and optimization as-pects of parallel programming models on hybrid architectures. Interna-tional Journal of High Performance Computing Applications, 17:49-62,2003. G. Krawezik and F. Cappello. Performance comparison of mpi and openmp on shared memory multiprocessors: Research articles. Con-currency and Computation: Practice and Experience, 18:29 61, 2006
15 B. V. Protopopov and A. Skjellum. A multithreaded message passing interface (MPI) architecture: performance and program issues. Journal of Parallel and Distributed Computing, 61:449-466, 2001. H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In Proceedings of 15th ACM International Conference on Supercomputing, pages 381-392. ACM Press, 2001
16 R. Rabenseifner. Some aspects of message-passing on future hybrid systems (extended abstract). In Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 8{10. Springer-Verlag, 2008
17 P. Giannozzi et al. J. Phys.: Condens. Matter 21, 395502 (2009)
18 R. Aubry, G. Houzeaux and M. Vázquez . Parallel performance of the Deflated Conjugate Gradient. PARCFD conference. Moffett Field (USA), May 18-22, 2009
19 F. Mut, R. Aubry, J. Cebral, R. Löhner and G. Houzeaux. Deflated Preconditioned Conjugate Gradient Solvers: Extension and Improvements. Submitted to 48th AIAA Aerospace Sciences Meeting, Orlando (USA), Jan. 4-7, 2010
20 "On the scaling of computational particle physics codes on cluster computers" by Z. Sroczynski, N, Eicker, Th. Lippert, B. Orth and K. Schilling, http://arxiv.org/abs/hep-lat/0307015v2
21 Prace D6.1: Identification and Categorisation of Applications and Initial Benchmarks Suite

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-RechenZentrum
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
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).

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.

Executive Summary

This document reports the results of Task 6.4 of Work Package 6 of the PRACE project. The overall objectives of WP6 are to identify and understand the software libraries, tools, and skills required by developers to ensure that their applications can use a petaflop/s system productively and efficiently. Task 6.4 explored and documented promising petascaling techniques. Related work has been done in Task 6.5 on serial optimization techniques and Task 6.6, which explores software libraries and programming models suitable for petascaling.

The approach taken to reach the objectives of Task 6.4 was to scale a set of applications to a large number of processors as possible on the PRACE prototypes. The scientific areas range from chemistry to geophysics codes (see [21]). The applications were chosen in tasks 6.1 and 6.2 by surveying leading HPC centres in Europe. Based on this survey the following 20 applications were chosen to be representative of codes used in HPC centers in Europe:

Alya, AVBP, BSIT, Code_Saturn, CP2K, CPMD, Echam5, EUTERPE, Gadget, GPAW, Gromacs, HELIUM, NAMD, NEMO, NS3D, Octopus, PEPC, SIESTA, QCD and Quantum_Expresso.

Each of the above applications was evaluated and scaled by a small team of researches from the PRACE partners, headed by a *Benchmark Code Owner* (BCO). The applications were scaled on the PRACE prototypes, which represent the current top of the line supercomputer architectures. This work has resulted in twenty application reports written by the BCOs and their teams, detailing their findings, while working with the applications. These document experiences, parallel optimizations, best practices, encountered bottlenecks, and future scaling work. This lays the groundwork for reaching petaflop/s performance on the future PRACE Tier-0 petascale machines. Furthermore other people wishing to scale their codes can use these reports, the synthesis of techniques given in chapter 2, and the experiences of the BCOs.

In summary, the petscaling of twenty applications has shown that it is not possible to recommend a single set of generic optimizations that is suitable for all codes, there are some which show good scalability for many codes or that should be pursued further in the future. One example is hybrid parallelization, which demonstrated promising results, although surprisingly few of the applications employ this technique fully at the moment. However, many are planning to do so in the future. Since each application has been ported to several of the prototypes systems, the application evaluation shows the suitability of a given architecture for running a particular code.

The above work was done in close collaboration with the main developers of the applications and useful optimizations have been reported back, so they can be merged in the main branch of the codes.

1 Introduction

To achieve petascaling is very hard and relatively few codes have managed to petascale to this date. However, as HPC machines are rapidly reaching and even surpassing the one petaflop barrier, more and more codes need to petascale to take advantage of such systems. There is therefore a need to understand the various approaches which are required to achieve applications scaling to petaflop/s performance. To achieve this goal, it is necessary to understand the limitations of scaling and to know which strategies are useful to explore to reach petascaling.

The Partnership for Advanced Computing in Europe (PRACE) has the overall objective to prepare for the creation of a persistent pan-European HPC service. PRACE is divided into a number of inter-linked work packages, and WP6 focuses on `Software Enabling for Petaflop/s Systems` and therefore addresses the above. This document is part of WP6 and its objective is to investigate the approaches required for applications to petascale and to explore promising petascaling techniques. The best way to do this is to take common HPC applications and optimize them on petaflop/s systems in order to understand the best practices to petascaling. Since PRACE did not have any petaflop/s systems at the beginning of this task, all petascaling work had to be done on prototypes. These prototypes represent promising architectures which might be expanded to petaflop/s size in the future PRACE project. The following systems are the prototypes, which were used:

- Jugene (BlueGene/P @ FZJ),
- Louhi (Cray XT5 @ CSC),
- Huygens (IBM p575 P6 @ SARA),
- Maricel (IBM PowerXCell @ BSC),
- HLRS-Vector (NEC SX-9 @ HLRS).

The processor types and site are given in parentheses.
The above prototypes are covered in detail in deliverable D5.2. It should be noted that this report can not be used to compare prototypes. It only addresses how to achieve parallel scalability on different platforms. The scaled and ported applications, however, will be used as a PRACE benchmarking suite (PABS) to assess prototypes in other WPs. PABS is being compiled in Task 6.3. For a comparison of prototypes see deliverable D5.4.

1.1 Structure of the report

This document has three main sections. In Chapter 2 five petascaling techniques which have been used by the Benchmark Code Owners (BCOs) to scale their application are described.

Chapter 3 contains the individual application scaling reports, written by the BCOs, which describe the actual techniques used to scale the applications and the resulting performance gains. This includes the effectiveness of each optimization, best practices, bottlenecks, and challenges in scaling. These reports are between 5-10 pages each.

Chapter 4 gives a preliminary conclusion of the work. It must be noted, that the size of the prototype systems is not at the petaflop/s level. The results indicate promising trends. The actual hard work to scale important application is yet to come.

2 Approaches to Petascaling

A petascale computer has a large number of processors, usually between 10^4 and 10^5 processors. This large number of processors worsens the common scaling problems caused by load imbalance, communication overheads, parallel I/O etc. The objective of task 6.4 of the PRACE project is to understand and mitigate these kinds of bottlenecks. This chapter distils the twenty application scaling reports from the BCOs to cover some important and common optimization techniques used to overcome bottlenecks, which typically limit scalability. However, before going any further, it is important to specify what is meant by optimization techniques. This report only covers optimization techniques which results in *out of core scalability*, such as node to node scalability. On core optimization techniques such as loop unrolling, vectorization etc. are equally important in reaching petascaling performance and are covered by deliverable 6.5.

The topics which have been distilled in this chapter are: *load balancing*, *parallel I/O*, *checkpointing*, *hybrid parallelization* and *minimization of communication overheads*. These are some of the most common approaches which have been used in the application reports from Chapter 3, to scale applications. It should be noted that these techniques are only considered best practice and are not necessarily suitable for all applications.

2.1 Load Balancing

Load imbalance is the source of performance degradation for a large variety of applications, since it leads to underutilised CPU resources. Load balancing in a petascale parallel system is an intricate task that needs to consider several factors such as even distribution of computation and communication, respect of dependence constraints and dynamic changes in the workload of the application. Load balancing in high performance computing has the ultimate goal of minimizing the overall parallel completion time of the application by efficiently utilising the execution platform's available resources (CPU, communication links, etc.).

Load balancing schemes used in parallel applications can be broadly classified into two categories: *static* and *dynamic*. If the workload pattern and the nature of imbalance are known or can be accurately predicted before the execution of the application, static balancing techniques are more appropriate. On the other hand, if the load variances occur during the execution of the application and are heavily dependent on dynamic input and calculations, the decisions on load balancing should be taken dynamically. Dynamic load balancing in this case is more efficient, but one needs to consider the cost of dynamic task orchestration and carefully design the balancing schemes to avoid the creation of hot spots in the execution of the parallel algorithm.

Load imbalance may occur in a variety of parallel applications. Algorithms expressed by loops with non-constant bounds lead to varying workloads per loop iteration. Linear algebra decompositions (e.g LU) involve a much larger number of computations on the lower-right part of the matrix than those that are performed on the upper-left. Naïve data and computation distribution in these cases can lead to severe load imbalance. Another classical example for the need of load balancing in scientific computing is the iterative solution of large sparse systems in parallel computers. Iterative solvers involve the multiplication of a sparse matrix with a dense vector. The distribution of the rows of the matrix to the processing nodes of the parallel platform needs to ensure both that equal computational load and the same volume of

data and number of communication messages are assigned to each node. This is formulated as a graph partitioning problem.

In several applications, as the simulation progresses different parts of the computational mesh has to be refined. This means that even if the initial workload distribution is balanced, dynamic mesh changes lead to load imbalances during the execution. In general, the simulation of dynamic systems (e.g. N-body simulations) very frequently leads to the need for dynamic and adaptive load balancing schemes. Finally, algorithms and computations performed on irregular data structures (trees, graphs, linked lists, etc) may also lead to load imbalance since the initial assignment of work cannot guarantee fair evolution of operations in all participating processing elements.

The following paragraphs describe several load balancing techniques that belong to the two aforementioned families of balancing strategies, static and dynamic. Although the general philosophy of load balancing remains the same, each application has its own features that favour specific approaches to be followed.

2.1.1 *Static*

Static load balancing is implemented with static mapping of tasks and data among processes prior to the execution of the algorithm. The choice of a good mapping in this case depends on several factors such as prior knowledge of workload and interactions between tasks. Even in the simple case that task loads are known, the problem of obtaining the optimal mapping is NP-complete for non-uniform tasks. However, for many practical cases heuristics can provide fairly acceptable solutions to the load balancing problem. Static mapping and consequently load balancing is frequently achieved by schemes distributing loop iterations, data (e.g. chunks of an array) or tasks (e.g. nodes of a task-dependency graph) to the available processes.

Loop scheduling requires to solve the problem of assigning proper iterations of parallelizable loops among n processors to achieve load balancing and with minimum dispatch overhead. In static loop scheduling the loop iteration space is divided into n chunks and each chunk is assigned to a processor. The volume of each chunk in static scheduling is based on the knowledge of the iteration space before the execution and follows the simple rule that each chunk of iterations assigned per process should contain the same volume of computations and not necessarily the same number of iterations. If a priori knowledge of the iteration space does not exist, then static loop scheduling could result in load imbalance. In this case dynamic schemes should be employed to achieve balancing.

Load balancing schemes that are based on *data partitioning* are suitable for algebraic computations on matrices implemented in a high-level programming language as 2-dimensional arrays. The data partitioning actually induces task decomposition, since data that are assigned to one processing element lead to the assignment of computations (writes) on them. This is known as the “owner-computes” rule. The most commonly used approaches for data decomposition, which in this case coincides with array distribution, are the following:

Block distributions are the most straightforward way to scatter array elements among the available processing elements. Following this scheme one assigns uniform contiguous portions of the array to different processes. Thus, the 2-dimensional array is distributed among the processes by chunks of rows, columns or blocks. This scheme achieves load balancing when the computations of the algorithm are evenly distributed among the array elements as, for example, in the case of matrix multiplication. This distribution scheme can be generalized for d-dimensional arrays. The choice between one, two or even higher dimensional distributions is not guided by load balancing criteria in this case, but rather by

other criteria such as cache utilization, communication pattern and good utilization of the available processes.

Cyclic and Block Cyclic distributions are used when the amount of computation is different between the elements of the array. In this case block distribution will lead to load imbalance. The classic example in this case is LU factorization of a matrix in which computation increases from the top left to the bottom right of the matrix. The central idea behind cyclic distributions is to assign chunks of elements cyclically to the available processes, so that each process has an almost equal set of elements in each region of the array. Block cyclic distributions are quite general, since block and cyclic distributions are special cases of it.

A large number of partitioning and load balancing schemes have been proposed for computations that can be described by *task graphs.* A task graph is a graph in which each node represents a task to be performed, while each edge represents dependences between the tasks. Each node has a metric associated with the task execution time, while edge weights can be used to express the required communication from one task to another. A large variety of applications can be described following the task graph model. The most characteristic example in scientific computing is the computation on meshes (e.g. in the Finite Element Method), where the computations on the mesh points are the task graph nodes and the connection between the elements lead to the task graph edges. In such scientific simulations, the structure of the computation evolves from time step to time step. These simulations require decompositions of the mesh (task graph) prior to the start of the simulation. The decomposition typically needs to fulfill two significant properties: *load balance* and *minimization of communication.* The distribution of mesh elements (task graph nodes) between processing nodes with the goal to keep the computations between the processors as even as possible, minimizing simultaneously the number of edges that cross-cut the partition's boundaries (*edge-cut)* is the well-known *graph partitioning* problem. The graph partitioning problem is known to be NP-complete. Therefore, in general it is not possible to compute optimal partitionings for graphs of interesting size in a practical amount of time. This fact has led to the development of numerous heuristic approaches.

Geometric techniques compute partitionings based on the coordinate information of the initial mesh nodes, and ignore the connectivity between the mesh elements. The goal of geometric techniques is to form groups of vertices that are spatially close to each other, whether or not these vertices are connected. Since the edge-cut metric is irrelevant for this family of techniques, interprocessor communication due to parallel processing is minimized by using an alternative metric, i.e. the number of mesh elements that are adjacent to nonlocal elements. Typically, geometric partitioners are extremely fast. However, they tend to compute partitionings of lower quality (in terms of interprocessor communications) than schemes that take the connectivity of the mesh elements into account. Coordinate nested dissection (CND), Recursive Inertial Bisection (RIB) and Space-Filling Curves schemes fall into the family of geometric techniques.

Combinatorial partitioners use an opposite notion to that of geometric partitioners. They attempt to group together highly connected vertices, whether or not these are close to each other in space. That is, combinatorial partitioning schemes compute a partitioning based only on the adjacency information of the graph and do not consider the coordinates of the vertices. For this reason, the partitionings produced typically have lower edge-cuts (interprocessor communication). However, combinatorial partitioners are much slower than geometric partitioners. The levelized nested dissection (LND) and the Kernighan–Lin/Fiduccia–Mattheyses Algorithm (KL/FM) are typical combinatorial partitioning techniques.

Another method of solving the problem is to formulate it as the optimization of a discrete quadratic function. However, even with this new formulation, the problem is still too difficult

to solve in practical times. To deal with this, a class of graph partitioning methods, called *spectral methods*, relaxes this discrete optimization problem by transforming it into a continuous one. The minimization of the relaxed problem is then solved by computing the second eigenvector of the discrete Laplacian of the graph.

A new class of partitioning algorithms is based on the *multilevel* paradigm. This approach is based on three distinct phases: graph coarsening, initial partitioning, and multilevel refinement. In the graph-coarsening phase, a series of graphs are constructed by grouping together selected vertices of the input graph in order to form a related coarser graph. This newly constructed graph then acts as the input graph for another round of graph coarsening, and so on, until a sufficiently small graph is finally obtained. Computation of the initial bisection is performed on the coarsest of these graphs and is very fast. Finally, partition refinement is performed on each level graph, from the coarsest to the finest (i.e., original graph) using a KL/FM-type algorithm.

The various graph partitioning strategies have advantages and disadvantages concerning the quality of the final partition, the partitinioning time, the parallelism and the applicability for certain families of graphs or meshes. In several cases, *hybrid* or *combined schemes* can lead to a better result for the input graph under consideration. For example, an initial partitioning can be computed by a fast geometric method, and then the relatively low-quality partitioning can be refined by a KL/FM algorithm.

A large variety of software tools are publicly available offering a wide choice of graph partitioners. The *METIS* package uses multilevel algorithms and has a parallel version (*ParMETIS*). The *Chaco* package offers a variety of algorithms, geometric, combinatorial and multilevel. The *JOSTLE* package also uses multilevel refinement schemes. The *PARTY* software library uses combined approaches and offers a large variety of algorithms, also interfacing to the Chaco packages. The *SCOTCH* library is based on static partitioning implemented by the Dual Recursive Bipartitioning (DRP) approach, enhanced with multi-level schemes as well. The *S-HARP* package uses spectral methods and provides a mixed-mode parallel version for fine-grain and coarse-grain parallelism.

2.1.2 *Dynamic*

In several cases during the simulation of a physical process, the computational grid may change. These are the cases when some areas of interest within the computational grid need to be more fine-grained to capture the process with higher accuracy, or when the computational domain changes structurally. In this kind of simulations the initial distribution of computations to the processors of a parallel platform may result to be unbalanced and redistribution needs to be carried out to accomplish rebalance. This dynamic load balancing can be achieved by using a graph partitioning algorithm. This problem is referred to as *adaptive graph partitioning* to differentiate it from the static graph-partitioning problem that arises when the computations remain fixed, as explained in the previous paragraph.

A repartitioning of a graph can be computed by simply partitioning the new graph from scratch. Since no consideration is given to the existing partitioning, it is unlikely that vertices will be assigned to their original subdomains with this method. Therefore, this approach will tend to require much more data redistribution than is necessary in order to balance the load. An alternate strategy is to attempt to adjust the input partitioning just enough to balance it. This can be achieved using the following simple repartitioning method: subdomains that suffer from excess load are relieved from this load, which in turn is assigned to subdomains with lower load. Adjacency of interacting domains is not taken into consideration. This method will optimally minimize data redistribution, but it can result in significantly higher

edge-cuts compared with more sophisticated approaches and will typically result in disconnected subdomains. For these reasons, it is usually not considered a viable repartitioning scheme for most applications. A better approach is to use a diffusion-based repartitioning scheme. These schemes attempt to minimize the data redistribution costs while significantly decreasing the possibility that subdomains become disconnected.

Load imbalance also occurs in parallel tasks with substantially different and difficult to predict execution times. This problem can be considered as a fully parallel loop of the form:

```
for (i=0; i < N; i++)    /* parallel loop */
    do_task(i,input_variable_list);
```

If function `do_task()` does not have constant execution time for the values of its input parameters, or its execution time cannot be estimated in any way, then the straightforward distribution of the above parallel loop in processing elements may lead to severe imbalance. If P processing elements are available, the straightforward, static approach would be to assign N/P chunks of the above loop to each processor. However, several dynamic scheduling approaches have been proposed that lead to a substantially more balanced distribution of the workload than the static one.

Dynamic scheduling schemes are either *centralized* or *distributed.* In the centralized case, a master processor coordinates the task distribution by communicating with the worker processors that execute the tasks and providing chunks of work dynamically. The simplest self-scheduling algorithm, called *Self-Scheduling* (SS) assigns just one task to each worker per request. This algorithm achieves almost perfect load balance. All workers are expected to finish at nearly the same time, with maximum difference of a task execution time. However, SS may suffer from excess scheduling overheads. *Chunk Self-Scheduling* (CSS) assigns constant size chunks to each worker. The chunk size is usually chosen by the user. A large chunk size reduces scheduling overhead, but at the same time increases the chance of load imbalance. As a compromise between load imbalance and scheduling overhead, other schemes start with large chunks to reduce the scheduling overhead, which are gradually reduced in size throughout the execution to improve load balancing. These schemes are known as reducing chunk size algorithms.

In *Guided Self-Scheduling* (GSS), each worker is assigned a chunk given by the number of remaining tasks divided by the number of workers. GSS allocates most of the work in the first few scheduling steps and the amount of the remaining work is not adequate to balance the workload, so that in some cases the load balancing achieved by GSS is poor. The *Trapezoid Self-Scheduling* (TSS) scheme linearly decreases the chunk size. In TSS the first and last chunk size pair may be set by the programmer. All the aforementioned dynamic scheduling methods provide flexibility concerning the tradeoff between load-balancing and scheduling overheads. Depending on the nature of a particular application (i.e. the minimum and maximum computation times of chunks, the distribution of loads among chunks, the scheduling overhead times, etc.) and the features of the underlying computational platform, a particular method may outperform the rest in terms of total parallel execution time.

Clearly, centralized approaches may face a severe communication bottleneck, since the master that orchestrates the scheduling process constitutes a hotspot in the execution of the application. In petascale environments where thousands of processing elements need to receive scheduling information, centralized dynamic scheduling schemes cannot serve as a viable scheduling approach, especially when also one considers fault tolerance. Distributed dynamic scheduling algorithms are executed on each worker processor and are based on local information. This strategy does not create any hot spots in the parallel execution and is more fault tolerant. However, distributed approaches pay the cost of suboptimal load balance.

Standard distributed scheduling approaches are based on the concept of *work-stealing*. All processors that participate in the execution maintain a local queue of assigned jobs. If one processor becomes idle, i.e. its local queue is empty; it tries to steal work from the queues of other processors. The selection approach of the target processor designates to a large extent the success of a particular scheduling scheme. Random, round-robin and more sophisticated techniques can be applied to select the target processor, according to the application features and the ability to keep and possibly exchange scheduling information. Some of the issues that need to be taken care of in this kind of schemes are the following: (a) the consistency of the local work queues (e.g. how simultaneous requests to a work queue are handled) (b) the decision whether it pays off to apply work stealing (e.g. when the computations are rather well balanced) given that on one hand work stealing leads to communication needs and on the other hand it removes the job from the cache of processor in charge, to allocate it in new data structures of the new processor.

2.2 Parallel I/O

Due to the large size of petascale systems, data can quickly grow to terabytes in a typical run. This data often needs to be saved or read periodically and it is therefore mandatory to have reasonable I/O performance to obtain good scalability of applications. This section will described how this can be done on parallel computers.

Parallel I/O systems combine many individual components (e.g. disks, servers, network links) together into a coherent whole, used to provide high aggregate I/O performance to parallel applications. Many improvements have been made in I/O systems, both by the HPC community and by outside groups. One key category of improvements has been in the organization of I/O software and in defining standard interfaces to various layers, both software and hardware. I/O software has moved from monolithic serial I/O libraries to software stacks with at least three layers: parallel filesystem, I/O middleware, and high level I/O interface. These three layers are shown in Figure 1.

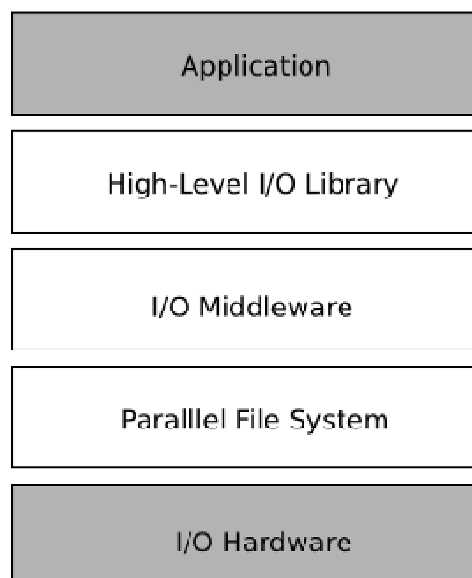

Figure 1 I/O software stacks provide the connection between applications and I/O hardware

Each of these three layers provides a subset of the overall functionality.

The high-level I/O library is responsible for applying structure to files in order to maintain a self-describing, portable data container and present a data abstraction to the application programmer that is close to the model used in the application. Examples of such a library are Parallel netCDF and HDF5. Parellel netCDF is built on top of the MPI-IO interface, which is part of the MPI-2 standard. The HDF5 high-level I/O library also layers on top of MPI-IO.

The second key component of an I/O stack is I/O middleware. This component is responsible for providing the base on which high-level I/O libraries may be built. This layer provides a mapping from the relatively simple interfaces of parallel filesystems into an interface that introduces concepts from the programming model, such as communicators and datatypes in the MPI programming model. The best example of I/O middleware is the MPI-IO interface.

The parallel file system (PFS) is responsible for managing all the storage hardware components. It presents a single logical view of such hardware, which can be used by the other software layers. It also enforces a consistency model so that the results of concurrent access are well-defined. One example of parallel file systems is the General Parallel File System (GPFS) from IBM.

The next sections in this chapter will cover implementations of each of the above mentioned layers.

2.2.1 *Parallel NetCDF*

NetCDF is an abstraction that supports a view of data as a collection of self-describing, portable, array-oriented objects that can be accessed through an interface. It stores data in an array-oriented dataset, which contains dimensions, variables, and attributes. Physically, the dataset file is divided into two parts: file header and array data. The header contains all information (or metadata) about dimensions, attributes, and variables except for the variable data itself, while the data part contains arrays of variable values.

The header part describes each variable by its name, shape, named attributes, data type, array size, and data offset, while the data part stores the array values for one variable after another, in their defined order. For fixed-size arrays, each array is stored in a contiguous file space starting from a given offset. To support variable-sized arrays netCDF introduces record variables. All record variables share the same unlimited dimension as their most significant dimension and are expected to grow together along this dimension. The other, less significant dimensions all together define one record of the variable. Figure 2 shows the storage layouts in a netCDF file.

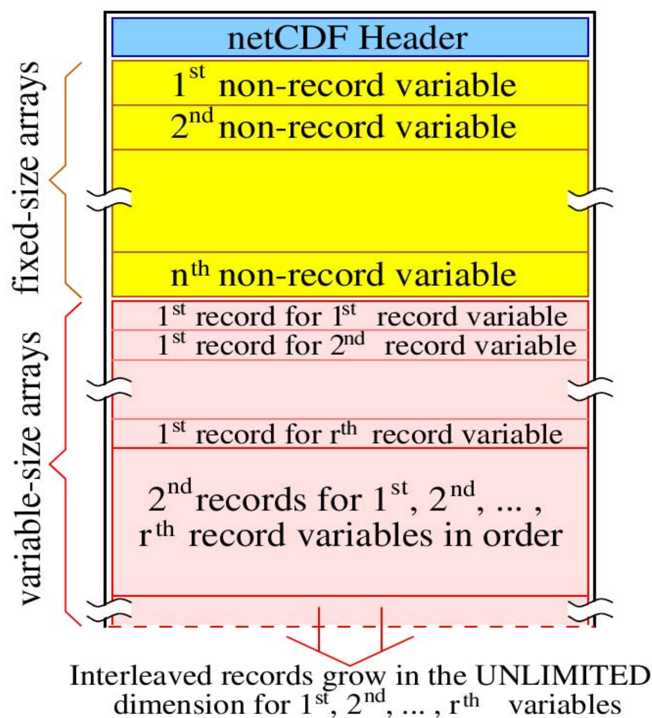

Figure 2 NetCDF File Structure

The original netCDF API was designed for serial codes to perform netCDF operations through a single process. In the serial netCDF library, a typical sequence of operations to write a new netCDF dataset is to create the dataset; define the dimensions, variables, and attributes; write variable data; and close the dataset. Reading an existing netCDF dataset involves first opening the dataset; inquiring about dimensions, variables, and attributes; reading variable data; and closing the dataset.

The original design of the netCDF interface is proving inadequate for parallel applications because of its lack of a parallel access mechanism. Because there is no support for concurrently writing to a netCDF file, parallel applications writing netCDF files must serialize access. This serialization is usually performed by passing all data to a single process that then writes all data to netCDF files. However, serial I/O access is both slow and cumbersome to the application programmer. To facilitate convenient and high-performance parallel access to netCDF files, there has been defined a new parallel interface called Parallel-netCDF (PnetCDF). Since a large number of existing users are running their applications over netCDF, PnetCDF retains the original netCDF file format (version 3) and introduces minimal changes compared to the original interface. The parallel API is distinguished from the original serial API by prefixing the C function calls with “ncmpi” and the Fortran function calls with “nfmpi”.

In PnetCDF a file is opened, operated, and closed by the participating processes in a communication group, an MPI communicator is added in the argument list to define the participating I/O processes within the file’s open and close scope. An MPI info object is also added to pass user access hints to the implementation for further optimizations. The same syntax and semantics is kept for the define mode functions, attribute functions, and inquiry functions. PnetCDF has two sets of data access APIs: the high-level API and the flexible API. The high-level API closely follows the original netCDF data access functions. These calls take a single pointer for a contiguous region in memory, just as the original netCDF calls. The

flexible API provides a more MPI-like style of access and relaxes the contiguous memory constraint. Specifically, the flexible API provides the user with the ability to describe noncontiguous regions in memory, which is missing from the original interface. These regions are described using MPI datatypes. The most important change from the original netCDF interface with respect to data access functions, is the split of data mode into two distinct modes: collective and noncollective data modes. In order to make it obvious that the functions involve all processes, collective function names end with “all”. Using collective operations provides the underlying PnetCDF implementation an opportunity to further optimize access to the netCDF file.

2.2.2 *MPI-IO*

Instead of defining I/O access modes to express the common patterns for accessing a shared file (broadcast, reduction, scatter, gather), MPI-IO chooses another approach in which data partitioning is expressed using derived datatypes.

To understand how MPI-IO works, first we must define some key concepts:

- *File*: An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items.
- *Displacement*: A file displacement is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins.
- *Etype*: An *etype* (elementary datatype) is the unit of data access and positioning. It can be any MPI predefined or derived datatype.
- *Filetype*: A filetype is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single *etype* or a derived MPI datatype constructed from multiple instances of the same *etype*.
- *View*: A view defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype.

Figure 3 shows an example of how all these concepts are used to achieve a global data distribution.

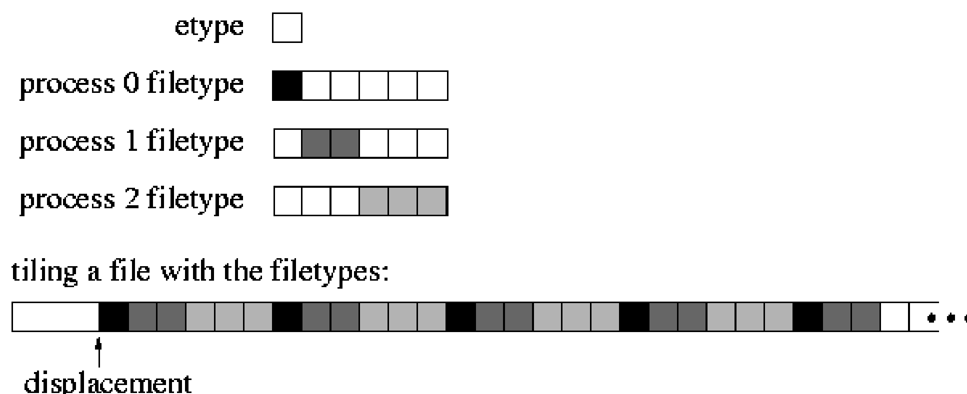

Figure 3 Partitioning a file among parallel processes

The MPI-IO API offers routines to open, close, delete and resize a file. Furthermore, it is possible to query the system about file parameters and preallocate space for a file. As in high-level libraries it is allowed to use some potentially useful hints. These hints mainly affect access patterns and the layout of data on parallel I/O devices. In addition, some hints are context dependent, and are only used by an implementation at specific times.

Regarding data access operations, there are three orthogonal aspects to data access: positioning (explicit offset *vs.* implicit file pointer), synchronism (blocking *vs.* nonblocking and split collective), and coordination (noncollective *vs.* collective). Table 1 enumerates all combinations of these data access routines, including two types of file pointers (individual and shared):

Positioning	**Synchronism**	**Coordination**	
		noncollective	*Collective*
Explicit offsets	*Blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
Individual file pointers	*Blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
Shared file pointer	*Blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Table 1: Data Access Routines

MPI-IO guarantees full interoperability, the ability to read, understand and represent correctly the information previously written to a file, within a single MPI environment, and supports increased interoperability outside that environment through the external data representation as well as the data conversion functions.

2.2.3 *GPFS*

The General Parallel File System (GPFS) is a high-performance shared-disk clustered file system developed by IBM. A clustered file system is a file system which is simultaneously mounted on multiple servers.

GPFS provides concurrent high-speed file access to applications executing on multiple nodes of clusters and can be used with AIX, Linux and Microsoft Windows. In addition to providing filesystem storage capabilities, GPFS provides tools for management and administration of the GPFS cluster and allows for shared access to file systems from remote GPFS clusters.

GPFS provides high performance by allowing data to be accessed over multiple computers at once. Most existing file systems are designed for a single server environment, and adding

more file servers does not improve performance. GPFS provides higher input/output performance by "striping" blocks of data from individual files over multiple disks, and reading and writing these blocks in parallel.

Other features of the file system are:

- Distributed metadata, including the directory tree. There is no single *directory controller* or *index server* in charge of the filesystem.
- Efficient indexing of directory entries for very large directories. There is no limitation on the number of files on a single directory.
- Distributed locking. This allows for full POSIX file system semantics, including locking for exclusive file access.
- Filesystem maintenance can be performed online. There is no need to unmount the filesystem to, for example, add new disks. It can be performed while the filesystem is live.

2.3 Checkpointing

Large petascale computers are expected to have a fairly short mean time between failures due to the large amount of components it comprises. For long runs it is therefore important to save its state regularly so that in case of a failure one can resume from the most recent state instead of restarting completely. This is known as checkpointing and is a technique for adding fault tolerance into computing systems. It basically consists of storing a snapshot of the current application state, and using it for restarting the execution in case of failure. Checkpointing provides the backbone for rollback recovery, playback debugging, process migration and job swapping and is often implemented on top of parallel I/O.

In Distributed Memory Systems there are two main approaches to checkpointing: coordinated checkpointing, in which all cooperating processes work together to establish a coherent checkpoint, and communication induced independent checkpointing. Coordinated checkpointing is not easy to implement due to the difficulty of obtaining a global consistent state, or even the existence of a global clock. In communication induced checkpointing, each process checkpoints its own state independently, whenever this state is exposed to other processes.

A number of practical checkpointing packages have been developed for the UNIX family of operating systems. These checkpointing packages may be divided into two classes, those which operate in user space and those which operate in kernel space. Checkpointing package used by *Condor* and the *Portable Checkpointing Library* developed by The University of Tennessee are examples of user space checkpointing packages. *Chpox* and the checkpointing algorithms developed for the *MOSIX cluster computing environment* are examples of kernel based checkpointing packages.

There are several research lines in the field of application checkpointing. These will be covered here briefly.

2.3.1 *Diskless Checkpointing*

The major source of overhead in all checkpointing systems is the time it takes to write checkpoints to stable storage (i.e. disks). Diskless Checkpointing is a novel technique which uses the philosophy of RAID (Reliable Arrays of Inexpensive Disks) by employing extra

processors to provide fault-tolerance instead of disks. This eliminates stable storage as the bottleneck in checkpointing, and places the burden on the network. In algorithms for diskless checkpointing, processors make local checkpoints in memory and an extra m checkpointing processors maintain encodings of these checkpoints, so that if up to m processors fail, their contents may be restored by the local checkpoints and checkpoint encodings of the surviving processors. See [8]

2.3.2 *User-Directed Checkpointing*

This is research based on the notion that a few hints by the user can result in drastic improvements in the performance of checkpointing. This is due to *memory exclusion*, meaning checkpointing less than the complete memory image of the program, since the jettisoned portions are unnecessary for a correct recovery. Memory exclusion has been employed effectively in incremental checkpointing, where pages are not checkpointed when they are *clean*. In other words, their values have not been altered since the previous checkpoint. However, memory exclusion has not been employed to jettison *dead* variables, variables whose current values will not be used by the program following the checkpoint. With user-directed checkpointing, the user may judiciously place checkpoints to maximize memory exclusion due to clean and dead variables. See [9].

2.3.3 *Compiler-Assisted Checkpointing*

The obvious "next step" for user-directed checkpointing is to employ the compiler. The reasons are clear. The user may miss potential savings due to memory exclusion, or even worse, make erroneous memory exclusion calls. There have been developed data flow equations that enable the compiler to generate correct memory exclusion calls for both clean and dead variables. See [10].

2.3.4 *Fast Checkpoint Compression*

Standard compression algorithms have proven unsuccessful at improving the overhead of checkpointing, because the time it takes to compress the checkpoint is greater than the time it takes to write the original checkpoint to disk. There is ongoing research on algorithms for performing fast compression on incremental checkpoints. See [11].

2.4 Hybrid Parallelization

The peak speeds of high-end supercomputers have grown at a rate that exceeded Moore's Law, which says processor power doubles roughly every 18 months. Moores law is an empirical law: its meaning has changed over time. However, during the last years the law still holds, but it has now been realized differently.

The current steady trend in high performance architectures is to build large clusters of shared memory (SMP) nodes. Today cluster manufacturers are replacing single processors in their existing systems with powerful and sophisticated multi-core CPUs. Parallel programming on these machines ought to combine the distributed memory parallelization between the nodes with the shared memory parallelization inside each node. Nowadays, most of the existing

codes are developed using only the pure message-passing paradigm. The Hybrid Programming paradigm (also Mixed Programming called) however, can potentially exploit features of the SMP cluster architecture better, thus resulting in a more efficient parallelization strategy and potentially better performance.

Message passing codes written in MPI are obviously portable and should transfer easily between different SMP systems. Intuitively, a parallel paradigm that uses memory access for intra-node communication and message-passing for inter-node communication seems to match better the characteristics of an SMP cluster. Combining shared-memory and distributed-memory programming models, is an old idea [12]. In the wide spectrum of possible solutions for hybrid shared/distributed memory code development, the joint use of MPI and OpenMP is emerging. Both MPI and OpenMP are two well-established industry standards with solid documentation and different tools are available to assist program development.

The majority of hybrid MPI/OpenMP codes are based on a hierarchical model, which makes it possible to exploit large- and medium-grain parallelism at MPI level, and fine-grain parallelism at OpenMP level. Hence, hierarchical hybrid code is structured in such a way that only a single message passing task, communicating using MPI primitives, is allocated to each SMP processing element, and the multiple processors with shared memory in a node are exploited by parallelizing loops using OpenMP directives and run-time support. The objective is to take advantages of the best features of both programming styles. Considerable work has gone into studying the hybrid model. Some examples can be found in [13].

Recently, the hybrid model has begun to attract more attention, for at least two reasons. The first is that OpenMP compilers and MPI libraries are now solid commercial products, with implementations from multiple vendors. The second reason is that scalable parallel computers now appear to encourage this model. A lot of scientific work enlightens the complexity of the many aspects that affect the overall performance of hybrid programs [14]. Also, the need for a multi-threading MPI implementation that will efficiently support the hybrid model has been spotted by the research community [15].

Hybrid programming with two portable and consolidated APIs would be impossible unless each made certain commitments to the other on how they would behave together. In the case of OpenMP, one important commitment is that if a single thread is blocked by an operating system call (such as file or network I/O) then the remaining threads in that process will remain runnable. This means that a MPI blocking call, such as MPI_Recv or MPI_Wait, only block the calling thread and not the entire process. This is a significant commitment, since it involves the thread scheduler in the compilers runtime system and interaction with the operating system.

The MPI-2 standard defines four different levels of thread safety. These are in the form of how an MPI implementation can perform communication between processes:

_ MPI_THREAD_SINGLE: there is only one thread in the application.

_ MPI_THREAD_FUNNELED: only one thread may make MPI calls.

_ MPI_THREAD_SERIALIZED: any threads may make MPI calls, but only one at a time.

_ MPI_THREAD_MULTIPLE: any thread may make MPI calls at any time.

An application can find out at run time which level is supported by the MPI library using the MPI_Init thread routine. The level of multi-threading provided appears to be heavily dependent on the hardware and how MPI is implemented. All MPI implementations support MPI_THREAD_SINGLE. The nature of typical MPI implementations is such that they probably also support MPI THREAD FUNNELED, even if they do not admit it by returning

this value from the MPI_Init thread. Usually when people refer to an MPI implementation as thread-safe they mean that the implementation supports the maximum level of functionality or, to be more precise, MPI_THREAD_MULTIPLE is returned by the MPI_Init thread.

2.4.1 *Main Rules for Mixed Programming*

There are many different approaches to mixed parallel programming. Rabenseifner [16] distinguishes different models depending on process/thread hierarchy, overlap of communication with computation and the number of threads calling communication routines. In all the cases MPI is used for coarse-grain parallelism (i.e. principal data decomposition), and OpenMP for fine-grain parallelism inside each MPI process. There are three main different mixed mode programming models depending on the way the MPI communication is being handled:

- *Master-only*, where all MPI communication takes place outside of OpenMP parallel regions.
- *Funnelled*, where communication may occur inside parallel regions, but is restricted to a single thread.
- *Multiple*, where more than one thread can call MPI communication routines.

The master-only approach defines the simplest hybrid programming model with MPI and OpenMP, because no particular features are needed. The Funnelled mixed model can be achieved by surrounding MPI routines with OMP_CRITICAL, OMP_MASTER or OMP_SINGLE directives, inside of a parallel region. One must be very careful, however, since the OMP_MASTER directive does not imply an automatic barrier synchronization or an automatic flush operation, neither at the entry to nor at the exit from the master section. If the application wants to send data computed in the previous section or wants to receive data into a buffer that was also used in the previous parallel region, then a barrier (which implies a flush operation) is necessary prior to calling the MPI routine. If the data is also used in the section after the exit of the MPI routine, then also a barrier is necessary after the exit of the OMP_MASTER section. In this way, while the master thread is executing the MPI routine, all other threads are sleeping.

Only the multiple mixed model allows a direct message passing from each thread in one node to each thread in another node. However, it requires more complicated communication handling which may result in additional overhead.

Based on these descriptions and because a large number of MPI implementations cannot be guaranteed to be thread-safe, to ensure that the code is really portable and runnable over a high number of different architectures, all MPI calls should be made to follow a Master-only parallelization scheme.

2.4.2 *Analysis of Hybrid Parallelization in QuantumEspresso*

The next subsections will cover approaches in hybrid parallelization by analyzing the application characteristics and the parallel performance achieved by using the hybrid programming model for a real scientific application, Quantum ESPRESSO (QE). QE has been chosen for this purpose since it employs hybrid parallelization extensively. QE is an

integrated suite of high performance computing codes for electronic structure calculations and materials modelling at the nanoscale. See also [17] and references therein.

The name "ESPRESSO" stands for "opEn Source Package for Research in Electronic Structure, Simulation, and Optimization", while "Quantum" stresses about its scope: first-principle (i.e. based on the electronic structure) calculations within Density-Functional Theory [25] (DFT) in a Plane-Wave (PW) Pseudo-Potential [27] (PP) approach.

The QE distribution is built around a number of core components, PWscf (Plane Wave self consistent field) and CP (Car Parrinello), designed and maintained by a small group of core developers. Interoperability of different components is granted by the use of common formats for the input, output, and custom work files. Parallelization is achieved using the Message Passing paradigm by calling standard MPI libraries.

High performance on massively parallel architectures is achieved by distributing both data and computations in a hierarchical way across available processors, ending up with multiple parallelization levels that can be tuned to the specific application and to the specific architecture. In more detail, the various parallelization levels are geared into a hierarchy of processor groups, identified by different MPI communicators. In this hierarchy, groups implementing coarser-grained parallel tasks are split into groups implementing finer-grained parallel tasks. The first level is image parallelization, implemented by dividing processors into n image groups, each taking care of one or more images (i.e. a point in the configuration space, used by the NEB method). The second level is pool parallelization, implemented by further dividing each group of processors into npool pools of processors, each taking care of one or more k-points. The third level is plane-wave parallelization, implemented by distributing real- and reciprocal-space grids across the nPW processors of each pool. The final level is task group parallelization [79], in which processors are divided into ntask task groups of nFFT = nPW/ntask processors, each one taking care of different groups of electron states to be Fourier-transformed, while each FFT is parallelized inside a task group. A further paralellization level, linear-algebra, coexists side-to-side with plane-wave parallelization, i.e. they take care of different sets of operations, with different data distribution. Linear-algebra parallelization is implemented both with custom algorithms and using ScaLAPACK [80], which on massively parallel machines yield much superior performances. The table below contains a summary of the five levels currently implemented:

group	distributed quantities	communications	performance
image	NEB images	very low	linear CPU scaling, fair to good load balancing; does not distribute RAM
pool	**k**-points	low	almost linear CPU scaling, fair to good load balancing; does not distribute RAM
plane-wave	plane waves, **G**-vector coefficients, **R**-space FFT arrays	high	good CPU scaling, good load balancing, distributes most RAM
task	FFT on electron states	high	improves load balancing
linear algebra	subspace Hamiltonians and constraints matrices	very high	improves scaling, distributes more RAM

From an algorithmic point of view QE relies on the following basic kernels: 3D FFT, Linear Algebra (Matrix multiplications and Eigenproblem solution), space integrals and point function evaluation, where most of the execution time is spent.

2.4.3 *Limits of MPI*

QE like many other HPC applications is parallelized only using MPI, meaning that, in order to use the whole power of a given machine, one MPI task has to be scheduled on each available core. As pointed out in the introduction of this chapter, the number of cores per node is rapidly increasing, so that, maintaining the 1 MPI task / 1 core ratio could easily end up stressing the network and the OS, due to the high number of messages delivered inside each node. Moreover, many MPI applications like QE make use of global communications like MPI_ALLTOALL where the number of messages exchanged grows with the square of the number of MPI tasks, making this problem even worse. To make things more difficult for pure MPI applications, there is clear tendency in architecture design to reduce the memory / core ratio. In fact every MPI task of QE, like many other MPI codes, needs some auxiliary data structures to coordinate the activities among tasks. The size of this data structure does not decrease with the number of MPI tasks. It rather slightly increases, leaving less space for the other data structures.

On the other hands the number of nodes in a HPC system does not seem to grow as fast as the number of cores, making hybrid parallelization attractive for most MPI applications. It is important to underline that this does not solve the scalability problem of MPI applications on large HPC systems completely, but at least can increase the scalability by a factor that is comparable with the number of cores inside each node. In fact MPI application like QE, for a given dataset, usually scale almost linearly with the number of processors up to a given processor count, above which the speed-up saturates, most often because the data distribution becomes too fine grain. The hybrid parallelization can help by differentiating between fine grain parallelism (inside the node) and coarse grain parallelism (outside the node), so that MPI tasks can scale with the number of nodes and not with the number of cores. In this respect the scalability of a hybrid application can gain up to a factor of ten, or even more. Therefore, using the hybrid programming model can be a good way to make an application that scales up to say 100TFlops, reach 1PFlops.

2.4.4 *Hybridization strategy*

After having analyzed the scalability of an MPI code and found that hybridization will be a good opportunity to make the code scale to sustained petaflop performance the programmer is faced with the problem of how to start hybridizing the MPI code with OpenMP. There are two main possible approaches that present somehow different problems, namely the “implicit” approach and the “explicit” approach, and there is obviously the combination of the two.

Implicit approach

Because most vendors provide their own libraries in both single- and multi-threaded versions, the simplest approach to mixed parallel programming is linking with the multi-threaded libraries. Using a multi-threaded library is quite simple but not always immediate. It requires recompiling most of the auxiliary libraries to be compatible with the multi-threaded version,

to adapt the various makefiles needed to compile the source code and probably rewriting pieces of code to conform to new routine prototypes inside the library. It is important to note that every time a multi-threaded routine is called, an additional overhead required to spawn threads and distribute the work is incurred. This short overhead, multiplied thousands and thousands of times, impacts the global performance. Regarding our test case (the QE code) an implicit approach has been used for the linear algebra subtask.

Explicit approach

The second approach consists of explicitly instructing the code on using multi-thread parallelism. Unlike the previous approach, deeper knowledge of the OpenMP standard is required. Starting from an existing code, there are two types of errors that can arise during the programming practice. These are correctness mistakes (errors impacting the correctness of the program) and performance mistakes (errors impacting the speed of the program). Concerning QE, the explicit approach has been used for the ad-hoc 3D FFT implemented in QE.

Implicit and Explicit approach

Regardless of an application spending most of its time inside library subroutines, or an application not calling any external libraries, in general one has to use both the implicit and the explicit approaches to get the best OpenMP scalability. In this case one has to pay attention to OpenMP regions which contain calls to external libraries. If a library is multithreaded it may happen that it is incompatible with OpenMP and thus one has to substitute the library with a thread safe version. This is not always easy since in general multithreaded and single threaded libraries contain the same symbolic names. So the linking should be selective.

In what follow we present two typical examples of the implicit and explicit approaches that can be found in real world applications. In particular it is explained how Linear Algebra (LA) and FFT have been parallelized using the implicit and explicit approaches in QE, however, the approach is absolutely general and can be applied to any other application.

Linear Algebra (implicit approach)

In QE, like in many other applications, parallelization of the linear algebra subtask is performed using the ScalaPACK library. ScalaPACK guarantees an almost linear scalability if the size of the problem is increased with the number of processors. On the other hand, linear algebra inside an application is often used to solve only a subtask of the whole solution, and the dimension of this task is constant for a given problem. Therefore, increasing the number of MPI tasks reduces the size of the local blocks of the matrix equations solved with ScaLAPACK and at a certain point, depending on the architecture; the solution does not scale any further. The hybrid approach offers an optimal solution to this problem; one can combine the efficiency of the ScaLAPACK to communicate between nodes and the efficiency of the SMP LAPACK and BLAS library to scale inside the node, extending the scalability of the application to a factor proportional to the number of cores per node.

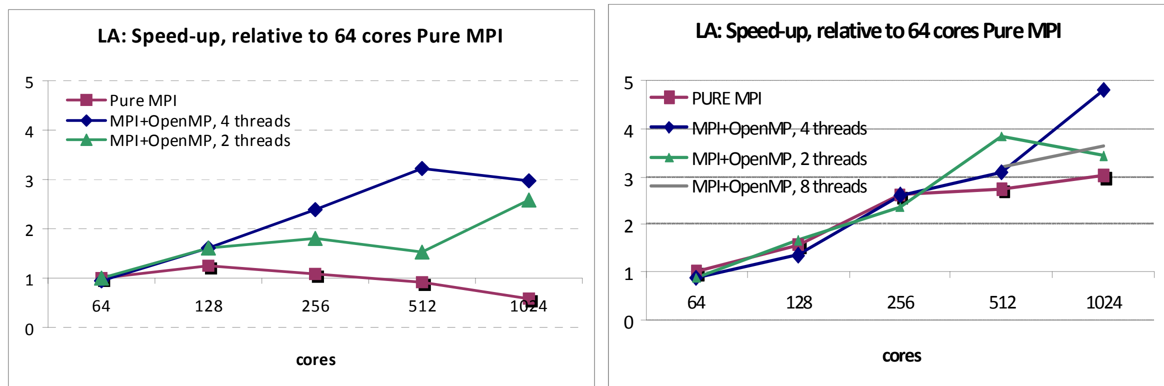

Figure 4 Speed-up of the Linear Algebra(subtask of a Car-Parrinello simulation of 256 water molecules

Figure 4 shows the scalability of the LA subtask on two different architectures performed with QE on BCX(CINECA linux cluster with 4 Opteron cores/node,left) and on HPCx (EPCC p575 P5 cluster with 16 cores/node, right. Matrix size is 1024x1024. On BCX the pure MPI version saturates at 128 cores, while the hybrid version saturates at 512 cores. On HPCx the LA subtask saturates at 256 cores, while the hybrid code, combining MPI tasks and OpenMP threads (256 tasks and 4 threads) scales up to 1024 cores.

Note that different combinations of tasks/threads can lead to different scalability behaviour. This behaviour can be related to the particular data distribution of ScaLAPACK, where a change in one of the parameters (processor grid, block size, ext.) has a great impact on performance and therefore parameters need to be changed when changing the number of cores. A common experience with hybrid codes is that for a given dataset and a given architecture one has to try different combinations of tasks/threads to get the best performance.

FFT (explicit approach)

The FFT implementation in QE is done around the idea of being as modular as possible, following the general guidelines of the distribution. There are three main modules that interact with each other: fft_parallel.f90 containing one routine, called tg_cft3s, that contains the whole main logic behind the ad-hoc FFT algorithm (with and without task groups); fft_scalar.f90 containing the "scalar" routines for 1D FFTs along z (cft_1z) and 2D FFTs along x and y (cft_2xy); fft_base.f90 containing the routines that handle the communication among the MPI tasks.

During a typical execution, there are two different grids involved in the FFT calculation: one for the charge density and for potentials, and another one for the wave-functions. The information about the grids is stored in a data structure, called fft_descriptor, replicated on all the processes involved in the computation. This descriptor contains: the dimensions of the grid, the number of planes, the number of sticks (columns of values in the z direction) for each processor, the task group subdivision and also all the information required to perform the packing and unpacking of the data before and after the redistribution among the processors. The definition of the fft descriptor is placed in types.f90 together with the routines to initialize it correctly.

Both CP and PWscf QE kernels call one unique routine, tg_cft3s, in order to invoke the FFT calculation on a specified grid. Using different parameters we can choose to enable (or disable) the task group strategy and to perform a forward (from G- to r-space) or backward (from r- to G-space) transformation. The grid descriptor is explicitly passed to the routine as an argument because QE is designed to operate on different grids together. The routine algorithm for FFT on charge density and for potentials grid is summarized below:

```
subroutine tg_cft3s ( isgn , fftdescriptor )
allocate auxiliary space
if ( isgn > 0 ) then
  call cft1z ( . . . )
  call fw_scatter ( . . . )
  call cft2xy ( . . . )
else
  call cft2xy ( . . . )
  call bw_scatter ( . . . )
  call cft1z ( . . . )
end if
deallocate auxiliary space
end subroutine
```

The above is a pseudo code version of the main 3D FFT driver of QE. Here: cft1z performs fft along the z direction being the z columns of values distributed among MPI tasks, fw_scatter & bw_scatter perform forward and backward data redistribution among MPI tasks, cft2xy performs fft along the x and y directions being the xy planes of values distributed among MPI tasks.

2.4.5 *Implementation design*

When implementing explicit hybrid codes one also has to decide how to make MPI functions cope with threads. For QE the master-only scheme was chosen, where the calls to the MPI library are all done outside OpenMP regions. This strategy was chosen because it is considered the best suited for portability, in fact not all MPI versions support all the possible schema of mixing MPI and OpenMP, and also due to the fact that it is well suited for well structured and modular applications like QE, where the bulk of communications is performed in well defined points in the code.

As an example of the explicit master-only approach we consider the OpenMP parallelization of the forward 3D FFT (the backward is analogous). Below the MPI version and its hybrid equivalent are shown.

```
call cft1z ( . . . )       ! FFT along z
call fw_scatter ( . . . ) ! data exchange
call cft2xy ( . . . )     ! FFT along x and y
```

Pseudo code relative to the MPI forward 3D FFT. During the FFT along z, x and y are distributed, while during the FFT along x and y, z is distributed. fw_scatter performs the data redistribution.

```
!$omp parallel do
do i = 1 , nsl
   call 1DFFT along z ( f [ offset( threadid ) ] )
end do
!$omp end parallel do
call fw_scatter ( . . . )  ! Only the master performs data exchange
!$omp parallel
do i = 1 , nzl
!$omp parallel do
  do j = 1 , Nx
    call 1DFFT along y ( f [ offset( threadid ) ] )
  end do
!$omp parallel do
  do j = 1, Ny
    call 1DFFT along x ( f [ offset( threadid ) ] )
  end do
end do
!$omp end parallel
```

The above is pseudo code relative to the master-only hybrid implementation of the forward parallel 3D FFT (the backward is analogous). Here: nsl is the number of z columns assigned to each MPI task, nzl is the number of xy planes assigned to each MPI task, Nx and Ny are the global dimensions of the 3D FFT in the x and y directions. Along each direction (z, x and y) the series of 1D FFT are performed distributing the computation among OpenMP threads, the array to be transformed (f in the pseudo code above) is shared among threads, so that each thread works on a different portion of the array using a private offset depending on thread ID.

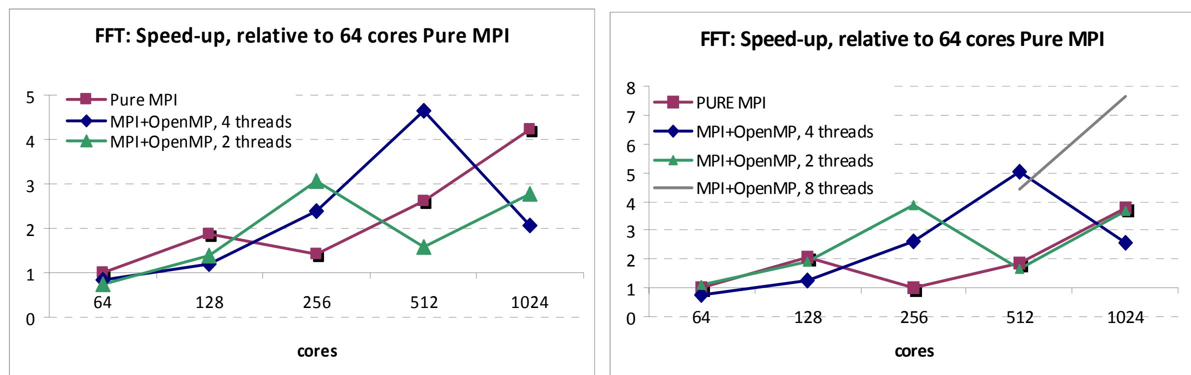

Figure 5 Speed-up of the ad-hoc 3D FFT subtask of a Car-Parrinello simulation of 256 water molecules. Grid size is 200x200x200. See text for comment.

Figure 5 shows the speed-up results of the hybrid FFT subtask performed with QE on BCX(CINECA linux cluster with 4 Opteron cores/node,left) and on HPCx (EPCC p575 P5 cluster with 16 cores/node, right).. The behaviour is quite similar on BCX and HPCx. Pure MPI code scales between 64 and 128 cores, then there is a negative scalability between 128 and 256 cores, before starting to scale again between 256 an 1024 cores. The negative scalability between 128 and 256 cores depends on the grid size and data distribution.

Regarding the curve obtained with the hybrid run, it is possible to see that they replicates the behaviour of the pure MPI curves but at higher number of cores and with a speed-up proportional to the number of threads.

2.4.6 *Other explicit OpenMP parallelization.*

The FFT and Linear Algebra subtasks are the heaviest computations in QE, but there are also many other algorithms that start to be significant, when the number of OpenMP threads is increased. These other parts of the code are mainly explicit loops over the real and reciprocal grids, or over the number of atoms. The loops over the real and reciprocal space is performed to compute space integrals or to evaluate point functions. Both can be easily parallelized with OpenMP, with a parallel do plus reduction for space integral, and with a simple parallel do for the point functions. Loops over the atoms are a little bit trickier because many times a given quantity is computed only for a subset of the atoms. In practice this means that inside the loop there is an 'if statement' to select whether or not an atom have to be processed. From the point of view of OpenMP this means that one has to manage the parallelization explicitly using thread IDs to balance the load among threads.

Just to give an idea of the OpenMP parallelization statements used in QE, here we present how we have parallelized the loop to compute the exchange and correlation energy (XC). The subroutine computing this quantity (xc) is a function to be evaluated for each point in the grid, so that with OpenMP this can be parallelized with a "parallel do" directive.

```
!$omp parallel do private( rhox, arhox, ex, ec, vx, vc ), reduction(+:etxc)
     do ir = 1, nnr
        rhox = rhor (ir, nspin)
        arhox = abs (rhox)
        if (arhox.gt.1.d-30) then
           CALL xc( arhox, ex, ec, vx(1), vc(1) )
           v(ir,nspin) = e2 * (vx(1) + vc(1) )
           etxc = etxc + e2 * (ex + ec) * rhox
        endif
     enddo
!$omp end parallel do
```

Point Function and Space Integrals evaluate Exchange and Correlation (XC) potential and energy. Where: rhor is the electron density, xc is the XC function, v is the XC potential, etxc is the XC energy and nnr is the number of grid points for each MPI task.

As stated above there are loops with a more complex structure that are more difficult to parallelize. Below there is an example of a loop over the atoms to compute a component of the atomic pseudopotential (deeq), where the number of threads and the thread index are used explicitly.

```
!$omp parallel default(shared), private(na,qgm_na,is,dtmp,ig,mytid,ntids)
              mytid = omp_get_thread_num()
              ntids = omp_get_num_threads()
              ALLOCATE(  qgm_na( ngm ) )              !
              DO na = 1, nat             !
                IF( MOD( na, ntids ) /= mytid ) CYCLE
                IF ( ityp(na) == nt ) THEN
                   qgm_na(1:ngm) = qgm(1:ngm)*  &
                      eigts1(ig1(1:ngm),na)*eigts2(ig2(1:ngm),na)*eigts3(ig3(1:ngm),na)
                   DO is = 1, nspin_mag
                       dtmp = 0.0d0
                       DO ig = 1, ngm
                          dtmp = dtmp + aux( ig, is ) * CONJG( qgm_na( ig ) )
                       END DO
                       deeq(ih,jh,na,is) = fact * omega * DBLE( dtmp )
                       deeq(jh,ih,na,is) = deeq(ih,jh,na,is)
                    END DO
                 END IF
              END DO
              DEALLOCATE( qgm_na )
!$omp end parallel
```

Explicit loop parallelization, loop over the atoms. Where: nat is the number of atoms, ityp is the atom type and nt is a parameter to select a given atom type. Other parameter are not relevant for our purpose.

2.4.7 *Performance of the hybrid code*

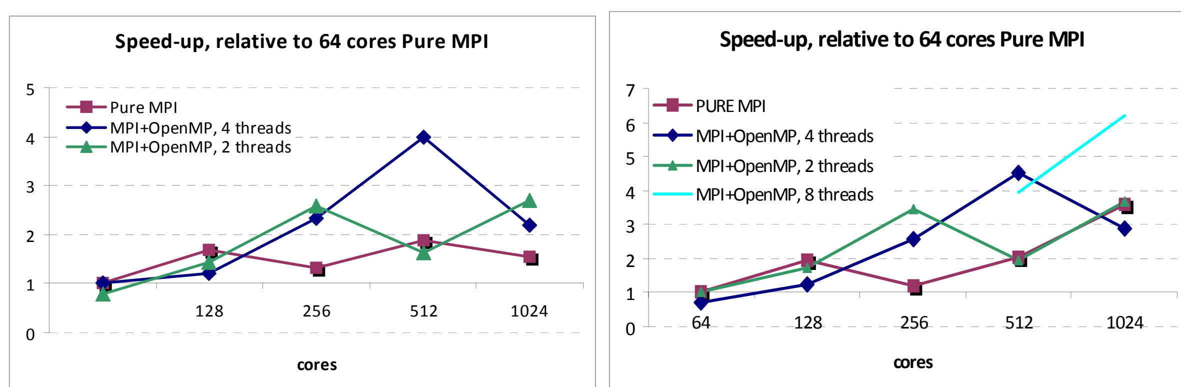

Figure 6 Speed-up relative to 64 cores of a Car-Parrinello simulation of 256 water molecules

Figure 6 shows the speed-up results of a Car-Parrinello simulation using QE with Hybrid parallelization, performed with QE on BCX(CINECA linux cluster with 4 Opteron cores/node,left) and on HPCx (EPCC p575 P5 cluster with 16 cores/node, right). The two graphs are more or less a combination of the linear algebra and FFT subtasks of Fig. 1 and 2. This is quite obvious since LA and FFT are the heaviest kernels in the code. Analyzing the speed-up curves while on BCX, the pure MPI curve above 128 MPI tasks oscillates between 1 and 2, on HPCx the curve oscillate up to 512 tasks and between 512 and 1024 tasks, there is a speed-up of almost a factor of 2. This behaviour is due to the fact that the LA subtask on HPCx does not show a negative speed-up (see Figure 4). Regarding the curve obtained with the hybrid run, it is possible to see that it replicates the behaviour of the pure MPI curve, but at a higher number of cores and with a speed-up with respect to the pure MPI curve that is proportional to the number of threads.

2.4.8 *Conclusion*

For tightly coupled applications implemented using MPI, the number of MPI tasks is a critical factor for scalability. This is especially true if they rely on global communication. In the past these applications could run with one MPI task per core to exploit the power of parallel architectures. However, this ratio is not sustainable any longer, since on present and future architectures the number of cores per node is increasing and the ratio of memory per core and bandwidth per core are decreasing. OpenMP in combination with MPI allows a way to distinguish between intra and extra node parallelism and can thus diminish the above bottleneck. This technique is called hybrid parallelization and applications can take advantage of it by rewriting the parallelization of key algorithms and their data structures to cope with this hierarchy. Such a hierarchy can be achieved by following an implicit or an explicit approach, or a combination of the two, to combine the efficiency of multithreaded libraries, available on most architectures, with explicit OpenMP parallelization of the most compute intensive loops.

It is important to underline that a hybrid application is also more complex in terms of data distribution and therefore the relation between performance and the number of cores is less obvious. Different combinations of tasks/threads may show quite different behaviour, depending on the dataset. With respect to pure MPI applications more performance tests have to be done to get the best performance for a given dataset. As a general rule, if hybrid parallelization has been employed, one can expect to gain roughly a scalability increase of a factor proportional to the number of cores per node, compared to the pure MPI version.

2.5 Minimizing of Communication Overheads

Nowadays, message passing is one of the most popular parallel computation models. For MPP and clustered SMP architectures, message passing is the only way to communicate between different nodes. MPI parallelization, which is portable and efficient, has been widely used on variety of architectures to achieve high applications performance. However, its communication overheads can not be fully avoided. The overheads will affect the parallel performance more and more with increasing processors/cores used. Petascaling using standard processors will involve a large number of processors/cores. It is therefore very important and necessary to minimize the MPI communication overheads for a better parallel performance at the peta scale.

The factors which can affect the MPI communication performance are numerous and complex, so it is difficult to generalize the MPI code optimizations. Tuning the performance of a MPI application depends on the code features, the MPI implementations, the usage of MPI in the application and the given platform.

Lack of sufficient parallelism could be a bottleneck for the application parallel performance. In such cases, further parallelization should be applied based on the code investigation and analysis where possible. On the other hand, targeting the additional parallelization will typically lead to more MPI communications. A larger number of MPI communications will reduce the application performance because of the extra overheads. Therefore a balanced ratio between the communications and the computations in code is very important.

The MPI communication functions can be divided into two major types: the *point-to-point* functions and the *collective* functions. A point-to-point communication involves exactly two processes, i.e. the message will be passed from one process to another. A collective communication will involve all processes in a specific group at one time. For example, the collective functions include barrier, broadcast and global reduction operations, etc.

For the two communication types, the main reasons for high overheads are different and thus varying techniques should be applied to reduce the overheads. For instance, when using the point-to-point functions, the communication performance will be more affected by the message passing modes (blocking/non-blocking, synchronous/asynchronous), message buffering, message passing protocols, message size and number, etc. The optimization techniques for point-to-point communications will be discussed in more detail in section 2.5.1.

The collective communications could be costly as they may enforce global synchronizations, i.e. all processes have to wait for the slowest process. This means the collective functions should be used only when it is absolutely necessary. For collective communications, the load balance and synchronization are usually the key factors for the communication overheads. This will be discussed in the section 2.5.2.

From the hardware aspect, there are also many factors for the MPI performance, such as the memory subsystem, the network, etc. MPI parallelization should try to adjust the application codes to the given platforms. For example, a good mapping of the virtual processes onto physical processors will be quite helpful to improve the communications between nodes. This will be discussed in section 2.5.3. Also, on some platforms, e.g. MPP-BG/P, different networks will be provided separately for the point-to-point and collective communications. Using proper networks will improve the communication performance effectively.

To optimize MPI applications and minimize the communication overheads, it is important to first identify the key bottlenecks. Different HPC tools are provided to help identify and analyze the MPI communication behaviors. Commonly used MPI tools include the Craypat tool, the IBM MPI Trace tools, Scalasca, Paraver, Vampir, etc. In the PRACE WP6, many application optimizations were implemented based on the HPC toolkit profiling results.

2.5.1 *Point-to-Point Communications*

The point-to-point communication is the communication pattern only between two processes. There are different point-to-point functions which implement the sending and receiving of messages between processes in many ways. The communication performance can vary considerably depending on the implementations.

- **Blocking / Non-blocking communications**

  The *blocking* communication functions only return once the communication has completed, i.e. the message must be successfully sent or received safely and the buffers should be available for reuse. This communication mode is obviously quite costly with respect to time.

  The *non-blocking* communication functions will return without waiting for completion of the communication. Therefore, the communication can continue in the background and the process may carry on with other work, returning at a later point to check that the communication has completed successfully. This kind of communication will reduce the overheads significantly, especially for the regular communication patterns, e.g. when many processes perform halo swapping.

  For a better parallel performance, the non-blocking communication functions should be used primarily and the blocking ones should be replaced by the non-blocking implementations where possible. An `MPI_Wait` has to be used in this case to guarantee that the buffers are safe to use or modify. Not using `MPI_Wait` will also lead to memory leaks.

- **Message buffering**

  In the point-to-point MPI functions, the s*ynchronous* send operations will complete only after knowing that the message has been received by the receiving process, while the *asynchronous* send operations will not wait for an acknowledgement of a message being received but will complete even though the receiving process has not actually received the message. Therefore, synchronous operations have a higher latency as they need to wait for the acknowledgement from the receiving process. For large messages this is typically insignificant, but it is important for smaller messages.

  Message buffering offers the possibility of asynchronous message passing to implement more effective communications and reduce the waiting overheads. It solves the issue that a message is sent without a receiving a matching acknowledgment. Using the buffered

sending function, the message data will be stored in the system buffer until the receiving occurs. This allows the sending process to continue with other work rather than waiting.

However, the message buffering has some disadvantages. The programmer will have to ensure the buffer space is sufficient and usually needs to explicitly attach enough buffer space. Otherwise it will cause program failures and/or errors. Also, a working buffering implementation under one set of conditions may fail under another set.

Many MPI implementations provide environment variables to control the buffering size used by the communication subsystem to buffer early arrivals. For example, the MPI environment variable `MP_BUFFER_MEM` is provided by the IBM MPI implementation on the MPP-Power6 platform, which allows the buffer size to be increased within the maximum limit, when running out of buffer space during an application execution.

- **Message passing protocols**

  The method of sending message data immediately after sending the message envelope is called the *eager* protocol. The eager protocol is an asynchronous protocol which assumes the receiving process can handle the sending message data and allow the send operation to complete without receiving an acknowledgement. The eager protocol is good for reducing the message latency, but can only be used for small messages due to limits on buffer space. Also, large messages may be costly for the receiving process to get from the network or copy into the buffer space.

  A "handshaking"-like protocol, *rendezvous* protocol, is usually used for large message communications. Under this protocol, the sending process will only send the message envelop first to the destination process. When the buffering space is available, the destination process will reply with a data requirement to the sending process. The real message data will then be sent from the sending process to the receiving process. The rendezvous protocol is more scalable compared to the eager protocol and can prevent the buffering exhaustion that may cause program failure. The rendezvous protocol is best used for the large message size communications, when the performance penalty due to the initial negotiation phase is negligible.

  Based on the discussions above, the eager protocol is usually used for the small message size communications and the rendezvous protocol is used for the large message size communications. Tuning the eager limit properly, above which the rendezvous protocol is used, can help to get a better application performance. For applications with intensive small message passing, setting a higher eager limit so as to use the eager protocol instead of the rendezvous protocol may reduce the "handshaking" overheads. On the other hand, setting a smaller eager limit could be helpful to reduce network congestion by cutting down on the number of messages in flight. Many MPI implementations allow users to control the eager limit. For example, the MPI environment variable `MP_EAGER_LIMIT` is provided on the IBM MPP-Power6 platform to change the threshold value for message sizes. The maximum value is 64K.

  Other protocols could be provided on specific platforms. Using the most suitable protocols will benefit the MPI application performance and reduce the communication overheads effectively.

- **Message size and number**

  Message sizes have a significant influence on the MPI application performance, especially within the small to mid-size message range. For applications with intensive communications, reducing the message sizes may help to take advantage of using different communication protocols and thus improving the performance. For some other

applications, increasing the message sizes will lead to a fewer total number of messages and it can improve the parallel performance by reducing many overheads caused by frequent communications. Derived datatypes can be helpful when combining several small messages into a single larger message.

2.5.2 *Collective Communications*

The collective communications involve all the processes within the specific communicator. The collective MPI functions perform distributing data, gathering data, global reduction operations as well as barrier synchronizations, etc, for a group of processes. The collective functions provide a convenience for the parallel programming but do not guarantee a faster performance. The same global operations can usually be implemented by using point-to-point functions. In the case where collective MPI functions are optimized to match the platform hardware, using the collective communications may help to improve performance. However, if this is not the case, the performance may be even slower by using collective communications. Therefore, when trying to reduce the communications overheads, it can be worth considering replacing the ordinary point-to-point functions with the collective MPI functions, and vice versa.

For the collective communications, synchronization and load balance are the two key factors for the collective communications performance. Global barrier synchronizations are usually quite expensive as all processes will have to wait for the slowest process, before being able to continue. Therefore global barrier synchronizations should be avoided if possible, but in some cases it is necessary to ensure the operation's correctness. Some collective functions may enforce a global synchronization. Reducing the unnecessary global barrier synchronizations will improve application performance.

The global reduction values (e.g. the global sum) should be calculated only when the data will be used. An unnecessary global results transfer will also cause high overheads. For example, consider the case where a global sum is required to be calculated and which is only needed by one process. Although using `MPI_Reduce` and `MPI_Allreduce` can both achieve the correct result, broadcasting the global sum to every process involved rather than only to the root process will be a waste of time and bandwidth. It is therefore important to select the most proper collective functions for the parallelism implementation. A user defined complex operator could be helpful to reduce the frequency of collective operations.

As every process in the same specified communicator will do the same communication or operations in the collective functions, a good load balance is therefore particular important to reduce the global synchronization overheads. A serious load imbalance will cause significant waiting overheads, especially for executions with large numbers of processors/cores.

Some platforms provide specific optimizations for the collective communications. For example, the MPP-BG/P system provides a separate network for the collective communications/operations and a separate network for the global barrier. Selecting the proper network to implement the MPI collective communications will highly improve application performance and reduce its overheads.

2.5.3 *Logical Communication Mapping in the Physical Communication Layer*

Mapping the logical communications onto the real physical communication layer is very important as it can influence the application performance significantly. Ideally, the processes domain could be mapped to a similar shape physical processors domain with a proper rank ordering. However, the default/random mapping may not always be the most suitable one for

the given MPI applications and can cause high overheads. A new manual mapping strategy or a new process topology may be required in order to achieve a performance improvement.

Both the processes mapping shape and the rank order can affect the communication performance. E.g. for a MPI application having many halo swappings between neighbouring processes, it will be more reasonable to map the processes onto the processors within a square shape subnetwork, compared with a diagonal mapping shape. Also, in this case, it will be ideal to allocate the neighbouring processes physically next to each other when setting the processes rank order.

The effect of process mapping depends a lot on the platform architectures. The communication efficiency will be affected by the distance between processes and their position in the network as a whole. On the clustered SMP or multi-core MPP systems, communications between the processes on the same node could be much faster than between processes on different nodes. Therefore processes with frequent communications should ideally be mapped on the same node or nodes near each other. For an application which requires large memory, placing less MPI tasks on each node could be a useful strategy to solve the memory limit issue.

Many systems provide specific environment variables and options for the job launcher, e.g. `mpirun`, to control the processes mapping. On some platforms, such as the MPP-BG/P, programmers can even manually remap the MPI tasks onto the physical hardware by using a mapping file.

Also, a virtual topology may be created to help improve the communication convenience and efficiency. The MPI topology can provide a convenient naming mechanism for the group processes within a communicator. Although it is not the same as changing the mapping onto the physical structures directly, some heuristic virtual topologies may assist running systems to optimize the process mappings based on the architecture features.

3 Applications

This section includes the *application scaling reports,* which the Benchmark Code Owners (BCO) have written to document their work. The BCOs have been responsible for heading a wider team, originating from more than one PRACE partner, in order to scale their applications as much as possible. This work was done in collaboration with the original developers of the applications. The reports are thus about the approaches the BCOs have taken to scale the application they have been responsible for and their experiences, best practices, and bottlenecks they have encountered during this work. Each report contains an application description section, a petascaling section about what has been done to scale the application, a result section and finally a conclusion section. There are a total of twenty application reports.

For this work all PRACE HPC member centers have been involved, which includes many people spread across multiple countries. To coordinate this work, bi weekly telcons were arranged by the task leaders of tasks 6.4 and 6.5, where BCOs reported their progress and scalability problems, to the task leaders and other BCOs. In this way best practices could be propagated faster and in case of problems other BCOs could suggest possible solutions. Furthermore the work of each BCO was documented on a dedicated WP6 website before each telcon and used later when writing the application reports.

It should be noted that some optimization information is not available in the below application scaling reports, due to restrictions on licensing regarding what can be disseminated. Instead such information appears anonymously in the previous distilled sections.

The codes are listed in alpabetical order.

3.1 Alya

Written by: Guillaume Houzeaux (BSC)
Collaborator: Raul de la Cruz (BSC)

The Alya System is a Computational Mechanics (CM) code with two main features. First, it is specifically designed for running with the highest efficiency in large scale supercomputing facilities. By specifically designed its meant that Alya has been designed as a parallel code from the beginning and to solve in a flexible yet clear way every kind of CM model.

Second, 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 Alya 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).

3.1.1 *Application description*

Alya is based on Finite Element Methods. These types of applications can be divided into two main computational tasks executed inside a time-step loop: the element loop and the solver. These two tasks are executed as many times as the number of time-steps we want to run in the simulation.

In the first task (sparse matrix and RHS assembly), 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 (algebraic solver for symmetric and unsymmetric sparse systems), 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-Seidel).

The relative weight of the two previous tasks depends on the problem and type of elements used. It can be 10%-90% or 60%-40%. The speedup up depends on this relative weight. The assembly depends only on the load balance as almost no communcation is needed: only some global communcaiton to compute convergence critaria or some point-to-point communcaitons if gradients are needed. The solver is definitely the bottleneck for petascaling and almost all the effort has been concentrated on this part.

Looking at the code, 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 optimization support. Finally, the modules solve different sets of PDE's describing a physical problem.

The code is modular in the sense that different physics (modules) can be pluged in and out at compilation or runtime. The solvers are shared by all the physical modules, so any petascaling effort is automatically repercuted to all modules. In principle, the petascaling of the module only depends on the load balance supplied by the mesh partitioner. This is not necessarily the case for optimization which can also be applied at the module level.

Depending on the problem to solve, sometimes pre/post-processing steps are be required. Their descriptions and the computational loads compared to the main algorithms are as follows:

1. Pre-process: the mesh generation is excluded from the preprocessing. It thus includes the mesh partitioning, the organization of the data for the slaves, the communication scheduling, the computation of some permutations arrays, and the writing of all this information in restart files, one for each slave. To give an example, for the 27M mesh of the cavity benchmark used in PRACE, the pre-process is 25 minutes on SARA power6, while each iteration is around 12 seconds on 1024 CPU's.
2. Postprocess loads depend on the problem. If a stationary solution is sought, postprocess is only needed at the end of the run. For transient simulations (for example using LES turbulence modeling), it could become important.

To enable parallelization and to achieve petascaling in Alya, two external libraries are used in the code:

- LibMPI, as a message passing library, and
- LibMetis, a mesh partitioner for domain decomposition among the MPI tasks.

The parallelization strategy is based on MPI and OpenMP hybrid programming models. The parallelization is achieved through a mesh partitioning technique implemented inside the algebraic solvers. This implies that at each solver iteration the same results are obtained as the sequential counterpart.

At this time I/O is serialized and executed only over the master task. Parallel I/O using HDF5 is under implementation. No special strategy was originally used.

Three datasets which are available for Alya testing and benchmarking:

1. A simple 3D cavity flow which enables the use of a simple mesh (generated inside the code). For this dataset, the element assembly dominates and speedup is almost perfect.
2. Airflow in a nose. This data set consists of a real biomechanics problem. It uses a hybrid mesh so the load balance is now an issue: the scalability of the element assembly depends on the relative weight of the elements while the solver scalability depends on the number of degrees of freedom. These two criteria are incompatible.
3. Hemodynamics in brain. This is a also a real biomechanics case. It consists of 140 arteries, basically split by METIS in two parts, and a confluence region where METIS gives subdomains with up to 9 neighbors (depending on the total number of subdomains). This case is interesting for testing the communication as neither the work nor the communcations are well balanced by METIS.

All these datasets are big enough to be used for petascaling tests. At the same time, the two last presented datasets are based on real world problems, which can be run in a suitable amount of time.

3.1.2 *Petascaling Techniques*

Petascaling techniques have been implemented and tested on MareNostrum and the SARA Power6 machines, as the Maricel prototype is a bit limited for such tests. The techniques implemented for Alya to enable petascaling are the following:

Hybrid parallelization:

1. OpenMP has been implemented in the solvers. The symmetric graph used originally in the symmetric solvers was replaced to have better CACHE access and to enable it to use OpenMP directives for the Matrix-vector products (they were not productive when using the symmetric graph).

Load balancing:

2. Nothing has been done. However, other mesh partitioners are going to be tested to solve the hemodynamics in the brain benchmark, which exhibits pathological load balancing using METIS.

Minimization of the communication overheads:

3. Some global communications were put together in the algebraic solvers.
4. One AllReduce have been substituted by an AllGather in the DCG solver.

Parallel I/O:

5. HDF5 has been implemented.

Checkpointing:

6. Solution is dumped into files at each n time steps to be prescribed by the user.

Algorithms:

7. Most of the work was dedicated to this aspect. A new solver (DCG) has been implemented to solve the continuity equation. Details are given in the following point.

Three main parts of the Alya code should be considered during the petascaling effort: the preprocess, algebraic solvers and postprocess. The details of such applied techniques for each part are as follows:

1. **Preprocess**: Generally the preprocess is not included in the petascaling strategy. However, in some cases it could be a large part of the complete simulation cycle. Three approaches are being followed:
   - We parallelized some of the loops using OpenMP during this phase.
   - A non-negligible CPU time was spent in writing the restart files for the slaves. Virtual buffers have been created to accelerate this step.
   - In addition, a parallel automatic element splitting is under implementation. This will enable Alya to obtain huge meshes starting from a coarse mesh in parallel. Parallel efficiency is expected to be good for this phase and will only involve point-to-point communication.
2. **Algebraic solvers**: One possibility to increase the speedup is to decrease the relative weight of the solver during the computation. This was achieved by an algorithmic modification in order to converge the solver in less iterations:
   - We implemented a Deflated Conjugate Gradient solver to solve the pressure equation. In addition, we have implemented two parallelization strategies and reordered some operation to decrease the number of global communications. One is based on a global communication to exchange the coarse space vector and the other is based on an all-gather communication. Some more testing is needed on Power6 and on a more favourable example than that chosen to test the two algorithmics. See next figures.

     More details about the parallelization of the algebraic solvers can be found in the following references [18, 19].

We also worked on a linelet preconditioner to accelerate the solver convergence in boundary layers and treated its implementation in Parallel.

- GMRES speed-up has been treated by considering classical orthogonolization instead of modified Gram-Schmidt, which can be beneficent in some cases.

  The figures below compare the classical CG, the DCG and the DCG with linelet preconditioning for a 2D turbulent and thermal cavity flow:

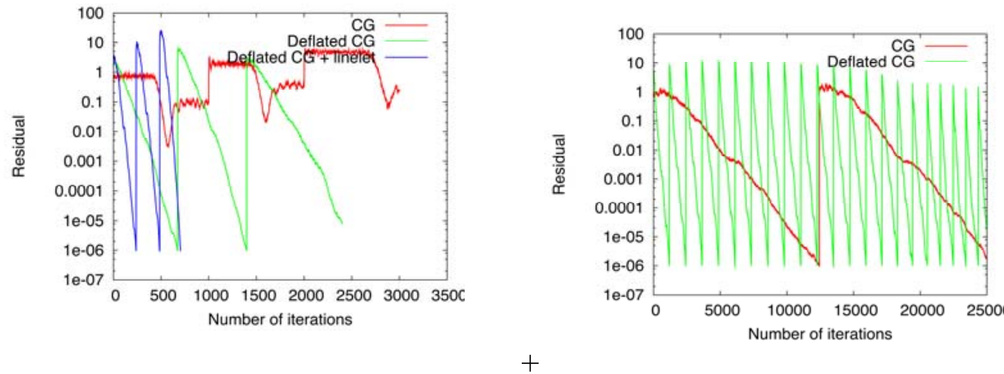

Figure 7 Comparisons of classical CG with Deflated CG. (Left): 2D thermal and turbulent tall cavity. (Right) Hemodynamics in brain, PRACE benchmark.

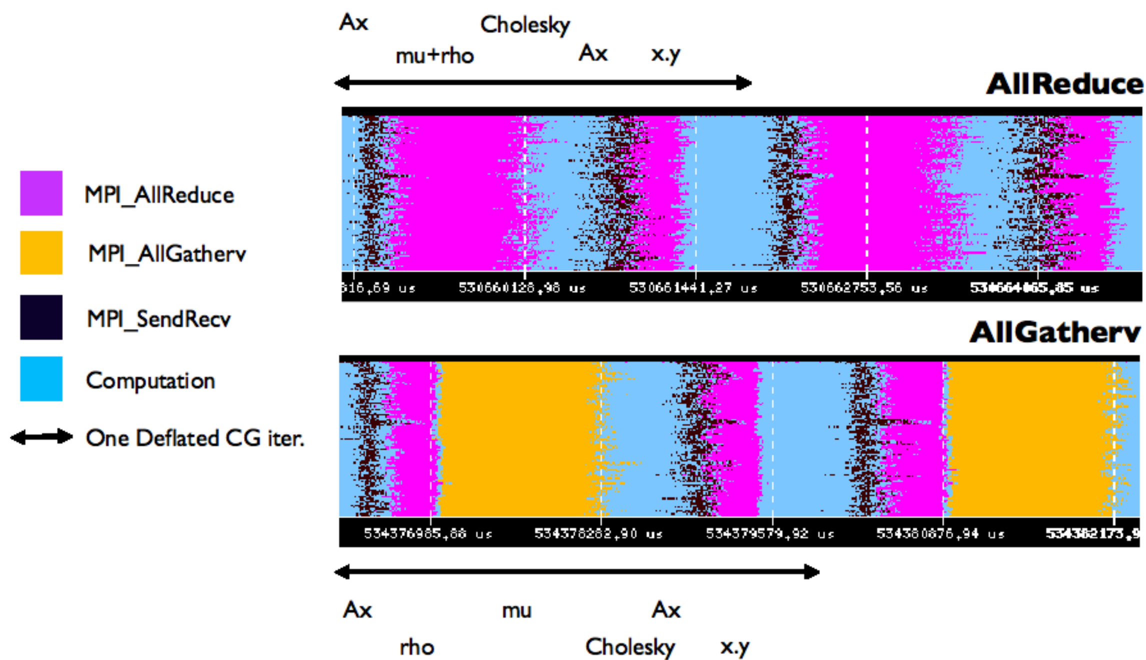

Figure 8 Comparisons of AllReduce and AllGather for the coarse space vector in the deflated CG. Hemodynamics in brain PRACE benchmark.

1. **Postprocess**: Parallel I/O is under implementation using netCFD and preliminary results are still not available.

Understanding the results of the two MPI implementations tested for the DCG are the main challenges for the previous petascaling techniques. Some traces have been obtained on MareNostrum. They show that the AllGather does not perform as expected. More traces are needed on other platforms to understand why. On Exascale machines, a drastic algorithmic change will be needed as the basis of the Deflated solver is a direct solver.

An estimated effort involved in employing the different petascaling techniques are:

- Preprocess. 5 PM.
- Algebraic solvers. 5 PM.
- Postprocess. 1 PM.

In the Deflated Conjugate gradient, huge point-to-point or all-gather communication patterns are required, which consume most of the solver time with respect to the very few computations involved.

Preprocess (apart from mesh generation) has some serial parts which can be parallelized using OpenMP directives for shared memory treatment. As the preprocess requires lots of memory, it is usually carried out on a large shared memory computer (e.g. ALTIX).

3.1.3 *Results*

Very good improvements have been achieved for the pressure Algebraic solver. Orders of magnitude can be achieved in the number of iterations. Further work is needed to treat pathological cases.

METIS can fail in some cases, like the benchmark hemodynamics in the brain. Other mesh partitioning libraries will be tested in the future for pathological problems.

The implementation of the DCG has permitted not only to go much faster but also in some cases to obtain convergence that could not be achieved with classical solvers.

One order of magnitude has been achieved by changing the pressure solver from the CG to the linelet preconditioned Conjugate gradient. This was achieved by an algorithmic approach.

Although the number of iterations have been reduced, the solver requires large all-reduce or all-gather operations. Therefore, the speedup of the particular algorithm will be lower than that of the original CG, even if the sequential counter part can be orders of magnitude lower.

The expected speedup has not been reached for the AllGather DCG implementation. More tests must be carried out to complement the traces obtained on MareNostrum. For now, no definitive conclusion can be drawn.

On the other hand, definitely, pre-processing is a bottleneck for the petascaling task of the Alya code.

3.1.4 *Conclusions*

Some lessons learned are that the all-gather implementation for the Deflated Conjugate Gradient was expected to perform better. Thus, some traces are being analyzed to understand the reason.

Never the less, some of the employed petascaling techniques have been efficient. For instance some iterative solvers have been treated for petascaling, resulting in quite good results. On the other hand, parallel I/O is under implementation and it remains unsolved, thus requiring some effort to improve the petascaling version.

3.2 AVBP

Written by: Bertrand Cirou, CINES/GENCI

3.2.1 *Application description*

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).

AVBP is written in Fortran90. It uses 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. The Fortran90 code is commented, and well organized.

AVBP is based on Finite Element Methods, and is 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). These two tasks are executed as many times as time-steps we want to run in the simulation.

AVBP was already ported on IBM BG/L, IBM power5, XT4 and Itanium, therefore the porting task was easy.

3.2.2 *Petascaling techniques*

MPI collective call usage

We noticed that multiple point to point communications did not scale. We have re-ordered the arrays data in order to be able to issue collective communications on these arrays. This is not a platform specific optimization. The first difficulty is to identify in the code a set of point to point calls that could be a good candidate for being replaced by a collective call. Then the data arrays used in these point to point calls must be reordered. Before this optimization the code did not scale beyond 4096 cores. After this optimization the code can scale beyond 12288 cores.

BLAS library usage

We noticed that about 15 % of the time was spent in some self written computation loops. It is well known that hardware vendors provide optimized computation libraries that perform

better than self written code. We have replaced the computation loops by equivalent calls to the BLAS library. This is an agnostic optimization as BLAS is available on any parallel machine. However, due to indirect access in arrays, the self written code did not match to exactly one BLAS call. We had to copy the data and use two BLAS calls. Before the optimization the speedup was good but after the optimization the speedup was worst.

3.2.3 *Results*

SGI Altix ICE 8200EX (CINES)

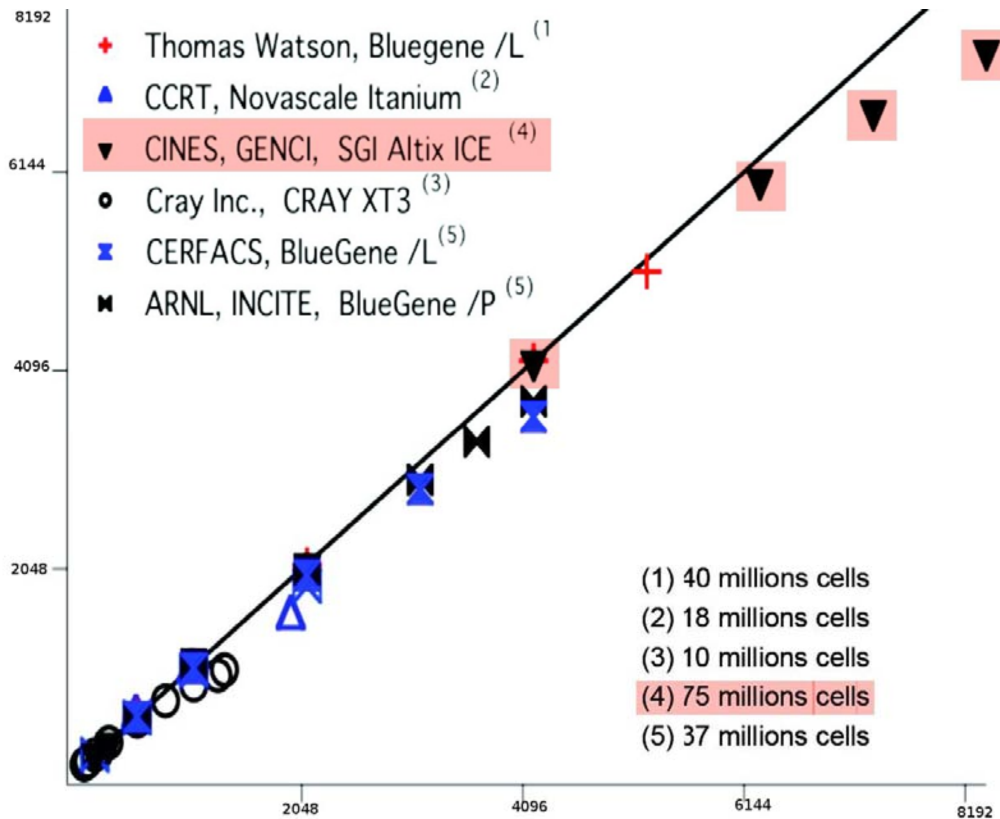

Figure 9 Performance of AVBP

# cores	total time (seconds)
128	1540
256	815.53
512	384.42
1024	181
2048	89
4096	46.24
6144	33.29
7168	29.78
8192	27.27

Table 2 Total computation duration in seconds for helicopter turbine benchmark on the SGI ICE 8200EX

BlueGene P (Argonne)

Low memory version of the benchmark with 1 master node:

# cores	total time (seconds)
511	1590
1023	810
2047	420
3071	290
4095	230

Table 3 Total computation duration in seconds for the helicopter turbine benchmark (low mem) on the BG/P

In this benchmark test case, AVBP is used without chemistry (which lead to a maximum of 15% of time added). The Navier – Stokes model uses two stencils (which is more stable). With these parameters, we obtain two kinds of information:

First, execution time is essentially spent in the slave routines. More precisely, the subroutine slave_temporal, which made the computation use more than 90% of the execution time. In fact, one particular subroutine is more interesting: compute. On average it uses 88% of the execution time. It is called by slave_temporal once and afterwards it is used as many times as the number of iterations. It is hard to improve this routine directly, but it appears to depend on another one, which can probably be threaded to reduce 5% to 8% of the execution time, on average. Second, long time is spent on some communications. During the computation, a lot of MPI calls are made.

3.2.4 *Conclusions*

We managed to improve the scalability of AVBP. Now, thanks to the insertion of collective calls the code scales beyond 12288 cores. The use of BLAS seemed to be a good idea but as the matching of the existing code with a single routine was not perfect we had to restructure and copy the data, which lead to performance degradation. So we do not use BLAS.

The code execution is well balanced due to the use of Metis. We could improve this static load balancing by distributing dynamically the workload inside an MPI process onto two or more threads (OpenMP). These threads could migrate on other cores of the node where some MPI processes are waiting for communication.

3.3 BSIT

Written by: Mauricio Araya, BSC

Reverse Time Migration (RTM) is a technique used in geophysics to deliniate sub-surface structures. This technique is based on a two-way wave propagation equation (PDE). In order to solve this PDE we need to deal with a laplacian calculation in a time loop, which is the most compute demanding segment of the RTM

3.3.1 *Application description*

From a high level point of view BSIT follows a master-worker scheme as shown in Figure 10. The master performs the following tasks: distributes the work, coordinates the workers and assembles the final result. The master source code has 15.800 lines of code (LOC). BSIT was developed mainly for the Cell processor which is a heterogeneous processor with two different types of processor cores. These are the PowerPC Processor Element (PPE), which is a conventional processor, and the Synergistic Processor Element (SPE), which is a vector processor. Every worker executes the RTM kernel, where some subtasks are executed in the PPE and others in the SPE. The kernel PPE code has 3007 LOC and the kernel SPE code has 3388 LOC and the master code has 15500 C LOC and bash scripts. Also, the application includes some 400 LOC of bash scripts and some LOC of Makefiles.

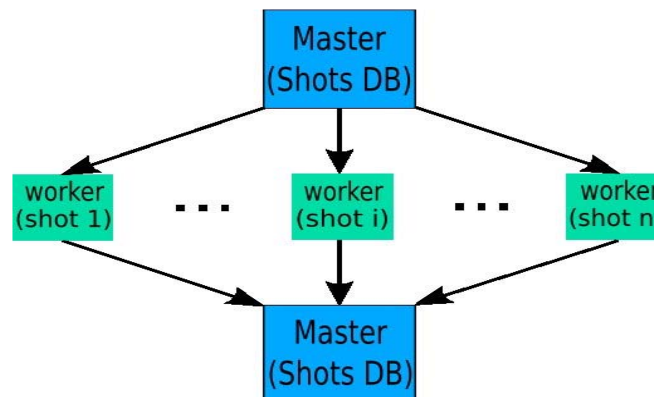

Figure 10 Master-Worker Scheme

The application uses the following libraries: librt for asynchronous IO, libnuma to enable the NUMA capabilities, libspe2 for SPE intrinsics, libpthread for the pthread management intrinsics, and libm for the mathematics function intrinsics.

The readability of the code is quite good for the master and the PPE code and is also well commented although not so well documented. The readability of the SPE code, however, is lower than the former. The SPE code, where the most optimization efforts were deployed, is almost completely written with assembler-like intrinsics, thus exposing the classical trade-off between readability and full optimization. It is clear that the only way to keep an acceptable

level of readability of the code is through well placed comments, in particular if other layers of optimization will be added later.

The algorithms are naturally separated based on their purpose: the master pre-processes and coordinates the work and the slaves only focus on crunching numbers. The master pre-processes the “raw” data which consists of a gigantic velocity model and seismic traces. The model is divided in pieces (“submodels”) that fit the resources of the computational nodes available. Every one of these sub models and corresponding seismic traces are the input for the wave propagation simulation (RTM). The preprocessed data is organized in a DB and distributed among the workers.

RTM is based on a two-way wave PDE, this implies solving two times the acoustic wave equation, called forward and backward propagations (see Figure 11). The Finite Difference solver kernel computes the stencil and the time integration for every iteration of the time dependent loop. This kernel is the most computationally demanding code of the RTM.

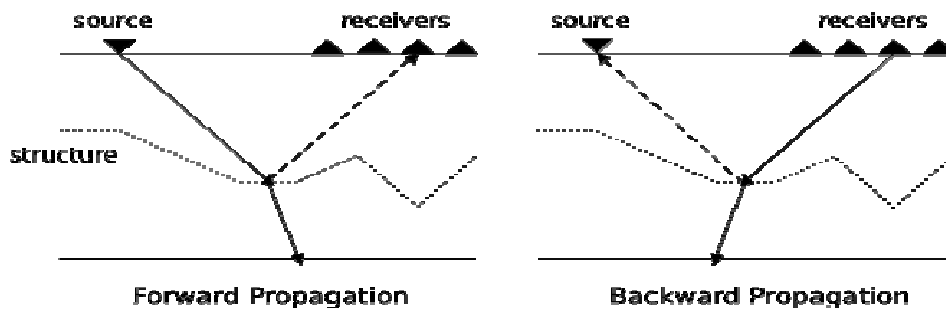

Figure 11 RTM two-way wave propagation

Furthermore, RTM includes the following tasks: the source wave introduction (shot), the receivers trace introductions and the absorbing boundary conditions (ABC) computation (see Figure 12 for the relative execution times). Finally, in every step of the backward propagation the correlation of the forward and backward wave fields are carried out.

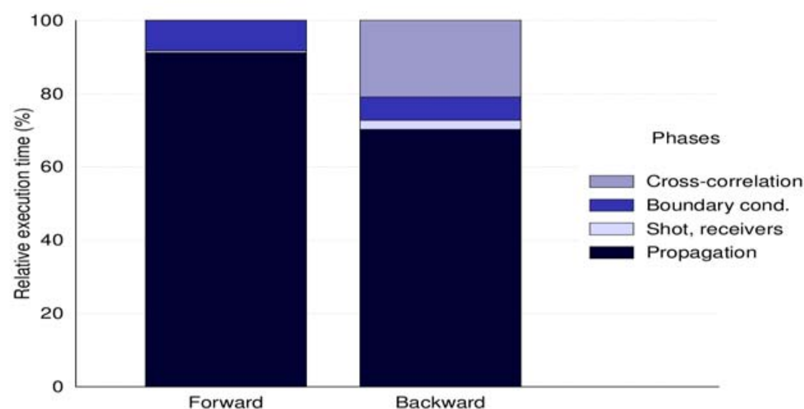

Figure 12 RTM time breakdown

At the same time that the workers (RTM kernel) are sending back their results to the master, where the data is post-processed in order to generate the final image result. In total and for a regular run, the master elapsed time accounts for 10%, and the worker elapsed time for 90%.

The master performs both pre and post processing, which is mainly an I/O intensive set of tasks. The runtime cost of the master is a function of the size of the problems, the number of shots to be executed and some architectural parameters (net topology, filesystem and I/O), but generally speaking accounts for 10% of the whole execution.

The Pre-processesing tasks are: Organization of the data for the workers (DB), the communication scheduling, the computation of some global parameters, and information files setup, one for each worker.

The Post-processesing task is basically summation of the workers output to generate the final resulting image. Furthermore, filters and imaging conditions are applied to the composed image.

Parallelization is based on MPI and OpenMP. The original code uses a hybrid parallelism approach, where the Domain Decomposition (DD), done by MPI, is used to distribute work off-node. DD looks for well balanced communications and workload, and due to the fact that the computational domain is structured, the resulting domains resemble the shape of the initial domain. OpenMP is used to take advantage of the node's cores. Our experience shows that for this kind of problem (PDE-FD-structured grid) the pthreads approach is not efficient enough.

The application is highly intensive in terms of I/O. On the worker side, we overlap I/O with computations, thus asynchronous I/O (librt in linux) is used. On the master side, the application takes advantage of distributed filesystem to reduce to some extend the I/O pressure and the communication among master and workers.

Many synthetic and academic datasets are available, in particular the SEG data sets, which are a sort of benchmark/acid test for geophysics codes at http://research.seg.org/3dmodel/salthome/intro.html.

Due to IPR restrictions we can not provide commercial real-life data sets. Unfortunately, the SEG data sets are not designed for petascaling and are only designed to test algorithms. For instance the SEG data set size is 210x676x210 (120 MB) single precision. In order to have access to big enough data sets, an extra effort is required to generate synthetic examples. The generation of such data sets is as complex as the model intended, and the size is not a limiting factor.

3.3.2 *Petascaling techniques*

The following petascaling techniques have been implemented and tested on MareNostrum and Maricel:

1. Hybrid parallelization
   OpenMP has been deployed in the kernels. MPI is used for the master and workers in case of DD.
2. Load balancing
3. Double-buffering
4. Minimization of the communication overheads
   Optimizing communication patterns so as to reduce communication as much as possible and using sendrecv functions.
5. Parallel I/O
   Asynchronous I/O

Efficient use of GPFS

6. Checkpointing
   Implemented, but not used in the production code version.
7. Algorithms Optimizations
   Most of the work was dedicated to this aspect. In particular, to optimize the mathematical kernels, and the distribution of data among workers.

Two main parts of the code have been reviewed:

Worker:

- We parallelized all task within the main loop of the worker using OpenMP. In the Cell/B.E. case pthreads were used and double-buffering technique employed (see Figure 13).
- Domain decomposition well balanced with minimal communication overhead.
- Parallel I/O is implemented.

Master:

- Parallel I/O is implemented.
- Master to workers communications is partially replaced for file distribution through GPFS.

Figure 13 Double-buffering technique

During the petascaling optimizations we have faced many challenges in, particular with regard to I/O, since the computations have been well optimized and contain very few communication overheads. Therefore on the I/O front, the distributed files system still lacks good performance and reliability, and requires many tweaks to keep pace with the computation.

To get a balanced work distribution, every node should have the optimal amount of data which means that the initial raw data is divided in such a way that every computational node fully utilizes their resources in terms of memory.

The main communication in BSIT is between the master and the workers which takes place two times. One at the beginning and one at the end of the worker execution. This communication is always point to point. Another type of communication which may take place if the model is big enough to force DD, in that case the nodes that are processing an overlap domain exchange information at every time step. This communication is also point to point and works as a synchronization point among nodes.

Both pre and postprocessing tasks (master) are performed in parallel. The workers return their results in an embarrassing parallel way, where the master is constantly receiving data and combining them.

3.3.3 *Results*

We have achieved full and efficient utilization of the Cell platform. BSIT is in fact of industrial production calibre at the moment. Our first implementation of the system reduced the elapsed time of a regular run of an order of magnitude, at worker level, from hours to minutes. Then our Cell/B.E. implementation went even further, reducing the kernel elapsed time by one order of magnitude. The RTM kernel implementation is close to optimal according to performance indicators (e.g., 93% of the peak bandwidth throughput).

Also, the application shows at least 13.0x speedup when compared against a reference traditional multi-core platform based on a PowerPC 970MP processor:

Platform	Avg power [W]	Execution time [s]	Arithmetic Throughput [GFlops]	Scalability	Energy Efficiency [GFlops/W]
JS21	267	45.0	8.3	3.8	0.03
QS21	370	2.5	104.6	15.2	0.28

The only possible dark cloud remaining is the filesystem, which depending on the size of the dataset and the number of nodes available might be a problem.

3.3.4 *Conclusions*

The main issues solved are: speeding up the computation and reducing the impact of I/O in the global performance of the application. Also, the application was ported to a novel platform, which required an important effort in algorithm mapping.

We do not foresee any additional major optimization to this application. Eventually minor optimizations will be deployed. This is because the performance achieved indicates that we reach various limits, in particular bandwidth. Thus, the main algorithm becomes memory bounded.

The above exposes the main architectural shortcoming, the bandwidth. Increasing the bandwidth of the HPC architecture could improve the performance of BSIT further. This is particular true for the Cell/B.E. platform. Also, it would be useful to have larger local store (QS22) and L1/L2 cache (JS21). Finally, the I/O performance will improve if faster storage systems are available.

3.4 Code_Saturne

Written by: Andrew Sunderland, STFC

Code_Saturne® is a multipurpose 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 starting version of Code_Saturne in PRACE was v.1.3. Recently v2.0 beta has been released which includes the optimizations described here (apart from the improvements to partitioning which is an external feature).

3.4.1 *Application description*

Source Code and Languages Used

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 benchmarked versions (no-view versions). The code base is very large, consisting of 500000 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 later this year, the Fortran77 will be replaced by Fortran95 structures.

Libraries Used

No external libraries are required for compilation of the code. Routines undertaking equivalent operations to Basic Linear Algebra Routines (BLAS) operations are provided within the code suite. 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)

Readability of the code

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.

Main algorithms in the original code and their relative computational load.

- Finite Volume
- Turbulence Models
  - RANS (Reynolds Averaged Navier-Stokes Simulations)
  - LES (Large Eddy Simulations)
- Sparse Iterative Linear Solver for both Pressure and Velocity
  - CG (Conjugate Gradient (v1.3.2)
  - MG (Multigrid) (v2.0)

Full simulations generally involve many hundreds or even thousands of timesteps. Overall compute time is usually dominated by time spent in the Sparse Iterative Linear Solver, solving the Poisson equation for the pressure (if incompressible).

Pre/post-processing steps

- Grid Partitioning via METIS, SCOTCH (not included in PRACE benchmark)
- Post processing via ENSIGHT, PARAVIEW (not included in PRACE benchmark)

Parallelization Strategy

Code_Saturne uses a distributed memory parallelism using domain decomposition. The MPI API is used for both point-to-point and global communications between processes. The characteristics of the domain decomposition are determined in the mesh partitioning stage of the calculation. Modern partitioning software is very adept at load-balancing the domains across processes, which is crucial to good parallel performance.

The code uses a classical ghost cell method for both parallelism and periodicity

- Most operations require only ghost cells sharing faces
- Extended neighbourhoods for gradients also require ghost cells sharing vertices

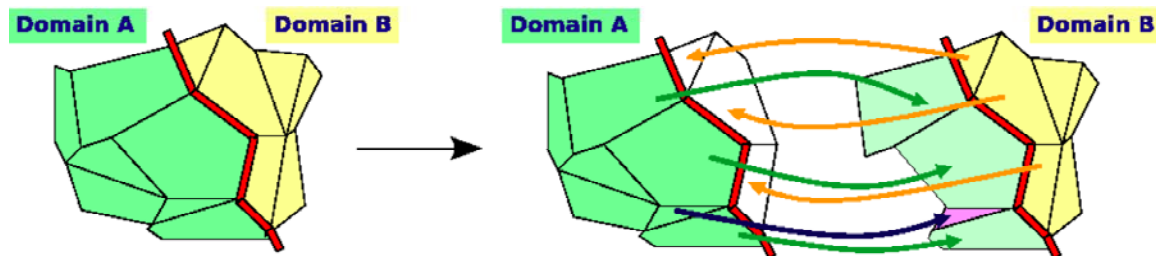

Figure 14 Ghost Cell Method

I/O strategy

Reading Data

The grid partitioning software produces a separate file for each process. These files are then read-in in simultaneously by the complete set of processes from one directory. Although in theory this is parallelised IO, in practice system limitations usually prevent this approach scaling on even relatively modest core counts.

Writing Data

Blocks of data are assembled in succession on processor rank 0 for writing to disk. The outputs in the original codes are therefore serial in nature. Improving the I/O has been a major focus of the petascaling approach.

Latest Release

EDF recently released Version 2.0 Beta of Code_Saturne, which includes the parallel optimizations summarized below:

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 more necessary to redo the full preprocessing (mesh pasting especially)
- Optional parallel algorithm for mesh pasting (not yet compatible with periodicity)

Parallel Solver

- New algebraic 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.

Parallel IO

- Restart files are now read and written using MPI-IO optimised libraries, when available

Description of available datasets for the application:

10 Million Cell Dataset

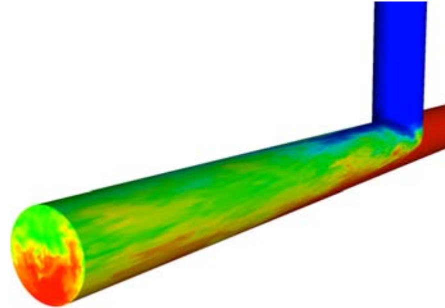

Figure 15 T-junction Dataset

This case deals with an isothermal Large Eddy Simulation in a T-junction. Only the dynamics of the flow is investigated. The runs are carried out for 100 time steps, starting from an already developed flow.

100 Million Cell Dataset

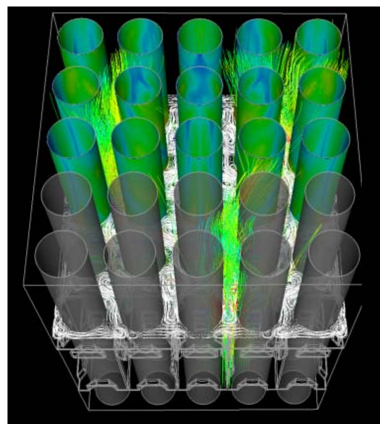

Figure 16 Mixing Grid Dataset

This dataset involves an unstructured grid and 100 million cells. The simulation represents the flow around a bundle of tubes in a nuclear reactor, where the geometry is too complex to be represented by structured gridding. The problem represents a very large-scale computational challenge (requiring high-end systems) as the flow is strongly three dimensional, with secondary vortices existing between pipes. The simulation uses a k-epsilon model and starts from an already developed flow.

Both the 10M and 100M are computational challenges suited to petascale architectures. Work is currently underway to produce a 500M cell dataset, based on the existing 100M cell dataset.

3.4.2 *Petascaling techniques*

Summary of Petascaling techniques undertaken

- Replacing the Conjugate Gradient method in the linear solver with a Multigrid solver
- Replacing Serial I/O with Parallel I/O
- Parallel Grid Partitioning

Main algorithm parallelization & challenges

Linear Solvers

In general, upwards of 80% of run time is spent undertaking the solution of sparse linear equations. Most algorithms for such solvers are built upon vector-vector and matrix-vector operations.

Conjugate Gradient Solver

The Conjugate Gradient Method is an iterative algorithm for the solution of large, sparse, positive-definite systems of equations. As described in netlib.org, it is the oldest and best known non-stationary iterative method. An initial guess is made for the solution, usually based upon a result from a previous iteration. The method proceeds by generating vector sequences of iterates (i.e., successive approximations to the solution), residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. Although the length of these sequences can become large, only a small number of vectors need to be kept in memory. In each iteration of the method, two inner products are performed in order to compute update scalars that are defined to make the sequences satisfy certain orthogonality conditions. For a symmetric positive definite linear system these conditions imply that the distance to the true solution is minimized in some norm.

In a distributed-memory parallel scheme this results in distributed matrix-vector and distributed vector-vector operations, where all operations are of fixed dimension through out (i.e. the dimension of the system). The initial version of the code is optimised to take advantage of IBM ESSL libraries if available for these operations.

Multigrid Solver

The strategy of multigrid methods is to define a hierarchy of grids, ranging from a fine grid (the dimension of the complete system) to a coarse grid with a much reduced set of grid points. The main idea of multigrid is to accelerate the convergence of a base iterative method by correcting, from time to time, the solution globally by solving a coarse problem. Points on the coarser grids are a subset of points on the finer grids. The grid hierarchy can be traversed in V or W-cycles. On each level of the hierarchy an iterative solver (here the Conjugate Gradient solver) is called. Transformations of the grid between fine and coarse structures usually take place via a Restriction process or matrix and a Prelongation process or matrix.

I/O

The development of efficient methods for reading data from and writing data to disk has become hugely important for codes that use high-end computing resources. This issue is particularly relevant for Code_Saturne, as the large-scale simulations that the code undertakes often require the input of huge datasets from disk and the output of large amounts of results files. The outputs are usually required at frequent intervals in order to model a system that changes with time. Visualizing the results is generally undertaken in a separate post-processing stage.

Serial I/O

In serial IO mode, data for each block is written or read successively by rank 0. A similar mapping of partitions to blocks is used for both serial and parallel I/O. However for serial I/O, blocks are assembled in succession on rank 0. Each block is written before assembling the next in order to avoid very large buffer requirements. A minimum buffer size limit is enforced in order to enforce the number of blocks in small cases. This avoids potentially expensive latency overheads. Figure 17 demonstrates the cost of serial IO overheads for a typical parallel simulation on the Cray XT4 system. The two lines represent the parallel performance of the overall code (Total Time) and the time spent in the parallel solver (Iteration Time). The difference between the two, which increases markedly as the processor count increases, mainly represents the cost of IO in the code.

Parallel I/O

As with serial I/O, the parallel I/O implementation uses an 'fvm_file' intermediary layer. This includes a subset of MPI I/O routines, with collectives prioritised. The fvm_file layer handles offsets and other metadata and provides functions such as 'fvm_file_write_global', 'fvm_file_write_block' with corresponding read functions. The implementation uses a global numbering scheme for redistributing partitions to blocks in preparation for parallel IO as shown in the example in Figure 18. Minimum block sizes may be set to avoid the generation of many small blocks. A block to partition redistribution is used when reading, a partition to block partition is used when writing. The parallel I/O is fully implemented for reading of preprocessor and partitioner output and restart files. The implementation allows for using (synchronous) explicit offsets and individual file pointer implementations. Experiments have taken place on shared pointers, but have been removed due to issues on some filesystems, and limited interest.

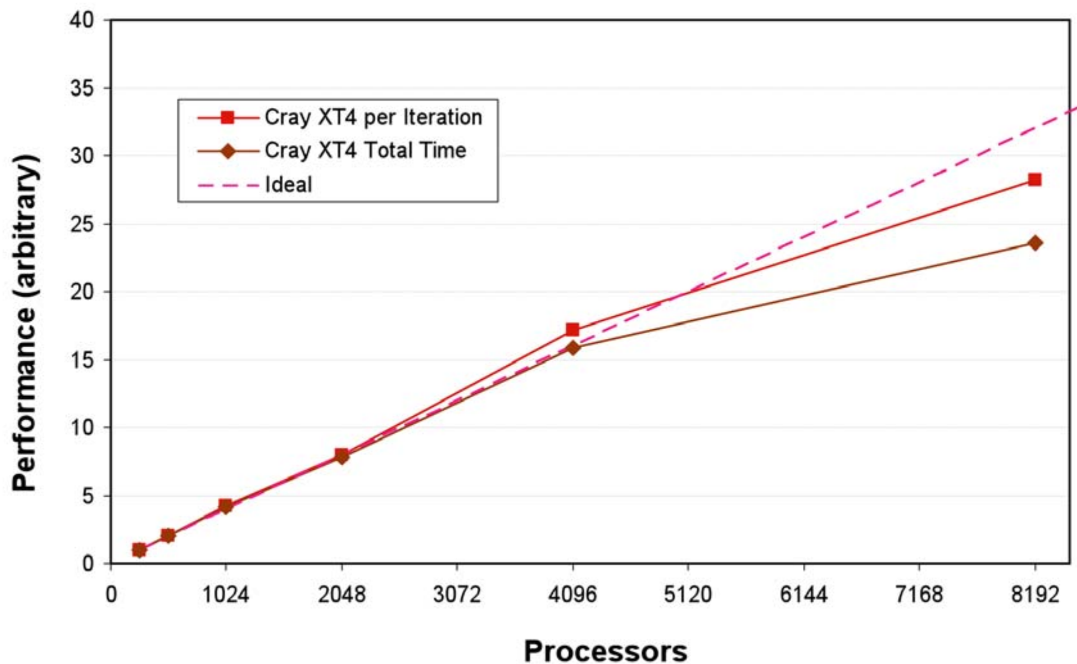

Figure 17 I/O Overheads of Serial I/O on the Cray XT4

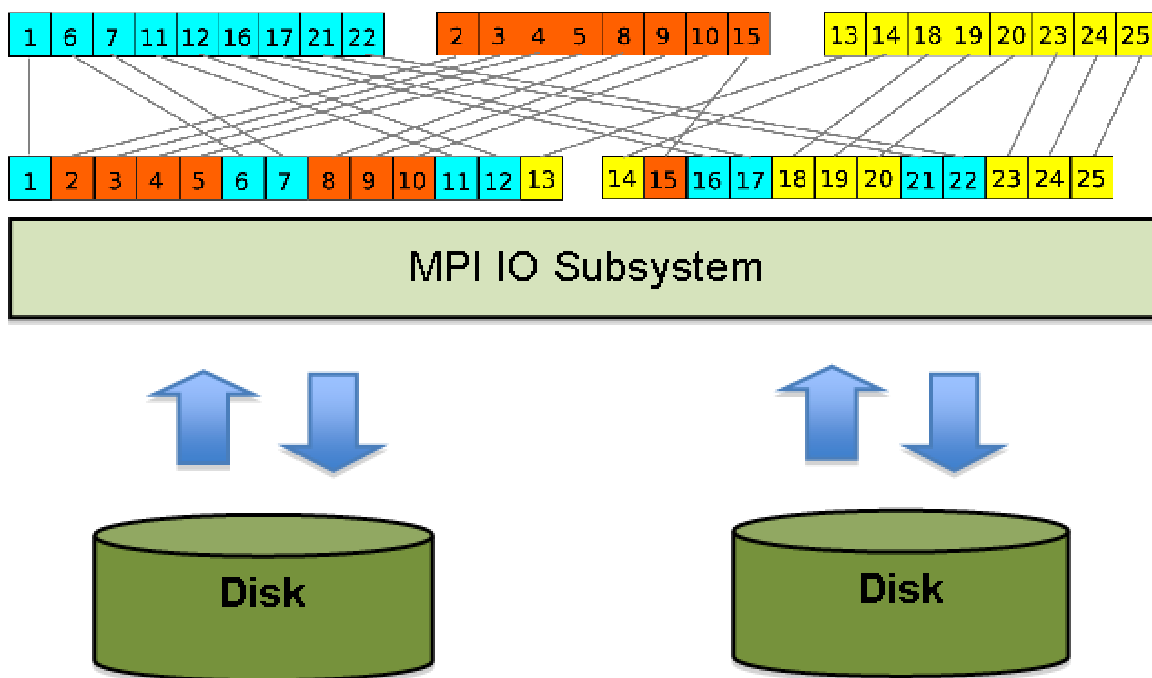

Figure 18 Parallel MPI/IO implementation in Code_Saturne v.2.0

All communications are undertaken using MPI. Collective and point-to-point MPI communications are used throughout the program. The program takes advantage of asynchronous communication modes for point-to-point messages between processes. On Cray

architectures, mechanisms in the hardware allow communications to occur concurrently with computation when the message passing mode is asynchronous.

Version 2.0 of the code is already prepared for parallel mesh partitioning. The mesh is read by blocks via a canonical / global numbering and is redistributed using cell domain number mapping. It is therefore straightforward to apply a new mesh partitioning algorithm in order to obtain an alternative cell → domain mapping. The redistribution infrastructure implemented in v2.0 of Code_Saturne and a separate project is investigating the use of third-party parallel partitioning software such as PARMETIS and PT-SCOTCH.

3.4.3 *Results*

Cray Pat Analysis on XT platform:

Profiles of v1.3 suggested that the collective communication MPI_Allreduce is prevalent. Otherwise most communications consist of asynchronous point-to-point message passing.

```
| 100.0%  | 37132 |       -- |      -- | Total
|------------------------------------------------
|  35.7%  | 13272 |       -- |      -- | MPI
||-----------------------------------------------
||  10.7% |  3987 |  437.43 |   9.9% | MPI_Allreduce
||   9.3% |  3471 | 2243.79 |  39.3% | MPI_Waitall
||   4.7% |  1750 |  282.24 |  13.9% | MPI_Barrier
||   4.6% |  1714 |   12.16 |   0.7% | MPI_Recv
||   3.8% |  1418 | 1124.28 |  44.3% | MPI_Isend
||   1.0% |   356 |  215.20 |  37.8% | MPI_Irecv
```

Removing calls to MPI_Barrier may increase parallel performance, though it is often the case that removing such synchronizations merely transfers synchronization overheads into other MPI routines.

The predominant message exchanging routine in Code_Saturne is cs_halo_sync_var. This routine involves halo data exchange. Where the structure is implemented with isend & irecv, with a global barrier between isends and irecvs to ensure irecv is posted before the send. Better performance may be obtained by reordering the isends and it would also be desirable to not issue isends & irecvs if the message length is zero. Additional performance improvements may be observed if there is some work to do between irecvs and isends as this will allow the communication to happen asynchronously. It has also been observed that in some cases the calls to cs_halo_sync_var could be combined to send one message rather than four.

Multigrid performance

Figure 19 shows the relative parallel performance of the new multigrid/conjugate gradient (MG) solver against the original conjugate gradient only (CG) solver for the 10M dataset. The performance improvement shown is highly significant as this stage of the calculation usually dominates execution time. On lower processor counts MG-based solves are around three times faster than CG-based solves. However on higher processor count the performance difference narrows to a ratio of around two-fold. Moreover MG solves on 2048 cores are

slower than those on 1024 cores. The parallel scaling limitations of MG are discussed in the section on ‘Key Bottlenecks’.

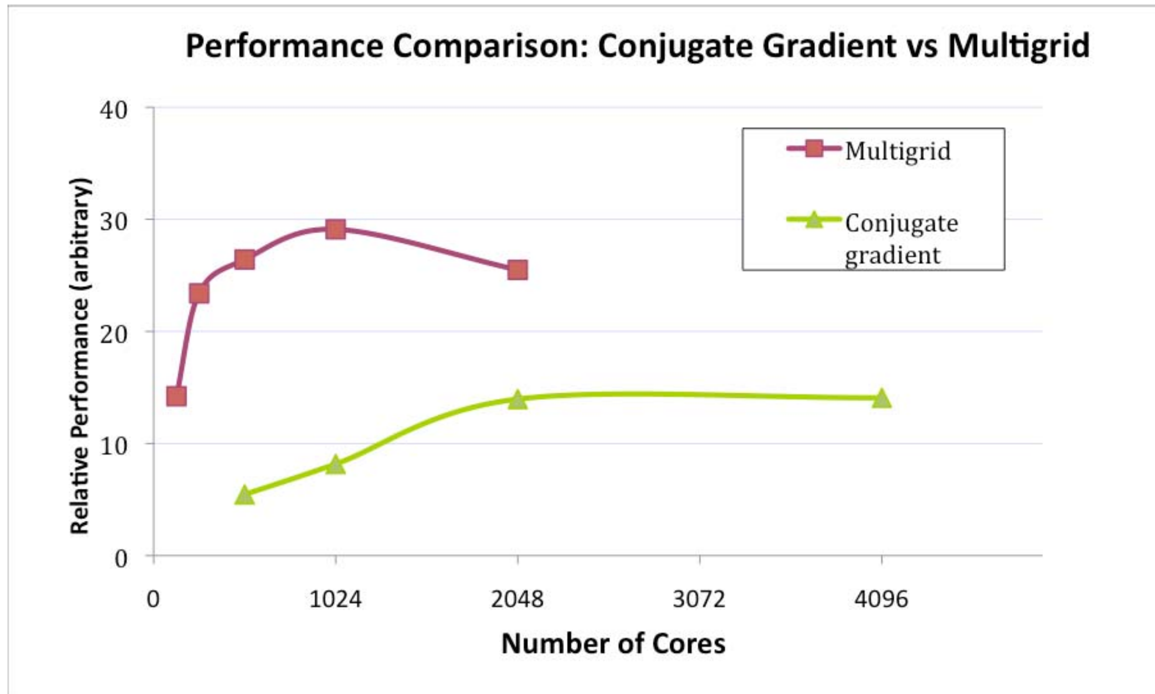

Figure 19 Relative Performance of Conjugate Gradient and Multigrid on the Cray XT4

Parallel I/O

Testing and analysis of MPI Parallel I/O in the newly-released version 2.0 of Code_Saturne is ongoing at the time of writing. Measuring accurately the impact of parallel I/O can be difficult on machines with large numbers of users, where data loads on I/O servers can vary dramatically. Future work will complete this analysis.

Parallel Partititoning

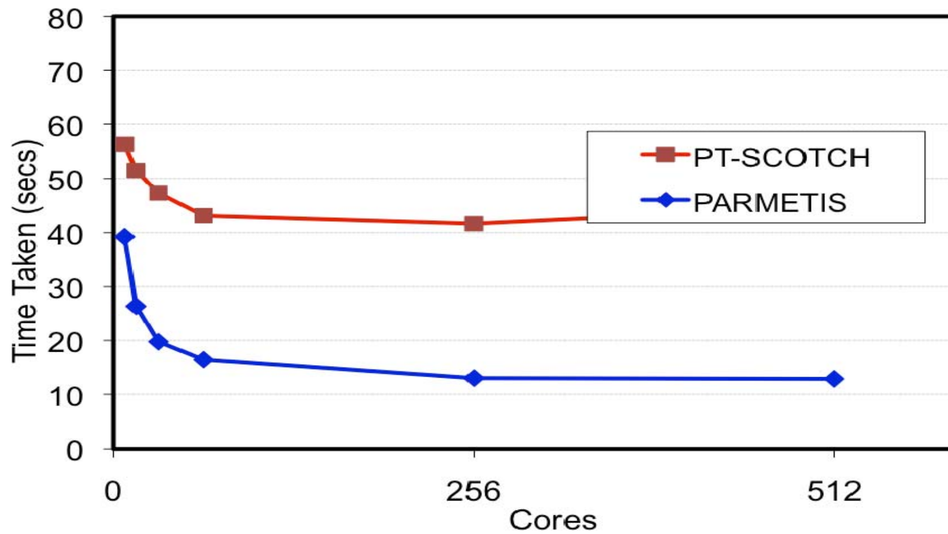

Figure 20 Parallel Performance of Mesh Partitioning software on the IBM BG/P

A performance analysis of two freely-available parallel mesh partitioners is shown in Figure 20. The case contains around 10 Million cells (though it is not the PRACE benchmark case). The parallel performance of neither package is good when using above around 100 cores. However it can be observed that PARMETIS runs several times faster than PT-SCOTCH. The quality of the partitioned meshes may have a far greater impact on overall performance than their associated cost. For further explanations see the discussion in the 'Key Bottlenecks' section below.

Parallel Performance of Code_Saturne on the Prototype Systems

Figure 21 demonstrates the parallel scaling of the Code_Saturne, based on timings from the parallel solver, for both the starting and final versions of the code (v1.3 and v2.0). The performances generally scale very well up to at least 512 cores for the 10M cell dataset case. Version 2.0 includes all the new petascaling techniques described in this section. On Huygens (IBM PWR6) performance results are shown up to the point that performance does not increase any further with extra core counts. Beyond this point (512 cores for v2.0, 1024 cores for v1.3) performance degrades quite markedly. It is believed that firmware problems on the current Huygens switch are responsible for this deterioration in performance, which it not observed on other prototype platforms. The parallel performance of v2.0 on Louhi (Cray XT5) has improved markedly compared to v1.3, both in overall speed and parallel scalability. Parallel scaling on the Jugene IBM BG/P remains excellent for both versions up many thousands of cores, though we generally observe faster performance with v2.0.

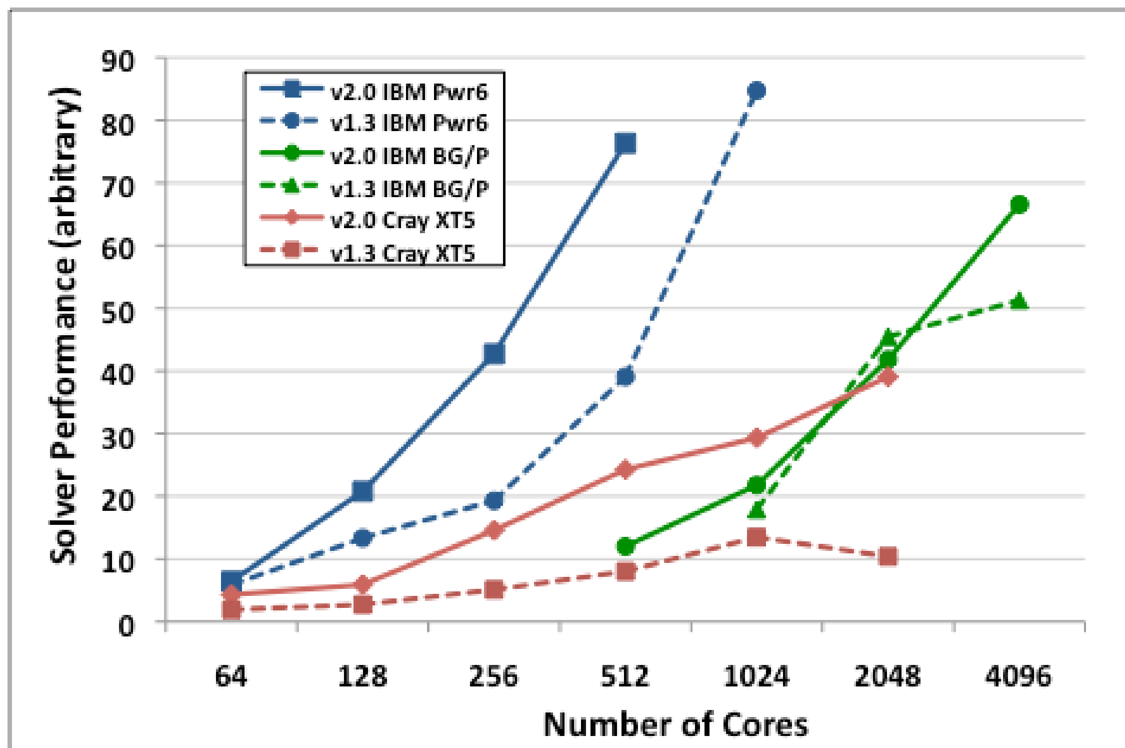

Figure 21 Parallel Scaling of Code_Saturn on the PRACE Prototype Systems for the 10M cell T-Junction dataset

Key bottlenecks for scaling:

Multigrid – Key Bottleneck to Scaling: Each task has a minimum of one grid point in the coarsest mesh allowable.

Although the introduction of multigrid methods can produce much more effective solvers the transitions through the hierarchy to the coarsest of the grids can cause scaling problems on large processor counts. In the current implementation the number of grid points on the coarsest grid cannot reduce to less than the number of processes in the parallel run. With a high processor count, fewer grid levels will be used, and solving for the coarsest matrix may be significantly more expensive than with a low processor count. This effect reduces the overall scalability of the code. This restriction may be resolved in future optimizations to the parallel code – see the Future Work section below.

Parallel Partitioning - Key Bottleneck to Scaling: Load Imbalance

In common with other unstructured mesh codes, the decomposition of the problem into equal parts for parallel processing is imperative for good scaling. Small imbalances may be inconsequential in the relatively long compute times associated with smaller processor counts. However compute jobs on petaflop architectures may contain hundred of thousands of tasks, where the granularity of the decomposition means relatively small imbalances can severely impact upon scalable performance. Moreover, partitioning the problem for such huge processor counts presents an extreme challenge for mesh partitioning software, which itself does not scale past a few hundred processors.

For example, a RANS simulation with 100 M tetrahedra + polyhedra (most I/O factored out) partitioned using METIS, has the following mesh decompositions:

96286/102242 min/max cells at 1024 cores = 5.8% imbalance

11344/12781 min/max cells at 8192 cores = 8.9% imbalance

As load imbalance increases with processor count, scalability decreases. Alternatively, if load imbalance reaches a high value (say 30% to 50%) but does not increase, scalability is maintained, but processor power is wasted. Future developments may attempt to reduce load imbalance by introducing weighting for domain partitioning, with Cell Weight = 1 + f(n_faces).

Load Imbalance and cache miss rates

Another possible source of load imbalance is imbalance in the speed of operations with different cache miss rates on different ranks. This is common with unstructured meshes and is very difficult to predict in advance. For example, with otherwise balanced loops, if a processor has a cache miss every 300 instructions, and another a cache miss every 400 instructions, considering that the cost of a cache miss is at least 100 instructions, the corresponding imbalance reaches 20%.

3.4.4 *Conclusions*

The parallel scaling of the original version of the code used in PRACE (v1.3) was very good on all the targetted prototype platforms, and our analysis shows that this excellent parallel performance has been maintained with the introduction of the faster multigrid solver in version 2.0. MPI-IO has also been introduced in order to remove bottlenecks when reading and writing intermediate and restart files on parallel platforms. The code has also been adapted to facilitate the use of 3rd party parallel mesh partitioning software during the pre-processing stage of the calculations.

Imrovements to the Parallel Implementation of the Multigrid Solver

In order to address the limitations of multigrid with the coarsest grids discussed above (i.e. a minimum of one grid point per core), the planned solution involves moving grids to nearest rank multiple of 4 or 8 when mean local grid size is too small. Most ranks will then have empty grids, but this will not incur any inefficiencies as latency usually dominates during this stage. The communication pattern is not expected to change radically, as partitioning is of a recursive nature (whether using recursive graph partitioning or space filling curves), and should already exhibit some sort of "multigrid" nature. This may be less optimal than some methods using a different partitioning for each rank, but setup time should also remain much cheaper.

Planned Future Improvements to the Parallel I/O

- *fvm_file_read/write_list* in the future using indexed datatypes to have all data shuffling inside MPI IO instead of outside.
- Infrastructure is being developed for parallel postprocessor output

- Indexed output data will be distributed to blocks - preferably by data blocks rather than index blocks for good balancing
- "*fvm_gather_...*" type functions are to be replaced by "*fvm_part_to_block_...*" type functions in file output and output helper code
- Multiple layers will be introduced due to handling of several file formats

Acknowledgements

The authors wish to express their gratitude to Yvan Fournier, Jerome Bonelle and others in the Code_Saturne development team at EDF for their major contributions to this work. We would also like to thank Kevin Roy from the Cray Centre of Excellence, UK for his assistance.

3.5 CP2K

Written by: 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.

3.5.1 *Application description*

CP2K is written completely in Fortran 95. The code is well-structured, polished and easy to maintain. Code is easy to read, and its clarity poses no problems to petascaling efforts. There are also some post-processing utilities, but they are not computationally demanding and thus not considered further in this deliverable.

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 (in the context of the general use of the code in the computational chemistry community) test case the DGEMM routine of Lapack was the most intense. Tuned libraries are used by the code whenever possible.

CP2K is parallelized with MPI. Details of the parallelization strategy can be adjusted in the user input, but in general it is done by affiliating atoms, or more precisely, their basis set functions, into MPI tasks. In an experimental part under development a hybrid OpenMP/MPI approach is used, but mostly to overcome a memory bottleneck. CP2K does not use MPI-I/O. Most of the I/O is carried out with the „spokesman strategy", i.e. the root task handles I/O. In rare cases all the tasks write scratch files.

For PRACE purposes we had two ab initio molecular dynamics simulations of water (one with 512 and other with 1024 molecules). They are actual research cases, in fact published some years ago. The simulation time was reduced to a couple of femtoseconds. The datasets were not sufficiently large to reach petaflop/s performance, as it seems that CP2K requires some tens of basis set functions per PE to scale sufficiently. However, ab initio simulations of systems featuring hundreds of thousands of basis set functions (corresponding to a petaflop/s machine assuming 10 TF PE's) are hitherto unseen, and real-world calculations are 2-3 orders

of magnitude smaller. The quadratically scaling memory requirement also blocks the use of datasets much larger than the present ones. In that respect the current one is at the limit of being feasible.

3.5.2 *Petascaling techniques*

It was decided with the developers that it is not worth rewriting the parallelization strategy itself, as they have plans for a major rewrite of the code in the future. The petascaling efforts within the PRACE project were the following, all of them done in close collaboration with the code developers:

- Load balance improvements of the collocation/integration of the Gaussians on the grids by fine tuning the code.
- An implementation of the routine that assigns atoms/basis set functions into PE's that scales linearly in memory requirement and would thus enable a calculation of larger datasets. The outcome is, however, much slower than the default one and therefore more or less useless in practice.
- New strategies for how to optimize the assignment were implemented: optimization target being the variance in the load, instead of the maximum effort.
- Preposting receiving routines in non-blocking communication featured in the intense routines for a better scalability in mind. This had no effect in practice.
- In a development version of the code, replacing 4 or 8 MPI tasks with OpenMP threads overcame a major memory bottleneck as it avoids replicated data needed in each process. The role of the BCO in this was very minor, mostly in discussions and in providing a development platform.

The main challenge is that there were no "low hanging fruits" to be picked - the code was very well written and parallelized. The throughout analyses and aforementioned minor efforts are approximated to have taken 1.5-2 pm.

As regards to parallelization practises; point-to-point (mostly non-blocking) and collective communication are both employed in the code. No exotic MPI features are involved. Most of the communication is done with non-blocking point-to-point communication overlapped with computation. The parallelization is "isotropic" meaning that the MPI tasks are equal regardless of their placing in the hardware. In MPP-Cray prototype different rank placements were tried. They did not show much difference between them.

Pre- or postprocessing is not necessary in all uses of the code and always computationally cheap, therefore no optimization or petascaling effort was put in considering them.

3.5.3 *Results*

There was not much to be done for the "conventional" part of the code (quick-step DFT): it is a well-written and tuned software very suitable for production runs in existing supercomputers, with not much potential for employing tens or hundreds of thousands of CPU cores.

The only major improvement with PRACE involvment was the OpenMP/MPI hybridization of the Hartree-Fock exchange part of the code, which enabled calculations not feasible otherwise. This part of the code is however very difficult to install at the moment and requires a large set of hacks and outside code (e.g. the LibInt library), glue and duct tape. This part of the code is shown to scale to over 10,000 processors in a Cray XT5 and would be of potential

petascaling code. It also compiles at the moment only with one compiler and in one platform, therefore these calculations are not listed in other PRACE deliverables.

The impact of the other performed improvements by the BCO may be described as minor. However, the long and throughout discussions with the developers on the performance of the code in different platforms as well as quantitative performance analysis data have most likely (read: hopefully) been useful for the development of the code, even though the improvements were not done by the PRACE BCO.

The mostly considered QS-DFT seems to scale up to 5 TF partition in the MPP-Cray and FN-Power6 prototypes. The available memory in MPP-BG/P is not sufficient for the larger dataset. Other prototypes were not considered. The test dataset does not enable scaling any further. It is also expected that the amount of communication would block the scaling also with larger datasets, even if the memory shortages would be somehow overcome. It should be noted that these statements apply only to one application area of the code (ab-initio molecular dynamics), the other ones featuring perhaps better scaling prospects, with the novel Hartree-Fock code, could prove to do so. The QS-DFT is however clearly the main application area of the code - to BCO's best knowledge - and therefore a rather fair restriction.

3.5.4 *Conclusions*

Lessons learned:

- Replacing MPI tasks with OpenMP threads (in some nomenclature "coarse-grained" hybrid programming) is a very good way to reduce communication and memory requirements if replicated data (in this case the sc. Fock matrix) is unavoidable.
- Otherwise, the current parallelization strategy of assigning variable-size blocks of basis set functions into MPI tasks have likely reached its limits in scaling: there are no tuning tricks left to try.

In general, faster CPUs (higher frequency, larger caches) and wider interconnect for non-blocking point-to-point communication will always accelerate CP2K. Possible improvements in different prototypes that would enhance the performance of the application:

- MPP-BG/P: memory increase to 1 GB/core (allows for calculation of larger systems), larger interconnect bandwidth, increased processor frequency
- MPP-Cray: larger caches and/or frequencies in CPUs to allow for faster linear algebra (DGEMM), more interconnect bandwidth/core (the effect is visible between the XT4 and XT5 parts of CSC's Louhi machine; the scalability is indeed better in the XT4 due to almost double bandwidth/core)

  FN-Power6: better inter-node bandwidth.

3.6 CPMD

Written by: Albert Farres, BSC

3.6.1 *Application description*

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 Dipartimento di Fisica Teorica, Università di Trieste, 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).

Two datasets have been used for PRACE. The first one is a dataset with 32 water molecules in the liquid phase at 100 Ry and MT pseudopotentials, and the second one contains 64 Ionic Liquids. The 64 Ionic Liquids dataset is big enough for petascaling.

3.6.2 *Petascaling techniques*

The original code from developers scales well on all the platforms and is actively maintained by the developers, therefore no special techniques were considered for the PRACE project.

3.6.3 *Results*

Below are the execution times on several platforms for 64 ionic Liquids dataset:

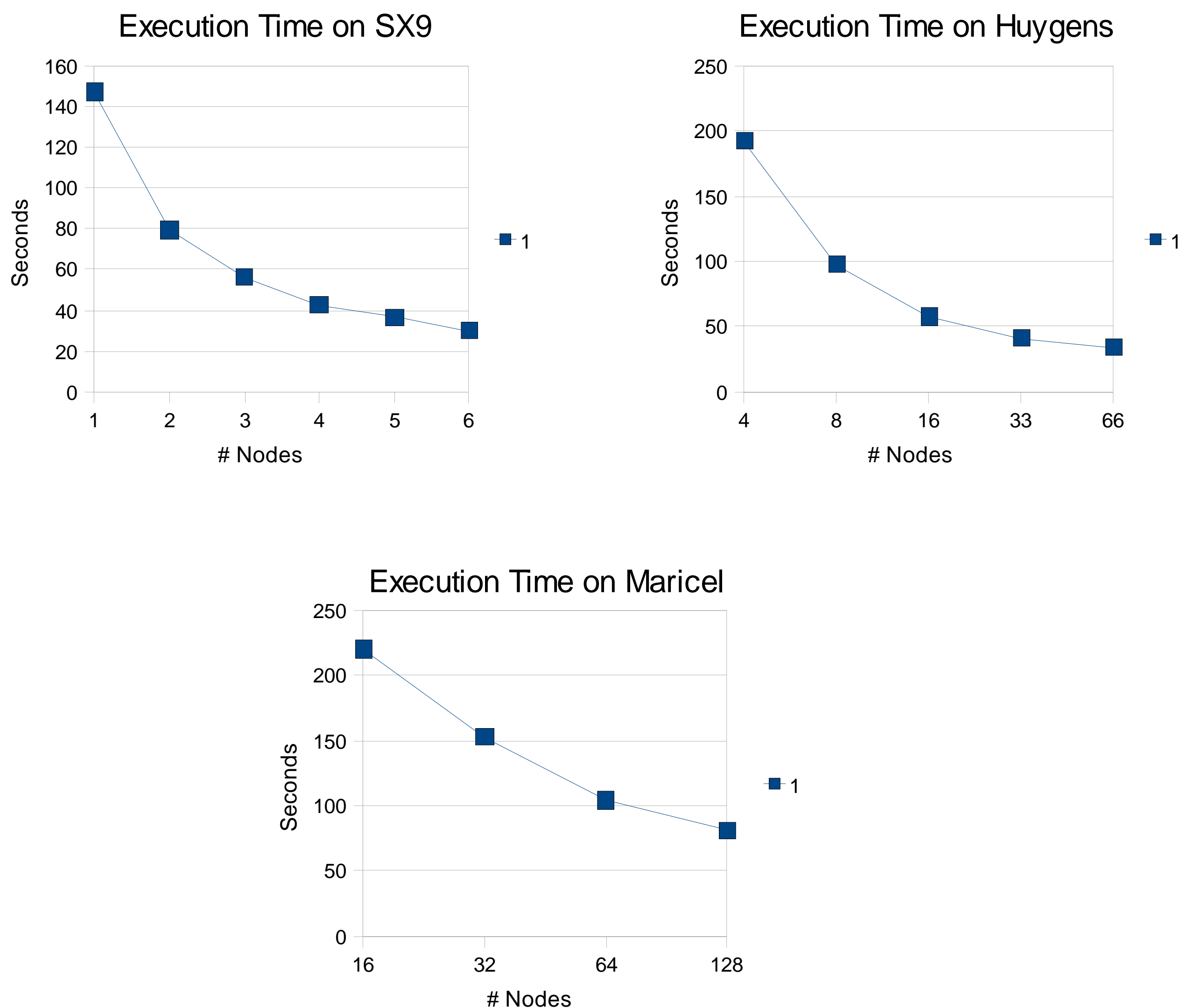

Figure 22 Scalability of CPMD

Unfortunately it has not been possible to run CPMD on the BlueGene/P. The reason for this is that the dataset with 32 water molecules is too small and the 64 Ionic Liquids dataset, suitable for petascaling, requires more memory than available per node.

3.6.4 *Conclusions*

The code was already very well optimized and so it was difficult to improve it further. Original code scales correctly, so it's not necessary to apply any modification. To get the optimum performance one needs to tune the input parameters correctly. In order to be able to run on BlueGene/P, more memory per node needs to be available.

3.7 ECHAM5

Written by: Sami Saarinen, CSC

ECHAM5 is an atmospheric circulation model developed at the Max Planck Institute for Meteorology in Hamburg. Depending on the configuration the model resolves the atmosphere up to 10 hPa for tropospheric studies, or up to 0.01 hPa for middle atmosphere studies (often referred to as MAECHAM5).

3.7.1 *Application description*

The program has been written in Fortran95 with only few utility routines coded in C. The NetCDF 3.6.2 library, AMD Core Math Library (ACML) and MPI-library were linked to the program. The program has been parallelized against the MPI-library, but also contained some OpenMP directives. However, in this exercise OpenMP has been ignored. Runs were performed on CSC's Cray XT5 equipment in Finland with 2.3GHz AMD Barcelona quadcores, 8 cores per node.

The program was written in a modular fashion with simplified interface to the MPI. However, quite a lot of replication to handle different data types and multidimensional arrays is present. These make changes to the MPI-wrapper quite a delicate task.

3.7.2 *Petascaling techniques*

Petascaling efforts of this code have essentially been destroyed because of the use of NetCDF I/O and piping the output to a single processing element (PE) in order to keep postprocessing (outside ECHAM) unmodified. Also, since the program is meant mainly for climatological runs, its typical usage usually comes with relatively low resolution and thus preventing good scalabilities (even if the impact of I/O-degradation is ignored).

The algorithms of the original code have not been changed. Instead computational processes were let to do computational work only and the separate I/O processes (one or more) were assigned to perform output I/O. The first PE still did reading and distribution of input data during the initialization phase.

The principal task to speedup the program was to reorganize its single PE I/O. In the original code data was collected from other PEs to the master PE. Since all the PEs were also computational ones, this arrangement introduced an inevitable bottleneck, while the first PE performed field gather and their output. Delays of several seconds per I/O-timestep were experienced compared to non-I/O-steps being often a fraction of seconds.

In the optimized version additional PEs were allocated to handle the data gather from computational PEs and to perform I/O (still through NetCDF, though) independently of computational tasks. This arrangement let I/O to progress in the background whilst computation could progress often several timesteps ahead of I/O.

The available parallelism (in theory) in the I/O is comparable to the available NetCDF files written. Output is organized in terms of streams, which could be 2D or 3D grid fields or spectral components. However, due to limitations imposed by postprocessing needs, many of these streams were in fact written to the same physical file. We measured that there was only up to 10 separate files in our runs, with one file taking majority of the streams. Before these kind of inherent limitations are removed, there is very little one can do to improve the performance. And all this on top of built-in bottlenecks found in the NetCDF-file I/O itself.

Despite having computational and I/O tasks separated, the program was run using the SPMD paradigm. But in an interesting manner, described below.

The first NPES-tasks were assigned to do the computation only and additional NUM_IOPES tasks on top of the computational ones tackled the I/O only. Whilst the computational PEs served the regular computational loop, the I/O tasks were left in an event-loop by the main-program. That is to say, the I/O tasks waited for doing I/O-work.

Events were triggered by computational tasks, usually by the master (first) computational PE. In practice this arrangement meant that the dedicated I/O-tasks executed only selected subset of ECHAM-code, and that the computational tasks did not call NetCDF I/O at all. The event delivery was accomplished through prearranged messages using separately allocated communicators across computational and I/O PE boundaries.

As mentioned earlier, the resolution and thus datasets were not big enough to achieve good scaling. Model size T106 (ca. 120km grid spacing) was used which was already considered big for climatological runs. Another test with T42 (ca. 300km) was also tried, but did not lead to any gains in performance.

3.7.3 *Results*

A powerpoint presentation about results is found under:
http://www.esnips.com/web/echam5/ECHAM.ppt

The source code has been shipped back to MPI/Hamburg in June 2009. Source code is also found under SVN: https://trac.csc.fi/pracewp6-echam/browser

3.7.4 *Conclusions*

Whilst ECHAM was not a very successful example for petascaling, its I/O optimization demonstrated an important design pattern that could be employed elsewhere: a separation of computational and I/O tasks from each other whilst still running the same program. This approach also shows perhaps the minimum effort that eventually needs to be put in order to reap the benefits of I/O performance.

In ECHAM's case in order to achieve petascaling, much more effort than available within PRACE-project must be put in place. Potentially nearly a complete re-design is maybe necessary plus introducing sufficiently large datasets (grid spacing of approximately 10km).

3.8 EUTERPE

Written by: Xavier Saez, BSC

EUTERPE is a gyro kinetic particle-in-cell (PIC) 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.

3.8.1 *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 the code 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 must be computed. 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 linear solver (poisson_solver) and the noise reduction with fast fourier transformation (ppcfft2).

EUTERPE has two pre-processing steps. Their computational loads are insignificant compared to the main part because they are only executed once.

The first step is to generate the electrostatic fixed equilibrium (for example, a cylindrical equilibrium) as initial condition. This step is done by an external application that maps coordinates generated by the equilibrium into the toroidal system required from EUTERPE, while the quantities generated by the application (temperature, pressure and fluid parameters of the fusion reactor) are calculated and stored in a rectangular grid.

The other step is related with the solving of the quasi-neutrality equation. The finite element matrix contained in the equation is time-independent, so it has to be calculated only once at the beginning.

EUTERPE is parallelized using MPI. The domain is decomposed in one-dimension and the resulting subdomains and electrostatic field grids are assigned to different processes together with the Monte Carlo particles that reside on them.

As particles move from one region to another, the particles that move out of region are transferred to the process that is associated with the new region. Thus, the number of particles resident on each process is not a constant of time and it could cause load imbalance. During

the calculation of the gyro-averaged electric field at the particle position, field information of neighboring regions of space must be accessed, so that extra guard cells are kept in each process. Special routines take care that the particles are passed to the correct process when leaving a domain and that the guard cells are updated after a calculation of the electrostatic potential.

Moreover, EUTERPE adds another feature to parallelize the code: the clones. The clones are copies of the same domain to distribute the particles between them. So each clone is assigned to a group of processors with its own set of particles. The processors of each group are on the same node, so that the data intensive particle communication is restricted to the shared memory into nodes, and between the nodes only the moderate field communication takes place via the network. The one-dimensional decomposition is performed in such a way that each process has a corresponding subdomain clone.

EUTERPE mixes serial and parallel access to data files. The initial electrostatic equilibrium is divided in files which are read by the corresponding processes. The finite element matrix contained in the equation to solve is also divided in a file per process. Finally, the files with the results are sequential and the master process is the one that writes into them.

Regarding datasets, there is one available for the application. It uses a cylindrical equilibrium (meaning that the geometrical domain is a cylinder) where the electromagnetic field is parameterized and fixed. The domain is divided in a grid of 32x512x512 pieces, so the problem can be distributed into 512 processes at the most. The plasma simulation consists of 1000 million of particles into the cylinder.

This dataset is not big enough for petascaling, but it is easy to prepare one. One only has to increase the number of grid subdivisions and particles to simulate. The final objective is to simulate a real fusion reactor, and this problem can only be done on a petascale machine.

3.8.2 *Petascaling techniques*

The main strategy to face the petascaling challenge has been to introduce OpenMP into the EUTERPE code. OpenMP has allowed us to assign all the memory of a node to a single MPI process, so we will be able to execute big real datasets in the future. Additionally, as the communication between threads is via shared memory, there is a reduction of the communication costs between processes.

After analyzing the distribution of the time between routines, three routines were identified, which consumed more than 50% of the main execution time. These are: push (moves the particles in the domain), grid (determines the charge density) and petsc_solve (solves the field equation).

These routines were candidates for introducing OpenMP. In the "push" and "grid" routines, the particle loops were parallelized. In the "grid" routine, as different particles could write in the same position of the "rho" array, it was necessary to allocate private rho arrays (one per thread) and to make a reduction at the end.

Regarding the petsc_solve routine, there was a problem with the PETSc library. This library is a well known package for solving Partial differential equations on parallel machines, but it is not thread-safe. So, we had to develop a hybrid version of the solver. This new solver is a preconditioned conjugate gradient with Jacobi, and we parallelized all the loops. The only dependences between iterations came from the dot products and they were solved with reductions.

The chosen schedule was static, because there is the same workload per iteration. In the case of the multiplication subroutine, we know that the distribution of the sparse matrix in EUTERPE is like a kind of band matrix, where each row has identical number of non nulls, and as a result, the same workload.

The matrix contained in the field equation is distributed in a rectangular grid between the processes. If the size of the global field matrix is (Nx * Ny), each process holds a local matrix of size (Nx/nx * Ny), where nx is the number of processes. Each process is assigned to a node.

Regarding the communication patterns, for each iteration of the solver, borders are exchanged with 2 neighboring processes (the previous and next process by id) with point-to-point communications. Furthermore, there are broadcast communications, for example, into dot products of the solver.

The two pre-processing tasks have not been parallelized with OpenMP, because they are only executed once. And because of this, their computational loads are insignificant compared to the load of the main part that it is placed inside a steps loop.

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

3.8.3 *Results*

The performance achieved is more or less the same as the original code with MPI. The reason is that the scalability of the MPI version was already excellent, and there are parts of the code that are not parallelized with OpenMP, so only one thread per process is working. Also, there is a small overhead related to spawning, synchronization and destruction of threads.

The reason for developing a hybrid version is to allow us to run the code in a petascale machine. Before EUTERPE could only use processes and as result, each process could only use a part of the memory of the node. Now, with the hybrid version, the application can create threads and a process can use all the memory of a node.

The key bottleneck for improving the scalability of the hybrid version is to introduce threads into the parts of the code that have not been parallelized yet. But perhaps there are no more suitable loops to parallelize with OpenMP, so further work is needed to investigate this.

3.8.4 *Conclusions*

EUTERPE was parallelized using OpenMP and MPI. MPI is used globally to exchange border-values and particles, while OpenMP is used to parallelize the computation loops.

Although, OpenMP is the easiest way to parallelize since we simply add "#pragma" directives to the for-loops, in reality it is difficult to parallelize all the code because there are zones without such loops. Furthermore, it is difficult to reorganize the communication between the processors since the communication is implicit, not like MPI where the communication is explicit.

The techniques used for petascaling have produced the expected result: to create a functionally equivalent hybrid version of EUTERPE.

In the future, an optimization technique which may be helpful, could be the implementation of Deflated Conjugate Gradient. We think this new solver will allow EUTERPE to run much faster because the number of iterations required to find the solution will be less.

Regarding architecture improvements, we are dealing with a code that is memory bound, since the cost of the memory operations is linear with respect to the cost of the computation.

For example, the equation of the movement of particles has the same order of floating point operations as memory accesses. Likewise, in the multiplication of a sparse matrix by a vector, the data is not reused and there are not enough floating point operations to take advantage of the hardware. Therefore, increasing the memory bandwidth would improve the performance of the application.

3.9 GADGET

Written by: Orlando Rivera (LRZ)
Collaborator: Jose Gracia (HLRS)

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.

3.9.1 *Application description*

In the benchmarking set a non-open version (3.0) of GADGET was used. This new version includes mainly improvements in the memory management and better statistics about load balancing.

The source is written in Ansi C (C99) and contains more than 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 future versions should also work. Another library necessary for compilation is the Fast Fourier Transform 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 offer better performance, have a different interface and the MPI-supported implementation is still at the beta stage.

The code is fragmented into 144 source files inside of a single directory. The code is highly procedural and one can find up to 5 levels in the function call hierarchy. The names used in the source files are descriptive and correspond to the function names or sections that form the code. In general, the code is well commented and documented.

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 have been generated. For this benchmark a cosmological, commoving box with periodic boundary conditions was used.

The code uses only MPI for its parallelization. The domain is particle-based and the domain decomposition is done using Hilbert-Peano Space Filling Curves, for load efficiency. The initial conditions are stored in binary format and can be split into several files. These files are distributed for input and output in an evenly way and each MPI-Task is responsible for reading/writing a file. In case there are more MPI-Tasks than files, some MPI-Tasks will not perform any I/O operations.

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 as:

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

This is roughly 57 Bytes of information for each particle. For this benchmark, GlassTileFacs of 25 to 50 were used. Larger values are possible for petascaling, but care in memory usage, storage space and wall time need to be taken.

3.9.2 *Petascaling techniques*

Load balancing has to be addressed properly with larger data sets and increasing number of cores/cpus. Since particles are moving, the domain decomposition is calculated at every time step with an efficient Hilbert-Peano space filling curve algorithm that splits the data among MPI-Tasks and uses less than 1% of the wall time.

An imbalance in the number of particles for each processor (sub-domain) is required to achieve an optimal load balancing. However, this can also lead to an uneven memory usage. The user can control this memory imbalance with a parameter (PartAllocFactor) and can also specify the maximum memory required for a MPI-Task. Shared memory or fat node architectures are best suited for this kind of data input.

A previous version of GADGET had a memory bottleneck, which prevented using more than 8000 MPI-Tasks. This problem has been partially solved with the new version of GADGET (GADGET version 3) which gives better information about memory requirements, allowing a proper tuning.

3.9.3 *Results*

Since one important bottleneck in scalability is the memory required, we have some estimations about the memory pressure. We can calculate the memory required using the following expressions:

$$\text{mem(MB)} \sim \text{PartAllocFactor} \times (N \times (68 + 0.65 \times 64) + \text{Nsph} \times 84) + 12 \times \text{PMGRID}^3/\text{MPITasks}$$

Where:

$$N=(\text{Number Total of particles})/(\text{Average Number of Particles/MPI Task})$$

Nsph is the Number of NSPH particles and PMGRID is a factor for long forces calculations and it was set to 128 so far. We are investigating the impact of the PMGRID factor on the memory pressure and how this factor influences it, but obtaining physically correct results at the same time.

The communication buffer has to be also added to these numbers. For example a system using 64 millions of particles and 256 MPI-Tasks requires up to 1.6 GB of memory. A petascale-range problem will possibly contain at least 8 times more particles (512 millions) or more. Using more cores imposes more memory pressure, since every MPI-Task needs concurrent information from other MPI-Tasks. This extra pressure also follows a power-of-2 law. The memory usage can be segmented into 3 groups: calculation of algorithms, neighbours information and communication buffer.

The practical limit for GADGET, either version 2 or 3, is not given by the information about the domain (sub-domain) that need to be computed, but by the information about other sub-domains, which needs to be stored for each MPI-Task. This information is the number of particles that need to be sent/received to/from other sub-domains; this is translating into the allocation of several matrices and vectors, in particular one 2D matrix (Sendcount_matrix), which has the MPI-Tasks on each axis. Therefore the size of this matrix is MPI-Tasks x MPI-Tasks, O(MPI-Tasks^2), and is filled with integer values. In systems with 4-bytes integers and 8192 MPI-Tasks the amount of memory required only for this matrix is 256MB. After that, you have to add the memory required for the algorithm. For 16384 and 32768 MPI-Tasks it is 1GB and 4GB respectively. As we can see with 32768 MPI-Task we approach the limit in terms of memory of current systems. By doubling the number of MPI-Tasks we require 4 times more memory.

Fortunately, it is possible to eliminate this 2D matrix from the code in a relatively simple way. This is because a given MPI-Task does not really need the full MPI-Tasks x MPI-Tasks matrix. It only needs the row and the column that corresponds to its task number (rank).

Instead of filling this matrix with a call to MPI_Allgather, like this:

```
MPI_Allgather(Send_count, MPI-Tasks, ...,Sendcount_matrix, MPI-Tasks,...)
```

And then extracting "Recv_count" from it with:

```
for(j = 0; j < MPI-Tasks; j++)
Recv_count[j] = Sendcount_matrix[j * MPI-Tasks + rank]
```

One can simply get "Recv_count" (which has length MPI-Tasks) directly, with:

```
MPI_Alltoall(Send_count, 1, ..., Recv_count, 1,...)
```

Once this is done the Sendcount_matrix can be eliminated from the code.

Power6 (SARA)

GADGET3's scalability is almost linear as long as enough memory is available. As shown in the table, the speed-up reached on the Power6 Huygens system at SARA using 2048 MPI-Tasks has an efficiency of 83%. Lower counts of cores give better efficiency rates (95-90 %).

In the Power6 system we used SMT in which each node was set up to deploy 64 MPI-Tasks. The data set contains 512 millions of particles. The code was compiled with IBM's C/C++ compiler version 10.1 and the following options:

```
-q64 -O5 -qipa=exits=endrun,MPI_Finalize -qipa=inline=auto
```

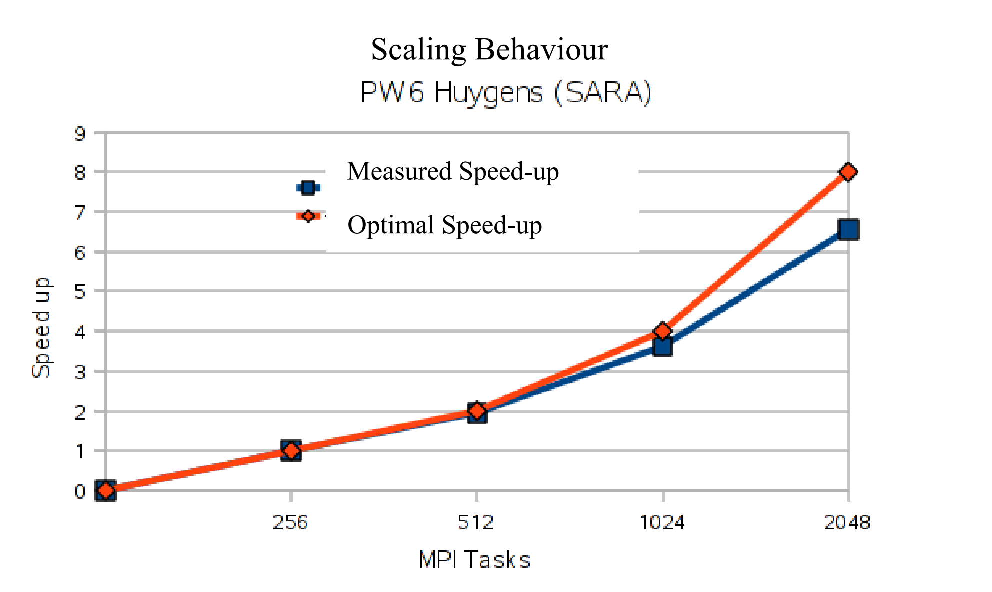

Figure 23 Gadget Scaling Behaviour on Huygens

BG/P (Jugene)

In the BlueGene/P platform, by using a smaller data set as the one used on Power6 for memory considerations (266 millions of particles) it is possible to scale GADGET up to 8192 MPI-Tasks. It is interesting to note that if a run with 2048 MPI-Task is taken as a basis super linear scalability has been achieved. The reason could be a better memory usage.

Using the run with 4096 cores as base line we have an 87% speed-up efficiency compared to 8192 MPI-Tasks. This result is in accordance with the results obtained on other platforms.

It is important to mention that jobs running on this BG/P system were running with SMP mode on. Each MPI-Task was running on a compute card. The data set as well as the Number of cores made it impossible to launch 4 MPI-Tasks per compute card (1 MPI-Task per core).

On Jugene the IBM's C/C++ compiler version 9.0 with the following optimization options has been used: *-qtune=450 -qarch=450 -O5* was used

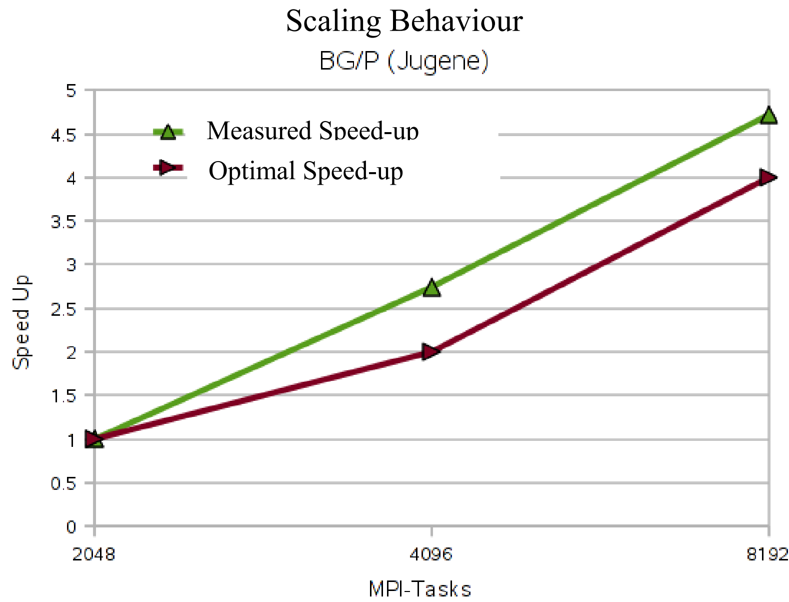

Figure 24 Gadget Scaling Behaviour on Jugene

Nehalem Cluster (HLRS-Baku)

The cluster consist of 700 nodes, each with two quad-core Intel Xeon (X5560) Nehalem at 2.8 GHz, with 8MB Cache and the memory per node is 12GB. 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 build 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

Two sets were use on Nehalem. One with 512 x106 particles (GlassTileFac=50). In this case some problems were also encountered regarding memory usage for this data set because 1.5GB per MPI-Tasks was available.

MPI-Tasks	Timing (sec.)	Speed up *
64	87.34	1
128	43.94	1.99

MPI-Tasks	Timing (sec.)	Speed up *
56	100.36	1
112	56.11	1.79

Table 4 Experimental Speed on BAKU(HLRS) 512 x106 particles (Calculated with respect to the wall time of half the core count)

The other data set contains 64 x10^6 particles (GlassTileFac=25). The speed up and wall times are presented here:

MPI-Tasks	Timing (sec.)	Speed up *
8	75,044	1,00
16	39,41	1,90
32	21,4	1,84
64	13,738	1,56
128	9,006	1,53

MPI-Tasks	Timing (sec.)	Speed up *
27	25,508	1,00
28	25,552	1,00
55	16,072	1,59
56	16,048	1,59
111	13,926	1,83
112	14,08	1,81

Table 5 Experimental Speed on BAKU(HLRS) 64 x106 particles (Calculated with respect to the wall time of half the core count)

It is easy to see that using cores counts of power of two is more efficient than arbitrary core counts. This is because the internal algorithm of GADGET uses Fast Fourier Transformations.

3.9.4 *Conclusions*

We can not expect an optimal petascaling out of the box. There are many factors to be taken into account. A mid range generic problem should be used as an initial point. The user should choose, depending on the system, either a better scalability at the expense of load balancing or better performance using more memory and smaller numbers of MPI-Tasks.

As the problem gets bigger the memory will be a limiting factor. An MPI-Task will require as much memory as possible. In terms of a single die, that means one core making all calculations while the others simply stall. In that sense the multi/many core technology can be advantageous by using OpenMP or threads. The same is applicable to accelerators specially working on shared memory.

Another improvement could be the use of MPI topologies to adapt the domain decomposition to the physical interconnect.

The memory problem is endemic in many implementations. Petascaling programs, for SIMD problems require at MPI level, that each task can perform calculations with minimal information about other tasks. In GADGET's case we have seen that this information scales with the square of number of MPI-Tasks. This bottleneck is being addressed with a sophisticated algorithm, as described. Without this modification GADGET3's limit, in best case, is close to the range of 32000 MPI-Tasks. Future hybrid extension of GADGET, OpenMP or pthreads, will increase this limit by a factor which is equal to number of threads that each MPI-Task can hold.

3.10 GPAW

Written by: 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. The program uses the projector-augmented wave (PAW) method for presenting the wave functions in terms of smooth pseudo wave functions. GPAW is GPL-licensed open-source software and it is developed in several universities and research institutes.

3.10.1 *Application description*

GPAW is written with a combination of Python and C programming languages. Currently, there are ~50 000 lines of Python and ~13 000 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 are on the other hand harder to read.

The program contains extensive test suites which help in confirming the correct behavior of the program during software development.

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

Construct Hamiltonian O(N)

  1. Solve Poisson equation
- Subspace diagonalization $O(N^3)$
  1. Calculate Hamiltonian matrix from wave functions
  2. Diagonalize Hamiltonian matrix
  3. Rotate wave functions
- Iterative refinement of wave functions $O(N^2)$
- Orthonormalization $O(N^3)$
  1. Calculate overlap matrix
  2. Cholesky decomposition
  3. 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 algorithms are:

- Construct Hamiltonian O(N)
- 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.

The parallelization is done using MPI. There are MPI-calls in the low level routines implemented in C, and there are also Python interfaces to certain MPI-functions which are needed in the higher level algorithms implemented in Python.

The basic parallelization strategy is domain decomposition. The real-space grid is divided to processors so that the size of the domain is the same across the processors. This puts some limitations on the number of cores that can be used with the program, as the real-space grid has to be divisible by the number of cores.

The number of computations for finite-difference operations using domain decomposition with P processors is O(N/P), while the communication cost is $O((N/P)^{2/3})$. Even though the computation to communication ratio is rather good, it appears that for efficient parallel scaling the minimum grid-dimension per processor is 10-20. In the current real-world applications the maximum grid dimensions are typically 160, and it is very likely that in the near future the grid sizes increase at most by factor of two. Thus, with current datasets the domain decomposition can scale to ~1000 cores and also with future datasets it is unlikely to be able to go beyond a few thousand cores with domain-decomposition. A further restriction of domain-decomposition is that memory requirements scale $O(N^2/P)$.

Originally, in ground-state calculations the matrix diagonalization and Cholesky decomposition were done serially. Even though these operations have a very small prefactor, they scale $O(N^3)$ and thus become a serious scalability bottleneck with larger datasets. The code version in the beginning of the PRACE project had a preliminary Scalapack-implementation which has now been refined.

In addition to domain decomposition, density-functional theory and real-time time-dependent density-functional theory offer additional parallelization possibilities.

In magnetic systems, the wave functions have a spin degree of freedom, and parallelization over spin is nearly trivial. There are only two spin values, so spin-parallelization enhances scalability by a factor of two at most. In periodic systems there is an additional k-point degree of freedom. Parallelization over k-points is also nearly trivial and it is very similar to the parallelization over spins. However, the number of required k-points decreases with increasing system size, so in large systems there are only few k-points which limit the scalability of k-point parallelization. The real-time TDDFT formalism is not compatible with periodic boundary conditions, so k-point parallelization cannot be used. The datasets used in current benchmarks do not employ spin or k-point parallelization.

The final natural degrees of freedom for parallelization are the electronic states. Real-time TDDFT parallelization over electronic states is nearly trivial and requires very little communication. Thus, theoretically real-time TDDFT is well suited for petascale calculations.

In ground-state calculations the parallelization over electronic states is more difficult, as subspace diagonalization and orthonormalization require all-to-all communication of the wave functions. This communication cost is a constant $O(N^2)$. The code version in the beginning of the PRACE-project had the electronic states parallelization implemented for real-time TDDFT and a very preliminary implementation for ground state DFT calculations.

There is relatively little compulsory IO in ground-state calculations, however for restart-purposes wave functions can be written to disk. Wave function IO is required for starting a TDDFT calculation. Only a single process writes and reads the data, which is gathered from or distributed to other processes.

The preprocessing step in ground state DFT is the generation of an initial guess for the wave functions, which is obtained from atomic-orbitals. The atomic-orbital calculation is parallelized but does not scale as well as the actual calculation. However, in real-applications the initialization time is typically insignificant compared to the full time.

The preprocessing step for real-time TDDFT is a separate ground state DFT calculation.

The current dataset for ground state calculation is a system of 256 water molecules, and few self-consistent iterations of the total energy calculation are performed. This dataset is not large enough for petascaling, but it is fairly easy to obtain or generate larger datasets.

For real-time TDDFT calculations there are two datasets, a smaller 60 atom C_{60} fullerene and larger 55 atom Au_{55} cluster. While these datasets are not suitable for petascaling, larger real-world datasets exist. Generally, the TDDFT calculations are significantly heavier than DFT calculations and exhibit also better scalability. Thus, at least 10-20 times more processors can be used in the TDDFT calculation than in the DFT calculation for the same dataset.

3.10.2 *Petascaling techniques*

Most of the effort in petascaling has focused on improving the parallelization over electronic states in the ground state calculations. The work has been done in close collaboration with other GPAW developers.

The basic communication patterns in the software are point-to-point send and receives. In the domain decomposition, point-to-point communication is needed only between nearest neighbours, in addition there are reduce operations over all domains which are needed for example when evaluating dot products. Point-to-point communication is mostly non-blocking, however it should be possible to increase the overlap of computation and communication by optimizing the program code more.

In the ground state calculations, parallellization over electronic states is based on a pipeline where processes have a one-dimensional arrangement. Each process receives data only from the previous process in the row. After the process has performed the necessary computations, the data is passed on to the next process in the row. The communication is performed with non-blocking point-to-point operations. Reduce operations are needed when summing for the charge density.

It was found that in MPP-Cray, rank placement is important when using domain decomposition and state parallelization together in DFT calculations. When using over 512 processors, there were large random variations in the execution time of the program. For certain parts of the algorithm the times varied almost by a factor of two. In MPP-Cray it is possible to specify only whether the ranks are placed on the same node. By using a custom rank placement, where processes exchanging lots of data i.e. those communicating in the electronic state parallelization are in the same node if possible (there are eight cores in the

node in Cray XT5), the execution time remained practically constant between different runs, the time being also sligthly shorter than the best case with the default rank placement.

There are no direct results about the importance of rank placement in FN-Power6, but the experiences from the GPAW developers with Blue Gene platforms indicate that the rank placement is important also in MPP-BG.

The matrix diagonalization step of the subspace diagonalization is performed with Scalapack using a subset (typically 4x4 or 8x8) of all the processors.

In the time-dependet calculations, the operations for different electronic states are nearly independent and only reduce operations are needed when constructing the charge density.

3.10.3 *Results*

The work done during the PRACE project in collaboration with GPAW developers has increased substantially the parallel scalability of the DFT calculations. Previously, the dataset of 256 water molecules scaled to 128 cores on MPP-Cray and now the same dataset scales to 2048 cores with 75 % parallel efficiency when doubling the number of CPU-cores. Other platforms and bigger datasets should now be investigated in order to determine the limits of scalability.

The main figures limiting the parallel scalability are the number of grid points per domain and the number of electronic states per processor. Minimum domain size is determined largely by the computation to communication ratio of finite-difference operations, and the minimum grid-dimension is currently 10-20 points. Other limiting factor for the domain decomposition are the PAW-method related operations within atomic spheres which introduce load imbalance. Optimizing the communication patterns of finite-difference operations could decrease the minimum grid size, but large benefits are not expected.

The minimum number of electronic states per core seems to be 100-200. Limiting factors for scalability are the computation to communication ratio of matrix construction and wave function rotation. Test calculations indicate that on MPP-BG the overlapping of computation and communication does not work very well, the execution times with blocking and non-blocking calls in the state-parallelization produce nearly identical execution times. In MPP-Cray the benefits of overlapping computation and communication are larger.

The matrix diagonalization with Scalapack is done using only a subset of processors and does not scale very well, thus it can be considered a „serial“ part in the calculation, limiting the scalability. Further limitation in the current Scalapack implementation is that the input matrix is not distributed, and especially on systems with limited memory per core e.g. MPP-BG, memory requirements limit the maximum system size to a few thousand electronic states.

The parallel scaling of TDDFT calculation has been good in all tested platforms, the Au_{55} example scales to 2048 cores both on MPP-Cray and FN-Power6. For petascale calculations it is possible that reading the wave functions from disk can limit scalability, thus parallel IO should be considered. In principle, TDDFT requires a standard DFT calculation as a preprocessing step which can be thought of as scalability bottleneck. However, as the DFT calculation is done in a separate step, and with the same system size, the scalability of the TDDFT calculation is considerably better. Therefore petascaling of the TDDFT calculation should be possible.

3.10.4 *Conclusions*

Close collaboration with the GPAW development team has been important in the petascaling effort. Efficient overlapping of communication and computation is important especially for the parallelization over electronic states in ground state calculations, however, our experiences are that in MPP-BG the overlapping does not work very well.

For the ground state calculation, the main remaining issue for larger scale calculations is the distribution of Hamiltonian and overlap matrices in the subspace diagonalization steps. Currently, the memory needed for storing the matrices limits the maximum size of the possible datasets.

In time-dependent calculation, there are no clear bottlenecks for petascaling.

In both calculation modes, further optimization of the program code could improve the scalability but very large enhancements are not expected.

As the program retains mostly to point-to-point communication, and algorithms offer natural means for overlapping the computation and communication, it is important to have efficient implementation for these routines both in hardware and in MPI-implementation level.

As only MPI is used currently, a threaded implementation e.g with OpenMP could be considered in the future.

3.11 GROMACS

Written by: 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.

3.11.1 *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 classical MD simulations 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 update 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 Coulomb (long ranged) forces, van der Waals 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 a domain decomposition scheme. The domain decomposition is able to dynamically load balance the workload.

Long ranged forces can be computed in several ways in Gromacs; the two relevant ways are here reaction field (RF) and particle mesh Ewald (PME). The algorithmic complexity of RF is O(N) while it is Nlog(N) for PME, N is here the number of particles. When using RF long ranged forces are essentially approximated using short ranged forces and the whole system is solved using domain decomposition. This algorithm shows excellent scalability up till the largest partitions we tested it on (40Tflop/s). PME is more accurate than RF. In PME the long ranged forces are computed in fourier space and a 3D-FFT has to be performed that involves a MPI_Alltoall routine. This is the main scalability bottleneck for Gromacs when PME is used. PME thus only scales up to 5 TFlops. In Gromacs 4.0.x the FFT is only parallelized in 1D, in version 4.1 it is planned to be parallelized in 2D. There are several problems with having it parallelized in only 1D. First, the grid dimensions of the grid in Fourier space are often quite small which severely limits the number of processes that can be assigned to the PME task. This is a the main problem on Blue Gene/P as one would need to scale to tens of thousands of processors and the grid dimensions is in the order of hundreds of slices. Second, one would not need a global all-to-all routine in the 2D case.

The parallelization strategy when using PME is designed to extract maximum performance from multicore supercomputers. The program is essentially divided in half with some processes dedicated to computing PME forces, while the other processes are dedicated to calculating everything else. The program should be placed so that each node has both kinds of processes. The main benefit of this scheme is reduced network congestion as only one, or a few, processes per node execute the all-to-all operation.

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 Gromacs developers.

3.11.2 *Petascaling techniques*

In improving the scalability we concentrated on PME, as that is the main problem. We improved it in two ways and tested the solutions on the Cray XT5 prototype.

1. Improved Alltoall routine
2. Hybrid (MPI+OpenMP) version

1. Improved collective routines

We improved the all-to-all on Louhi by creating a new library that wraps the native MPI library routine. This change is transparent to the program and only requires a recompilation. Essentially we first aggregate all data on each node using low-level shared memory routines (memory mapped files), even as the whole program is still a pure MPI program. In addition to MPI_Alltoall, similar optimized versions of MPI_Alltoallv, MPI_Allgather and

MPI_Allgatherv were also created. The benefits are dependent on how efficiently the MPI implementation is able to utilize the SMP nature of the nodes. Using this routine with Gromacs increases its performance as the all-to-all routine is its main bottleneck in PME (Table 6).

2. Hybrid (MPI+OpenMP) version

Here we created a prototype quality fine-grained hybrid version of Gromacs where only the PME processes spawn threads. The results show that some speedup is obtained (Table 7). Speedup is achieved through improved performance of collectives, message aggregation and improvement in limited parallelism. The collectives, in this case the all-to-all routine, is much faster as it utilizes the same benefits as the shared memory library described above without having the data copy overhead. In addition to this we also get reduced number of messages as there are fewer slices in the FFT parallelization. Also very significant is the fact that one can now utilize more cores for PME computation as each PME task is parallelized with threads. In essence we use another level of parallelization in the OpenMP part.

If also the short-range part was parallelized with OpenMP one could run one PME process and one short-range process per node, and achieve improved load balance by balancing the number of threads of the two MPI processes.

3.11.3 *Results*

1. Improved collective routines

For the Cray XT5 the benefits of the shared memory all-to-all routine are significant, as can be seen from the measurements depicted below.

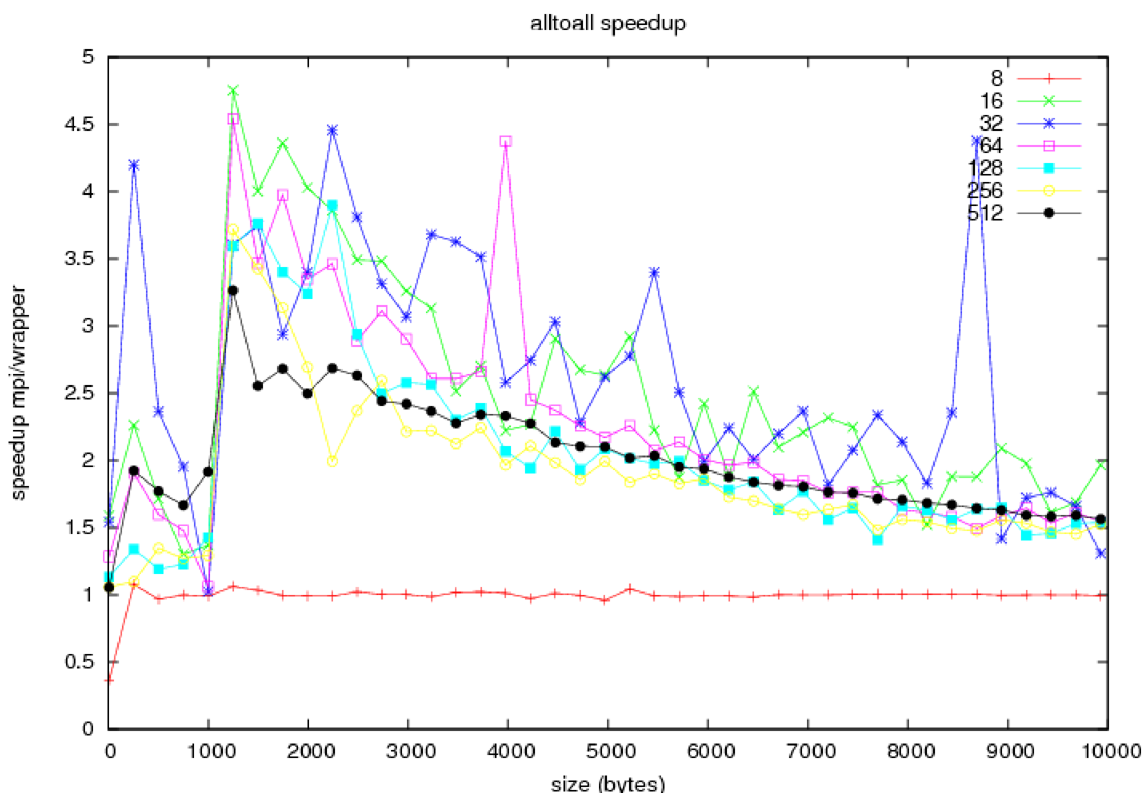

Figure 25 Speedup of shared memory all-to-all routine compared to the original MPI routine.

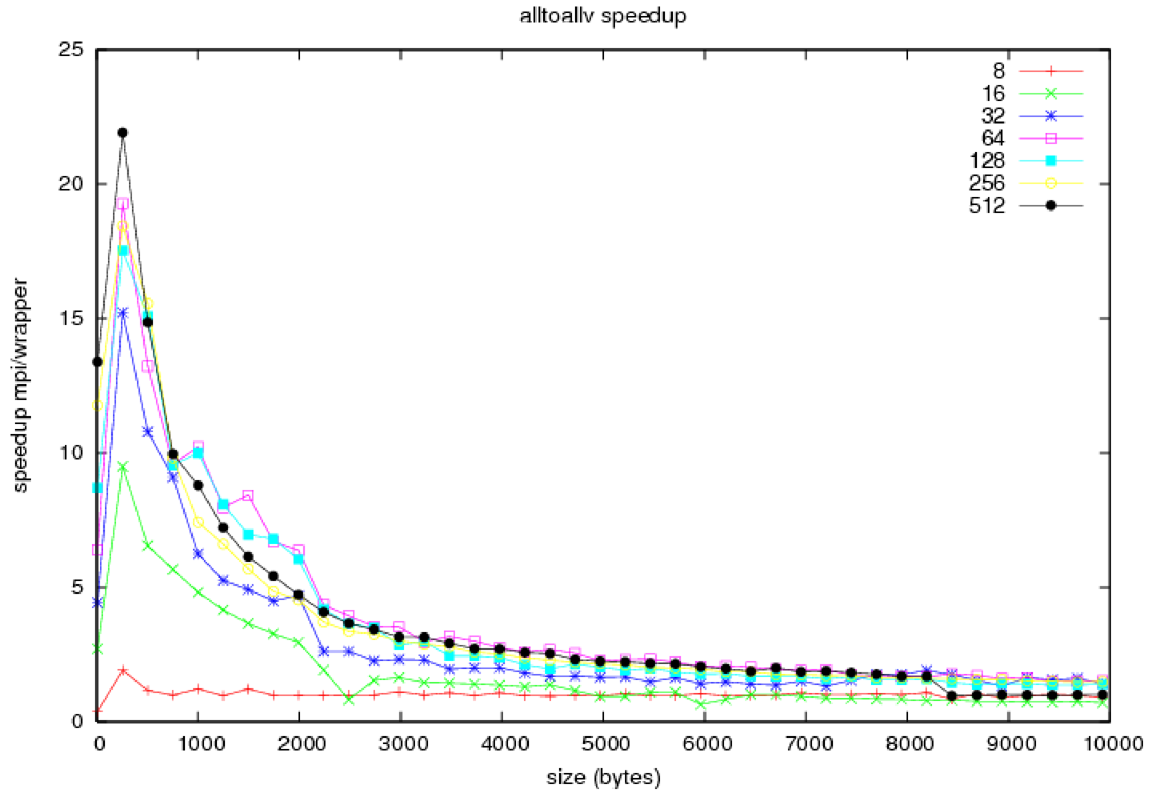

Figure 26 Speedup of shared memory all-to-all vector routine compared to the original MPI routine.

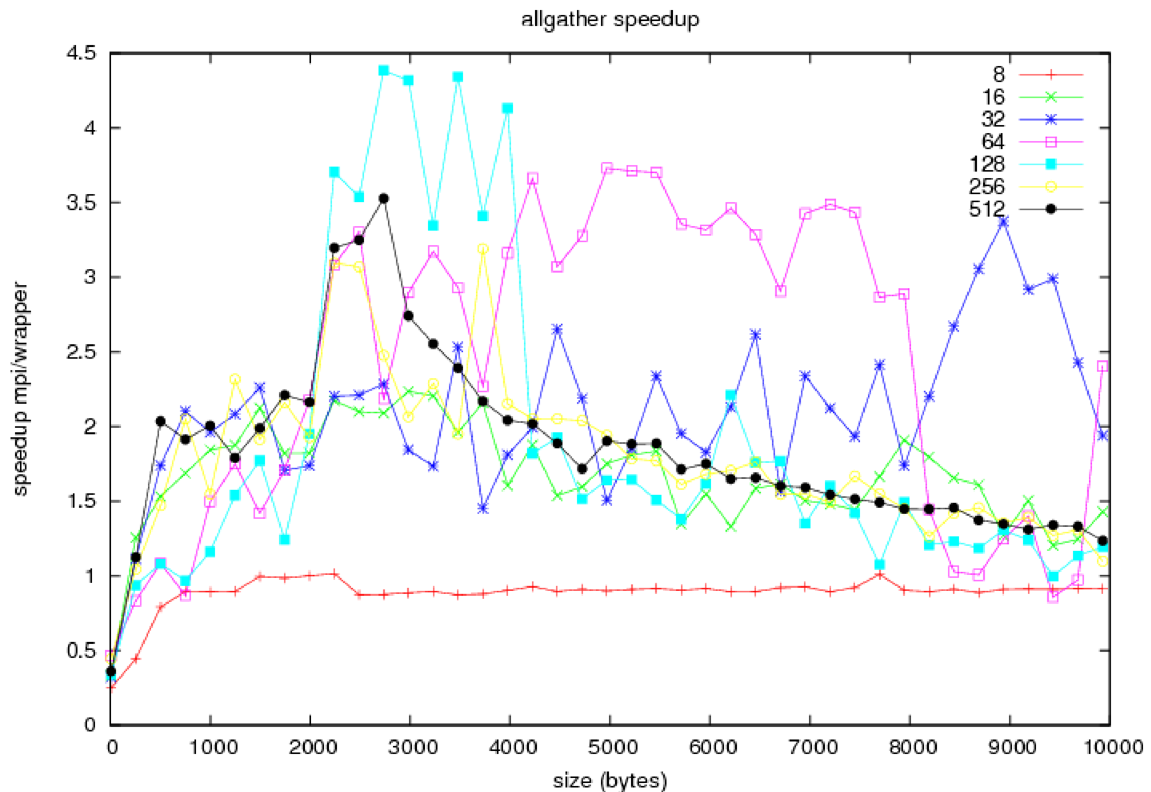

Figure 27 Speedup of shared memory all-gatherl routine compared to the original MPI routine.

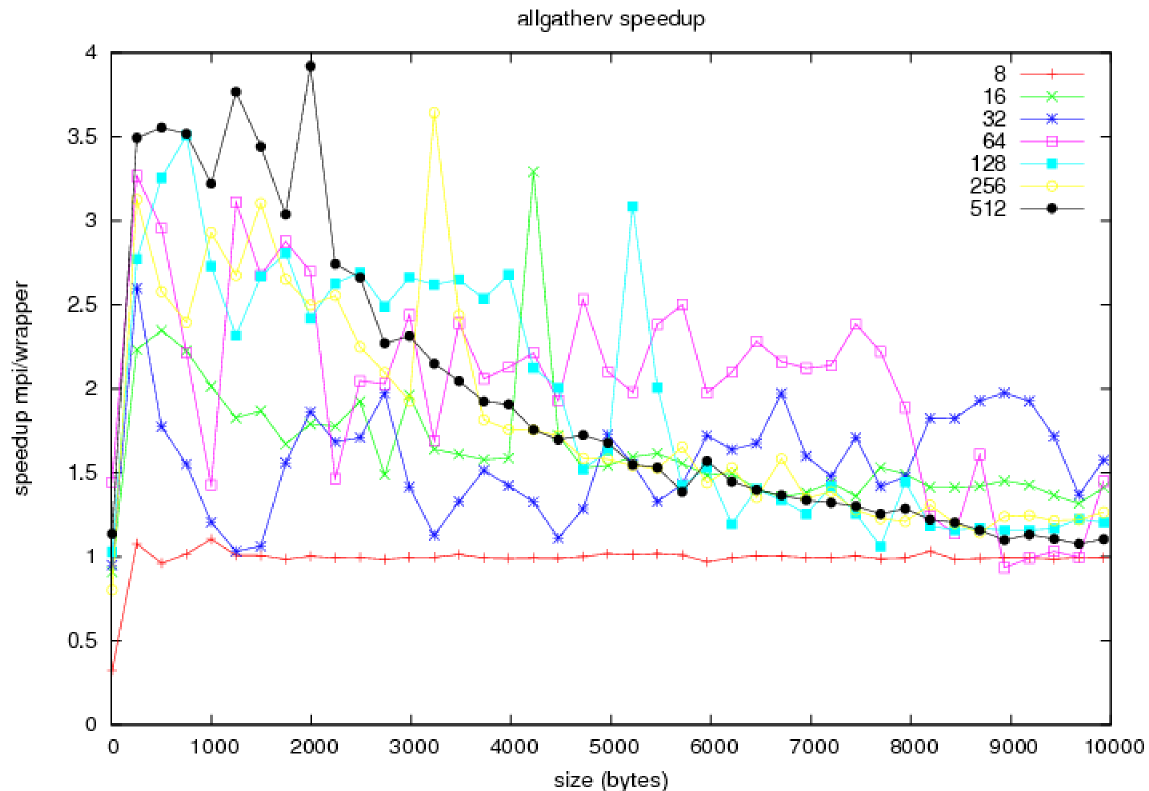

Figure 28 Speedup of shared memory all-gather vector routine compared to the original MPI routine.

When using the routine with Gromacs we see the following speedup.

NP	NP-PME	Original wall-time(s)	Optimized wall-time(s)	Speedup
352	88	402	384	1.05
704	176	316	270	1.17
1408	176	249	251	0.99

Table 6 Speedup of Gromacs using the shared-memory MPI_Alltoal on the Cray XT5 prototype. NP is the total number of cores (about 10 GFlops per core). NP-PME is the total number of cores dedicated to PME computation

The performance at 352 & 704 cores is better, and especially the scaling to 704 cores is improved. The case with 1408 cores is essentially the same. This is not surprising as we in that case have only one PME process per node. In that case no aggregation can be performed.

2. Hybrid (MPI+OpenMP) version

The speedup for the hybrid version is presented in Table 6. The results are promising taking into account the simplicity of the implementation. Speedup is limited by the fact that the implementation is of prototype quality and not all aspects of the PME calculation were parallelized with OpenMP.

NP	Speedup with four threads
352	1.10
704	1.34
1408	1.26

Table 7 Speedup of shared-memory MPI_Alltoall. NP is the total number of cores

3.11.4 *Conclusions*

The main bottleneck for scaling Gromacs is the PME algorithm. Using RF Gromacs is a petascale code (PRACE report D5.4), but using the PME approximation it is not even close to that scale. The developers are working on a 2D implementation of the parallel 3D FFT. Getting this done is of utmost importance. Additionally Gromacs benefits from a hybrid MPI+OpenMP approach. To get good scaling from the PME algorithm the machine has to provide a high performance MPI_Alltoall routine

3.12 HELIUM

Written by: Xu Guo, EPCC

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

3.12.1 *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 relatively straightforward to port HELIUM on different architectures as no specific libraries or environments are pre-required for the compiling and execution on most of the systems. However, due to the one-file large length source code, it could be difficult when trying to optimize HELIUM via manual code modifications.

The original HELIUM code only used MPI for the parallelization. The calculation is split to each process for implementation. Both point-to-point and collective communications are used in the HELIUM code.

The HELIUM application will write out results once every several time steps. The output frequency can be specified in the source code by particular parameters. In task 6.4, the total time steps was set to 80 and the writing operation was set to be implemented once every 20 time steps.

For task 6.4 of PRACE, the common used test cases on the prototypes Cray XT5 (Louhi@CSC) and Power6 (Huygens@SARA) have a 1540-block problem size. A larger problem size of 3060 blocks was also benchmarked successfully on BG/P (Jugene@FZJ), which should be large enough for the petascaling. The 1540-block problem size may not be sufficient for the petascaling on some prototypes, such as BG/P, but it will be very straightforward to create the new petascaling test cases by defining suitable parameters in the HELIUM source code and then rebuilding the application.

3.12.2 *Petascaling techniques*

The performance profiling results on the Power6 and Cray XT5 reveal that there are three main possible bottlenecks for the HELIUM petascaling performance:

- The memory limit was always the most possible issue for a scaling failure as the HELIUM code is a big memory consumer;
- When scaling the code to a large number of cores, the MPI communication cost will increase rapidly and result in being the biggest bottleneck;
- There are some high time expenses in the large loop calculation routines;

Based on the analysis of performance profiles, the main effort on petascaling techniques focused on improving the calculation/execution so as to be more efficient, as well as reducing the MPI communication costs.

- *Selecting proper compiler flags*

  Proper compiler flags were selected based on the prototype architectures and the code features. It is a good starting point to improve the efficiency of the calculation obviously without modifying too much of the original source code.

- *Reducing the communications cost*

  The MPI communications are quite costly when scaling to a large number of cores. However, some of the MPI communications were not necessary, and were therefore removed to reduce the communications cost.

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

  The MPI environment variables could be helpful to other applications on the prototypes, but there was not much to do to for the HELIUM source code with the MPI environment variables on Power6, Cray XT4/XT5 and BG/P. The MPI environments had very little effect on the MPI communications performance on these three prototypes.

- *Better memory/cache usage*

  When benchmarking on Cray XT5, Power6 and BG/P for this task, most of the executions were on fully populated nodes. Since the memory limit is a big issue for the scaling, sometimes utilizing all cores in a node can help make HELIUM run better. For example, when running with 630 cores on the BG/P, the SMP mode was used for a successful execution, i.e. with four MPI tasks per node. However, the performance could be poor with when all nodes are not fully populated.

  Some manual code modifications were made to get a better memory/cache usage. This focused on merging the large loops together in proper orders or introducing temporary variables during the long calculations. The code modifications reduced the calculation routine time cost and cache misses effectively. The modified routines included `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`.

Besides the three techniques above, another potential petascaling technique that can be applied is using OpenMP/MPI hybrid parallelization. Some experiments were done by implementing a hybrid version of HELIUM on both Power6 and Cray XT4/XT5 but the performance was very poor. This hasn't been solved currently due to the time limit and may need further investigation (estimated at around 6 pm).

It was a challenge to find the exact bottlenecks for the petascaling in the original source code, even with the profiling results. Not all possible petascaling techniques were suitable for optimising the HELIUM application. Sometimes the effect was not as good as expected.

It is important to make sure the correctness of the code after every applied petascaling technique. Sometimes it is necessary to ask the developers to ensure the validity of the code modifications. For task 6.4, all the correctness tests were passed for the final results.

- Power6 (Huygens@SARA):

Compiler flags	`-qfree=f90 -O4 -qessl -qarch=auto -qtune=auto –qhot`
Tasks allocation	Fully populated
MPI communications optimisation	Remove the calling of `Test_MPI` at the initialisation stage
Manual code modifications	1) Merged some loops with the same boundaries 2) Changed some loops' iteration orders to have a continuous data access in cache 3) Introduced some temporary variables to reduce floating point operations

- Cray XT5 (Louhi@CSC):

Compiler flags	`-O4 -OPT:Ofast:unroll_analysis=ON -LNO:fusion=2:full_unroll_size=2000:simd=2 -LIST:all_options=ON.`
Tasks allocation	Fully populated
MPI communications optimisation	Remove the calling of `Test_MPI` at the initialisation stage
Manual code modifications	1) Merged some loops with the same boundaries 2) Changed some loops' iteration orders to have a continuous data access in cache

- BG/P (Jugene@FZJ):

Compiler flags	`-O4 -qarch=450d -qtune=450 -qlanglvl=extended -qfree=f90 -qrealsize=8 -qsuffix=f=f90 -qessl`
Tasks allocation	Fully populated (VN mode) and Quad populated (SMP mode)
MPI communications optimisation	Remove the calling of `Test_MPI` at the initialisation stage
Manual code modifications	1) Merged some loops with the same boundaries 2) Changed some loops' iteration orders to have a continuous data access in cache

3.12.3 *Results*

The 1540-block size HELIUM test case scaled successfully on the prototype Cray XT5 (Louhi) up to 2485 cores. On Power6 (Huygens), the same problem size HELIUM test case was benchmarked up to 2485 cores, which scaled well to 1540 cores, but gave a poor scaling when using 2486 cores. On the prototype BG/P (Jugene), the 1540-block test case scaled up to 3003 cores. A larger test case with 3060 blocks was also scaled successfully up to 46971 cores on the BG/P prototype. The scalability and performance of HELIUM was improved when using proper petascaling techniques.

- Cray XT5 (Louhi@CSC)

The original HELIUM performance was compared with the performance of the new version of the code with the petascaling techniques, as shown below in Figure 29. Figure 30 is the corresponding cost plot. Note: cost = execution time * core number. I.e. a horizontal curve implies a linear scaling. It can be seen from the plots below that the HELIUM scaling performance and efficiency was improved on the Cray XT5 prototype after using the petascaling techniques.

Table 8 below shows the MPI profiling results before and after removing the unnecessary communication routine `Test_MPI`. The result values are the percentage of communication time cost out of the total execution time cost. Both the communication percentage and synchronisation percentage were reduced according to the profiling results.

	MPI_Sendrecv_replace	MPI_Barrier_(sync)
Original	7.00%	28.10%
With no `Test_MPI`	6.6%	13.7%

Table 8 MPI profiling comparison on Cray XT5 before and after removing Test_MPI.

Table 9 shows the detailed routines profiling in which the code modifications (merging loops and setting the loops in proper orders) were applied. A better memory/cache usage lead to a reduced time cost/percentage of these loops used for calculations.

	Original		New	
INCR_WITH_1ST_DERIV_OP_IN_R1	25 s	7.80%	19 s	6.20%
INCR_WITH_1ST_DERIV_OP_IN_R2	20 s	6.20%	19 s	6.20%
INCR_WITH_2ND_DERIV_IN_R1	6 s	1.90%	5 s	1.90%
INCR_WITH_2ND_DERIV_IN_R2	6 s	1.90%	6 s	1.90%

Table 9 Routine profiling comparison on Cray XT5 before and after merging loops.

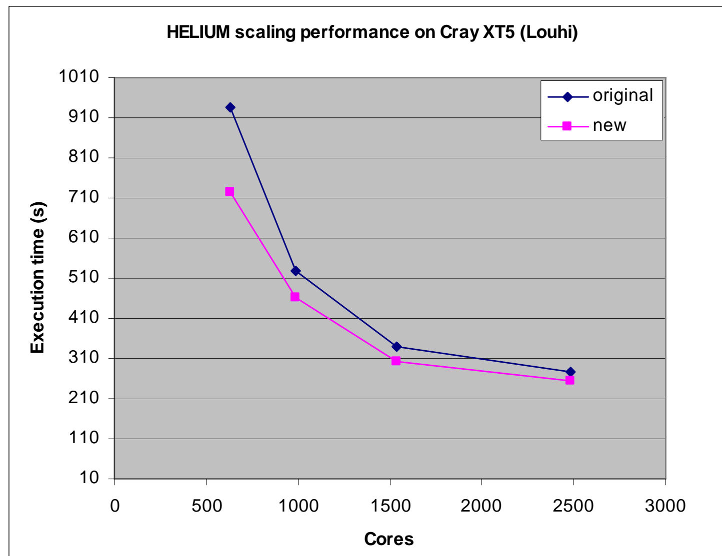

Figure 29 HELIUM scaling performance on the prototype Cray XT5 (Louhi)

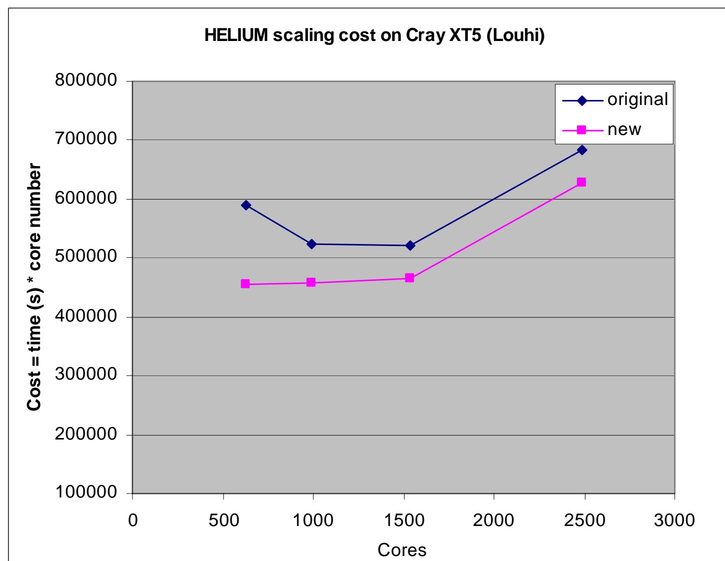

Figure 30 HELIUM scaling cost on the prototype Cray XT 5 (Louhi)

- Power6 (Huygens@SARA)

The original HELIUM performance was compared with the performance of the new version of the code using the petascaling techniques, as shown below in Figure 31. Figure 32 is the corresponding cost plot. Note: cost = execution time * core number. I.e. a horizontal curve implies a linear scaling. It can be seen that after using the petascaling techniques, HELIUM scaled with a better performance on the Power6 prototype.

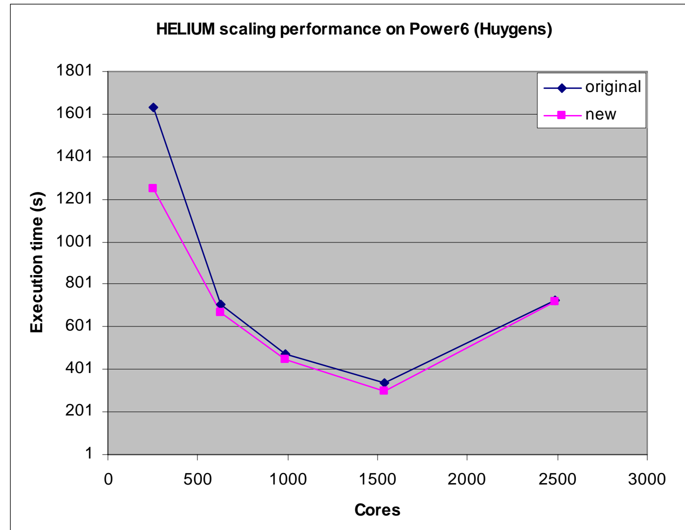

Figure 31 HELIUM scaling performance on the prototype Power6 (Huygens)

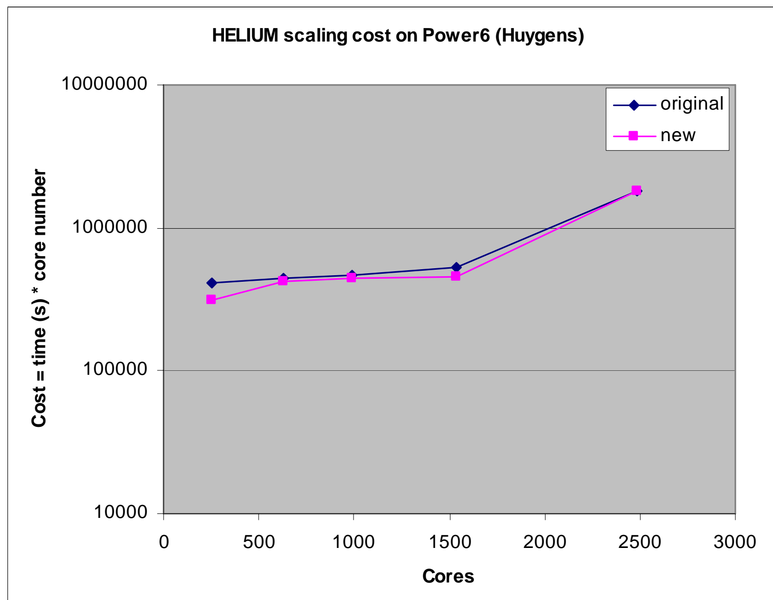

Figure 32 HELIUM scaling cost on the prototype Power6 (Huygens)

Table 10 below shows the MPI profiling results before and after removing the unnecessary communication routine `Test_MPI` on Huygens. The values are the communication percentage out of the total execution time, including the point to point communications, collective communications and the synchronisations.

	Point to point	Collective	Synchronisation
Original	10.82%	1.2%	23.56%
With no `Test_MPI`	7.22%	0.26%	23.83%

Table 10 MPI profiling comparison on Power6 before and after removing Test_MPI.

- BG/P (Jugene@FZJ)

The Jugene machine consists of 72 racks, each with 32 nodecards containing 32 four-way compute nodes. This represents a total of 24912 processing cores clocking at 850 MHz with an overall peak performance of 1 petaflop. 2 Gbytes RAM per node is provided.

The run-time environment on BlueGene/P provides three different execution 'modes':

- SMP mode, with one MPI task per node
- DUAL mode, with two MPI tasks per node
- VN mode, with four MPI tasks per node.

Tasks which 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. VN mode limits memory availability per core to 512 Mbytes. This was sufficient RAM for all the tests apart from the 630 core case, which required SMP mode to be set.

The BlueGene/P compute nodes are connected via five specialized network connections. Users can choose between MESH and TORUS topologies for a partition. These scaling tests specify TORUS wherever possible.

The original HELIUM code runs were based on an L_max value of 20. This parameter determines the number of angular symmetries in the calculation. As a result of problems with this setting on other PRACE prototype systems, this was reduced to a value of 16 in order to reduce overall problem size.

Weak scaling tests keep the local problem size constant as the core count increases. Thus the global problem size grows linearly as the number of cores increases. Ideal weak scaling is represented by similar timings for each core count (a flat profile).

Strong scaling tests keep the global problem size constant as the core count changes. Thus the local problem size per core reduces as the core count increases. Ideal parallel strong scaling is represented by timings that decrease linearly as the core count increases. Due to the very large range of processor counts available on Jugene it is appropriate to provide two problem sizes: 1540 global blocks for low processor counts and 3060 global blocks for higher processor counts.

For both weak and strong scaling tests, parallel scaling is very good up to the 40TF mark (11781 cores). However, beyond this core count parallel scaling performance degrades considerably due to communication overheads and parallel I/O inefficiencies.

Figure 33 shows the strong scaling of HELIUM on BG/P prototype, Jugene. The results of *flag* show the performance of using the proper compiling options with 1540 and 3060 block test case. The results of *new* are the data after removing unnecessary MPI routine and merging loops, etc. Figure 34 is the cost plot. The difference between the results of only using flags and using other techniques was quite small. Figure 35 and Figure 36 are the plots of weak scaling on BG/P prototype.

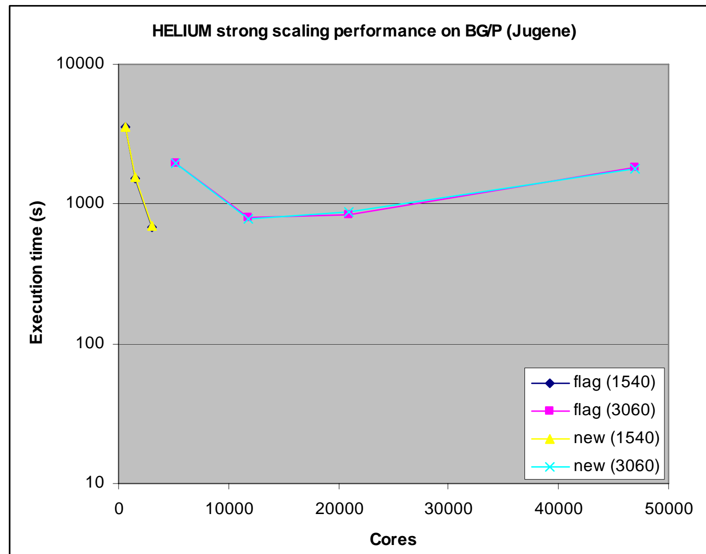

Figure 33 HELIUM strong scaling performance on the prototype BG/P (Jugene)

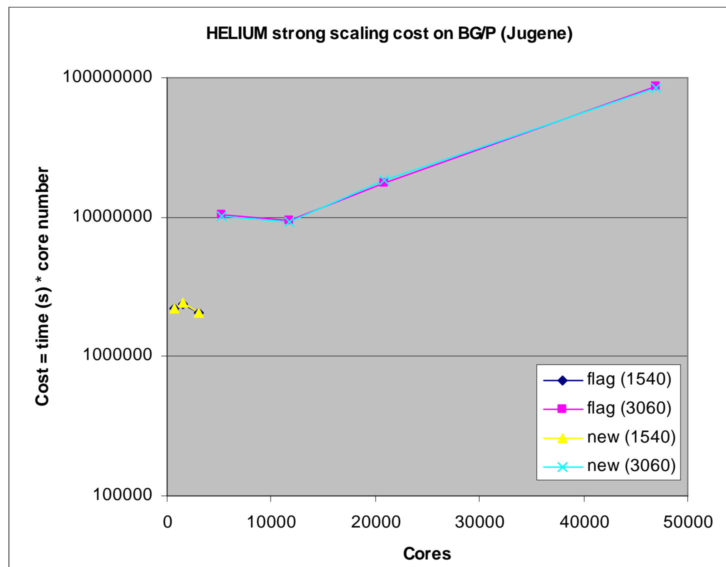

Figure 34 HELIUM strong scaling cost on the prototype BG/P (Jugene)

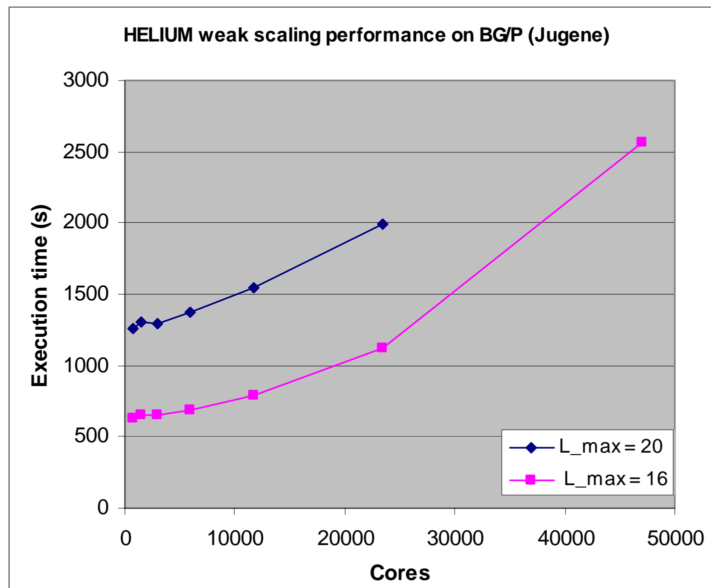

Figure 35 HELIUM weak scaling performance on the prototype BG/P (Jugene)

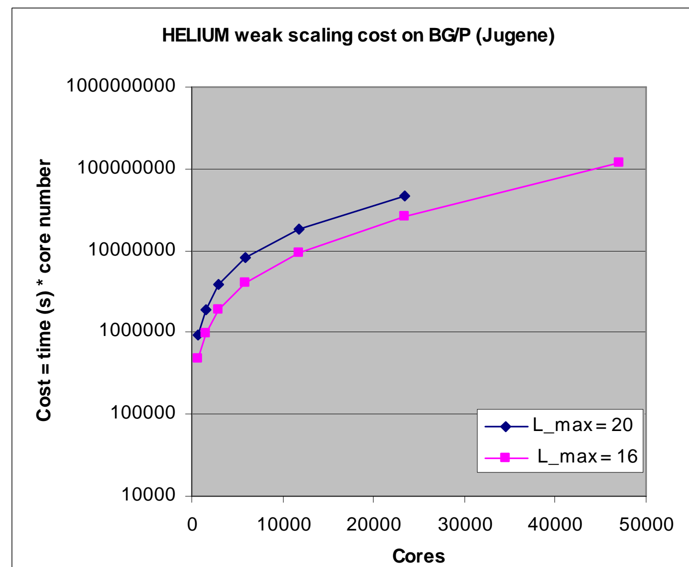

Figure 36 HELIUM weak scaling cost on the prototype BG/P (Jugene)

3.12.4 *Conclusion*

The employed petascaling techniques were efficient and improved the scaling performance on the Cray XT5 (Louhi@CSC) and Power6 (Huygens@SARA) prototypes. The HELIUM scaling on BG/P (Jugene@FZJ) scaled well up to 46971 cores for a large petascaling test case.

Selecting suitable compiling options is a good starting point to improve the scaling performance without too much manual code modifications.

MPI profiling and detailed user routine profiling are helpful to find out the scaling performance bottlenecks, but need to relate to the exact source code part. Reducing the unnecessary MPI communications can improve HELIUM scaling performance effectively. However, the MPI environment variables did not have too much effect on the HELIUM scaling performance on the Cray XT5, Power 6 and BG/P prototypes.

Proper manual code modifications may lead to a better memory/cache usage, but it depends a lot on the given architecture. It is very important to check the correctness after each code modification. Discussion with the original code developers may be helpful to make sure that the code modifications are valid.

HELIUM is still a big memory consumer however and not much can be done to change this. For the very large problem size, the executions can fail even with a successful build due to the memory limit. Therefore not all the core numbers can be used to run a large problem size. Selecting a proper core number is essential for HELIUM to scale well on a given system. Half populated or quad populated nodes may be helpful for the porting and scaling, but the performance could be poor.

The memory/cache, CPU architecture, CPU rate and interconnect can all affect the application performance if any changes occur. The petascaling performance and profiling should be tested on more architectures/prototypes if possible.

A basic hybrid parallelization optimization was applied to HELIUM but didn't get the performance improvements as expected. Further optimisations may require more challenging investigations and implementations. How to keep the code tidy and correct while improving the performance and scalability should also be noticed.

3.13 NAMD

Written by: Joachim Hein (EPCC)
Collaborators: 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.

3.13.1 *Application description*

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 Charm++ source is distributed with the NAMD source. In the case of NAMD 2.7b1, Charm++ 6.1 is included. For this investigation we have built Charm++ on top of the MPI library provided on the prototype architectures. In addition, on the prototype MPP-BG, we also build Charm++ directly on top of the system's "native" comms layer, for which the Charm++ distribution provides the required architecture description files. 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 position on the home patch and the 26 neighbouring patches is all that is required.

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 a simulation. These slower initial steps need to be taken into consideration when estimating the performance of a large production run from a test simulation lasting only a few hundred steps.

For this investigation we have obtained input data sets containing TCR-pMHC-CD complexes in a membrane environment [4]. These systems are interesting for studying 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 2fs. A scientifically meaningful simulation of such a system requires a trajectory length of at least 10 ns[4], which is equivalent to 5,000,000 simulation steps.

3.13.2 *Petascaling techniques*

The investigations done earlier as part of the PRACE project used NAMD version 2.6. A key problem with NAMD which was identified was the memory footprint of the application.[5] This proved to be a particular problem for the MPP-BG and MPP-Cray prototype architectures, offering comparably small amounts of main memory per core. We have strong indications that the memory requirements were the key reasons behind the inability to run the largest 9 million atom system with NAMD 2.6 on any architecture available to us. The memory footprint was seen as the main obstacle to petascaling.

In March 2009 the NAMD development team made a beta release of the forthcoming NAMD version 2.7 publicly available. Among new features and improved performance it also promises improvements with respect to the memory footprint [6]. A simulation with reduced memory foot print requires a NAMD executable, which has been specially compiled for this. Simulations with reduced memory footprint are classified as “experimental” by the developers.

Apart from compiling a separate executable, compressing the protein structure file (psf) is also required. To compress the psf-file a namd executable built without the memory compression is needed. For our 2 million atom test system, the uncompressed psf file measures 295 MB, while the compressed psf file measures 116 MB. Using a compressed

input file also reduces the startup time significantly. For example on a 256 cores of the MPP-Cray prototype startup time reduces from 53s to15s.

In addition to compiling a NAMD version with reduced memory foot print the NAMD team recommends [6] using the options `noPatchesOnZero` and `ldbUnloadZero`. These options ensure that no compute work is placed on rank 0 and the processor with rank 0 is left for tasks such as data I/O and load balancing.

Porting NAMD 2.7b1 to the prototype platforms and assessing how the new version addresses the problems encountered with version 2.6 was one of the main goals of the project in the recent months. Porting NAMD 2.7b1 also requires porting the latest version 6.1 of Charm++ to the target platform.

3.13.3 *Results*

The project succeeded in porting NAMD 2.7b1 to the MPP-BG, MPP-Cray and SMP-FatNode-pwr6 prototypes. Using the reduced memory footprint did indeed show an improvement for the problems encountered with NAMD 2.6

- On the MPP-BG with reduced memory footprint we had successful runs of the 1 Million atom benchmark when using all four cores per node. Without reduced memory foot print we could use two cores at most. For the two Million atom benchmark the executable with reduced memory footprint allowed utilising 2 cores per node while without reduced memory footprint only a single core per quad core node could be utilised.
- Running the 2 Million atom benchmark on the MPP-Cray system without reduced memory footprint requires fine tuning of the resources (e.g. buffer space) given to the MPI library for the application to run reliably without either the application being unable to allocate memory or the MPI library being swamped by unexpected messages. Using the executable with a reduced memory footprint is more stable and requires less tuning.
- We succeeded in running the 9 Million atom benchmark on the MPP-Cray and the SMP-FatNode-pwr6 system.

To quantify the memory consumption further we used the xt-craypat tool on the MPP-Cray prototype to measure the memory foot print of the NAMD 2.7b1 when using the 2 Million atom benchmark. It turned out that MPI rank 0 (master task) has a significantly different memory footprint from the other ranks. The investigation was repeated for three different values of the task count and we observed only a very limited dependence of the memory footprint on the task count. We also investigate the effect of the NAMD options `noPatchesOnZero` and `ldbUnloadZero` on the memory consumption by the master task. The following table summarises our findings:

Number of tasks	Memory reduction	No Patch on Zero	Unload Zero	Footprint rank 0	Average footprint
256	No	Default	Default	1.58 GB	0.87 GB
512	No	Default	Default	1.58 GB	0.85 GB
1025	No	Default	Default	1.52 GB	0.85 GB
256	**Yes**	Default	Default	0.60 GB	0.44 GB
512	**Yes**	Default	Default	0.58 GB	0.44 GB
1025	**Yes**	Default	Default	0.60 GB	0.43 GB
256	**Yes**	**Yes**	Default	0.60 GB	0.43 GB
512	**Yes**	**Yes**	Default	0.58 GB	0.43 GB
1025	**Yes**	**Yes**	Default	0.59 GB	0.42 GB
256	**Yes**	**Yes**	**Yes**	0.56 GB	0.43 GB
512	**Yes**	**Yes**	**Yes**	0.60 GB	0.43 GB
1025	**Yes**	**Yes**	**Yes**	0.60 GB	0.42 GB

Table 11 NAMD memory footprint

The table shows that even with the reduced memory foot print, rank 0 still consumes significantly more memory than the other tasks. The table also shows the NAMD options `noPatchesOnZero` and `ldbUnloadZero` do not yield a significant effect on the memory consumption. At least not for the benchmarks used within this project.

On the prototype MPP-BG we also experimented with the NAMD option `ShiftIOtoOne`, which moves the data IO from rank 0 to rank 1. It was observed that this lead to an executable, which would crash during writing of the final results.

The above has demonstrated that when using NAMD 2.7b1 it is possible to simulate very large systems with millions of atoms on the prototype architectures. The next question is scalability, when a large number of processors are deployed. An initial assessment obtained detailed performance figures for the three input configuration files on the MPP-Cray and the SMP-FatNode-pwr6 prototypes.

NAMD uses the initial steps to optimise its load balance, as discussed above. To obtain a more reliable estimate of the performance of a production simulation from short test runs we report on the NAMD benchmarking time. Our test runs print three benchmark times, the average over steps 301-400, 401-500 and 501-600. We typically note the best of these times, however, typically the differences between the three are at the few percent level. In the following figures we multiply the benchmark time with the numbers of computational cores utilised for the simulation, which is the total cost per step in core seconds. For a perfectly scaling code this would result in a flat line.

The following figures show the cost for the MPP-Cray and SMP-FatNode-pwr6 prototype, when using NAMD 2.7b1 with memory reduction. Comparing the performance of the NAMD version 2.6, 2.7b1 and 2.7b1 with memory reduction, the latter proved to be the best performing version [7]. On the SMP-FatNode-pwr6 system best performance is observed per physical processor when simultaneous multithreading (SMT) is used [7]. In case of NAMD this means placing two computational tasks on a single physical core.

The results show that NAMD can efficiently utilise several thousand compute tasks on the PRACE prototypes. This holds even for the smallest benchmark with 1 million atoms. When increasing the task count from a few hundred to a few thousand tasks, the efficiency dropped by less than 50%.

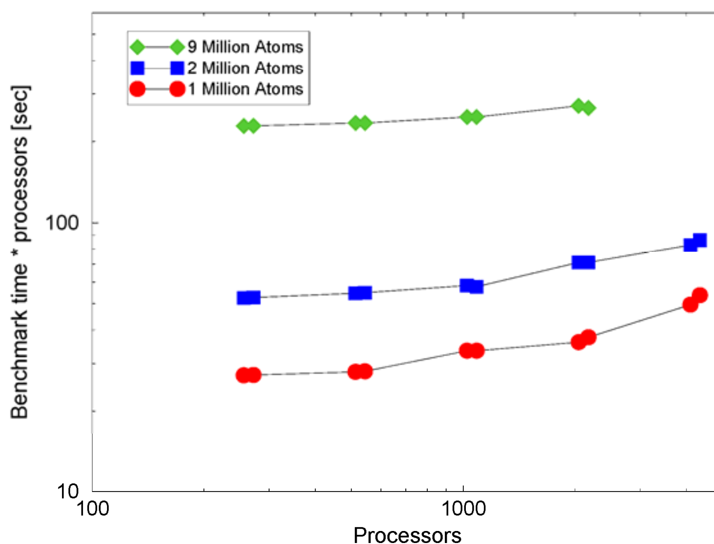

Figure 37 Computational cost per step for the MPP-Cray prototype forNAMD 2.7b1 with memory reduction

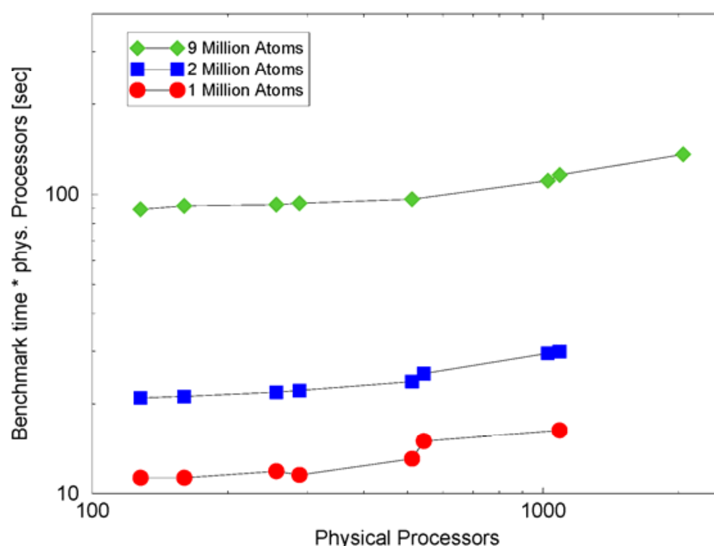

Figure 38 Computational cost per step for the SMP-FatNode-pwr6 prototype when using NAMD 2.7b1 with memory reduction. The results use the SMT feature of hardware – when using 1024 physical processors, 2048 computational task are placed on the hardware.

The next question we would like to address is whether multimillion atom benchmarks make sense from a bio-science point of view. We estimate the wall clock time required for a 10ns trajectory. This is shown in the following figure as a function of the peak performance of the partition of the prototype.

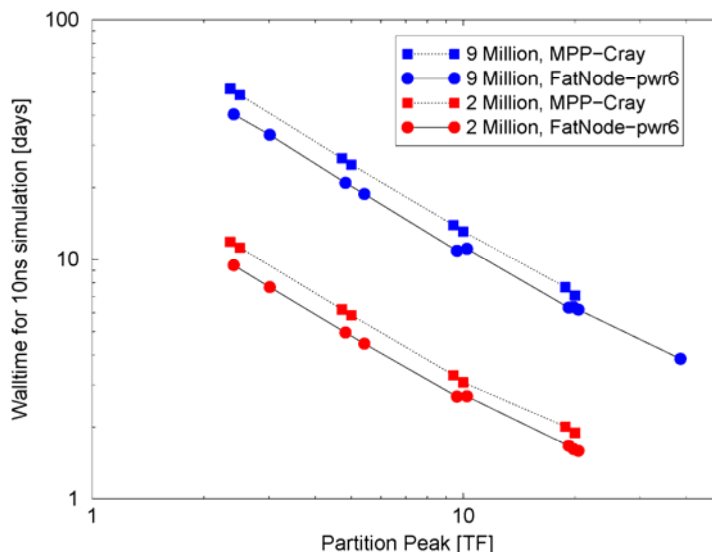

Figure 39 Wall clock time required for a 10ns simulation for NAMD 2.7b1 with memory reduction.

The figure shows that NAMD 2.7b1 is capable to create a 10ns trajectory for the 2 Million atom benchmark in less than 2 days of wall clock time. To create such a trajectory for the larger 9 million atom system would take less than 8 days of wall clock time when using a 20 TF machine partition. The time waiting for the solution and the resource level required are by no means unreasonable for a peta-scale system, given the scientific interest justifies the consumed resources.

3.13.4 *Conclusions*

The possibility to reduce the memory footprint in the new NAMD 2.7b1 is crucial to utilise NAMD at the peta-scale. A reduced memory footprint enables utilisation of compute architectures offering limited memory per computational core, such as the MPP-BG prototype. The reduced memory footprint is also required to simulate the system with 9 million atoms.

There are still some challenges left concerning the memory consumption of NAMD. On the MPP-BG prototype, we did not succeed in utilising all cores on the nodes for the 2 million atom benchmark and have not been able to get the 9 million atom benchmark running at all. On the MPP-Cray prototype with 1 GB of memory per core, running the 9 million atom benchmark on 4096 cores failed. The application indicated it could not allocate the memory it requires.

One of the key issues left concerning memory consumption is that NAMD appears not to feature a proper distributed memory model – the memory consumed per task does not seem to reduce when more tasks are deployed. This will prevent NAMD from efficiently running scientific cases with tens of millions of atoms on a future multi petaflop system, unless the system offers significant memory per compute core. Whether it is possible to re-engineer NAMD to offer a proper distributed memory architecture, requires more insight into the software architecture than we have been able to gain during the course of this project. An alternative mitigation strategy might be to rewrite NAMD or parts of it, to utilise more than a

single thread per compute task. This would allow having more memory available for each compute tasks without leaving a significant number of cores idle.

In summary our investigation shows NAMD is highly efficient when utilising several thousand computational tasks. This makes the simulation of bio-molecular systems with several million atoms feasible on a peta-scale architecture.

3.14 NEMO

Written by: John Donners, SARA

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

3.14.1 *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 of only 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, ie. to determine the domain decomposition that would have the most 'empty' domains. Andrew Sunderland has implemented irregular (but still rectangular) domain sizes in NEMO (not as part of PRACE). To achieve a good load balance, the workload per processor or area needs to be known well to choose the domain decomposition. Probably the workload varies during the simulation, which requires a dynamic or easily changeable decomposition.

NEMO 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 seems from benchmarks that the I/O is a prominent factor above 1000 MPI tasks.

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 sea ice 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. Their relative computational load (however, mostly due to communication) is about 20% and quickly increases with numbers of tasks.

3.14.2 *Petascaling techniques*

The 'north fold' is a line in the model across the north pole, which is shared by several processors. The exchange of data is slightly more complicated than the halo exchange at other domain edges, because it has either one or two neighbours. It is therefore implemented in three steps: gather the boundary data from all local domains onto one processor apply the boundary condition on this one processor and scatter the updated boundary data back to all local domains. This 'serial' operation starts to become a bottleneck when using more than 200 processors. On the Power6 system we have tried a few different approaches for the communication at the north fold:

1. to replace gather/scatter with one allgather operation and remove the scatter operation to save communication time. Unfortunately, the runtime increased by 25% on the huygens-Power6 system.
2. to initially determine locally the neighbours at the north fold and then send to and receive from the neighbours. At first, we used array slices together with asynchronous communication, but that does not work. The fortran compiler creates a temporary copy of the sliced array where all data is available contiguously in memory. This temporary copy is destroyed after the call is made. However, the nature of the asynchronous MPI call needs the array data available after the call is completed. This can only be solved using permanent copies, or an MPI derived type. We have chosen to use permanent copies using persistent communication channels. Because NEMO uses staggered grids, the neighbours are slightly different for variables on different grids. Therefore, different communication channels are required for every grid. The results are binary compatible and the scalability (speedup at double the amount of cores) improves significantly from 2.5TF to 5TF on Power6 (1.48 => 1.66) and Cray XT4 (1.13 => 1.33). (1PM)

NEMO has the option to use two different solvers for the surface boundary equation: preconditioned conjugate-gradient (pcg) and successive overrelaxation (sor). With the original communication routines, both methods give a very bad scaling characteristic, because both require several hundreds of iterations with halo exchanges. Both methods require a periodic Allreduce call (not necessarily every timestep) to determine the convergence. However, from 1000 cores it becomes clear that both methods have different bottlenecks and different possible solutions for further scalability:

1. The pcg algorithm needs two further Allreduce calls every iteration, and it usually requires a few hundred iterations every timestep. The large amount of Allreduce calls becomes the bottleneck at high task numbers. The deflated conjugate-gradient method could significantly reduce the number of iterations that the pcg solver needs to converge (maybe even a factor of 10) and extend the scalability to several thousands of cores. This method divides the complete mesh into a small number of deflation groups, and the idea is that the convergence speeds up if the solution has been found in a separate group. Ocean models like NEMO are usually converged in a large part of the domain, except for the region around the north pole. This indicates that the deflation method could be succesful in ocean models. The deflation method is not yet implemented.
2. The sor algorithm needs more iterations, but doesn't need any additional Allreduce calls. The bottleneck is therefore the exchange of the halos.

Derived MPI datatypes are implemented to speedup the performance of the east-west halo exchange in the communication subroutine. The arrays are laid out in memory as east-to-west,

north-to-south and top-to-bottom. This method is implemented for 2D arrays (no vertical axis) and an implementation for 3D arrays is similar and planned. The domain is split in two horizontal dimensions, so halo exchanges are needed in the east, west, north and south directions. The halos along the northern and southern edges are contiguous in memory and can therefore easily be communicated with MPI. The halos along the eastern and western edges are not contiguous in memory and are now copied to and from temporary arrays. The implementation of this array copy interchanges both array dimensions, which results in lots of cache misses. MPI allows for a vector type that consists of a series of equally spaced blocks. The size of the blocks needs to be the same on the sending and receiving end, but the spacing can be different. With this method the temporary arrays can be removed completely and many vendor-specific MPI libraries contain optimizations for this MPI types that are much better than naive manual code. The scalability has improved remarkably on Power6 (1.66=>1.85 up to 10TF) and on Cray XT4 (1.33=>1.44). Unfortunately, this method did not yet work on BlueGene/P.

3.14.3 *Results*

- The use of Scatter and Gather operations is replaced by sends and receives, which improves scalability on all platforms.
- The use of MPI vector types to exchange halos that are non-contiguous in memory also improve scalability on all platforms.

3.14.4 *Conclusions*

- Scatter and Gather operations are essentially serial in nature, and when used with large arrays this can quickly become a serious bottleneck for scalability.
- The key bottleneck is the solver, solver algorithms suffer from their very low computation/communication ratio. The conjugate-gradient algorithm could profit from a decrease in the number of iterations through the deflated conjugate-gradient method, while the successive over relaxation method would gain mostly from faster halo exchanges.

3.15 NS3D

Written by: 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.

3.15.1 *Application description*

The Code is completely written in Fortran 95 and neatly separated into modules. The code authors try to keep a uniform format throughout the application. Generally useful comments are found in most parts of the code, but mostly in German.

The most time consuming algorithm in the application is the solving of tridiagonal equation systems for the compact finite difference scheme. This roughly amounts to 40 % of the total computing time.

The next computational intensive algorithm is the FFT which amounts to around 25 % of the total computing time. The third major computing component with 17 % is the evaluation of long complicated terms for time derivatives.

Preprocessing within the code itself consists mainly of reading the data files and performing some minor operations, the needed time for this is negligible and already parallel. Preprocessing outside the actual code has to be done in the sense of setting up a so called baseflow, which is a 2D solution of the larger flow structures, for which the DNS is performed. However the computational load for this is very minor and done serially. The decomposition is done in two dimensions and the partitions are computed in parallel using MPI. In the third dimension the FFT has to be solved for the spectral ansatz, so a decomposition is not easily possible. Within each partition, shared memory parallelization is implemented by using the NEC Microtasking feature. The decomposition has to be done before the compilation, as the partition sizes are hard coded into the program, and need to be known at compile time.

The IO is done using the eas3 library and its binary format. Periodically a restart file is written to disk, these files are the largest possible ones, as they contain all necessary information to start the simulation from that point again. Beside the restart files the actual output with only selected values is done, either once at the end of the simulation or for transient simulations every given timestep interval. Principally there is no difference between the output and restart files, thus the restart file output is representative for the complete application.

The dataset we are using is derived from a testcase where two flows meet after a plate and create a shearwake when they are intermixing. This setup is for example appearing at the outlet of a turbine, where especially the noise generation is of great interest.

The dataset can be easily enlarged, by extending the computational domain in flow direction, as this extension is currently of most scientific interest in order to follow the generated structures further downstream. For the current benchmarking analysis a simply extended baseflow is used, derived from the original realistic benchmark, however a more realistic baseflow for a very large simulation area could easily be generated, however from a computational point of view there would not be a large difference between these testcases.

3.15.2 *Petascaling techniques*

In the code, a hybrid parallelization concept of domain decomposition based on MPI and shared-memory parallelization with Microtasking is implemented. Based on the domain decomposition a graph-communicator is created which distributes the processes according to the connectivity of the nodes. Data exchange is mainly point-to-point (blocking and non-blocking) and broadcast communication is used only during initialization.

Concerning scaling, the tridiagonal solver for the compact finite differences is the most relevant part. In order to avoid serialization due to the recursive loops of the Thomas algorithm, a pipelined version of this method is implemented. It uses the fact that not only one but up to 25 derivatives must be computed and works as follows: the first domain starts with the recursive loop of the first derivative. After its completion, data is transferred to the next domain which continues with this derivative. Concurrently the first domain starts to work on the second derivative. This procedure continues accordingly for the following domains/derivatives. It yields a speed-up of $(m \cdot n)/(m+n-1)$ for the tridiagonal solver, where n

is the number of domains in the current direction and m is the number of derivatives (here m=25). This limits the speedup of the tridiagonal solver to m=25 in each direction. However all other parts of the code including I/O are local for each MPI process and thus the overall scaling is better. Such theoretical analysis of scaling is confirmed by previous tests up to moderate numbers of processors. Optimizations of the tridiagonal solver (see D6.5) automatically improves the overall scaling by reducing the time spent in the pipelining subroutines.

If the described limitations, due to solving of the tridiagonal system are too restrictive, the code also allows to switch to explicit finite differences. These are less accurate but then all computing operations are local and data transfer at the interfaces is required only once per intermediate time step.

As an alternative, it might be possible to use the compact scheme only within each domain, but larger explicit stencils at the domain interfaces. By increasing the stencil, similar numerical properties of explicit and compact finite schemes can be achieved but data transfer increases.

In order to avoid data exchange of the complete flow field, domain decomposition is not applied to the spanwise direction with its Fourier discretisation. However an alternative formulation based on compact finite differences is already implemented. Thus there is also the option to extend the domain decomposition to the third direction in the future.

Up to now shared-memory parallelization is only implemented via NEC specific Microtasking. To use shared-memory parallelization on other machines as well, it would be necessary to convert those compiler directives to OpenMP, which is a major effort.

The load balancing and MPI communication system in the application is already quite sophisticated, and as only nearest neighbor communication is deployed this promises only little improvements. Thus focus has been laid upon the tridiagonal solver and reducing its share of the computational time consumption. However the deployed strip mining mechanisms also help the shared memory parallelism, as it provides blocks of reasonable size for each thread to act upon.

The restructuring of the shared memory parallelism within the NEC-MicroTasking framework already took around 2 PM. One of the most essential parts of the work was to ensure the correctness of the results after modifying the code.

Pre- and Postprocessing are negligible, even on large partitions their time consumption accounts for less than 2 %.

3.15.3 *Results*

The scaling behaves quite as expected on the SMP-ThinNode-x86 and the SMP-ThinNode+Vector, however on the SMP-FatNode-pwr6 there is an extraordinary good scaling behavior even with the compact finite difference scheme, which is the only one analyzed so far. On this system even a superlinear speedup can be seen, which is supposed to be due to caching effects in the strong scaling analysis, and points to possible single core optimizations by better cache usage.

On the MPP-Cray it seems to be quite obvious, that explicit finite difference schemes are necessary to make efficient use of large system partitions. However even the compact finite differences are already scaling quite well on that system. An opportunity for improvement might have been shared memory parallelism within the MPI communication layout, so we started out exploring this scheme. However, as this feature was implemented in a NEC-SX

specific way, this analysis was done on the NEC-SX9 which is part of the SMP-ThinNode+Vector prototype.

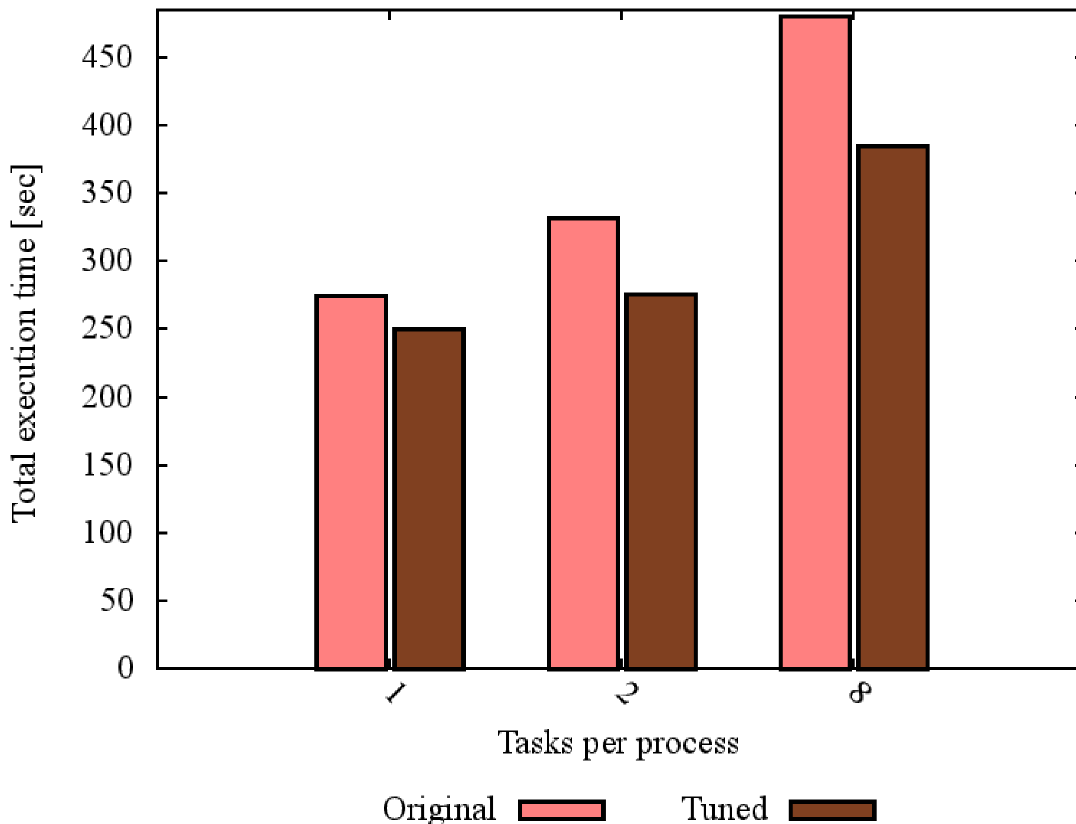

Figure 40 Effect of shared memory parallelism before and after tunings on a single NEC-SX9 node

In Figure 40 the impact of using shared memory parallelism together with MPI processes is shown. This analysis was done on a single dedicated NEC-SX9 node with 16 processors. As can be seen, the execution is slowed down when using shared memory parallelism in combination with MPI. After the tunings, where the shared memory parallelism is deployed on code blocks instead, this effect got less bad, but still remained. In fact, if the shared memory parallelism is completely disabled at compile time, the execution is even faster by 10 % than depicted for the single task in the graph.

As can be seen in Figure 41, this still holds true for larger partitions of the machine, in fact even on the complete machine with 12 nodes, it is beneficial to use a pure MPI-implementation. In the light of this result it is questionable, if a significant benefit can be drawn by applying OpenMP directives for other platforms. For systems with large numbers of processors it is most probably better to use explicit finite differences instead.

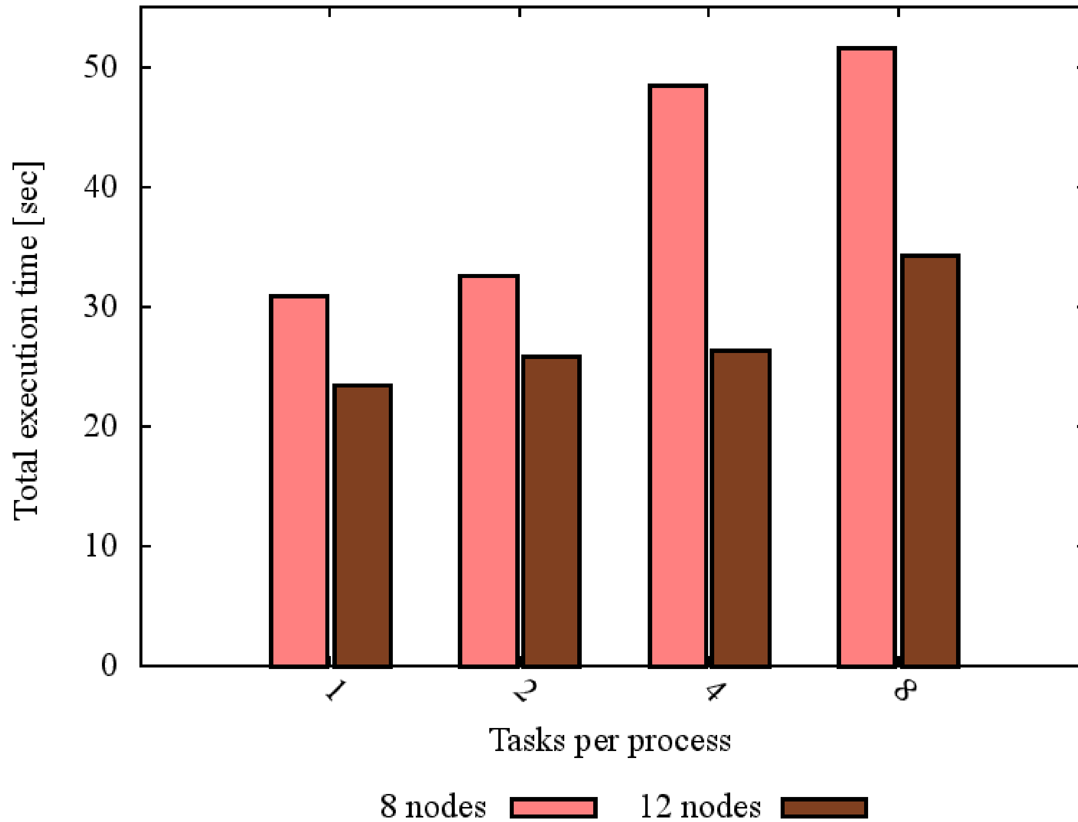

Figure 41 Analysis of several task per process combinations on 8 and 12 NEC-SX9 nodes

An overview for the scaling enhancements gained by the deployed blocking mechanisms and increased efficiency of the tridiagonal solver is given by Figure 42. As can be seen, the scaling improved together with the single node performance by the applied modifications. However for this smaller test case after 4 nodes the efficiency breaks down, as the problem sizes per processor get too small for the vector architecture.

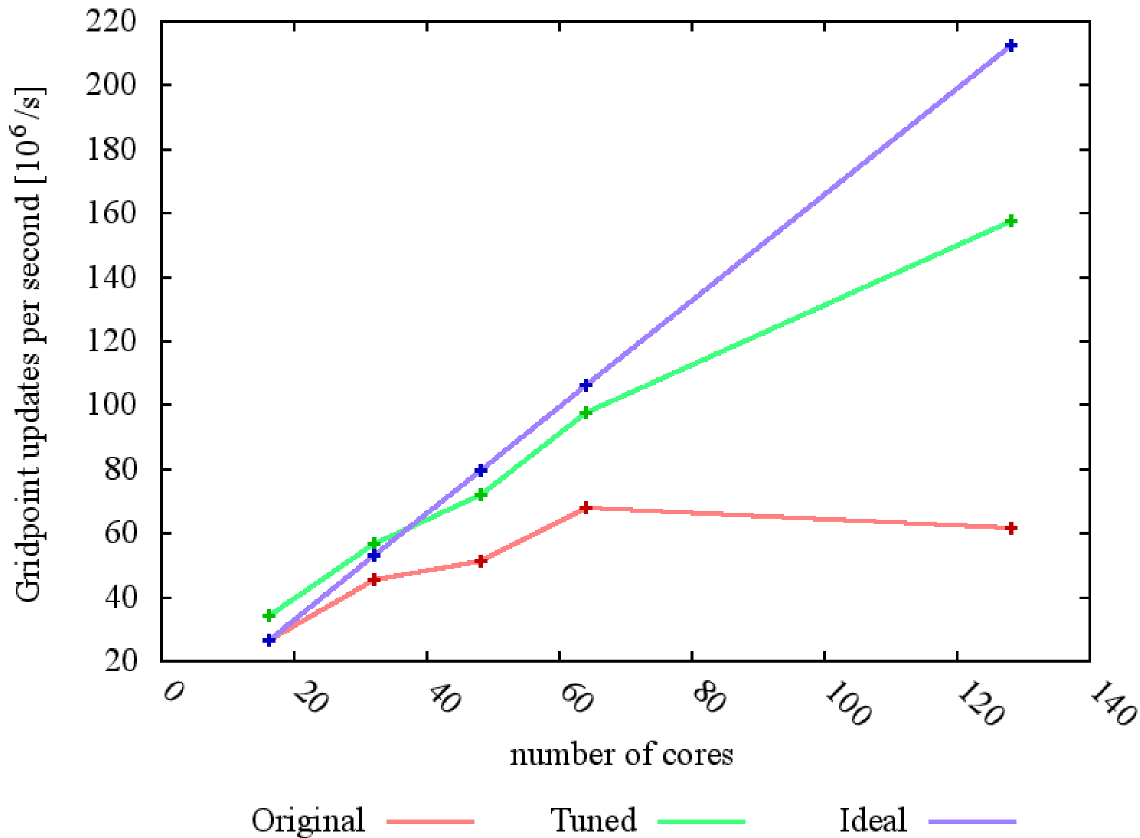

Figure 42 Scaling before and after optimisation on the NEC-SX9 processors in comparison to ideal scaling of the original code

3.15.4 *Conclusions*

NS3D has been developed on and for the NEC SX series, where each processor is quite powerful, thus only a few of them are needed to gain short execution times. Therefore scaling was not the main focus of the developers so far and the most efficient but inherently sequential compact finite differences are the preferred discretisation method. Even so the first results on the investigated PRACE platforms showed reasonable scaling with pure MPI parallelization and without tunings with respect to scaling.

The analysis of hybrid parallelism for this code showed no advantage for shared memory parallelism within the MPI processes. This is probably due to the well decomposable problem in MPI with only nearest neighbor communication involved. Yet, scalability is limited by the solving of the tridiagonal equation system, which therefore has to be dropped on systems with very high numbers of processors, like MPP-Cray. This has yet to be analyzed. Rewriting the shared memory parallelism to OpenMP for those systems, however, is deemed as not worth the effort for this code.

The main scaling improvement in this application was gained by decreasing the time fraction used by the inherent sequential tridiagonal solver. In other terms there is only little room for improvement of the scalability, as only nearest neighbor communications are deployed in an already quite sophisticated way. Different communication layouts may be tested for further small improvements, though. It may also be possible to combine some of the messages to larger ones.

The main task still subject to investigation is the behavior of the code when using explicit finite difference schemes.

3.16 Octopus

Written by: 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/.

3.16.1 *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.

Besides an 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 optimizations, 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 step 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.

3.16.2 *Petascaling techniques*

Octopus is designed so that data is partitioned among processors. However there is some amount of data that has to be stored in every processor and is independent of the number of processors in the particular parallelization level. When the systems to be studied are large and a large number of processors are used, the amount of non-distributed data becomes considerable, so one of the first tasks to achieve petascaling was to reduce the amount of this data.

One of the main improvements performed in the code was to rewrite performance-critical routines so that they work with blocks of vectors at a time, instead of working with single vectors. This allows us to group communication operations in the code, reducing the effect of communication latency. Working by blocks is also beneficial in some cases for single threaded execution performance, since more efficient matrix-matrix multiplication operations can be used.

Since memory constraints were determined to be one of the key limitations for parallel runs, a hybrid OpenMP + MPI parallelization was implemented. This is particularly critical for systems like the BG/P were the memory per processor is small in comparison with the requirements of Octopus. The operation by blocks mentioned above was useful to improve the OpenMP implementation, since it exposes more parallelism to the low-level routines.

3.16.3 *Results*

For the PRACE benchmark two different datasets were considered: the first consists of a fullerene molecule containing 240 atoms (960 electrons) and the second is a fragment of the light-harvesting complex in spinaches, containing 650 atoms (1654 electrons). These are physically relevant benchmarks designed to run up to several hundreds of processors, so that they would fit in PRACE prototypes. They are, therefore, not suitable to scale to PetaFlop/s machines. It is trivial to extend these tests to such machines by taking a larger fullerene or a larger fragment of the light-harvesting complex.

In the tables below, green indicates superlinear scaling, blue indicates suspicious results in need of confirmation and red points to possible bad scaling. “tdtime” is the time, in seconds, to perform 10 time-steps of the propagation of the Kohn-Sham wavefunctions. A realistic simulation would require several thousand time-steps. All time-steps take strictly the same time. Scaling was computed by comparing the ratio of “tdtime” values to the ratio of the total number of threads. The mflops values are real, measured internally by octopus. These values are an underestimation as not all parts of the code are considered, but work is currently in progress to have better estimates through the use of performance counters throughout the entire code (using PAPI).

ncpus	nodes	taskspeode	threadsask	tdtime	tdmflops	total mflops	**scaling**
BlueGene							
256	64	1	4	103.1	607.6	38886.4	**1.00**
512	128	1	4	56.16	565.9	72435.2	**0.93**
1024	256	1	4	32.65	486.7	124595.2	**0.80**
FN-pwr6							
64	2	32	1	107.23	1012.6	64806.4	**1.00**
128	4	32	1	70.58	759.2	97177.6	**0.75**
256	8	32	1	43.06	603.6	154521.6	**0.60**
512	16	32	1	15.59	833.8	426905.6	**0.82**
TN-x86							
16	2	8	1	301.21	1465.8	23452.8	**1.00**
32	4	8	1	160.28	1382.5	44240	**0.94**
64	8	8	1	81.77	1365.1	87366.4	**0.93**
128	16	8	1	42.86	1321.7	169177.6	**0.90**
Cray XT							
32	4	8	1	243.66	909.4	29100.8	**1.00**
64	8	8	1	133.66	835.2	53452.8	**0.92**
128	16	8	1	72.83	777.8	99558.4	**0.86**
Pwr6+Cell							
8	4	2	1	2949.58	299.4	2395.2	**1.00**
16	8	2	1	1478.58	298.6	4777.6	**1.00**
32	16	2	1	756.26	293	9376	**0.98**
64	32	2	1	657.95	169.7	10860.8	**0.57**

Table 12 Results for C240.

ncpus	nodes	taskspeode	threadsask	tdtime	tdmflops	total mflops	**scaling**
BlueGene							
512	128	1	4	112.18	506.9	64883.2	**1.00**
1024	256	1	4	59.35	479.1	122649.6	**0.95**
2048	512	1	4	31.84	453.2	232038.4	**0.89**
FN-pwr6							
128	4	32	1	99.72	989.6	126668.8	**1.00**
256	8	32	1	53.08	915.4	234342.4	**0.93**
512	16	32	1	27.26	880.4	450764.8	**0.89**
1024	32	32	1	74.61	152.9	156569.6	**0.15**
TN-x86							
32	4	8	1	367.32	1051.7	33654.4	**1.00**
64	8	8	1	169.75	1137.9	72825.6	**1.08**
128	16	8	1	88.48	1091.4	139699.2	**1.04**
256	32	8	1	47.82	1009.8	258508.8	**0.96**
Cray XT							
64	8	8	1	270.96	712.8	45619.2	**1.00**
128	16	8	1	146.37	659.7	84441.6	**0.93**
256	32	8	1	80.16	602.3	154188.8	**0.84**
Pwr6+Cell							
32	16	2	1	1897.79	203.6	6515.2	**1.00**
64	32	2	1	797.39	242.2	15500.8	**1.19**
128	64	2	1	697.9	138.4	17715.2	**0.68**

Table 13 Results for 650-atom light-harvesting complex.

3.16.4 *Conclusions*

Scaling seems to be excellent for each problem size and runs could be performed efficiently up to 2 to 4 threads per atom. Note that the second benchmark is roughly a factor of two more demanding than the first one. Comparing both sets of results, there seems to be a linear improvement of the scaling with the system size. This is extremely promising, as there are

many systems, typically of biological interest, that contain from 10 to 1000 times more atoms than those used in the benchmarks. This indicates that calculations for these larger systems would scale to tens of thousands of processors, being thus ideal candidates for the new PRACE PFlop/s machines.

The superlinear behaviour in the second benchmark, for some architectures, can be easily explained. In single node machines, most of the methods used in octopus are typically limited by memory access. By increasing the number of nodes, the required memory per node decreases, alleviating the memory access problem and increasing overall efficiency. However, the increase of the number of nodes brings communication overhead problems, that eventually dominate the overall scaling, lowering the efficiency.

The results marked in blue are suspicious as they do not follow the trends. In the Cell prototype, these suspicious cases appeared together with a network warning regarding a missing network interface. This might point to a misconfiguration of the machine. The same can be said of the FN-power6 results.

Finally, in the BlueGene, the code seg-faulted when using non-blocking communications (with two different implementations, one using LIBNBC and the other using directly non–blocking point to point communications). This can limit the parallel performance. A solution is still being actively investigated.

3.17 PEPC

Written by: 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(squared)) 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.

3.17.1 *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 mulitpole 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.

3.17.2 *Petascaling techniques*

The current status of the petascaling effort is to test the new PEPC-E frontend. Up to now the code runs for up to 8192 cores on the MPP-BGP and can thus saturate all target prototypes but the MPP-BG/P.

The scaling of the old version (PEPC-B) is shown in Figure 43 and the scaling of the new PEPC-E frontend on the target architectures is shown in Figure 44. Note, that the two figures cannot directly be compared, as the solver and the problem size are different. The important point is the possibility to use larger partitions with the new version of PEPC.

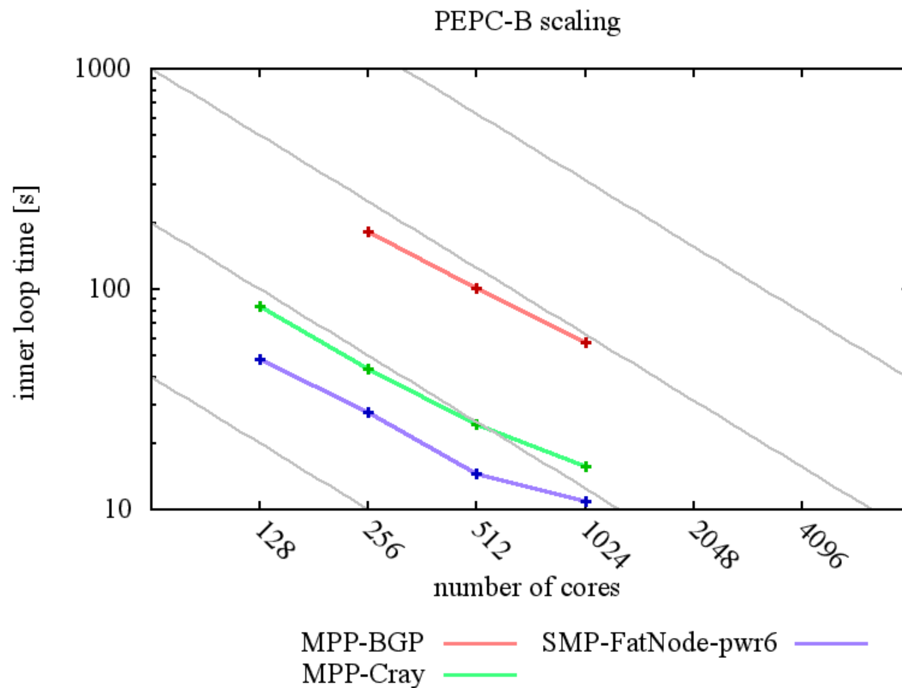

Figure 43: Scaling of PEPC-B on target architectures

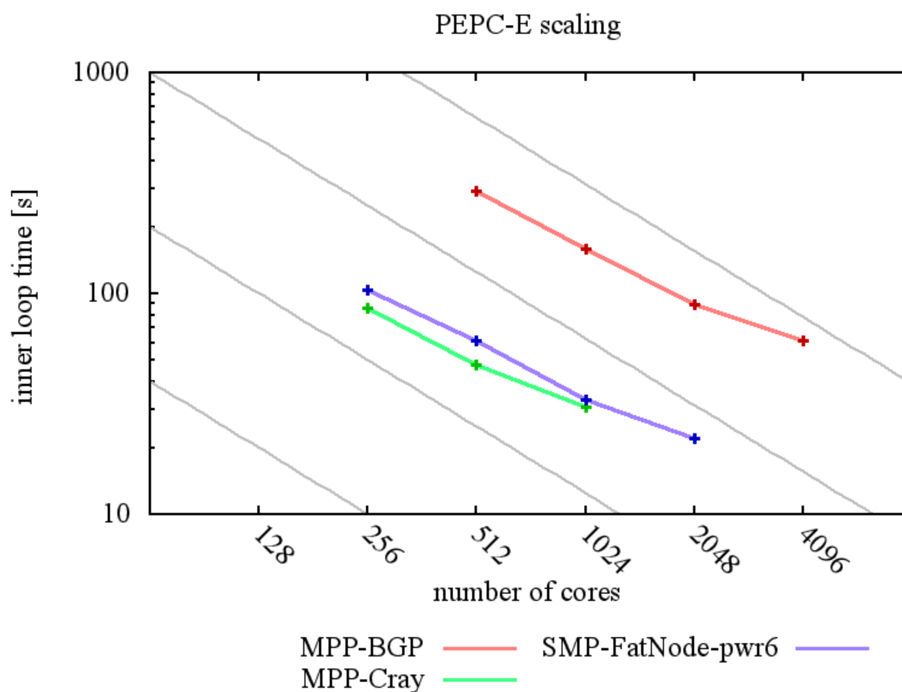

Figure 44: Scaling of PEPC-E on MPP-BG architecture

The main difference between the two PEPC versions is the memory management. The limiting factor for using large partitions is the memory available for each core, as this is nonlinearly dependent on the number of MPI tasks, i.e. it grows very strongly with the number of tasks. Due to this, PEPC-B is, at least at the current state of development, limited to about 1024 MPI tasks. The PEPC-E frontend is, in contrast to PEPC-B, using just the PEPC core functionality, the tree code, and no additional physical features, like laser pulses for example. These additional features were and are not used for benchmarking, but they

generate a lot of overhead, especially in the memory footprint. In addition to this, some not needed structures have been removed to further decrease the memory needs.

Although this resource usage reduction increases the scaling range of PEPC, the fundamental limits remain and avoid the scaling on a petaflop machine. The main development effort done right now is to fundamentally redesign PEPC to run and scale on partitions in the order of a petaflop machine. The currently followed way is to leave the homogenous path, this means that all MPI tasks are equal, and to setup a heterogeneous strategy. This will include two types of actors: so called data brokers and computing nodes. The computing nodes will not have any global information, like they do now, but will rather know which information they have and which they need to do the computation. All detailed physical information, i.e. the particles, are stored and computed on these computing nodes. On the other side, the data brokers will have global information, but with a limited scope. They will know all other data brokers and their corresponding computational regions. Their main functionality will be the creation of communication paths between the computing nodes and the caching of frequently used information.

The following example demonstrates the new strategy. A common situation in a PEPC simulation is the request for multipole moments of a distant particle cluster. What happens now is that the requesting node and its neighbours ask the distant node separately for the same information. The network path for such a request might be quite long, for example on the torus network of the MPP-BG/P architecture, and the same information is multiply transferred across the network. At this stage the data brokers come to their functionality. The neighbouring nodes indicate the data broker the need for distant information, which now assigns just one of them to do the distant communication and the others to ask for the requested information to their neighbour, which actually does the communication. Obviously the non assigned node can start other distant communication and share their communicated information with the others, using only neighbour communication. In this way the communication scheme will be more scalable and the diagonal elements of the communication matrix will become more dominant. Additionally, as the timescales of a single time step and the change in the multipole moments of large particle cluster are very different, the data broker will be able to speculatively prefetch data and even distribute it to its assigned computational nodes, using the overlap of computation and communication.

As this is in the early stage of development, no data has been collected. The effort to restructure the communication scheme and to integrate it in the current production workflow will be in the order of one person year.

Pre- and post-processing in PEPC-E is restricted to the creation of the initial particle position and to the cleaning up. The particle creation in the pre-processing is serial, due to the capability to verify the results on all prototypes and all partition sizes. As the initial particle positions are randomly set, the same random number sequence has to be used, i.e. it must not depend on the number of MPI tasks, the compiler and the architecture. To do so, each MPI rank computes, but does not store, the initial conditions for all particles in a well defined way and keeps just the one which are assigned to it. As this is in the order of a few seconds and does not depend on the number of MPI tasks, no effort is needed to reduce this procedure. The post-processing time is not significant.

3.17.3 *Results*

The speedup gained on the different platforms is in the order of 1.8, at small partitions, to 1.5, on the largest partitions. This speedup is not fundamentally increased by the change from PEPC-B to PEPC-E, but the maximal partition size was increased. As PEPC has a non

structured communication scheme, involving the communication of all tasks with all other ones, this scheme does not scale well and so does the application. Applying the new heterogeneous strategy will advance the parallelization of the communication scheme, resulting in a higher global speedup.

3.17.4 *Conclusions*

The current development stage focuses on a fundamental restructure, as described above, of the communication scheme which will not be optimized for any specific physical network, at least not at this stage. However, one might expect that the data broker and computing node separation could benefit from a hierarchical network topology. In addition to this, PEPC's scaling might also be improved if the MPI support of the computation and communication overlap is well supported, as the new communication strategy will include a prefetching scheme.

3.18 SIESTA

Written by: Rogeli Grima, BSC

SIESTA is a method to perform electronic structure calculations and *ab initio* molecular dynamics simulations of molecules and solids. It implements, in self consistent DFT, the order-N techniques developed for tight-binding.

3.18.1 *Application description*

SIESTA is a program written in Fortran with about 200.000 lines of code. Initially, it was a purely serial code. This was convenient for many "modest" users, who used single workstations, and this gave it a rapid popularity. Later, parallelism was added "on top", with the main priority being not to compromise serial execution, and without revising the algorithms for good parallelism. Unsurprisingly, this leads to unbalanced executions. The program uses several external libraries: BLAS, LAPACK, SCALAPACK, MPI, METIS and ARPACK.

SIESTA is a purely academic project. It is the result of the contribution of many different people with different backgrounds. The code style is not uniform and some parts of the code are well documented and readable, but some others not. Although it is not free software, it is distributed free of charge to all academic users.

These are the main parts of the program:

```
DO Time/movement loop
    •   Construction of Overlap matrix: S
    DO Self-consistency loop
        •     Construction of Hamiltonian
          matrix: H
        •     Compute eigenvalues: H·x = λ·S·x
    ENDO
ENDO
```

In the original code the construction of the Hamiltonian and the overlap matrices were parallelized with MPI. During this process SIESTA computes the contribution of every orbital to every point of a three dimensional mesh. The original implementation distributes this mesh equally among all the processes. This was a bad idea because the density of orbitals in the mesh can be different and this can lead to an unbalanced workload distribution.

Meanwhile, the computation of eigenvalues of the system was done using SCALAPACK. We found several problems in this library: It only works with dense matrices (The hamiltonian and the overlap matrices are sparse structures); it computes all the eigenvalues of the system (we do not need them all); finally it only scales to several hundreds of processors.

3.18.2 *Petascaling techniques*

Our first goal in order to improve the parallelization of SIESTA was to fix the problem with the unbalance in the workload during the construction of the Hamiltonian matrix. This unbalance arises because the work associated with every point of the mesh is different. In fact, the construction of the Hamiltonian matrix has four steps and every step has a different cost for every point of the mesh:

- *Rhoofd*: The amount of computation depends on the number of orbitals that intersects every mesh point.
- *Poison*: Computation of a 3-dimensional FFT. The amount of computation is uniform.
- *Cellxc*: We only have to make computations on those mesh points intersected by orbitals
- *Vmat*: Has the same pattern as Rhoofd.

We have created three data distributions that are optimal for every function. Every time we enter into a new function we use a new data distribution. This means that the processes have to communicate in order to get the proper data. These communications are point-to-point. The information to send or receive can be in any other process. For this reason we have precomputed a scheduling in order to overlap the communications and avoid blocking situations. After a self-consistency step the atoms and the orbitals of the system can move inside of the grid and we should recompute the data distributions and communications scheduling.

The parallelization is done with MPI, but we have also used OpenMP in order to reduce the number of domains. We use bigger domains which means that the number of communications between processes is reduced. The introduction of OpenMP inside of SIESTA has been straight forward because we have used a fine-grain parallelization. In order to guarantee the

workload balance we have used a dynamic scheduling inside of the parallelized loops. This hybrid parallelization will be helpful to adapt the application to many kinds of architectures.

The computation of eigenvalues is the most expensive part of SIESTA. For big problems it is an $O(N^3)$ task, while the construction of the Hamiltonian is just O(N). In order to deal with the problems that we have found in the SCALAPACK library we have tried an iterative method to compute eigenvalues. Iterative methods allow us to take advantage of matrix sparsity, because it does not need to make any transformation to the matrix. In fact, the matrix is only used to compute matrix per vector operations. The other advantage of iterative methods is that we don't need to compute all the eigenvalues and we can stop searching once we have found the desired ones. The iterative methods have 3 critical parts:

- Matrix per vector operation: It represents the main computation part of the method. It's very important to optimize the communications of this function.
- Orthogonalization of a vector: Most of the iterative methods use a base of orthogonal vectors. Every orthogonalization represents a global reduction among all the processes.
- Resolution of the projected system: In order to solve our problem we project it into a smaller one. This means that in every step of the method we should compute the eigenvalues of a smaller problem (using LAPACK). This is done in serial (in fact the computation is replicated in all the systems to avoid communications).

In order to guarantee the scalability of the sparse matrix per vector operation we have used the library METIS. METIS produces a domain decomposition that reduces the size of the border among domains and the number of neighbours. This means that we should do less communications and the data that we should send or receive is lower. We have also precomputed a scheduling in order to overlap communications.

We have implemented two different iterative solvers: Jacobi Davidson (JD) and Modified Lanczos Method (MLM). In the first version of the JD we found serious problems of stability so we decided to move to the MLM. This is much more stable than the other in that we always get good eigenvalues.

3.18.3 *Results*

The new data distributions used during the construction of the Hamiltonian matrix has increased the scalability of the program. The original code had very few communications but the load balancing was not good, especially on those problems where the orbitals were not distributed uniformly in the mesh. This has been solved by using a dynamic mesh distribution that adapts to the orbitals distribution.

Meanwhile, in the MLM, we haven't achieved good results. However, we have done a good work implementing an efficient matrix per vector computation. We have tested this part and we have seen that it can scale to more than 1000 processors. The problem is that with more than 100 processors the sequential part of the MLM becomes relevant.

The problem of MLM is that it needs to create a Krylov space which is bigger than the number of eigenvalues it needs to compute. If N is the size of the problem and M is the krylov space dimension, the complexity of the matrix per vector operation is O(N), while the complexity of the sequential part is $O(M^3)$. This means that MLM is not a good choice to compute a large number of eigenvalues.

3.18.4 *Conclusions*

We have done a good job in the construction of the Hamiltonian matrix and in the matrix per vector operation (used by iterative methods), obtaining good scalability, but we have failed in the selection of the modified Lanczoss method for SIESTA. This method is good when the number of eigenvalues to compute is small. However, the things that we have learnt implementing this method are useful for implementing other iterative methods. We are currently working with the Jacobi-Davidson method (JD). This method has a restart step that reduces the size of the Krylov space. This means that the sequential code doesn't grow with the size of the problem. For the time being we are happy with the scalability of this method, but we are still working on its stability and its convergence. We have tested many SIESTA examples that generate matrices with a bad condition number. This makes work harder for iterative methods.

3.19 QCD

Written by: 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.

3.19.1 *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 and 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 neighbors 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.

3.19.2 *Petascaling techniques*

The QCD benchmark kernels do not include any special petascaling techniques, but are rather implemented in a portable way. However, due to the locality of the solver, only nearest-neighbor communication is needed. These implementations work very well and scale on all target prototypes, see Figure 45 toFigure 47. Note that all the kernels except for C use strong scaling, i.e. a fixed overall problem size; kernel C uses weak scaling, i.e. a fixed problem size per processor.

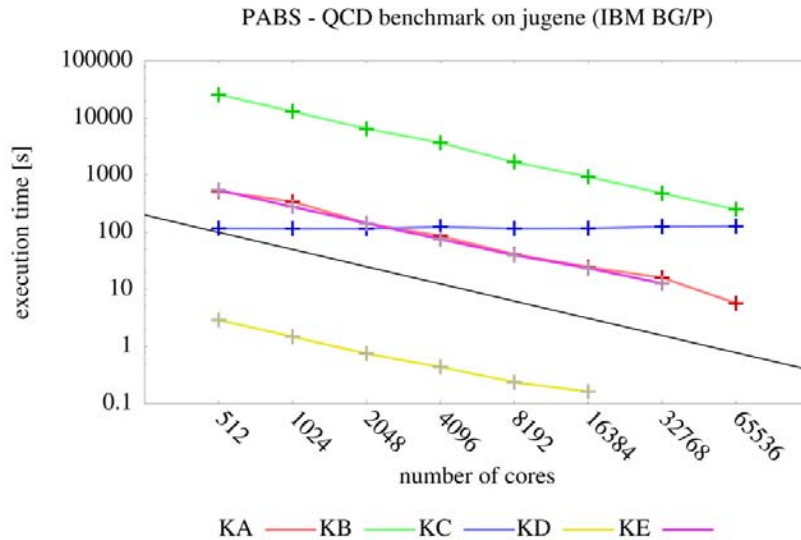

Figure 45: QCD kernel scaling on MPP-BGP. All kernels, but KC, are run for strong scaling.

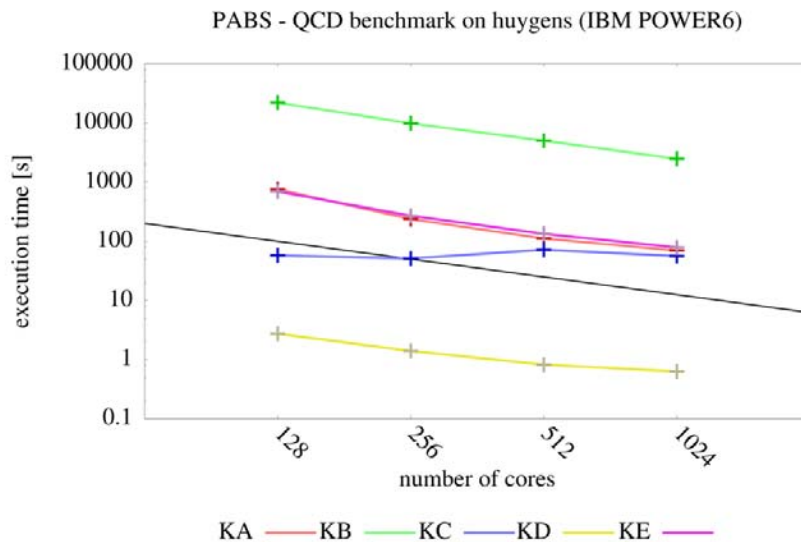

Figure 46: QCD kernel scaling on SMP-FatNode-pwr6. All kernels, but KC, are run for strong scaling.

This communication scheme works especially well on the MPP-BG, as this scheme benefits from the low communication latency. This is true for all kernels. However, to achieve better overall performance, the production codes use in some cases communication techniques which do not use MPI but low level, strongly hardware dependent programming. For instance, on the MPP-BG the network unit can be programmed directly to further reduce latencies and increase the bandwidth. This programming technique depends, at least for the MPP-BG

architecture, strongly on undocumented as well as unsupported hardware interfaces. This implies a very high effort for the implementation and for maintenance, as it is not guaranteed that the interfaces do not change with any OS update. The effort needed (implementation and maintenance) for such a highly specialized version might consume up to 20% of the total, i.e. research, time.

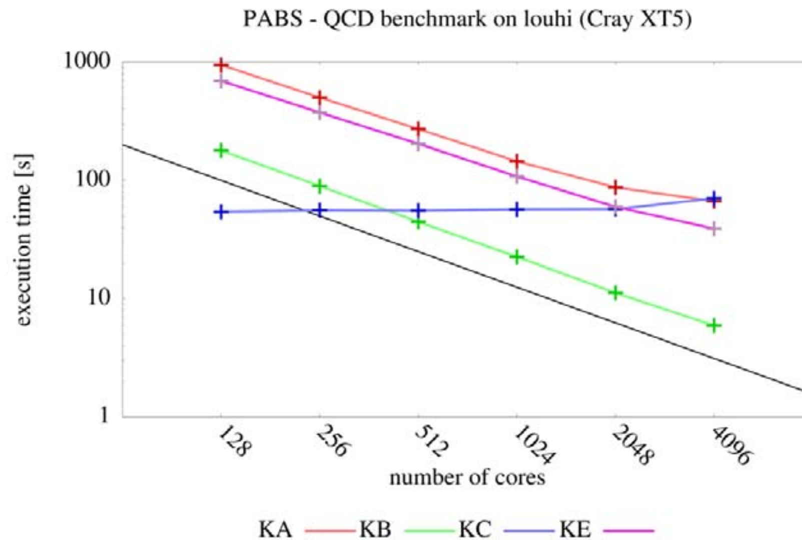

Figure 47 QCD kernel scaling on MPP-Cray. All kernels, but KC, are run for strong scaling

3.19.3 *Results*

No new scaling techniques have been applied to the lattice QCD benchmark, as the benchmark's techniques are already very efficient.

The locality of the involved solvers, i.e. only communication between a few partners, ideally been neighbours in the network geometry, has to be considered, enables the lattice QCD benchmark to scale to very large partition sizes.

The scaling has been already presented in Figure 45 to Figure 47, showing a very good, i.e. nearly linear, scaling behavior. The reason for a non linear speedup at the end of the scaling curves for the SMP-FatNode-pwr6 and the MPP-Cray architectures might reflect the scalability limits of the underlying network.

More details on parallelization can be found for example in [18].

3.19.4 *Conclusions*

The scaling of the lattice QCD benchmark is very good, even for the portable implementations provided for PABS. The main reason for this is the quite regular communication which is restricted only to nearest neighbours. In production codes this communication can be overlapped with computation, as the computational workload for each lattice point is high in the solver phase, i.e. the computational domain, can be split in two parts: the inner and the boundary part. Thus the boundary part is computed first and while the exchange of these boundary parts is in progress the inner part is solved.

The benchmarks provided here show, that a dedicated 4d torus network, like the one used in the MPP-BGP, works best for lattice QCD, as it fits exactly the applications communication scheme.

In general there is no fundamental scaling limit for lattice QCD. However, the lattice size has to be adapted accordingly to the given partition size. The main limiting factor is the scaling property of the computing system.

3.20 Quantum_Espresso

Written by: Carlo Cavazzoni, CINECA

QUANTUM ESPRESSO (QE) is an integrated suite of computer codes for electronic-structure calculations and materials modelling 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 Optimization. It is freely available to researchers around the world under the terms of the GNU General Public License. More information and reference are available at www.quantum-espresso.org

3.20.1 *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 if the core computational kernels (CP and PW) are roughly 50K lines each. The QE distribution is by default self contained, all what you need are a working Fortran and C compiler. Nevertheless it can be linked with most common external libraries, such as fftw, mkl, acml, essl, ScalaPACK and many others. External libraries for FFT and Linear Algebra kernels are necessary to obtain optimal performance. QE contain dedicated drivers for FFTW, ACML, MKL, ESSL, SCSL and SUNPERF FFT specific subroutines.

QE has been developed over time by many researchers who are not necessarily experts in code development and design, anyhow in the last years a lot of effort has been dedicated in making the code more readable and more easily extensible. 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 optimizations and petascaling techniques easy 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 half in FFT subroutines, but this proportion can vary very much with the simulated systems. The time spent for linear algebra is mainly time spent in matrix multiplications. There is no separate pre and post processing steps for the main computational kernels, even if the electronic states, charge density and potential computed with the main codes (CP and PWscf) many post processing computations can be applied to compute all ground state physical properties of a given system. Sometimes this computation can be heavier than the main computation itself.

Both data and computations are distributed in a hierarchical way across available processors, ending up with multiple parallelization levels that can be tuned to the specific application and to the specific architecture. More in detail, the various parallelization levels are geared into a

hierarchy of processor groups, identified by different MPI communicators. In this hierarchy, groups implementing coarser-grained parallel tasks are split into groups implementing finer-grained parallel tasks. The first level is image parallelization, implemented by dividing processors into n image groups, each taking care of one or more images (i.e. a point in the configuration space, used by the NEB method). The second level is pool parallelization, implemented by further dividing each group of processors into npool pools of processors, each taking care of one or more k-points. The third level is plane-wave parallelization, implemented by distributing real- and reciprocal-space grids across the nPW processors of each pool. The final level is task group parallelization, in which processors are divided into ntask task groups of nFFT = nPW/ntask processors, each one taking care of different groups of electron states to be Fourier-transformed, while each FFT is parallelized inside a task group. A further paralellization level, linear-algebra, coexists side-by-side with plane-wave parallelization, i.e. they take care of different sets of operations, with different data distribution. Linear algebra parallelization is implemented both with custom algorithms and using ScaLAPACK; on massively parallel machines this yields superior performance. Recently a first hybrid MPI+OpenMP parallelization has been implemented.

The input is constituted from plain text ASCII files with the description of the simulation and the tables for the atomic pseudo-potentials; the output is plain text for summary information, XML-like files for exchanging data with other packages and binary for wave functions. Binary wave function files are by far the heaviest output of the code and each processor writes its wave function components in a separate file.

For the PRACE benchmark two different datasets have been prepared, both scaling at least up to 20TFlops. The first dataset contains 443 atoms and run roughly in 20 minutes at 10TFlops, the seconds contains 686 atoms and run roughly 1 hour at 10TFlops. Physically the simulated system in both test cases represents an Iridium surface with a Graphene sheet (Graphite monolayer) on top of it. A smaller version (called test version) of the 443 atoms dataset has been used for optimization and profiling. The two dataset have been tailored for the PRACE prototype systems and probably are not big enough for petascaling.

There are a number of real world datasets of scientific and technological interest that are big enough for petascaling. The same datasets described above can be easily extended to build a new input suitable for petascaling. Enlarging the simulated system in this case is scientifically relevant because it can increase the accuracy of the simulation itself. Nevertheless for QE building a new input describing a new system is not an issues at all, starting atomic position is all what is needed.

3.20.2 *Petascaling techniques*

With QE, five main petascaling techniques have been applied: Mixed parallelism useful for all SMP node machines, substitution of blocking communication with non blocking one, vectorization of some key computations for vector machines, more MPI parallelism, and tuning of data distribution parameters to get the best scalability.

The first and most important petascaling techniques has been the implementation of a hybrid MPI & OpenMP version of the code. This effort was motivated by the need to improve the scalability of the code on new petascaling machines with many thousands of computing cores. PRACE in this regards has played an important role to convince the development team of QE that the hybrid parallelization could not be delayed. A lot of work has been done and further details could be found in the chapter 2 when discussing hybrid parallelization as a general technique for petascaling.

The second technique regards the substitution of an MPI alltoall communication in the data transposition contained in the parallel 3DFFT between FFT along z and FFT in the xy plane. Together with this new subroutine we have also introduced the option to remove all barriers in the code. In fact, using barrier could give some performance improvement on some architectures (InfiniBand cluster) but on some other architectures (like XT5) it can decrease the performance. On BG/P MPI barriers in the code have no effect.

The third petascaling technique is more specific for the SX9 architecture. Here some relevant function and loops have been vectorized. In particular we have vectorized all loops calling point functions that were relevant for the performance. In fact QE contains several loops over the real and reciprocal grid where for each point of the grid a function is evaluated. This does not cause problems on scalar architectures since the functions are quite complex and contain a lot of computation. On SX9 instead the loop over the grid destroys the vectorization with a big loss in performance.

The fourth petascaling technique originates from the runs on BG/P with a very high number of cores. These runs have shown that there are some parts of the code that unexpectedly do not scale at all, above a certain number of processors. Here is the explanation for this behavior and the solution we adopted.

Some contributions to the energy and the potential are computed with a loop over the atoms of the simulated system. This loop is parallelized over processors, so that every processor gets a subset of the atoms. This is fine if the number of atoms is larger than the number of processors, on BG/P instead we are always in the condition where the number of processors is much larger than the number of atoms (at least for the PRACE benchmark test case). Then we have to change the loop parallelization so that, when the number of processors is larger than the number of atoms, each atom is assigned to a group of processors. Then the computations required for each atom have been parallelized among the processors of the group. Files affected: flib/distools.f90 PW/paw_init.f90 PW/paw_onecenter.f90.

This technique is completely platform agnostic, but it affect mainly the execution on BG/P because, as explained is the only platform where the number of MPI tasks used is much larger than the number of atoms. Main difficulties were connected to the fact that we had to change the distribution of a given data structure (the atoms), substituting MPI tasks with MPI communicators, and arranging a new algorithm hierarchically distribute computations among communicators and inside communicators.

Finally, as a petascaling technique, it is possible to play with parameters governing data distribution to tune them to specific hardware. QE is written using a hieratical set of MPI communicators, and data can be distributed in different ways among processors changing the relative size of communicators. The parameters to play with are: number of pools, that controls how many MPI tasks should take care of each k-points in reciprocal space; the number of task groups, that controls how many orbitals (electronic wave functions) should be assigned to each task group; the size of diagonalization group that control how many MPI task should take part to the linear algebra subtask; and finally, after the Hybrid parallelization has been implemented, the ration between the number of MPI tasks and OpenMP threads. As can be seen the space of possible combinations is quite large and depending on the architecture and on the input dataset. Here the possibility to give guidelines for each architecture, a good experience of the users and the possibility to do warm up runs to test the different combinations before the production runs are fundamental to use QE in the most efficient way.

The effort required by the implementation of the Hybrid parallelization is roughly 8 PM, the effort for the implementation of a non blocking data transposition and vectorization is 0.5 PM

each, whereas the effort for the tuning of data distribution parameters could be estimated in 0.2 PM for each new run on a new architecture.

With the possibility to run a huge system on petascaling machines, initialization and post process become quite relevant tasks. In QE initialization of computation is parallelized and, even if it does not scale as well as the main iteration loop, usually it is not relevant since it takes a factor of 10-20 less than the rest of the computation. Nevertheless, this is not completely true on all architectures, from the run performed within PRACE some situation emerges where this limitation can become relevant, maybe not in the present generation of machines but probably in the next one. Therefore some work could be planned to improve the parallelization of the initialization. Concerning the post process, QE comes with many tools to give the possibility to the users to compute all quantities related to the Quantum Mechanical ground state of the system. All post processing codes are already parallelized, some of them scale even better than QE, some other scale less. The scalability of the post processing codes depends very much on the kind of computation they perform, but it has to be pointed out that less attention is paid to the parallelization of post processing codes rather than the main codes, so it is possible that with big petascale size simulations some of them can not be used due to limitation in the scalability.

3.20.3 *Results*

Using the hybrid parallelization model (MPI & OpenMP) the gain in terms of speedup is proportional to the number of cores per node there are in a given architecture. Unfortunately, this is not exactly equal to the number of cores per node, at least not on all architectures. There are architectures (like BG/P) where the overhead for thread management is quite high, others (like Power6) where it is negligible. Moreover, since changing the ratio MPI tasks/threads implies a change in the distribution of data, the gain in term of performance is not linear with the number of threads. Most of the time it is a matter of finding the best compromise between the number of MPI tasks and the number of Threads. Finally on thin node clusters, like BG/L, the usage of hybrid parallelism is mandatory for large simulations because of memory limitation, on fat node clusters with many cores per node it is possible to reduce the number of MPI tasks used, thus improving the scalability quite a bit. In fact the scalability of QE for a given simulation is limited by the data distribution among MPI tasks, at a certain point there are no more data to distribute.

The following table shows the performance difference between hybrid and pure MPI code in SMP execution mode on BG/P using the GRIR443_test testcase:

	Hybrid	Pure MPI
nodes used	1024	1024
cores	4096	1024
init_run	126s	230s
electrons	336s	634s
walltime	490s	891s

Table 14 GRIR443 test run using hybrid and pure MPI code

As can be seen there is roughly a factor of two, where in theory there should be a factor of 4. This is due to the overhead of the multithreading but also due to the fact the MPI tasks in the Pure MPI runs are alone in the node and can use all the memory bandwidth.

To better estimate the overhead of using the Hybrid code on BG/P a run with the same number of nodes and the same number of cores has been performed. Here the testcase is AUSURF112, smaller than GRIR443_test, in order to fit into the node memory.

	Hybrid	Pure MPI
cores	256	256
init_run	62s	61s
electrons	808s	681s
walltime	879s	754s

Table 15 AUSURF112 test run using hybrid and pure MPI code

Here Pure MPI code has been run in VN mode and Hybrid code in SMP mode. This is not exactly a measure of the overhead because the data distribution between the two runs are different.

The use of Hybrid programming mode has let QE scale on BG/P up to 65000 cores, with the possibility to scale even further, especially for the large dataset GRIR686, making QE for this dataset scale to the whole machine. Test case GRIR686 (run in SMP mode using hybrid MPI+OpenMP paradigm)

executable parameters	cores	init_run sec.	electrons sec.	walltime sec.
-ntg 8	16384	906	1955	2915
-npool 2 -ntg 8	32768	543	1419	2014
-npool 4 -ntg 8	65536	291	560	1012

Table 16 Test case GRIR686

Note that large discrepancies between computation (init_run+electrons) and job walltime seem related to the load of the I/O subsystem. The number of pools (-npool) is increased in order to use all available memory. When the flag –npool is not used this means that only one pool will be used.

The results in terms of performance improvement using the data transposition with non blocking communications in the 3D FFT and having eliminated the MPI barrier are reported below for Cray XT5 prototype.

Here we have used the input dataset GRIR443 (test version) with the craypat performance analyzer. The execution line is as follows:

```
aprun -n 512 -N 8 ./pw.x+pat -ntg 8 -input grir443.in
```

This results in terms of wall time of the to codes:

```
   Process |  Process |
      Time |    HiMem |
           | (MBytes) |
738.172904 |      817 | MPI_BARRIER and MPI_ALLTOALLV
704.416084 |      817 | no MPI_BARRIER and non blocking transp.
```

Regarding the fourth technique, it mainly affects the performance on BG/P where the number of cores used is much larger than the number of atoms of the system used for the benchmark. Below is the performance gain obtained on BG/P with the modified subroutines.

Test case GRIR443 test, number of cores 4096, execution command:

```
mpirun -exe ./pw.x -mode SMP -np 1024 -verbose 2 -args "-ntg 8 -
input grir443_test"
```

Performance improvement

timing	before	after
walltime	16m 7.18s	9m19.97s
init_run	141.63s	133.99s
electrons	636.47s	399.88s

Finally we present the result of the performance improvement that can be obtained by changing parameter for data distribution. In particular we show here the results using different number of task groups. The use of task groups is required in order to scale when using a number of MPI tasks greater than the radix of the 3D FFT along z, but they can also be used to improve performance or scalability per se, since changing the parameter changes the ratio between how often the data transposition is called and the total amount of data exchanged (e.g. increasing task groups decreases the number of times the transposition is called but increases the data exchanged at each call). Here again, as above we have used the dataset GRIR443 (test version) on XT5 for three values of task group sizes: 4, 8 and 16. From the test, a value of 8 was optimal for 512 cores.

Profile by Function using 4 task groups

```
aprun -n 512 -N 8 ./pw.x+pat -ntg XX -input grir443.in
```

XX is the number of task groups

```
   Process |  Process |
      Time |    HiMem |
           | (MBytes) |
774.952205 |      814 | XX = 4
738.172904 |      817 | XX = 8
808.389620 |      816 | XX = 16
```

Another data distribution parameter that can be tuned is the number of pools. When pools are in use k points in reciprocal space are distributed to pools and in this case the scalability is linear with the number of pools. The drawback is that the use of memory is also linear with the number of pools, so it is not always possible to use a larger number of pools. The number of pools is also limited by the number of k points in the input dataset.

3.20.4 *Conclusions*

QE is a tightly coupled application parallelized using MPI (before the work on hybrid parallelism) relying on global communications (especially MPI_ALLTOALL), where the number of MPI tasks is a critical factor for scalability. In fact to exploit the full power of parallel architectures QE had to be run with one MPI task per core. This ratio is not sustainable any more, since on present and future architectures the number of cores per node is increasing and the memory per core and bandwidth per core are decreasing, making the core too restricted for MPI tasks. OpenMP allow a natural way to distinguish between intra and extra node parallelism. For this reason the parallelization of the data and algorithms in QE has been rewritten to take advantage of this hierarchy.

To implement hybrid parallelism we tried to combine the efficiency of multithreaded libraries available on most architecture with the explicit OpenMP code on the heaviest loops. It is more difficult and perhaps too early in view of the state if various MPI implementations to mix MPI and OpenMP more intimately.

It is important to underline that a hybrid application is also more complex in terms of data distribution so the relation is less obvious between the performances and the number of cores, different combination of tasks/threads may show quite different behaviours, depending on the dataset. With respect to pure MPI applications, more performance tests have to be done to get the best performance for a given dataset.

As a general rule, if the hybrid paradigm has been implemented correctly, we can estimate that with respect to the pure MPI application, the scalability is increased by a factor proportional to the number of cores per node.

This is good for weak scalability but also for strong scalability in those contexts where one does not aim only at simulating larger systems but also to simulate the same system but for a longer time, like it happens in meteo-climate applications and molecular dynamics.

4 Summary

The petscaling of twenty applications has shown that it is not possible to recommend a single set of generic optimizations that is suitable for all codes, there are some which show good scalability for many codes or that should be pursued further in the future. One example is hybrid parallelization, which demonstrated promising results, although surprisingly few of the applications employ this technique fully at the moment. However, many are planning to do so in the future.

Since each application has been ported to several of the prototypes systems, the application evaluation shows the suitability of a given architecture for running different types of codes.

Although many applications have achieved scalability during Task 6.4, well above their original counterpart, it appears that some applications are not suitable for petascaling in their current state. There are several common reasons for this:

One is that the core algorithms do not inherently scale to petaflop/s performance. Rewriting or changing the algorithm seems the only option with inherent parallelism. A common bottleneck is the solver and many report that the Deflated Conjugate Gradient solver is an interesting solver to look into for future petascale optimizations.

A few applications also have problems with large datasets when petascaling. Quadratically or exponentially increasing memory requirement prevent the use of datasets sufficiently big for petascaling. The opposite is also true in that no useful datasets exist which are big enough for petascale runs.

Pre/post-processing also poses a bottleneck for petascaling, since it is done sequentially in some applications. This is usually not a problem for normal runs but in petascale machines the data which needs to be pre/post-processed will increase substantially, which can take more time than the computation itself.

Finally communications patterns also play an important role in petascaling. Some applications contain many synchronization points or global MPI calls and will thus have difficulty scaling further due to the number of nodes in a petaflop/s machine.

In conclusion, the experiences gained during the scaling of the applications on the prototypes will be essential when porting and scaling applications on the PRACE Tier-0 petascale systems. Furthermore, since each application has been ported to multiple prototypes the intricacies of each prototype are better know, which will make it easier to petascale applications in the continuation of the PRACE project.