

CONTRACT NUMBER 508830

**DEISA**

**DISTRIBUTED EUROPEAN INFRASTRUCTURE FOR  
SUPERCOMPUTING APPLICATIONS**

**European Community Sixth Framework Programme**  
RESEARCH INFRASTRUCTURES  
Integrated Infrastructure Initiative

Documentation for D-JRA2-3.2

Deliverable ID: D-JRA2-3.3

Due date: October, 31, 2006

Actual delivery date: October, 31, 2006

Lead contractor for this deliverable: EPCC, University of Edinburgh, UK

Project start date : May 1<sup>st</sup>, 2004

Duration: 4 years

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission	
CO	Confidential, only for members of the consortium (including the Commission Services)	

## Table of Contents

Table of Contents.....	1
1 Introduction.....	3
1.1 Motivation and project goals.....	3
1.2 Intended Audience.....	4
1.3 Outline.....	5
1.4 Conventions.....	6
1.5 Acknowledgements.....	6
1.6 References.....	6
1.7 Document Amendment Procedure.....	8
1.8 List of Acronyms and Abbreviations.....	8
2 Details of the DEISA Platforms Employed.....	9
3 Details of the Code employed.....	9
3.1 The Virgo Simulation Code: FLASH with StB Initial Conditions.....	9
3.2 Porting Process.....	9
4 Profilers Available.....	11
4.1 Profilers on RZG.....	12
4.1.1 Gprof and Xprofiler.....	12
4.1.2 The mpitrace tool.....	12
4.2 Profilers on BSC.....	13
4.2.1 The gprof tool.....	13
4.2.2 mpitrace and Paraver.....	13
4.3 Profilers on HLRS.....	15
4.3.1 Ftrace, prof and MPIPROGINF.....	15
4.4 Discussion of available profilers.....	16
5 Profiling Study for the Virgo Simulation Code.....	16
5.1 Potential Optimization Strategies.....	23
5.2 Summary of Findings.....	25
6 The Improved Virgo Simulation Code: FLASH with FFTW.....	25
6.1 Porting the Improved Virgo Simulation Code.....	25
6.1.1 Porting to HPCx.....	26
6.1.2 Porting to RZG.....	26
6.1.3 Porting to BSC.....	27
6.1.4 Porting to HLRS.....	27
6.1.5 Validation of the Results.....	27
6.2 Conclusions.....	28
7 Scaling Study.....	28
8 Profiling Study for the Improved Virgo Simulation Code.....	31
9 Code Migration using the DEISA Infrastructure.....	37
9.1 Overview.....	37
9.2 UNICORE Workflow.....	38
9.3 DESHL.....	39
10 Memory Usage.....	40
10.1 Overview.....	40
10.2 Estimating the memory usage.....	41
10.3 Memory Profiling Tools and Strategies.....	43
10.3.1 Using vmstat.....	43
10.3.2 Using half the number of processors available in a node.....	44
10.3.3 Using ulimit.....	45
10.3.4 Using libhmd.a.....	45
10.3.5 Summary of Findings.....	49
11 Discussion of Results and Future Work.....	50
12 Appendix: General Machine Information.....	53
12.1 HPCx.....	53

---

12.2	SARA.....	53
12.3	IDRIS.....	53
12.4	RZG.....	54
12.5	BSC.....	54
12.6	HLRS.....	54
12.7	ICC.....	55
13	Appendix: Machine specific scripts.....	55
13.1	HPCx.....	55
13.2	IDRIS.....	57
13.3	RZG.....	57
13.4	BSC.....	58
13.5	HLRS.....	59
14	Appendix: Compiler Flag Specifics.....	60
14.1	IBM-specific compiler flags.....	60
14.2	SUN-specific compiler flags.....	61
14.3	Altix specific compiler flags.....	61
14.4	NEC specific compiler flags.....	61
15	Appendix: Porting the Virgo Simulation Code to HLRS.....	62
16	Appendix: Implementation of the New Algorithm.....	63
16.1	The FFT Implementation.....	65
17	Appendix: Makefile.h for Various Platforms.....	66
17.1	Makefile for RZG.....	66
17.2	Makefile for BSC.....	67
17.3	Makefile for HPCx.....	68

## 1 Introduction

This documentation constitutes deliverable D-JRA2-3.3. This consists of documentation for an optimised version of a cosmological simulation code based on FLASH [10] which has been ported onto the DEISA infrastructure. The code has been adapted to include the ability to migrate between DEISA platforms during the course of a simulation. The report describes this code and also includes a description of the work carried out from May to October 2006. This report continues from the Specification Document D-JRA2-3.1 [4], submitted on the 30<sup>th</sup> April 2006, describing the work done leading up to this second phase of activity. The optimised code and the code migration scripts form the deliverable D-JRA2-3.2. Due to restrictions in the distribution of the FLASH code this modified version, now specific to Virgo, will not be a public deliverable and will be passed directly to the Virgo Consortium. However, all the processes undertaken to produce this modified version of the code and the code migration are documented in this report. We expect that our optimisations will be included in future releases of the publicly available FLASH.

### 1.1 Motivation and project goals

The main aim of this work package consisted of porting FLASH, a highly modular, Fortran90, parallel Adaptive Mesh Refinement (AMR) code, used by members of the Virgo Consortium to perform cosmological simulations [36], to efficiently utilise the various platforms available within the DEISA infrastructure.

FLASH can be used to model hydrodynamical and/or dark matter simulations by appropriately choosing the modules to be compiled into the code (see Section 3 in [4]). To ensure good performance on the DEISA infrastructure, the code was profiled in order to find out whether any optimisations could be performed on the code. For this task, we only used dark matter simulations using the code provided by Virgo (already introduced in [4]). This version of FLASH, henceforth referred to as the Virgo Simulation code, was adapted by the Virgo Consortium to read in the initial conditions for the dark matter simulations being used and acted as the base-line against which any code changes leading to performance and scalability improvements were compared with.

The Virgo Simulation code was based on FLASH version 2.5. The so-called 64k, 128k and 256k Santa Barbara (StB) data sets were used as initial conditions [12]. The data set names reflect the number of particles employed in the simulation, specifically  $64^3$ ,  $128^3$  and  $256^3$ , respectively. These data sets are a typical type of run that the Virgo Consortium may want to do. However, the 256k data set was made available but not used in the scaling and profiling studies as this data set proved to be too memory demanding for the platforms used resulting in run time crashes (calls to abort)<sup>1</sup>. This motivated the memory usage investigation reported in Section 9.

The Virgo Simulation code was ported and profiled on: HPCx, RZG, BSC and HLRS<sup>2</sup>; the methods and results for this are described in Section 5 with details about the port to HLRS described in Appendix 15. The multigrid-based algorithm used to solve the Poisson equation employed in the original FLASH code was replaced by a Fast Fourier Transforms (FFT) based algorithm to speed-up this computationally

---

<sup>1</sup> In addition, the run times required would have been too long – in the order of days - to run to completion.

<sup>2</sup> It is important to note that all the work carried out on at HLRS was not performed by DEISA funded staff. All results obtained on HLRS mentioned in this report were part of an MSc thesis [27].

demanding part of the code [34]. This new code, which we refer to as the Improved Virgo Simulation henceforth, was also ported to HPCx, RZG, BSC and HLRS; the process involved is described in Section 7. Scaling tests and profile runs were performed on RZG, BSC and HLRS; the scaling results are described in Section 7 and the profiling results in Section 8. The code was not profiled on HPCx due to the memory issues, described in [4], when profiling the Virgo Simulation for the 128k StB data set. Similarly, the IDRIS platform was not used at all as we were confronted with the same type of memory problem as experienced on HPCx.

Initially, we were to investigate the possibility of introducing multi-time-stepping into the code but, as has already been mentioned, during the porting process it was found that the code required more memory than expected on platforms such as HPCx. Thus, it was agreed to replace this planned task with an investigation into memory use on HPCx. This is reported in Section 9.

A follow-up investigation of the incorrect results produced on SARA, described in [4], did not resolve the issue. See Section 3.2 for more details.

To summarise, the following points have been investigated, either by us or R.H. Ostrowski [27], and are described in the remainder of this document over and above those already mentioned in the D-JR2-3.1 document [4]:

1. The Virgo Simulation code was ported to the HLRS platform [27].
2. Neither the Virgo Simulation code nor the Improved Simulation Code have been successfully ported to SARA – the codes run but produced incorrect results.
3. The 128k StB data set was employed on all platforms for the profiling and optimisation runs.
4. The simulation was profiled in detail on RZG, BSC and HLRS (the latter by [27]), in order to identify subroutines for subsequent optimization.
5. Platform-independent optimizations were introduced on the most computationally expensive routines by changing the algorithm used to solve the Poisson equation.
6. Platform-dependent optimizations were investigated on the HLRS platform [27].
7. The memory requirements of the code were investigated on HPCx using the 256k StB data set.

All results obtained on HLRS mentioned in this report were part of an MSc thesis [27]. This work was the first time that FLASH had been ported to an NEC vector supercomputer. The performance of FLASH on the HLRS platform was investigated and some automated and manual optimisations were introduced. The latter included the replacement of the existing multigrid-based Poisson solver algorithm with an FFT-based solver, which was investigated on the other DEISA machines.

Lastly, allowing simulations to migrate between DEISA platforms is a very important feature. Not only for large astrophysical simulations, but for any large simulations which require more time to run than any DEISA site permits within a single batch job. Here we have successfully utilised the DEISA infrastructure to allow long simulations to run to completion, seamlessly utilising several of the DEISA platforms if required.

## **1.2 Intended Audience**

The intended audience for this document are:

- The Virgo Consortium and other researchers that use FLASH who are, in essence, the users that apply this code to solve real scientific problems. It will benefit them to see how the code was ported on to these platforms and how it

scales and performs. In addition, we demonstrate the major bottlenecks in the code which we have tried to remove as well as providing other generic performance improvements. Using the types of architectures found in the DEISA infrastructure efficiently is a major goal for the type of computationally expensive and potentially long-running cosmological simulations that the Virgo Consortium wish to run.

- The FLASH developers. The usage of a variety of different machine architectures, including one which FLASH has never been ported to namely the NEC [5] platform at HLRS, helps to highlight any code portability issues and informs the developers about the way the code performs across these systems with the possibility of folding some of these changes into the main code base.
- The administrators and users of the DEISA sites on to which we port the code to. The porting of a major application, such as FLASH, requires common libraries, such as HDF5, to be available across the DEISA infrastructure. The process of porting allows us to inform the site administrators about any issues, problems or challenges related to the use, installation or availability across the DEISA infrastructure.

### 1.3 Outline

In the remainder of this document Section 2 summarises the DEISA platforms that have been used in this work package. Section 3 describes the general procedure of how to port the code and how the code was locally adapted for the platform at HLRS [27] (also see Appendix 15 for more technical details). The porting process onto HPCx, IDRIS, RZG and BSC was already presented in Section 4 of [4]. Section 4 in the present report lists the various profilers employed to investigate the code on RZG, BSC and HLRS. The major features of the profilers are discussed as well and instructions on how to use them are given. The result of the profiling study on these three machines is shown in Section 5. Section 5.1 discusses various possible optimization strategies for the Virgo Simulation code based on the outcome of the initial profiling study performed.

In Section 6 we introduce the Improved Virgo Simulation code which represents a possible optimization of the Virgo Simulation code. In this case the most time intensive algorithm, the Poisson solver based on a multigrid method, is replaced by a Poisson solver based on an FFT. The performance of the two codes investigated is clearly shown in the scaling study in Section 7. This Section is followed by another profiling study, now of the Improved Virgo Simulation code, to allow direct comparisons between the code with the old and new algorithms.

Code migration has been successfully implemented utilising the DEISA infrastructure, utilising either the UNICORE Workflow or the DESHL. This work is described in detail in Section 9.

Memory usage turned out to be an issue for large StB input data sets on HPCx. We show an investigation of memory usage of the Improved Virgo Simulation code on HPCx in Section 10.

We conclude this report with a discussion of the results and recommended future work in Section 11.

Appendices 12 to 14 recapitulate platform specific information which has already been provided in [4]. More details on how the original FLASH code and the Virgo Simulation code were ported, mainly onto HLRS as done by [28], can be found in

Appendix 15. We provide background information about the new algorithm to solve the Poisson equation in Appendix 16. Appendix 17 shows the `Makefile.h` for RZG, BSC and HPCx as used for the Improved Virgo Simulation code.

## 1.4 Conventions

In this document the following typographical conventions are used.

Text appearing in a blue box:

```
Indicates a setup script, profiler output, a compilation or link line.
```

While text in a yellow box:

```
Indicates the contents of a batch script.
```

Text in a `Courier` font indicates a compiler flag, file name, directory name, command line usage, tools or variable names.

## 1.5 Acknowledgements

The software used in this work was in part developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

We thank the Virgo Consortium for our ongoing collaboration. In particular, we thank Tom Theuns for his input to the success of this project. Tom wrote the subroutines responsible for handling the reading of the Santa Barbara [12] input data and the implementation of the new algorithm to solve the Poisson equation. We also thank John Helly for providing the HDF5 wrapper. Thanks are also due to Craig Booth, Richard Bowers and Carlos Frenk.

We thank Radoslaw Ostrowski whose work done for his MSc dissertation greatly added to this project. And we thank Stefan Haberhauer (NEC) and David Vicente (BSC) for their help with the porting of the code to these platforms.

Judit Gimenez (BSC) was especially helpful in interpreting our data on BSC with `Paraver`.

We thank Dan Sheeler, Paul Ricker and Anshu Dubey from the FLASH center for their guidance and support.

## 1.6 References

- [1] ASC / Alliances Center for Astrophysical Thermonuclear Flashes, [flash.uchicago.edu](http://flash.uchicago.edu).
- [2] Aster batch and job scheduling [www.sara.nl/userinfo/aster/usage/batch/index.html](http://www.sara.nl/userinfo/aster/usage/batch/index.html).
- [3] Barcelona Supercomputing Center, [www.bsc.es/index.en.html](http://www.bsc.es/index.en.html).
- [4] E. Breitmoser, G. Pringle and M. Antonioletti, Specifications Document for D-JRA2-3.1, EPCC, University of Edinburgh, UK, 2006.
- [5] DEISA Primer, [www.deisa.org.userscorner/primer.php](http://www.deisa.org.userscorner/primer.php).
- [6] Anshu Dubey, FLASH Center, Private Communication.

- 
- [7] European Centre for Parallelism of Barcelona, [www.cepba.upc.es](http://www.cepba.upc.es).
- [8] FFTE: A Fast Fourier Transform Package, [www.ffte.jp](http://www.ffte.jp).
- [9] FFTW, [www.fftw.org](http://www.fftw.org).
- [10] FLASH Code User's guide,  
[http://flash.uchicago.edu/website/codesupport/users\\_guide/home.py](http://flash.uchicago.edu/website/codesupport/users_guide/home.py).
- [11] FLASH user mailing list,  
[http://flash.uchicago.edu/website/maillinglists/home.py?submit=public\\_lists/flash-users/0802.html](http://flash.uchicago.edu/website/maillinglists/home.py?submit=public_lists/flash-users/0802.html).
- [12] Frenk, C. S., et al., The Astrophysical Journal, 525:554-582, 1999.
- [13] Judit Gimenez, CEPBA, Private Communication.
- [14] High Performance Computing Center Stuttgart,  
[www.hlrs.de/hw-access/platforms/sx8](http://www.hlrs.de/hw-access/platforms/sx8).
- [15] HPCx Service, [www.hpcx.ac.uk](http://www.hpcx.ac.uk).
- [16] HPCx FAQs, [www.hpcx.ac.uk/support/FAQ/stack.txt](http://www.hpcx.ac.uk/support/FAQ/stack.txt)
- [17] HPCx Users' Guide (v2.02),  
[www.hpcx.ac.uk/support/documentation/UserGuide/HPCxuser/HPCxuser.html](http://www.hpcx.ac.uk/support/documentation/UserGuide/HPCxuser/HPCxuser.html).
- [18] John Helly, Virgo Consortium, Private Communication.
- [19] Hydra\_MPI, [www.epcc.ed.ac.uk/computing/services/cray\\_service/t3e/virgo](http://www.epcc.ed.ac.uk/computing/services/cray_service/t3e/virgo)
- [20] IDL, The Data Visualization & analysis Platform, <http://ittvis.com/idl>.
- [21] Institut du Developpement et des Ressources en Informatique Scientifique,  
[www.idris.fr](http://www.idris.fr).
- [22] Institute for Computational Cosmology, Introduction to the Cosmology Machine COSMA, [www.icc.dur.ac.uk/Computing/cosma-new/cosma-new.html](http://www.icc.dur.ac.uk/Computing/cosma-new/cosma-new.html).
- [23] Max-Planck-Gesellschaft Rechenzentrum Garching der Max-Planck-Gesellschaft und des IPP, [www.rzg.mpg.de/computing](http://www.rzg.mpg.de/computing).
- [24] NCSA HDF Home Page, [hdf.ncsa.uiuc.edu](http://hdf.ncsa.uiuc.edu).
- [25] NEC SX-8, Man pages –fft, MathKeisan Fast Fourier Transforms.
- [26] NEC, SUPER-UX Performance Tuning Guide,  
[www.hlrs.de/hw-access/platforms/sx8/doku/new/PTG-1.0.pdf](http://www.hlrs.de/hw-access/platforms/sx8/doku/new/PTG-1.0.pdf).
- [27] Radoslaw Hubert Ostrowski, Porting, Profiling and Optimising an AMR Cosmology Simulation, MSc in High performance Computing, The University of Edinburgh, 2006
- [28] Radoslaw Ostrowski, Private Communication
- [29] Parallel Program Visualization and Analysis Tool,  
[www.cepba.upc.es/paraver/](http://www.cepba.upc.es/paraver/).
- [30] Rechenzentrum Garching, [www.rzg.mpg.de/computing/IBM\\_P/profiling.html](http://www.rzg.mpg.de/computing/IBM_P/profiling.html)
- [31] Paul Ricker, FLASH Center, Private Communication.
- [32] SARA, Description of the SGI Altix 3700 Aster system,  
[www.sara.nl/userinfo/aster/description/index.html](http://www.sara.nl/userinfo/aster/description/index.html).
- [33] Sedov L. I. 1959, Similarity and Dimensional Methods in Mechanics (New York: Academic).
- [34] Tom Theuns, Virgo Consortium, Private Communication.
- [35] Van der Linden, P., Expert C Programming, SunSoft Press, 1994
- [36] Virgo Consortium, [www.virgo.dur.ac.uk](http://www.virgo.dur.ac.uk).
- [37] UNICORE, [www.unicore.org](http://www.unicore.org).
- [38] Using Memory Debugging Routines for XL Fortran,  
[http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?topic=co\\_m.ibm.xlf91a.doc/xlfug/heapdbg.htm](http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?topic=co_m.ibm.xlf91a.doc/xlfug/heapdbg.htm).

[39] Using Memory Debugging Routines on HPCx,  
[www.hpcx.ac.uk/support/FAQ/mem\\_man.html](http://www.hpcx.ac.uk/support/FAQ/mem_man.html).

[40] xprofiler man pages on psi.rzg.mpg.de.

## **1.7 Document Amendment Procedure**

Intentionally left blank.

## **1.8 List of Acronyms and Abbreviations**

<b>AMR</b>	<b>A</b> daptive <b>M</b> esh <b>R</b> efinement
<b>BSC</b>	<b>B</b> arcelona <b>S</b> upercomputing <b>C</b> enter
<b>FFT</b>	<b>F</b> ast <b>F</b> ourier <b>T</b> ransform
<b>FFTW</b>	<b>F</b> astest <b>F</b> ourier <b>T</b> ransform in the <b>W</b> est
<b>HDF5</b>	<b>H</b> ierarchical <b>D</b> ata <b>F</b> ormat <b>5</b>
<b>HLRS</b>	<b>H</b> och <b>L</b> eistung <b>S</b> Rechenzentrum <b>S</b> tuttgart
<b>ICC</b>	<b>I</b> nstitute for <b>C</b> omputational <b>C</b> osmology
<b>IDRIS</b>	<b>I</b> nstitut du <b>D</b> eveloppement et des <b>R</b> essources en <b>I</b> nformatique <b>S</b> cientifique
<b>RZG</b>	<b>R</b> echen <b>Z</b> entrum <b>G</b> arching
<b>SARA</b>	<b>S</b> tichting <b>A</b> cademisch <b>R</b> ekencentrum <b>A</b> msterdam
<b>StB</b>	<b>S</b> anta <b>B</b> arbara

## 2 Details of the DEISA Platforms Employed

The following machines were available as part of the DEISA infrastructure plus a Sun Linux cluster cluster, Quintor, operated by the Virgo Consortium at the ICC [22], which is not part of DEISA. The DEISA machines used for this study were:

- The IBM at HPCx (hpcx), see Appendix 12.1 for more details.
- The SGI at SARA (Aster), see Appendix 12.2 for more details.
- The IBM at IDRIS (zahir), see Appendix 12.3 for more details.
- The IBM at RZG (psi), see Appendix 12.4 for more details.
- The IBM at BSC (MareNostrum), see Appendix 12.5 for more details.
- The NEC at HLRS (SX-8/576M72), see Appendix 12.6 for more details.
- The Sun at ICC (Quintor), see Appendix 12.7 for more details.

The general characteristics of these machines as present on April 2006 are summarized in Table 1.

Owner Platform	HPCx HPCx	SARA Aster	IDRIS zahir	RZG Psi	BSC MareNostrum	HLRS NEC SX-8	ICC Quintor
Location	Daresbury	Amsterdam	Paris	Garching	Barcelona	Stuttgart	Durham
No. of CPUs	1536	415	1024	812	4564	576	518
Computer type	IBM pwr5	SGI Altix 3700 Linux	IBM pwr4, pwr4+	IBM pwr4	IBM Power PC970 Linux	NEC SX8	SunFire V210SunFire
Peak performance [Tflops]	9.2	2.2	6.55	4.2	40	12.67	1
Interconnect type	IBM HPS	Shared memory (ccNUMA)	IBM HPS	IBM HPS	Myrinet & Gigabit Ethernet	IXS 16 GB/s per node	Gigabit Ethernet
Memory per CPU	2Gb	2Gb	1.5Gb-4.2Gb	2Gb-8Gb	2Gb	~124 Gb per node (8 CPUs)	2Gb

**Table 1: The DEISA platforms employed, plus ICC.**

For the sake of simplicity, we hereafter refer to each platform by the name of the institution that owns it, thus Quintor is referred to as ICC, etc.

## 3 Details of the Code employed

### 3.1 The Virgo Simulation Code: FLASH with StB Initial Conditions

The Virgo Simulation code is the FLASH 2.5 code, extended by Virgo to read in HDF5 files and adapted to run a particular dark matter simulation, using the Santa Barbara (StB) data set [12]. We described the Virgo Simulation code and the porting process onto: HPCx, IDRIS, RZG and BSC in Section 4 of [4]. Since this report, the porting of the Virgo Simulation code has been successfully extended onto HLRS [27]. A detailed description of this process, taken from [27], can be found in Appendix 15.

### 3.2 Porting Process

For the porting process we always used one of the following codes as our starting point:

- The standard FLASH 2.5 distribution tar file, as downloaded from the FLASH website [10] to test the hydrodynamical case, as described in Section 3 of [4] to check for any basic potential porting issues.

- The Virgo Simulation code tar file, which includes a dark matter only simulation and can read HDF5 data input files, as extended by the Virgo Consortium [34],
- The Improved Virgo Simulation code as received from [34]. This is basically the same code as the Virgo Simulation, however, the multigrid based algorithm to solve the Poisson equation is replaced by an FFT algorithm.

This test was run at: HPCx, IDRIS, RZG, BSC and SARA [4] and now runs at HLRS [27]. The Virgo Simulation code was run at: HPCx, IDRIS, RZG and BSC for a scaling study (see [4]) and has now been extended by runs on HLRS [27].

The gas-only simulation, namely the sedov problem, has ran successfully on SARA [4], however, neither the Virgo Simulation nor the Improved Virgo Simulation (see Section 6) produce the correct results, in that, the simulation converges to an unrealistic Universe.

Further intensive investigation of results produced at SARA, which were significantly different from those produced on all the other platforms [4], did not resolve the underlying problem. This meant that the process of porting the simulation on to the SGI at SARA was not completed as, after extensive investigations hampered by an unstable code base and a restrictive working environment<sup>3</sup>, the code still did not produce the correct results. To allow for a quick turn-around and ease in debugging, we reduced the complexity of the simulation being performed on both HPCx and SARA to single-processor runs of the Improved Virgo Simulation (see Section 6) using the 64k StB dataset and employed print statements.

It was found that the FLASH setup python script generates different forms of the file `dbase_defines.fh` on HPCx and SARA which, in turn, makes the layout of variables in memory quite different; however, we are confident that this is not the source of the incorrect results produced at SARA.

The results on HPCx and SARA diverge within the routine `init_flash` which, as the simulation progresses, produces `NaNs` (Not a Number) on SARA. It appears that the fault may lie in the installation of the FFTW library, as the library returns different values depending at SARA, however, this is only a conjecture at this time. It should also be noted that the version of HDF5 on SARA is v1.4.4p, whilst on HPCx it is v1.4.4 and these two libraries are currently incompatible. Specifically, an HDF5 created on HPCx is not readable at SARA. However, this may be ameliorated if the most up-to-date version of HDF5 is employed. Unfortunately, due to lack of time, we were unable to determine the source of the difference.

The Improved Virgo Simulation code was run on: HPCx, RZG and BSC for the scaling study. The scaling study was also run at HLRS [27]. IDRIS was not included this time as this platform proved to be very similar in behaviour to HPCx particularly in terms of its memory management and available batch queues, therefore we did not expect to gain any additional insight from employing IDRIS. The results on SARA for the Virgo Simulation code are not correct thus the Improved Virgo Simulation code was not ported to SARA.

The overall steps needed to install a code, i.e. the standard FLASH 2.5 code, the Virgo Simulation code and the Improved Virgo Simulation code, on to a DEISA platform were:

1. Install HDF5 version 1.4.4. We did not use the latest version of HDF5 as version 1.4.4 was used on the initial platform, ICC, where the Virgo Simulation code was

---

<sup>3</sup> SARA could only be accessed via the UNICORE ssh-plugin, which made debugging more difficult than on the other platforms, i.e. debugging had to be done using print statements as debugger GUIs failed to function over this connection.

originally run. As a default FLASH uses serial HDF5 for the output, as opposed to the parallel version available. We therefore also use serial HDF5 for the I/O to obtain all results presented throughout this report

2. Install the HDF5 wrapper for FLASH [18] for the Virgo Simulation code and the Improved Virgo Simulation code in order to read in the StB initial conditions. This is not always straight-forward as this code requires modifications for the different platforms – where the problems were mainly Fortran code calling C routines,
3. Run the FLASH supplied Python setup script to compile the corresponding modules (i.e. the Sedov setup, the Virgo Simulation code setup or the Improved Virgo Simulation code setup),
4. Adapt the `Makefile.h` to use the machine specific compilers, flags and library paths,
5. Adapt the code for a new platform if necessary (see Appendix 15),
6. Compile and link the code via `gmake`,
7. Create a batch script for the machine being used,
8. Choose a StB data set input file with a given number of particles; adapt parameters in the `flash.par` input file (not required for Sedov setup). The `flash.par` input file contains two parameters, `lrefine_min` and `lrefine_max`, which set the minimum and maximum level of refinements that the user allows for the code. For the Improved Virgo Simulation code there is an additional parameter, `pm_level`. We always choose `pm_level=lrefine_max`. In general, the smaller value of `pm_level` and `lrefine_max` determines the size of the FFT grid (typical values can be found in Section 7 in [4]). For the porting process we chose the smallest data set and parameters which allow a short run time of minutes only as provided by [34],
9. Validate results (via the `sfocu` tool -see [4] for details - for the Sedov problem, the `h5diff` tool [24] or an IDL script written by [34] (as applied in Section 6.1.5 and described in detail in [27]). We assume that the code has been ported correctly if these tools report an acceptable error.

We use 64-bit addressing throughout as it increases the amount of memory that a program can use. On IBM Regatta platforms, this also removes some of the restrictions on shared memory segments, which allows the 64-bit MPI library to include additional optimisations.

## 4 Profilers Available

In this report we use the `gprof` and `Xprofiler` tools on RZG, the `gprof` tool on BSC and the `mpitrace` library on RZG and BSC. Note that the `mpitrace` tool on BSC is not the same as the `mpitrace` tool available as it requires the Parallel Program Visualization and Analysis Tool `Paraver` to inspect the results. On HLRS, the profiling tools `ftrace`, `prof` and `MPIPROGINF` are used [27].

In addition, the FLASH code itself includes a routine, `perfmon.F90`, which allows the execution times for sections of the code to be generated. The user can determine the granularity of these sections by modifying the code to add timers where required. When a FLASH simulation runs to completion a profile is attached to the end of the log file that is produced whenever FLASH runs. This displays time consumed for the sections of the code that have been modified. This internal profiler can be switched off by setting the `NOOP` pre-processing directive within `perfmon.F90`. Consequently all related subroutines will simply return without executing any further. However, when this internal profiling was used the following errors were produced in the routine `h5_write.c`:

- Error writing `perfmon` timing data to data file,

- Error writing elapsed time attribute,
- Error writing number evolved zones attribute.

A consequence of the occurrence of any of these errors is that the routine `abort_flash` is called. To prevent this, the corresponding abort calls were removed. Note that the `NOOP` variable and its function are not mentioned in the FLASH User Guide [10].

More information on each of these profilers is given in the following subsections.

## 4.1 Profilers on RZG

### 4.1.1 *Gprof and Xprofiler*

To use `gprof` the code must be compiled with the compiler flags `-pg -g`, in conjunction with any level of optimisation. In this case, we employ `-O3`.

The `gprof` output is stored in `gmon` files produced at the end of the execution of the programme being profiled. Generally one file per processor will be produced. When `gprof` is run on these files output will be produced which shows the exclusive time (as a percentage of the total run time) taken by the subroutine – that is to say that it does not include time spent in the subroutine's descendents.

The `gprof` output provides the following information:

- `%time`: The percentage of the total running time of the program used by this function (self seconds/total run time).
- `cumulative seconds`: A running sum of the number of seconds accounted for by this function and those listed above it.
- `self seconds`: The number of seconds accounted for by this function alone. This is the column use to sort for the output file produced.
- `Calls`: The number of times this function was invoked, if this function is profiled, else blank.
- `self s/call`: The average number of seconds spent in this function per call, if this function is profiled, else blank.
- `total s/call`: The average number of seconds spent in this function and its descendents per call, if this function is profiled, else blank.
- `name`: The name of the function

The `gprof` tool also allows the user to get a line-by-line breakdown of the code with indications of where exactly most of the computing time is being spent.

The output includes time spent in MPI routines.

The `xprofiler` command invokes `Xprofiler`, a GUI-based AIX performance profiling tool. `Xprofiler` is used to analyze the performance of both serial and parallel applications. `Xprofiler` uses data collected by the `-pg` compiling option and presents a graphical representation of the functions in the application in addition to providing textual data in several report windows.

### 4.1.2 *The mpitrace tool*

The main reason for using the `libmpitrace.a` library [30] is to provide a very low overhead elapsed-time measurement of MPI routines for applications written in any of FORTRAN, C or C++. The overhead for the version on the IBM systems used at RZG, at the time of writing, is about 1 microsecond per call.

In order to use the `mpitrace` library it needs to be added at the linking stage of the compilation process. Hence, we have the additional line in `Makefile.h` on RZG:

```
LIB_OPT = -L/afs/rzg/@sys/lib -lmpitrace
```

However, problems were encountered when running the code that was linked to the `mpitrace` library and also enabled for `gprof`. In this case the code would abort after the first time step was reached. An error message was generated from the routine `perfmon.F90` which stated that it had run out of space on the timer call stack. Since profiling is enabled there are additional memory requirements which exceeded the available memory on HPCx for the 128k StB data set as reported in [4]. One work-around is to turn off all routines related to `perfmon.F90`.

## 4.2 Profilers on BSC

### 4.2.1 The `gprof` tool

The profiler `gprof` is also available on BSC, version 2.15.90.0.1.1. As before, the code must be compiled with the `-pg -g` flags. However, the optimization level needs to be lowered from our default of `-O3` to `-O2` otherwise the code stops before starting the calculation without producing any error message. Unlike RZG, the output of a `gprof` enabled code only produces a single `gmon` file by default.

In order to get a separate `gmon` output files per processors on BSC the environment variable `GMON_OUT_PREFIX` needs to be set:

```
export GMON_OUT_PREFIX=exe_name
```

This environment variable must be defined for all processes, so the export line can to be included in the `$HOME/.bashrc` file or set in a separate script which is launched by the `mpirun` command. The first option was used.

Note that the MPICH-GM libraries installed on BSC do not have a `gprof` version. For this reason the time spent within MPI routines do not explicitly appear in the `gprof` call graph.

### 4.2.2 `mpitrace` and `Paraver`

On the BSC platform there are no MPI profiling tools like those available for the Regatta servers, like the `mpitrace` library as yet. On the other hand, the `Paraver` tool [29], being developed by the European Centre for Parallelism of Barcelona [7], can be used to obtain an MPI-communications profile.

To do this, the code requires to be linked with a different `mpitrace` library (version 1.0). It is preferable to use a static over a dynamic library as the code will not then have to search for dependencies when running. This is more time efficient.

By default, MPI tracing data is not collected. To collect this data, one sets

```
export MPITRACE_ON=1
```

in the `$HOME/.bashrc` file or in the batch script. If the variables are only exported in the LoadLeveler script, the master process will be the only process with the right

environment variable and all the other process will fail when trying to load the profiling library.

To link with the static library remove the -L flag, one is required to treat the library as an object, for example:

```
mpif90 -O3 /gpfs/apps/CEPBATTOOLS/lib/64/libmpitracef.a mpi_test.o -o  
mpi_test
```

The resulting output files, namely \*.mpits, are only intermediate trace files and need to be transformed by the tool mpi2prv to then be readable by Paraver. This is done by running:

```
/gpfs/apps/CEPBATTOOLS/bin mpi2prv -syn -f *.mpits -o output.prv
```

This produces two files: output.prv and output.pcf.

In the resultant default trace obtained there are no references to the user routines. Currently there are two ways of including user functions information in the Paraver tracings on BSC. We chose to include references to the routines that call MPI library functions. For this purpose the code needs to be compiled with the debug compiler flag -g to be able to extract the names of the routines with the file and line number information from where it is called. In the batch script the environment variable MPITRACE\_MPI\_CALLER needs to be exported. This approach only includes details on the routines that call MPI:

```
export MPITRACE_MPI_CALLER=1-3
```

so we get 3 levels of callers that will provide more details than just 2 or 1. Once the code is run the resulting profile output files \*.mpits need to be merged. This is done by running:

```
/gpfs/apps/CEPBATTOOLS/bin/mpi2prv -syn -f *.mpits -e ./flash2 -o  
output.prv.gz
```

Then Paraver (version 3.4) can be launched:

```
$PARAVER_HOME/bin/paraver output.prv
```

An initial view of the run can be seen in Figure 1. This is a high-level view – it is possible to zoom in to view the process interactions at much higher resolution.

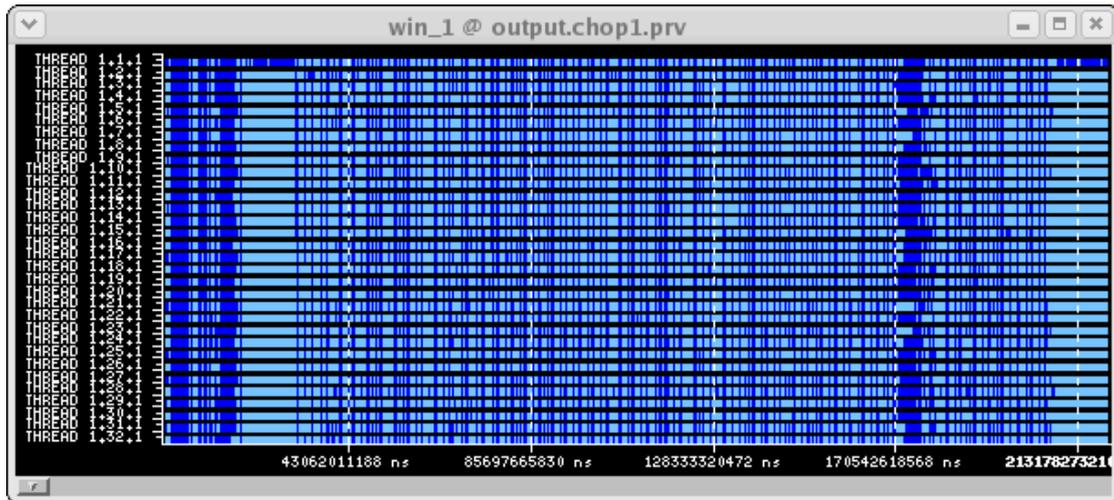


Figure 1: *Paraver* initial view for a run on 32 processors. Light blue lines represent computing and dark blue correspond to communication time and computing intensive sections smaller than 100ms.

### 4.3 Profilers on HLRS

#### 4.3.1 *Ftrace, prof and MPIPROGINF*

As reported in [27], the profiling tools employed on HLRS were *ftrace* and *prof* [26]. There is no line-by-line breakdown available; all information produced is at the routine level. The profiler *prof* (GNU *gprof*, version 2.11.90.0.8) is the profiler with lowest overhead. It is a statistical profiler which indicates the time spent in each routine. The code needs to be compiled with the `-p` compiler and linker. The profile results are stored in files named `mon.out.univ.rank`, where `univ` is the MPI communicator used for the communications profile and `rank` is the MPI rank of the process within that communicator. To obtain the profiling results, say on processor with rank 0, the user specifies:

```
prof -m -mon.out.0.0 executable
```

in the batch script, as there is no such profiler available at the front end. The code will run on the vector backend and not on the scalar front end.

The *ftrace* profiler (*sxftrace* command version : 2.2) shows more overhead than the *prof* profiler [27]. In contrast to the latter, *ftrace* is an execution trace profiler (see [27]). It reads the performance counters at the beginning and at the end of the analysed routines. Additionally, it allows the user to instrument regions of code which are of interest. This has not been used here, though. The code needs to be compiled with the `-ftrace` compiler and linker flag. The results are stored in files named `ftrace.out.univ.rank`. To obtain the profiling results, the user runs:

```
sxftrace -f ftrace.out.*.*
```

to gather information from all files and accumulated over the time spent on each processor. Unlike *prof*, *ftrace* does not display the time spent in any MPI calls explicitly but includes these times in the parent routine's execution time. This is due to the way MPI has been configured on HLRS.

In addition to the two profilers *prof* and *ftrace* there is at the environment variable level, `MPIPROGINF`, for profiling information to be collected that provides a summary of the efficiency of an application on the particular hardware it is using [26]. The code

needs to be compiled using the `-mpiprof` compiler and linker flag in order to gain this information. Also, the environment variables

```
export MPIPROGINF = ALL_DETAIL
export MPICOMMINF = ALL
```

need to be set. The first allows one to get more detailed information, the second allows for the analysis of the MPI communications. The `Min` and `Max` columns of the `MPIPROGINF` output show the minimum and maximum value of each metric, and the `universe` and `rank` of the process reporting the minimum and maximum values. The `Average` column shows the average per-process performance and the `Overall Data` section reports parallel performance for the entire program (see Table 8).

#### 4.4 Discussion of available profilers

The profilers on HPCx were the same as those on RZG, but were not employed for reasons given in the following Section.

The `gprof` profiler is a tool which is available on many machines providing very useful information, such as a line-by-line breakdown of the time-intensive (source) code. The `xprofiler` allows for easier accessibility of the `gprof` info. It is less interesting if the MPI library used has not been compiled with it and consequently no information about MPI communications can be obtained.

The `mpitrace` library as found on RZG complements the information obtained with `gprof` and shows a breakdown of the MPI communication routines used. The information summary produced about the various elapsed times is somewhat confusing and we have not found documentation that could clarify this matter.

The `mpitrace` library on BSC has to be used in conjunction with `Paraver`. `Paraver` requires a steeper learning curve than the tools previously presented and has not been used by the authors of this report themselves but by [13]. There appears to be the potential for a very detailed analysis which has probably not been fully exploited due lack of time.

## 5 Profiling Study for the Virgo Simulation Code

In this Section we present the profiling results obtained with the various profilers discussed in Section 4 for the Virgo Simulation code on RZG, BSC and HLRS.

For the profiling study the 128k StB data set was used unless explicitly stated otherwise. For 64 processors the 128k data exhibited about 70% parallel efficiency and thus made good use of the resources employed [4] and was thus used for the profiling runs. To contrast we found that the 64k data set did not scale well due to the relatively small problem size - the communications cost quickly dominated over the computation assigned to each processor while the 256k data set exhibited memory problems on HPCx which needed to be investigated separately and on RZG the simulations took too long to complete.

To run a job to completion for the 128k StB data set would require a 64 processor job to run for over eight hours<sup>4</sup>. Our previous scaling studies, reported in [4], had shown that the code ran fastest on HPCx closely followed by RZG (see also Figure 3 and

<sup>4</sup> This was the expected run time required on RZG, from [4].

Figure 4 in Section 7) but as HPCx does not provide mechanisms for 64 CPU jobs to run for longer than 6 hours we chose to run the profiling study on RZG<sup>5</sup>. Using checkpoint files to get a profile of a complete run using these as re-start files was not an option as it would have caused too much interference with the results.

Another reason for using RZG was that it had virtual memory enabled, which HPCx did not. This allowed runs on RZG to swap to disk when the 2 Gb per processor memory limit was exceeded. The HPCx runs on the other hand encountered insufficient memory problems for the 128k data set. Note in these instances that the total amount of real memory which can be used by a process is actually only about 1.7 Gb as the OS kernel requires to occupy some of the memory.

As previously stated we use the 128k StB data set to run the code. The `gprof` results on RZG obtained for the complete run (time step  $n=341$ ) and the run up to time step<sup>6</sup>  $n=25$  demonstrate clearly that the latter is already exemplary for the whole run as the same routines appear in the same order across both profiles (see Table 2 and Table 3). Thus we only investigate runs up to time step  $n=25$  hereafter.

%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
18.0	4353.33	4353.33	309738	14.05	18.63	.poisson_mg_residual [8] poisson_mg_residual.F90
15.1	8008.64	3655.31	169016712	0.02	0.02	.amr_cp_loc [14] amr_cp_loc.F90
5.3	9290.24	1281.60				._getvd_hndlr [16] /usr/lib/liblapi_r.a
4.5	10368.45	1078.21				._stripe_hal_init [19] /usr/lib/liblapi_r.a
4.0	11336.48	968.03	1085622	0.89	0.89	.amr_prolong_gen_work_fun [21] amr_prolong_gen_work_fun.F90
3.4	12164.89	828.41				.LAPI__Nopoll_wait [22] /usr/lib/liblapi_r.a
2.9	12871.18	706.29				._tag_waiting [23] /usr/lpp/ppe.poe/lib/libmpi_r.a
2.5	13483.32	612.14				._convert_lvector_to_ldgsp [26] /usr/lib/liblapi_r.a
2.3	14031.13	547.81				._check_one_vec [28] /usr/lib/liblapi_r.a
2.1	14547.92	516.79	43618224	0.01	0.01	.amr_restrict_work_fun [27] amr_restrict_work_fun.F90
2.1	15051.01	503.09	63042	7.98	8.62	.amr_restrict_bnd_data [32] amr_restrict_bnd_data.F90
2.0	15530.82	479.81	1349007	0.36	4.06	.mg_bndry [9] mg_bndry.F90

**Table 2: gprof result for the 128k StB data set run on 64 processors up to time step  $n=341$  (completion) on RZG for the Virgo Simulation code. Only routines which use 2% or more of the total run time are displayed.**

<sup>5</sup> To find out how much memory per processor is available and for how long a job can run for a given number of processor the LoadLeveler command `llclass` needs to be run.

<sup>6</sup> This end time can be set by specifying it in the variable `nend` in the input file `flash.par`.

%time	cumulative		self calls	ms/call	self ms/call	total name
	seconds	seconds				
18.9	560.93	560.93	40189	13.96	17.92	.poisson_mg_residual [8]
15.4	1017.87	456.94	21971728	0.02	0.02	.amr_cp_loc [14]
4.9	1161.63	143.76				._getvd_hdlr [18] /usr/lib/liblapi_r.a
4.5	1294.31	132.68				._stripe_hal_init [19] /usr/lib/liblapi_r.a
4.3	1420.77	126.46	141777	0.89	0.89	.amr_prolong_gen_work_fun [21]
3.1	1514.09	93.32				.LAPI__Nopoll_wait [22] /usr/lib/liblapi_r.a
2.9	1599.33	85.24				._tag_waiting [24] /usr/lpp/ppe.poe/lib/libmpi_r.a
2.3	1667.54	68.21				._convert_lvector_to_ldgsp [31] /usr/lib/liblapi_r.a
2.2	1732.63	65.09	5675128	0.01	0.01	.amr_restrict_work_fun [26]
2.2	1796.75	64.12	8121	7.90	8.54	.amr_restrict_bnd_data [30]
2.0	1856.25	59.50	215928	0.28	3.19	.mg_bndry [9]
2.0	1915.38	59.13				._check_one_vec [32] /usr/lib/liblapi_r.a
2.0	1974.26	58.88				._stripe_hal_close [33] /usr/lib/liblapi_r.a

**Table 3: gprof result for the 128k StB data set run on 64 processors up to time step n=25 on RZG for the Virgo Simulation code. Only routines which use 2% or more of the total run time are displayed.**

The two most time consuming FLASH routines are `poisson_mg_residual` and `amr_cp_loc` which respectively take 18.9 % and 15.4 % of the total run time. Note the routines related to the `liblapi` library relate to communication routines.

The total run time to completion for this particular simulation was 8 hours 3 minutes without any profilers being used and 8 hours 9 minutes when the compiler flags `-pg -g` were used (`gprof` enabled), thus profiling using `gprof` does not introduce a significant overhead.

In the above Tables we display the results which take 2% or more of the total run time. The routines with one preceding underscore are related to MPI-calls. It can be shown that, on RZG, 21% of the total run time is taken up by MPI-related calls, i.e. communication across processors. This result can be compared to one obtained from the `mpitrace` tool (see Table 7), where the total communication time appears to be 49 % of the total elapsed time. Note that the percent contribution of each of the routines to the total run time might slightly vary from processor to processor. This is also true for the ordering of the routines. These typical variations between the results of different profilers are probably due to implementation specifics.

The profiling results obtained with the FLASH internal routine `perfmon` are shown in Table 4. We show the percentage relative to the total run time. NB: a subroutine's descendants are included and indented; the associated percentage time is also included but not indented.

```

perf_summary: code performance summary
                beginning : 05-23-2006 17:51.45
                ending   : 05-23-2006 18:51.34
    seconds in monitoring period : 3588.231
      number of subintervals : 25
    number of evolved zones : 0
        zones per second : 0.000
-----
accounting unit                time pct
-----
initialization                 7.161
particles                      0.007
  redistribution                0.007
guardcell internal             0.005
boundary conditions            0.000
gc_srl                         0.004
gc_ctf                         0.000
gravity                         5.575
  guardcell internal            0.005
  boundary conditions           0.000
  gc_srl                       0.004
  gc_ctf                       0.000
i/o                            0.417
evolution                      92.421
  particles                    0.841
  redistribution                0.212
  particle forces               0.622
  move particles                0.008
gravity                         90.724
  guardcell internal            0.276
  boundary conditions           0.000
  gc_srl                       0.220
  gc_ctf                       0.004
i/o                            0.476
update refinement              0.171
  particles                    0.171
  redistribution                0.171
=====

```

**Table 4: perfmon result for the 128k StB data set run on 64 processors up to time step n=25 on RZG for the Virgo Simulation code.**

The most time consuming part of the code is the evolution loop (92%) and within the gravity calculation (91%). This is consistent with the results obtained from the other profilers.

To run the simulation to completion would take over 12 hours on BSC compared to a little over 8 hours on RZG. This is another reason why we only investigated runs up to time step  $n=25$ , which is, as we previously demonstrated, characteristic of the complete run. The `gprof` result on BSC is shown in Table 6. It can be seen that the most time consuming FLASH routines are basically the same as the ones for RZG.

In [27], the profiling study on HLRS was done for the 128k StB data set on 32 processors, which proved to be the number of processors where the code shows about 70% efficiency. The result of the `prof` profiler is shown in Table 5 for a run up to time step<sup>7</sup>  $n=48$ .

<sup>7</sup> This uses a slightly different end time step from the other results used in this report as the work on HLRS was done by an MSc student who chose slightly different criteria.

%Time	Seconds	Cumsecs	#Calls	msec/call	Name
29.7	1412.22	1412.22			mpisx_vtestv
8.3	393.95	1806.18	15469600	0.0255	amr_restrict_work_fun_
6.1	288.90	2095.08			MPID_SHMEM_Check_incoming
4.2	201.12	2296.19			mpisx_wait
3.8	178.63	2474.83	250020	0.7145	amr_prolong_gen_work_fun_
3.5	167.39	2642.22			mpisx_ixs_pkt_hrreg
2.4	114.89	2757.10	203783	0.5638	amr_restrict_cc_
2.2	102.61	2859.71	98	1047.0	mapparticlestomesh_
1.9	89.61	2949.32	35411	2.5306	poisson_mg_residual_
1.7	80.97	3030.29	18057	4.4841	poisson_mg_relax_
1.6	78.01	3108.30	19543	0.3992	amr_guardcell_cc_srl_
1.5	73.07	3181.37			f_cassigni
1.3	62.76	3244.13	150626	0.4167	b_int_sendrcv_
1.3	61.96	3306.09			H5S_hyper_compare_regions
1.3	60.40	3366.49	46806879	0.0013	dbasetree.dbaseneighborblocklist_
1.2	57.18	3423.67			mpisx_waitixs_long
1.1	53.44	3477.10			MPID_IXS_Rndvn_ack_queue
1.1	52.35	3529.45			xx_shmalloc
1.1	50.89	3580.34			_real_realloc
1.1	50.44	3630.78			mpi_ssend_
1.0	48.72	3679.49			f_secpsc
1.0	45.19	3724.69			f_arypsc

**Table 5: The `prof` result for the 128k StB data set run on 32 processors up to time step  $n=48$  on HLRs for the Virgo Simulation code. Routines which use 1% of the total run time or more are displayed. These results first appeared in [27].**

The routines listed which start with `mpisx_` are not FLASH routines but MPI communication calls. These MPI related routines take up over 40 % of the total run time. The most time intensive FLASH routines are the same as those identified for RZG or BSC but with a different priority ordering.

The `mpitrace` result obtained on RZG is shown in Table 7. A summary of the average times, the memory and their standard deviation can be found in Table 16 (together with the results for the Improved Virgo Simulation code).

Each sample counts as 0.01 seconds.							
%	cumulative	self		self	total		
time	seconds	seconds	calls	s/call	s/call	name	
41.21	5220.55	5220.55	39189	0.13	0.16	.poisson_mg_residual	
18.57	7573.30	2352.75	9318596	0.00	0.00	.amr_cp_loc	
15.50	9537.41	1964.11	4137812	0.00	0.00	.amr_prolong_gen_work_fun	
3.08	9927.73	390.32	204479	0.00	0.03	.mg_bndry	
2.86	10289.82	362.09	9318	0.04	0.05	.amr_prolong_cc	
2.47	10602.44	312.62	792	0.04	0.04	.amr_restrict_bnd_data	
2.35	10900.62	298.18	20189	0.01	0.23	.poisson_mg_relax	

**Table 6: The `gprof` result for the 128k StB data set run on 64 processors up to time step  $n=25$  on BSC for the Virgo Simulation code. Routines which use 2% of the total run time or more are displayed.**

MPI Routine	#calls	avg. bytes	time(sec)
MPI_Comm_size	96	0.0	0.002
MPI_Comm_rank	42	0.0	0.000
MPI_Send	8606	3452.8	0.445
MPI_Ssend	15185486	5987.0	997.827
MPI_Isend	56664	128288.0	3.633
MPI_Recv	19	4.0	0.000
MPI_Irecv	15218056	7084.1	39.905
MPI_Sendrecv	16	4.0	0.000
MPI_Waitany	1008326	0.0	201.682
MPI_Waitall	939317	0.0	183.594
MPI_Bcast	213	115.2	1.052
MPI_Barrier	4434	0.0	51.653

MPI_Gather	4	8.0	0.042
MPI_Scan	10	5.6	0.001
MPI_Allgather	492	6967.3	14.959
MPI_Reduce	26	88.0	0.032
MPI_Allreduce	94126	10.9	268.827
-----			
total communication time	= 1763.655 seconds.		
total elapsed time	= 3588.273 seconds.		
user cpu time	= 3376.230 seconds.		
system time	= 54.530 seconds.		
maximum memory size	= 740928 KBytes.		
-----			
Message size distributions:			
MPI_Send	#calls	avg. bytes	time(sec)
	7	4.0	0.000
	8	356.0	0.000
	8567	1632.0	0.029
	8	2136.0	0.000
	8	4806.0	0.000
	2	182272.0	0.397
	3	1146880.0	0.004
	3	3956736.0	0.015
MPI_Ssend	#calls	avg. bytes	time(sec)
	85124	4.0	3.999
	526	8.0	0.007
	84115	12.0	19.381
	84024	20.0	2.698
	341070	64.0	17.681
	420097	79.2	76.032
	252166	161.3	41.656
	167922	396.0	5.178
	3549016	2048.0	187.426
	4092112	4096.0	264.056
	5443520	8191.6	291.573
	657998	32451.8	83.814
	3325	57344.0	1.228
	4259	114676.0	2.773
	212	458752.0	0.326
MPI_Isend	#calls	avg. bytes	time(sec)
	33	4.0	0.000
	1199	1632.0	0.008
	55432	131104.0	3.625
MPI_Recv	#calls	avg. bytes	time(sec)
	19	4.0	0.000
MPI_Irecv	#calls	avg. bytes	time(sec)
	75696	4.0	0.177
	537	8.0	0.002
	272808	64.0	0.475
	68	112.0	0.000
	3550182	2047.7	9.779
	4092112	4096.0	11.872
	5445090	8191.5	9.977
	1007568	8204.0	3.974
	710744	32198.5	2.988
	3440	57344.0	0.013
	4273	114667.2	0.012
	55432	131104.0	0.635

	106	458752.0	0.002
MPI_Sendrecv	#calls	avg. bytes	time(sec)
	16	4.0	0.000
MPI_Bcast	#calls	avg. bytes	time(sec)
	180	4.0	0.929
	25	20.0	0.122
	1	108.0	0.000
	1	212.0	0.000
	1	608.0	0.000
	1	2160.0	0.000
	4	5060.0	0.000
MPI_Gather	#calls	avg. bytes	time(sec)
	4	8.0	0.042
MPI_Scan	#calls	avg. bytes	time(sec)
	6	4.0	0.001
	4	8.0	0.000
MPI_Allgather	#calls	avg. bytes	time(sec)
	124	4.0	10.925
	76	1800.0	0.099
	140	3961.4	3.160
	76	10800.0	0.242
	76	25200.0	0.534
MPI_Reduce	#calls	avg. bytes	time(sec)
	26	88.0	0.032
MPI_Allreduce	#calls	avg. bytes	time(sec)
	40042	4.0	257.855
	1790	8.0	3.248
	52201	16.0	5.173
	8	24.0	0.002
	8	112.0	0.073
	77	224.8	2.476

**Table 7: mpitrace result for processor 1 for the Virgo Simulation code run up to time step 25 for the 128k StB data set on 64 processors on RZG.**

The mpitrace output on RZG revealed that the most time intensive MPI calls are MPI\_Ssend (27.8 %), MPI\_Allreduce (7.5%) and MPI\_Waitany (5.6 %). The three most prominent MPI\_Ssend calls are the one where 2048, 4096 or 8191 bytes are sent on average.

We have no results for mpitrace interpreted with the Paraver tool on BSC for the Virgo Simulation code, due to lack of time.

A part of the MPIPROGINF output on HLRS is presented in Table 8 and is reproduced from [27]. This run was done with the Virgo Simulation code, the 128k StB data set up to time step  $n=48$  and on 32 processors. It can be shown from these results that the MPI calls (Max total real MPI time) take about 64% of the total run time, i.e. 50 minutes (3009 s). In addition, the MPIPROGINF output shows that the average vector operation ratio is 82% and the average vector length which processed by vector pipes is 43 [units]. A good vectorization means a vector operation ratio of over 95 % and an average vector length of 200 or more, the vector length cannot exceed 256 [26]. To improve the vectorization ratio, work on exposing fine-grain parallelism in the application would be required. This includes finding the most expensive loops and ensuring that they have no dependencies. Additionally, it

would be advisable to collapse loops in order to increase their length, or if they are always shorter than 256, using the `shortloop` or the `vreg` directives to reduce the number of memory accesses.

```

MPI Program Information:
=====
Note: It is measured from MPI Init till MPI Finalize.
      [U,R] specifies the Universe and the Process Rank in the Universe.

Global Data of 32 processes : Min [U,R] Max [U,R] Average
=====
Real Time (sec)           : 4601.426 [0,22]      4601.867 [0,6]      4601.564
User Time (sec)          : 4449.016 [0,0]      4553.510 [0,14]     4538.654
System Time (sec)       : 3.327 [0,12]      19.276 [0,0]        5.717
Vector Time (sec)       : 1863.847 [0,31]     2602.901 [0,3]      2226.611
Instruction Count        : 832397116104 [0,27]  70237647439 [0,31]  870123792868
Vector Instruction Count : 64362051625 [0,31]  179099446708 [0,3]  96122576309
Vector Element Count     : 3048938081971 [0,31]  4367744498486 [0,3]  3732796656409
FLOP Count              : 176223268270 [0,1]  347171769966 [0,28]  220142994236
MOPS                    : 870.732 [0,31]      108.194 [0,3]       992.964
MFLOPS                  : 38.830 [0,9]         76.436 [0,28]       48.494
Average Vector Length    : 23.940 [0,28]        3.150 [0,16]        43.019
Vector Operation Ratio (%) : 77.094 [0,31]       86.680 [0,3]       82.712
Memory size used (MB)    : 1070.065 [0,10]     1325.894 [0,0]      1106.454
Global Memory size used (MB) : 16.000 [0,0]        32.000 [0,6]        18.000
MIPS                     : 182.969 [0,7]      213.617 [0,31]      191.714
Instruction Cache miss (sec) : 139.319 [0,3]      217.748 [0,30]      187.230
Operand Cache miss (sec)  : 418.747 [0,3]     680.251 [0,30]     587.323
Bank Conflict Time (sec)  : 109.246 [0,28]    145.401 [0,26]     130.723

Overall Data:
=====
Real Time (sec)           : 4601.867
User Time (sec)          : 145236.939
System Time (sec)       : 182.928
Vector Time (sec)       : 71251.553
GOPs (rel. to User Time) : 31.775
GFLOPS (rel. to User Time) : 1.552
Memory size used (GB)    : 34.577
Global Memory size used (GB) : 0.563

MPI Communication Information:
-----
Real MPI Idle Time (sec)  : 1151.372 [0,28]      1545.792 [0,16]     1354.084
User MPI Idle Time (sec) : 1148.776 [0,28]     1542.439 [0,16]     1350.849
Total real MPI Time (sec) : 2433.721 [0,28]    3009.234 [0,8]      2821.817
Send count                : 15843695 [0,3]     30876977 [0,26]     26078976
Recv count                : 15934889 [0,3]     30689132 [0,26]     26078976
Barrier count             : 2314 [0,0]         2314 [0,0]          2314
Bcast count              : 397 [0,0]          397 [0,0]           397
Reduce count             : 49 [0,0]           49 [0,0]             49
Allreduce count          : 94864 [0,0]        94864 [0,0]          94864
Scan count               : 11 [0,0]           11 [0,0]             11
Gather count             : 4 [0,0]            4 [0,0]              4
Allgather count          : 650 [0,0]          650 [0,0]            650
Number of bytes sent     : 111996401376 [0,3]  201206190672 [0,13]  173673265447
Number of bytes rcv      : 109116062712 [0,3]  201188415028 [0,18]  173673265447

```

**Table 8: Output obtained with MPIPROGINF on HLRS for the Virgo Simulation code and the 128k StB data set on 32 processors and run up to time step n=48. These results are from [27].**

The MPIPROGING tool also identifies the most often executed MPI calls; they three most prevalent ones are `MPI_Send`, `MPI_Recv` and `MPI_Allreduce`. The results on RZG showed that `Ssends`, `allreduce` and `waitany` were the most prevalent communications. The most expensive routine on BSC were not identified due to lack of time.

### 5.1 Potential Optimization Strategies

The various profiling tools used in Section 5 for the Virgo Simulation code on RZG, BSC and HLRS highlighted several time-intensive parts of the code. These can be classified as follows:

- 
- i. Source code related:
    - o The most time intensive routines, such as `poisson_mg_residual` and `amr_cp_loc`, are all related to the solution of the Poisson equation. Since the simulation only does gravity it is not surprising that they dominate the CPU time. The profilers that provide a line-by-line breakdown of the source code revealed that the most time intensive parts of the FORTRAN code are all related to either initialising (setting whole arrays to zero, for instance) or copying arrays locally.
  - ii. MPI communication related:
    - o In general, a very high percentage of the total run time (20-50%) is consumed by calls to MPI routines, where the actually percentage depends on the platform and the profiler employed.

There are several possible approaches to tackle these problems:

- i. Source code optimization:
  - o Investigate re-ordering of loops, array indices etc. If this is not feasible and there are no obviously straight-forward improvements then:
    - i. Consider a complete re-write of routines identified via the profilers, e.g. use a new algorithm to remove the need to assign whole arrays to zero.
- ii. MPI communication related optimization.
  - o Replace sets of point-to point communications by single calls to MPI collective communications.
  - o Replace synchronous sends `MPI_Ssend` by the standard send `MPI_Send` to let the underlying MPI decide whether to buffer outgoing messages or not and thus potentially get a slight edge on performance.
- iii. Compiler based optimization, which is by its nature platform dependent. To get good performance on a given architecture it is usually worthwhile studying the architecture specific compiler flags. Many compilers automatically detect possible optimizations and compile the code to gain the best performance by exploiting the underlying architecture used.

We look into the potential optimization strategies proposed above in more detail below:

- i. The profiling results for the Virgo Simulation code show that most of the total run time of the code is used by the solution of the Poisson equation. The current implementation of the Poisson solver is an iterative solver based on a multigrid method. An alternative is to use Fast Fourier Transforms, a non-iterative spectral method. The iterative method to solve the Poisson equation is in general faster than the FFTW when used for serial code [34] since there are fewer operations to be carried out. For parallel code, however, a parallel FFT can be much faster than a parallel multigrid. In addition, the package used, i.e. FFTW [9], scale very well on parallel machines. Note, when employing vector platforms, that the FFTW library has been especially optimised for cache-based architectures. There are FFT packages, such as the MathKeisan Fast Fourier Transform [25] or the Fast Fourier Transform Package FFTE [8], which are especially optimized for vector machines but were not investigated on the HLRS machine by this project.
- ii. MPI communications: We replaced `MPI_Ssends` in the files `multigrid.F90` and `amr_guardcell_cc_srl.F90` by `MPI_Sends`. The latter routine is called very frequently and hence shows a high rate of `MPI_Ssends`. An investigation with the `mpitrace` library revealed that no performance gain could be achieved by this substitution on RZG. There was no time to

- incorporate the results obtained from the `Paraver` study on BSC regarding the most time intensive MPI communications as described in Section 8.
- iii. Compiler based optimization
    - As stated in [27], on HLRS a speed increase of a factor of 6.7 was be seen for the 128k StB data set up to time step  $n=48$  on 8 processors when using the compiler flags `-C vopt` instead of `-C debug` (except for the two files `amr_restrict_unk_fun` and `amr_restrict_work_fun`). Increasing the optimization compiler flags further did not show any improvement on the code's performance.
    - The compiler flags on RZG and BSC have already been outlined in [4] and proved to be the fastest (see also Appendices 13.3 and 13.4.)

## 5.2 Summary of Findings

The profiling study revealed that the most time consuming parts of the code are MPI communications and the solver for the Poisson equation. In addition, when the code is ported to a new architecture it is worthwhile to investigate which compiler flags are optimal on the given platform. We started replacing some MPI calls by different ones which did not speed up the code on RZG but might still be advantageous on other platforms. Further investigation of the MPI communications is recommended. The replacement of the Multi Grid based Poisson solver by a new algorithm is investigated in some further depth in the following Sections.

## 6 The Improved Virgo Simulation Code

The most time consuming parts of the code are related to the Poisson solver of the FLASH code as the profiling study for the Virgo Simulation code showed in Section 5 and discussed in Section 5.1. Hence the routines based on the multigrid solver were replaced by an FFT solver. A more detailed description of this algorithm is given in Appendix 16

We run the FLASH python setup script to generate the new code from the corresponding modules:

```
./setup epcc -auto -3d -maxblocks=MB
```

where the `-auto` option enables `setup` to generate a "first draft" of a Module file for the user, `-3d` generates Makefiles for three dimensional problems, `MB=450` is used for the 128k StB data set.

### 6.1 Porting the Improved Virgo Simulation Code

There were many calls to the non-standard Fortran flush routine in order to avoid I/O buffering. However, as these calls take a relatively long time to execute and are only there as a debugging aid, usually during the code development stage, we removed them. Moreover, the flush routine is not part of the Fortran standard and can thus cause portability problems. An alternative on IBM Regatta systems is to employ the environment variable

```
export XLFRTSEOPTS buffering=disable_all
```

within the batch script which produces unbuffered output. However, this also slows the code down considerably and should only be use during a debugging process.

For the porting of the Improved Virgo Simulation code the code was first ported to HPCx and then to RZG, HLRS and BSC. The compilers on the various machines showed different tolerances towards more relaxed coding practices.

On HLRS, HPCx and RZG the routine `fftw_f77.i` needs to be copied into the object directory.

### 6.1.1 Porting to HPCx

On HPCx it was necessary to:

- The Fortran compiler complained about several problems in the file `multigrid.F90` which had been changed in order to use the FFTW algorithm for the simulation [34].
  - Additional continuation marks ‘&’ needed to be added.
  - Twice, when a file was opened, the open statement contained the specifier `access` instead of `position`.

The Fortran function call `flush(6)` needed to be replaced by `flush_(6)`. This call avoids I/O buffering and is only advisable for testing purposes. For any scaling studies and further investigations of the code it was removed.

### 6.1.2 Porting to RZG

Several changes needed to be performed to allow the code run on RZG in addition to those already introduced for HPCx:

- Several DEISA modules need to be loaded before compilation and linking:

```
module load deisa
module switch fortranreal/4 fortranreal/8
module load fftw (default is version
2.1.5)
```

The `Makefile.h` can then be extended as follows:

```
# double precision fftw libraries required
FFTW_LIB      = $(DEISA_LDFLAGS)
INCLUDE_FFTW  = $(DEISA_FFLAGS)
```

Where the `DEISA_LDFLAGS` are defined locally as

```
-L/afs/rzg/common/soft/fftw/fftw-2.1.5/@sys/lib -ldfftw
-ldrfftw -ldfftw_mpi -ldrfftw_mpi -lm and the DEISA_FFLAGS
-q64 -qautodbl=dbl4 -qdpc=e
-I/afs/rzg/common/soft/fftw/fftw-2.1.5/@sys/include.
```

- The Improved Virgo Simulation code only runs on RZG when the files `init_from_scratch.F90` and `InitParticlePositions.F90` are compiled with the `mpxlf_r` compiler instead of `mpxlf90_r`. Otherwise a segmentation fault will occur in the routine `InitParticlePositions.F90` due to a subscript being out of bounds.

- In addition, a stack limit of 200 MB was chosen in the batch script. The previous choice of 20 MB made the code abort with a segmentation fault for the 128k data set.

### 6.1.3 Porting to BSC

The code was linked to the FFTW, version 2.1.5, 64-bit which can be found in the directory `/gpfs/apps/FFTW/2.1.5/64`.

In order to port the code to BSC the `lnblnk F77`-routine in the file `init_from_scratch.F90` had to be replaced by the routine `len_trim` as the compiler did not accept the F77 version.

### 6.1.4 Porting to HLRS

The porting to HLRS was carried out by [27] and is reproduced here for completeness.

In order to port the code, the code base being the port to HPCx, successfully onto HLRS several changes needed to be applied to the Improved Virgo Simulation code.

- According to the Fortran standard a Fortran name must consist of between 1 and 31 alphanumeric characters. Some variable names in the files `multigrid.F90` and `init_from_scratch.F90` needed to be shortened for this reason.
- In several routines, the statements `include "mpif.h"` and `include "fftw_f77.i"` came before declaring `implicit none`. The order needed to be reversed.

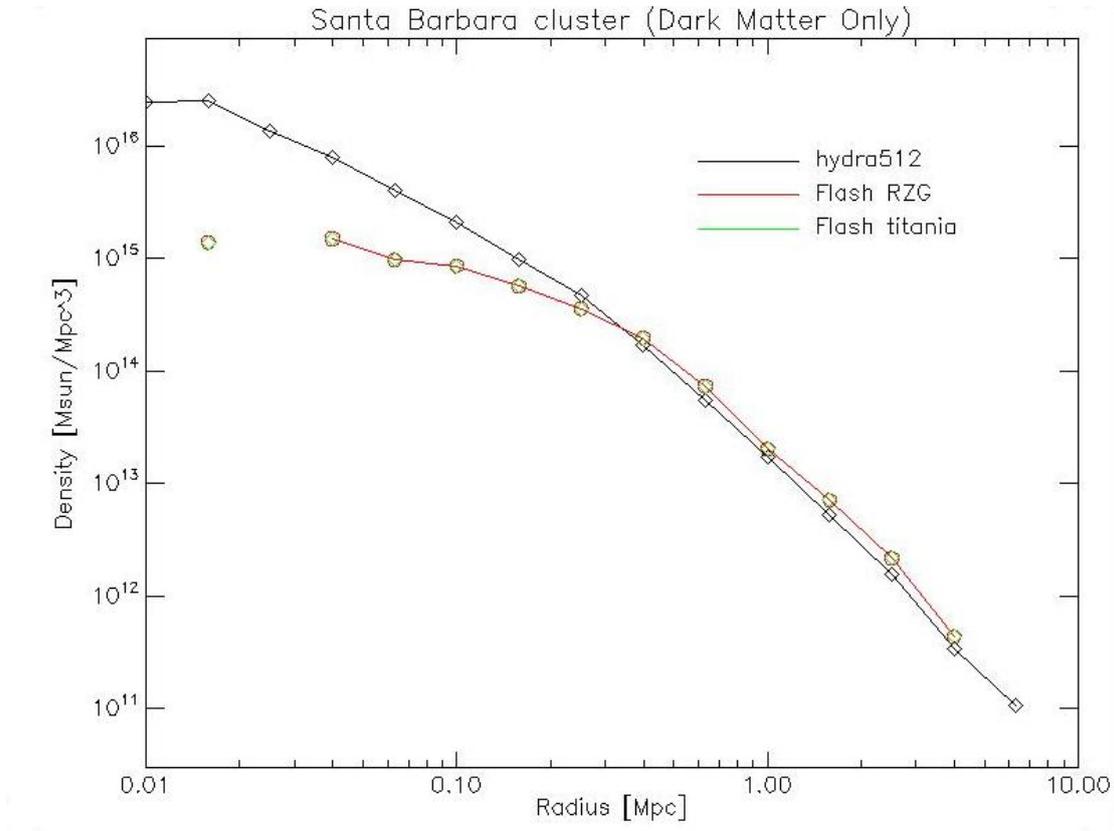
All compiling problems on HLRS are discussed in more detail in [27].

### 6.1.5 Validation of the Results

In order to check the correctness of the results of the ported code we use the output checkpoint files obtained from [34] for the same simulation done on ICC and compare the results obtained there to our results with the help of an IDL [19] script which was also provided by [34]. In other words, we checked if the numerical results between different machines agree and [34] checked the consistency to ensure that the physical results are correct via the IDL script.

In addition, results for the StB simulation are available from another of Virgo's cosmological simulation codes, i.e. `Hydra_MPI` [19] which can be used as a means to check for consistency. The `Hydra_MPI` simulation code was run for  $512^3$  particles. This simulation provides a means of validating the FLASH code with the new FFT algorithm. The graph produced by the IDL script shows the mass density as a function radius, where the radius is taken as the distance from the centre of mass. All results for the Improved Virgo Simulation code (obtained on ICC, HPCx, RZG, BSC and HLRS) are consistent; see Figure 2 for the results on ICC (titania) and RZG (psi). The differences between the Improved Virgo Simulation code and the Hydra code can be attributed to various reasons: The difference is especially visible for radii smaller than 0.05 Mpc. First, we use less particles, i.e.  $64^3$  as opposed to  $512^3$ , which accounts for a worse resolution. Second, there is an initial error in calculating the centre of mass which differs from run to run. Despite the apparent differences in the curves, the graph is used to demonstrate that port is successful, as the differences are within a tolerable range. Third, the main reason is low number statistics. The

program is binning particles so that for small radii there will be small numbers of particles and hence there will be fluctuations.



**Figure 2: Results for the Hydra code and the FLASH based Improved Virgo Simulation code. The baseline for the Improved Virgo Simulation code is the result from ICC (titania) obtained by [34] using the 64k StB data set. This result agrees with the ones from RZG, BSC, HLRS and HPCx. The RZG data points coincide with the titania data points and hence the titania data points are obscured. The BSC, HLRS and HPCx data points are not displayed as they coincide exactly with the Titania and RZG data points.**

## 6.2 Conclusions

In this Section we described how the Improved Virgo Simulation was ported to HPCx, RZG, BSC and HLRS. Now we can run a scaling study for the code based on the new solver for the Poisson equation and compare it directly with the results for the Virgo Simulation code. Then, we can profile the Improved Virgo Simulation code and compare these results with the ones obtained for the Virgo Simulation code. This is done in Sections 7 and 8.

## 7 Scaling Study

For the scaling study we compare results for both the Virgo Simulation code and the Improved Virgo Simulation code for the 64k and 128k data sets, up to time steps  $n=41$  and  $n=48$ , respectively<sup>8</sup>, on: HPCx, RZG, BSC, HLRS and IDRIS (the latter one not for all scenarios). All results for the 64k StB data set for both codes are displayed in Figure 3 and the times for the Improved Virgo Simulation code are listed in Table 9 for the 64k StB data set. The results for from HLRS for the Virgo Simulation code can

<sup>8</sup> Differences in the final simulation time step reached arose because these runs were submitted to a one hour batch queue, thus the different data sets reached different end simulation time steps and the range was determined by the slowest platform for each given data set.

be found in Table 11 and are from [27] and [28]. Note, that there are no results for the 64k StB data set run with the Virgo Simulation code on BSC as this machine proved to be too slow for this data set [4]. The Improved Virgo Simulation code allows running the code on BSC on a much smaller number of processors than before as it is much faster.

The fastest run for the 64k StB data set occurred on HPCx using 16 processors while the code consistently runs slowest on IDRIS with the Virgo Simulation code. The Improved Virgo Simulation code is faster than the Virgo Simulation code on all architectures. The more processors are used, the faster the code runs, albeit with decreasing efficiency. The fastest architecture is HPCx, followed by HLRS, RZG and HLRS. The Improved Virgo Simulation code is up to a factor of 3.7 faster than the Virgo Simulation code; this is for the run on 16 processors on HPCx and HLRS and on 32 processors on HLRS as well. The data set is not very large. That is why the Improved Virgo Simulation code does not show a more significant speed-up compared to the Virgo simulation code.

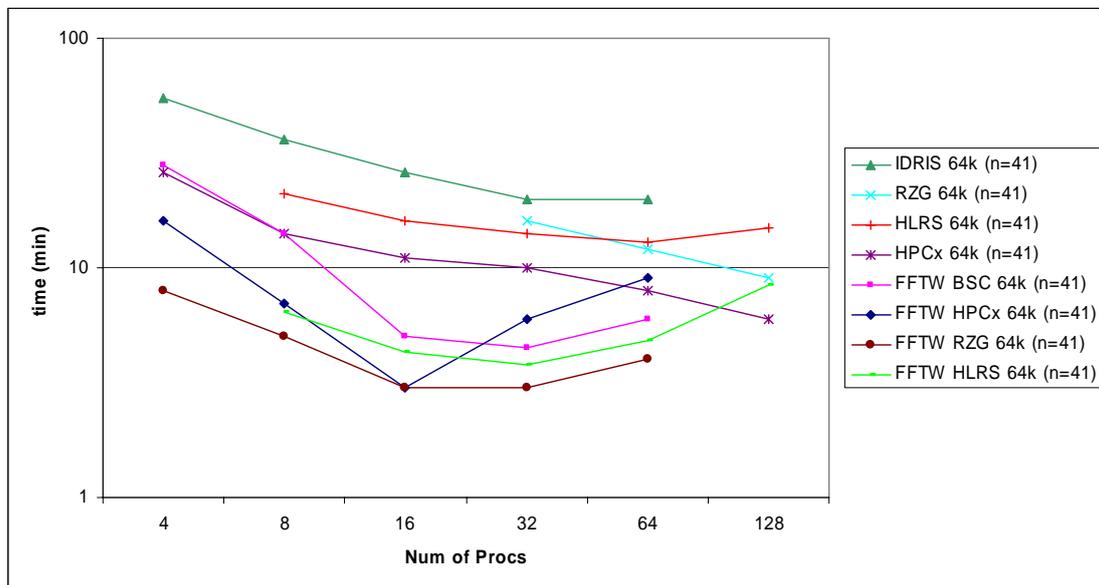


Figure 3: Results on HPCx, IDRIS, RZG and BSC for 4, 8, 16, 32, 64 and 128 processors and run up to time step n=41 for the Virgo Simulation code and the Improved Virgo Simulation code, labelled FFTW, and the 64k StB data set.

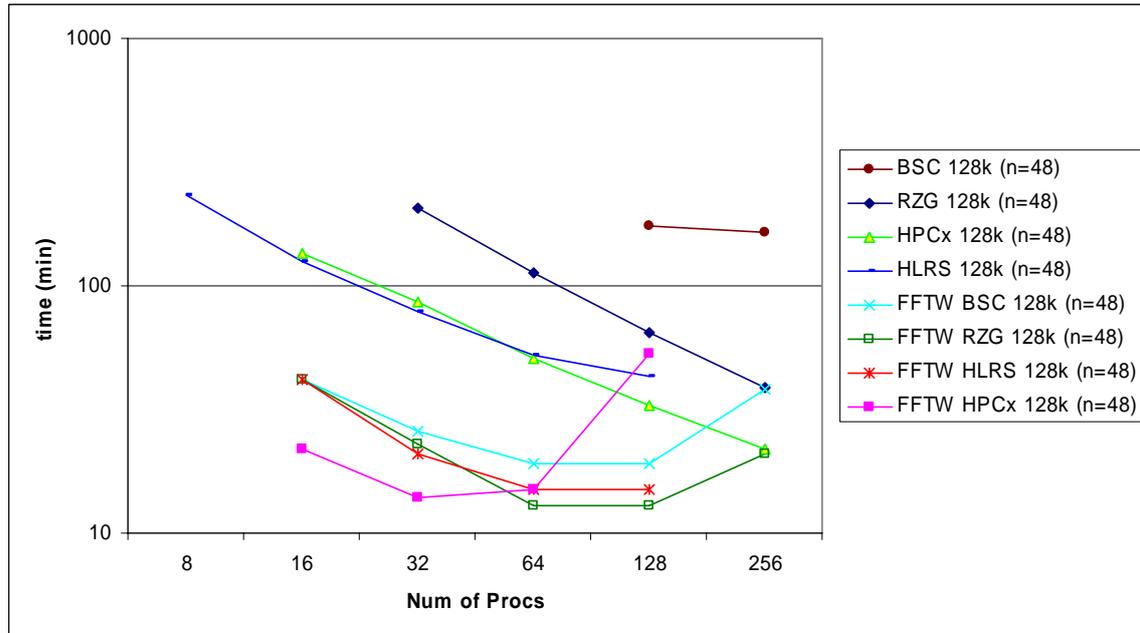
	HPCx		RZG	BSC	HLRS	
	n=41	n=178 (completion)	n=41	n=41	n=41	N=174 (completion)
<b>4</b>	16	-	8	28	-	-
<b>8</b>	7	35	5	14	6.4	28
<b>16</b>	3	17	3	5	4.3	17
<b>32</b>	6	26	3	4.5	3.8	13
<b>64</b>	9	42	4	6	4.8	14

Table 9: Time in minutes needed on HPCx, RZG, BSC and HLRS to reach a certain simulation time step n for 4, 8, 16, 32 and 64 processors using the 64k StB data set with the Improved Virgo Simulation code.

The scaling study results for the 128k StB data set and the Improved Virgo Simulation code in Figure 4 and Table 10 show that HPCx is the fastest architectures for up to 32 processors. The results for RZG and HLRS are very similar and on 64 processors or more they outperform HPCx. BSC is the slowest of the four IBM

machines. For the Virgo Simulation code HPCx and HLRS compete very closely, followed by RZG and then BSC.

For the 128k dataset running on 32 RZG processors, the Improved Virgo Simulation code runs a factor of up to 8.9 faster than the Virgo Simulation code. The fastest version is the Improved Virgo Simulation code run on 32 processors on HPCx. The trends observed for the Virgo Simulation code when it is run with the 128k StB data set is also true when it is run with the 64k StB data set.



**Figure 4: Results on HPCx, IDRIS, RZG, BSC and HLRS for 16, 32, 64, 128 and 256 processors and run up to time step n=48 for the Virgo Simulation code and the Improved Virgo Simulation code, labelled FFTW, and the 128k StB data set.**

	RZG		HPCx	BSC	HLRS	
	n=48	N=357 (completion)	n=48	n=48	n=48	n=357 (completion)
<b>16</b>	42	286	22	42	42	335
<b>32</b>	23	175	14	26	21	154
<b>64</b>	13	95	15	19	15	92
<b>128</b>	13	86	53	19	15	69
<b>256</b>	21	141	-	38	-	-

**Table 10: The time in minutes needed on HPCx, RZG, BSC and HLRS to reach certain simulation time step n for 16, 32, 64, 128 and 256 processors for the 128k StB data set for the Improved Virgo Simulation code. The results on HLRS are from [27] and [28].**

Note that the Improved Virgo Simulation code needs a different number of time steps to reach completion, i.e. 178 (357) time steps compared to 159 (341) for the 64k (128k) StB data set of the Virgo Simulation code (compare with [4]). This can be expected since a different numerical algorithm is used and thus the simulation requires a different number of iterations to converge.

It is also important to be aware of the fact that we only investigate relatively small runs (i.e. data sets). For these small data sets the relaxation technique used to solve the Poisson equation in the Virgo Simulation code is reasonably fast but would

perform much less favourable when used with bigger data sets. Consequently, the results obtained for the Virgo Simulation code look much better compared to the results for the Improved Virgo Simulation code in our study than would be expected for larger data sets [34].

In addition, we provide the results for the Virgo Simulation code on HLRS from [27] and [28] for both data sets in Table 11 since they had not been available for [4]. All other results for the Virgo Simulation code can be found in [4].

	64k StB		128k StB
	n=41	n=174 (completion)	n=48
<b>8</b>	21	93	231
<b>16</b>	16	67	125
<b>32</b>	14	56	79
<b>64</b>	13	-	52
<b>128</b>	15	-	43

**Table 11: Time in minutes needed on HLRS to teach a certain simulation time step  $n$  for 8, 16, 32, 64 or 128 processors for the 64k and the 128k StB data sets for the Virgo Simulation code. These results are from [27] and [28].**

## 8 Profiling Study for the Improved Virgo Simulation Code

In this Section we present the profiling results obtained with the various profilers discussed in Section 4 for the Improved Virgo Simulation code.

Running up to time step  $n=25$  instead of completing the whole run (i.e. time step  $n=357$ ) still provides a representative use case for the code's behaviour as demonstrated in Section 5

The `grprof` result on RZG and BSC for the 128k StB data set run on 64 processors up to time step  $n=25$  are displayed in Table 12 and Table 13, respectively.

%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
14.2	48.26	48.26	52	928.08	953.21	.mapparticlestomesh [8]
7.0	71.94	23.68				._stripe_hal_newpkts [10] /usr/lib/liblapi_r.a
5.6	91.15	19.21	4056	4.74	4.74	.prolong_block_to_uniformgrid[13] multigrid.F90
5.4	109.44	18.29				._lapi_dispatcher [14] /usr/lib/liblapi_r.a
4.9	126.12	16.68				._lapi_shm_dispatcher [15] /usr/lib/liblapi_r.a
3.9	139.45	13.33	2288	5.83	5.83	.exchangedata [18] multigrid.F90
3.5	151.29	11.84	1	11840.00	11840.00	.initparticles [21] InitParticles.F90
3.2	162.21	10.92	3417600	0.00	0.00	.gravaccelonerow [20] GravAccelOneRow.F90
3.0	172.29	10.08				.REG_3stream_store [22] /usr/lib/liblapi_r.a
2.9	182.19	9.90	52	190.38	2747.32	.gravpotentialallblocks [4] GravPotentialAllBlocks.F90
2.6	191.19	9.00	1	9000.00	9000.00	.amr_initialize [23] amr_initialize.F90
2.5	199.79	8.60	52	165.38	293.56	.send_localregion_to_planes[16] multigrid.F90
2.5	208.26	8.47	52	162.88	291.06	.receive_localregion_from_planes[17] multigrid.F90
2.4	216.53	8.27	104	79.52	79.52	.pm_allocate_deallocate [26] multigrid.F90
2.3	224.49	7.96				.__mcount [28]
2.3	232.25	7.76	52	149.23	149.23	.density_to_potential [30] multigrid.F90
2.1	239.46	7.21	52	138.65	1603.67	.multigrid [5] multigrid.F90

**Table 12: gprof result for the 128k StB data set run on 64 processors up to time step n=25 on RZG for the Improved Virgo Simulation code. Routines which use 2% or more of the total run time or are displayed.**

The gprof result for BSC for the same setup as the one on RZG is shown in Table 13.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
26.70	416.41	416.41	52	8.01	8.15	.mapparticlestomesh
8.28	545.49	129.08	4056	0.03	0.03	.prolong_block_to_uniformgrid
6.29	643.59	98.10	341760	0.00	0.00	.gravaccelonerow
6.08	738.46	94.87	2288	0.04	0.04	.exchangedata
5.20	819.56	81.10	52	1.56	20.73	.gravpotentialallblocks
3.99	881.87	62.31	52	1.20	11.01	.Multigrid
3.74	940.25	58.38	52	1.12	1.12	.density_to_potential
3.61	996.56	56.31	52	1.08	2.00	.send_localregion_to_planes
3.50	1051.22	54.66	1950	0.03	0.03	.particletimestep
3.26	1102.02	50.80	52	0.98	1.89	.receive_localregion_from_planes
2.54	1141.58	39.56	1	39.56	39.56	.initparticles
2.51	1180.77	39.19				.fftwi_no_twiddle_32
2.46	1219.07	38.30				.fftw_no_twiddle_32
2.22	1253.76	34.69	605913	0.00	0.00	.meshtoparticle
2.22	1288.42	34.66	104	0.33	0.33	.pm_allocate_deallocate

**Table 13: gprof result for the 128k StB data set run on 64 processors up to simulation time step n=25 on BSC for the Improved Virgo Simulation code. Routines which use 2% or more of the total run time or are displayed.**

The subroutines ExchangeData, Prolong\_Block\_to\_UniformGrid, Density\_To\_Potential, Send\_LocalRegion\_To\_Planes and Receive\_LocalRegions\_From\_Planes showing in the gprof profiles for the Improved Virgo Simulation code are all part of the file multigrid.F90 written by [34]. Note, as previously stated, that the profile obtained via gprof on BSC does not include any of the overheads from using MPI communications.

The ftrace profiling result on HLRS obtained by [27] for the Improved Virgo Simulation code run with the 128k StB data set on 32 processors up to time step n=48 is shown in Table 14. We display the information about the most time consuming subroutine, how often it is called and how much exclusive time is spent in it. The total time in seconds is summarized over the number of processors used.

PROG.UNIT	FREQUENCY	EXCLUSIVE TIME[sec]( % )
moduleredistributeparticles.redistributeparticles	4736	12887.906( 18.8)
modulemapparticlestomesh.mapparticlestomesh	3136	11486.173( 16.8)
exchangedata	86240	8667.689( 12.7)
density_to_potential	3136	5879.909( 8.6)
checkpoint_wr	96	5249.113( 7.7)
dbasedatabypointer.dbasegetdataptrsinglblock	1028593686	2682.654( 3.9)
modulemeshtoparticle.meshtoparticle	603979776	2027.196( 3.0)
b_int_sendrcv	85600	1604.245( 2.3)
dbasetree.dbaseblocksize	1259175338	1515.348( 2.2)
total	12365526791	68498.830(100.0)

**Table 14: ftrace result for the 128k StB data set run on 32 processors up to simulation time step n=48 on HLRS for the Improved Virgo Simulation code. Routines which use 2% or more of the total run time are displayed. The results are from [27].**

The mpitrace results on RZG are shown in Table 15. A summary of the average time spent and message size for the MPI communications for both the Virgo Simulation code and the Improved Virgo Simulation code is shown in Table 16.

MPI Routine	#calls	avg. bytes	time(sec)
MPI_Comm_size	98	0.0	0.000
MPI_Comm_rank	96	0.0	0.000

MPI_Send	6238	3676.8	0.095
MPI_Ssend	71182	5628.0	16.663
MPI_Isend	56138	129474.8	2.177
MPI_Recv	19	4.0	0.000
MPI_Irecv	126205	61117.7	0.626
MPI_Sendrecv	8752	745265.1	73.423
MPI_Waitany	6964	0.0	5.791
MPI_Waitall	9427	0.0	0.607
MPI_Bcast	213	116.8	1.185
MPI_Barrier	4370	0.0	40.976
MPI_Gather	3	8.0	0.044
MPI_Scan	10	5.6	0.001
MPI_Allgather	589	5787.2	10.432
MPI_Reduce	26	88.0	0.028
MPI_Allreduce	832	29.1	3.605
-----			
total communication time	=	155.654 seconds.	
total elapsed time	=	453.830 seconds.	
user cpu time	=	378.721 seconds.	
system time	=	12.609 seconds.	
maximum memory size	=	668432 KBytes.	
-----			
Message size distributions:			
MPI_Send	#calls	avg. bytes	time(sec)
	5	4.0	0.000
	6	356.0	0.000
	6209	1632.0	0.022
	6	2136.0	0.000
	6	4806.0	0.000
	2	182272.0	0.065
	4	3098624.0	0.007
MPI_Ssend	#calls	avg. bytes	time(sec)
	344	4.0	0.004
	526	8.0	0.008
	638	12.3	0.224
	552	20.2	0.585
	1326	63.0	0.069
	2965	79.8	10.112
	1948	160.7	1.749
	864	396.0	0.186
	16224	2048.0	0.861
	18720	4096.0	0.781
	24960	8192.0	1.506
	1872	32768.0	0.351
	52	34816.0	0.014
	88	69632.0	0.140
	99	138637.6	0.070
	4	557056.0	0.003
MPI_Isend	#calls	avg. bytes	time(sec)
	33	4.0	0.000
	673	1632.0	0.005
	55432	131104.0	2.172
MPI_Recv	#calls	avg. bytes	time(sec)
	19	4.0	0.000
MPI_Irecv	#calls	avg. bytes	time(sec)
	336	4.0	0.001
	537	8.0	0.002

	626	64.0	0.002
	68	112.0	0.000
	18171	1947.7	0.055
	18720	4096.0	0.064
	24960	8192.0	0.052
	5220	8204.0	0.021
	1872	32768.0	0.012
	52	34816.0	0.000
	96	69632.0	0.000
	55545	131118.7	0.418
	2	557056.0	0.000
MPI_Sendrecv	#calls	avg. bytes	time(sec)
	832	0.0	17.614
	16	4.0	0.000
	6552	263168.0	29.523
	416	2097152.0	7.817
	936	4194304.0	18.470
MPI_Bcast	#calls	avg. bytes	time(sec)
	180	4.0	1.013
	25	20.0	0.171
	1	108.0	0.000
	1	212.0	0.000
	1	608.0	0.000
	1	2160.0	0.000
	4	5140.0	0.000
MPI_Gather	#calls	avg. bytes	time(sec)
	3	8.0	0.044
MPI_Scan	#calls	avg. bytes	time(sec)
	6	4.0	0.001
	4	8.0	0.000
MPI_Allgather	#calls	avg. bytes	time(sec)
	122	4.0	2.478
	104	12.0	1.080
	76	1800.0	0.049
	135	3956.4	4.367
	76	10800.0	0.250
	76	25200.0	2.209
MPI_Reduce	#calls	avg. bytes	time(sec)
	26	88.0	0.028
MPI_Allreduce	#calls	avg. bytes	time(sec)
	524	4.0	1.499
	216	8.0	0.694
	8	24.0	0.021
	83	218.4	1.381
	1	2048.0	0.010

Table 15: `mpitrace` result for processor 1 for the Improved Virgo Simulation code run up to simulation time step 25 for the 128k StB data set on 64 processors on RZG.

	Virgo Simulation		Improved Virgo Sim.	
	Mean	Stand. Dev.	Mean	Stand. Dev.
<b>Tot Comm Time [s]</b>	1778.5	117.3	165.3	25.2
<b>Tot Elap Time [s]</b>	3588.3	0.0	453.8	0.0
<b>User CPU Time [s]</b>	3388.2	14.1	374.3	6.1
<b>Sys Time [s]</b>	51.8	2.6	13.0	1.3
<b>Max Mem Size [Kb]</b>	741193	1446	645147	26952

**Table 16: Mean and standard deviation for the total communication time, elapsed time, user time and maximum memory size of the 128k StB data set on 64 processors on RZG for the Virgo Simulation code and the Improved Virgo Simulation code up to time step 25.**

While 49 % of the total run time was taken by communications for the Virgo Simulation code (see Table 7), this is only 34 % for the Improved Virgo Simulation code. For the code with the new algorithm the most time intensive MPI calls are `MPI_Sendrecv` (16 %), `MPI_Barrier` (9.1 %), `MPI_Ssend` (3.8 %) and `MPI_Allgather` (2.2 %). The `MPI_Sendrecv` call which dominates the communications, not previously used in the Virgo Simulation code, is found in the new subroutine `ExchangeData`, part of the file `multigrid.F90`. The most time consuming `MPI_Ssend` calls for messages of 2048, 4096 and 8192 bytes for the Virgo Simulation code are negligible now. Most time, i.e. 2.2 %, for `MPI_Ssend` calls is taken up for the transfer of 79.8 byte messages in the Improved Virgo Simulation code. The `mpitrace` tool on RZG does not provide any details about which of the Fortran routines call the MPI communications. The location of the `MPI_SendRecv`s is only known as they are exclusively called from the `ExchangeData` subroutine. The `Paraver` tool on BSC, discussed below, provides information about the calling subroutines.

The `Paraver` trace file on BSC, obtained by linking with the `mpitrace` library, produced a 754 Mb for the Improved Virgo Simulation code when run on 32 processors with the 64k StB data set up to time step  $n=25$ . The first step to use this file with `Paraver` was to generate a filtered trace that only include the very time intensive computational sections of the code larger than a given size, we chose 100ms, in order to analyze the load balance. The resulting trace file was only 256 Kb. Unfortunately, we could not use the 128k StB data set for this analysis as it produced too much data before the filtering stage which caused the `Paraver` tool to crash while loading the data.

According to the analysis of the Improved Virgo Simulation code run on BSC with `Paraver`, done by [13], the most time consuming MPI calls were:

- `MPI_Sendrecv` (13.6%), at line 1403 in the subroutine `exchangedata`,
- `MPI_Ssend` (4.23%),
  - at lines 187 and 194 in file `amr_guardcell_cc_c_to_f.F90`,
  - at line 434 in file `amr_refine_derefine.F90`
  - and line 268 in subroutine `amr_restrict_cc.f90` and
- `MPI_Allgather` (3.32%).
  - at lines 451 and 452 in the subroutine `pm_periodic` and
  - at line 74 in the file `ReDistributeParticles.F90`.

This is in accordance with our findings on RZG discussed above. There are two calls to `MPI_Sendrecv` in the new subroutine `ExchangeData` which account for this subroutine requiring a high portion of the code's total run time within the Improved Virgo Simulation code. The large computationally expensive sections of the code are identified to be between two calls to `MPI_Allgather`. The first one is in the file

`ReDistributeParticles.F90` (line 74) and the second one is in the file `MapParticlesToMesh.F90` (line 205). This is in accordance with the `gprof` result obtained on RZG when investigating the code line-by-line with the `Xprofiler`. These computationally expensive parts visible in the `Paraver` investigation can be related to copying arrays to boundary regions and zeroing arrays in `MapParticlesToMesh.F90`. We have no results from [27] on HLRS for the Improved Virgo Simulation code which give any information about MPI communication. The `ftrace` profiling results on HLRS from [27] also identify the file `ReDistributeParticles.F90` as the most time intensive one (18.8 % of the total run time), followed by `MapParticlesoMesh.F90` (see Table 14).

Looking into the most time intensive routines for the Improved Virgo Simulation code, we took multiplications out of a loop which stayed constant within the loop and were recalculated several times without the need for it and assigned the product of it to a temporary variable outside of the loop. This was done to the file `Prolong_Block_To_UniformGrid.F90` at line 3497 onwards for the multiplication of two `weight` variables. We tested this on RZG only. This procedure did not show any improvement in run time. It can be assumed that most compilers take the multiplication out of the loop automatically, anyway. This might now be true for all compilers, though, and could be worthwhile testing on the other platforms.

## 9 Code Migration using the DEISA Infrastructure

### 9.1 Overview

As discussed in [4], code migration allows simulations to migrate between DEISA platforms, which, considering the fact that these simulations take a long time to complete, is an important feature. This is so not only for large astrophysical simulations, but for any other type of large long-running simulations which require more time to complete than any DEISA site permits within the scope of a single batch job.

Here, we describe how we can start a FLASH simulation at a DEISA site and then complete the simulation at another, utilising elements of the DEISA infrastructure, including the use of UNICORE Workflows [5] or DESHL [5] as well as code built into FLASH that allows a restart file to be output before the job execution is scheduled to end on a given batch queue.

### 9.2 Restart Files

For any simulation to be able to undergo code migration, the code must be able to create a restart file. A restart file contains sufficient information for the same simulation to continue running from the point at which the restart file was created.

In FLASH terminology, a restart file is called a Checkpoint file<sup>9</sup>. These Checkpoint files are complete dumps of the entire simulation at intervals specified by the user. These can files can be created in various ways and are suitable for restarting the simulation.

FLASH provides a number of methods to produce Checkpoint files, all of which are independent of each other and can be used in conjunction with each other if required.

<sup>9</sup> FLASH also allows the creation of Plot files, which contain all the information required to interpret the AMR tree structure and includes a user-defined subset of the variables. The data is stored at reduced precision to conserve space.

The files can be created at fixed intervals of: wall-clock seconds, so-called redshit units or at a fixed number of simulation time steps. Another option is to create the file and terminate the simulation in terms, again, of wall-clock seconds, a redshit value or at a fixed number of time steps. This latter option is specified in the `flash.par` file. Lastly, if a file named `.dump_restart` appears in the output directory where the master processors is writing, then the FLASH code produces a checkpoint file and stops the simulation. This is useful if a user knows that the machine is being taken down or that the queue window is about to end so that they can manually signal FLASH to create a restart file and allow it to terminate cleanly and create this file.

### 9.3 Restarting FLASH

According to the FLASH manual [10], there is a restart script which enables the user to simply restart the simulation. However, this script was missing from the available FLASH distributions. For this reason we had to create our own restart bash script which simply adjusted the `flash.par` file to read a specified checkpoint file and set the `restart` variable to `.true..`

This was successfully tested on HPCx using the Virgo Simulation code running on 16 processors with the 64k StB data set. We submitted the simulation to HPCx and, after 5 minutes, created a `.dump_restart` file in the output directory. The simulation then created a restart file and finished cleanly. We then ran our script to create the necessary `flash.par` file, and resubmitted the job to the batch queue. It was shown that the results from restarting the simulations were identical to a simulation which ran to completion within a single batch job.

### 9.4 UNICORE Workflow

A UNICORE Workflow was created, which specified a job to start on HPCx and then complete at either IDRIS or SARA. This was done using the UNICORE Workflow manager client GUI, where each dependent task is connected to denote a dependence. So in this instance, task 2 will commence once task 1 completes.

To ensure that all of this works correctly, the user must place all the necessary files required to run FLASH in a particular directory, namely the UNICORE `USPACE` [5]. Uploading these files in advance of the batch job starting is known as staging the data. Thus the StB data set and the `flash.par` parameter file are placed in the `USPACE`. The executable is automatically imported into the `USPACE` by UNICORE. Note that it is advisable to always employ an executable which has been installed on the target platform where it has been ensured that the code hyperscales<sup>10</sup>. Thus, when employing code migration, we recommend only including platforms where the executable has been installed locally and thus is not staged.

A restart bash script was created to adjust the `flash.par` file to enable the executable to restart accordingly. This script is the second task to run within the UNICORE Workflow, which means that this script is executed as soon as the initial FLASH run completes. The second task is then run on the second platform. The files associated with the initial run, which resides within the `USPACE`, are automatically copied inside this second task's `USPACE`, since this script is considered as a new job by the Workflow manager. The data is thus transported to the second platform automatically.

After the restart script (job) finishes, and files have been automatically transferred to the `USPACE` of the second FLASH job, which restarts and completes the simulation. This `USPACE` can be on any other DEISA platform, preferably where the executable has been hyperscaled for that platform and is available locally.

---

<sup>10</sup> Here, we define hyperscale to signify that a code retains high parallel efficiency when executed on hundreds or thousands of processors.

Finally, after the simulation is complete, one final script (job) exports the data out of `USPACE` to `$DEISA_HOME` [5], for instance.

This UNICORE Workflow described above has been successfully executed using HPCx acting as both the first and second platforms and using HPCx and IDRIS as the first and second platforms respectively. Since HPCx is currently not part of the DEISA global file system, this latter experiment demonstrates that all of this works in the heterogeneous extended DEISA infrastructure.

We also attempted to use SARA as the second platform. As SARA is a big-endian platform, whereas HPCx is a little-endian platform, this would have demonstrated that data could be migrated within a heterogeneous cluster. However, due to a temporary incompatibility of the HDF5 libraries (see Section 3.2), the initial conditions data file, 64k, failed to load. It should also be noted that the execution of FLASH was not as straightforward as HPCx or IDRIS, as, at SARA, the HDF5 library module had to be loaded before the task could execute the simulation. So the line which simply launches the executable was replaced by a simple script which first loaded the module and then launched the executable.

## 9.5 DESHL

From a user's point of view, DESHL might be considered as being a command line interface version of UNICORE, however, this is a significant over simplification. One major difference between DESHL and the UNICORE Workflow process is that dependencies between tasks cannot be declared explicitly. Thus we have created a bash script, which runs on a local workstation, that manages the job submissions, job monitoring and staging of data, using the DESHL as it engine. Using this script, we have successfully started the Virgo Simulation at HPCx and completed it at IDRIS, where the initial input files and the final output files all reside at the local workstation.

The key stages of this script are now described in some detail.

1. **Stage the startup files in the HPCx USPACE.** The startup files, namely the 64k Stb data file and `flash.par` and the job description file, are staged in the `USPACE` at HPCx using the `deshl copy` command. As with the UNICORE example, the executable is already installed at the target platform. Note that, when employing the `deshl copy` command, wildcards can only be used when staging data into the `USPACE`. At present, wildcards cannot be employed when copying data out of `USPACE`. Therefore, we stage files via a bash script. This script creates a tar file of all the files to be staged, which is then moved to the target machine by DESHL and another script unpacks the tar file upon arrival. Indeed, this method may be faster than employing wildcards for a large number of large files.
2. **Submit the executable to the HPCx batch queue.** The executable is then submitted to HPCx on the local workstation using `deshl submit` command.
3. **Monitor job.** The job's status is then determined by repeatedly calling the `deshl status` command at a fixed interval. This is currently set to one minute. A `sed` operation is performed on the output to isolate the 'state', where the state can be `Running` (which actually means running or pending), `DoneOK` or `DoneFailed`.
4. **Copy restart file back to workstation.** Once the simulation has completed successfully, the data is taken back to the workstation from which the job was started and the script then updates the `flash.par` file accordingly.

5. **Stage the restart files in the IDRIS USPACE.** The second job description file, along with the restart data and the updated flash.par file, are then staged to the IDRIS USPACE using the tar script and `deshl copy`.
6. **Submit the executable to the IDRIS batch queue.** The executable is then submitted to run at IDRIS using the `deshl submit` command.
7. **Monitor job.** Again, the status is monitored by polling the IDRIS batch queues with repeated calls to the `deshl status` command.
8. **Copy final files back to workstation.** Once the simulation has completed successfully, the output files are taken from the IDRIS USPACE back to the local workstation.

If a broker is introduced, then this method could be employed to repeatedly submit a job to the next available platform, thus reducing the user's time-to-solution. Furthermore, if the requested batch queues are small, i.e. a small number of processors and/or a short length of time, this method could be used to soak up any spare cycles being offered as national services drain their queues when switching from different modes of operation, such as preparing for a capacity run or when switching from day to night modes. Thus, in the near future, a user could submit a very long simulation, at his workstation, which would then run on any number of remote HPC platforms, without the need for multiple passwords or data staging.

## 10 Memory Usage

### 10.1 Overview

We memory encountered problems when running the 256k StB data set with the Virgo Simulation code on HPCx on 64, 128 or 256 processors as reported in [4]. This may have been due to errors in the IBM Regatta MPI runtime libraries. Running the Virgo Simulation code with `maxblocks=675` and a stack limit of 200 Mb is successful on 128 processors when turning the `perfmon.F90` subroutine off. When run on 32 or 64 processors, the code is terminated because the value of `maxblocks` is too low to allow the code to run. Because of the hard limit on memory per processor available on HPCx which is discussed in detail below, the value for `maxblocks` cannot be raised too much over 675. We found this out relatively late in the project's life cycle which is why there was no time to obtain scaling results for the 256k StB data set for the Virgo Simulation code.

Unfortunately we encountered memory problems again when running the Improved Virgo Simulation code. Again, we turned the `perfmon.F90` subroutine off. The original Virgo Simulation code was, more or less, FLASH 2.5 which does not allocate much dynamic memory, if at all, while the Improved Virgo Simulation code does. When the code was run with the 256k data set on HPCx using 64 processors the code would call an internal abort function in the `multigrid.F90` file as the 3D array `IRegionSize`, in the subroutine `pm_allocate-deallocate`, could not be allocated successfully. The subroutine `pm_allocate-deallocate` is part of the new FFTW-based algorithm and thus is not found in the Virgo Simulation code. The abort happens during initialisation, at a very early stage of the code's run time, before the time evolution loop is started.

To make the Improved Virgo Simulation code with cosmologically reasonably large sized data sets suitable for all platforms of the DEISA infrastructure, it is of much interest to be aware of the exact memory requirements of the code.

In the following subsection we try to estimate the total static memory requirements for the code. This should give us an idea about the minimum number of processors

required to run the 256k StB data set on HPCx. In the following subsections we then use various memory profiling tools to test these estimates and to gain more information about the memory problems being encountered. A summary of our findings are presented in subsection 10.3.5.

## 10.2 Estimating the memory usage

The total amount of physical memory that can be occupied by a process on HPCx is  $26.96 \text{ Gb} / \text{tasks\_per\_node}^{11}$  [15], where a full node consists of 16 tasks. This is the total amount of real memory which can be occupied by a process and can be divided into two areas: stack and data. If the `stack_limit` and the `data_limit` exceed this amount of memory the job will fail [16]. There is no virtual memory available on HPCx, i.e. there is no swapping to disk, as there is for example on RZG.

The total amount of physical memory available needs to be compared with the actual memory requirement of the Improved Virgo Simulation code.

The static memory usage for the code can be estimated using the `size` utility – as it has been stated, the original FLASH code does not allocate much memory so this is a good indicator as to whether the code will run, or not, within the memory of a given processor. Thus, the memory requirement for a FLASH code executable, `flash2`, can be determined using:

```
size -X64 -f flash2
```

where `-X64` signifies that a 64-bit object needs to be inspected and the `-f` asks for the section names of the object to be written in parenthesis following the section size and the object to be inspected, `flash2`, is at the end. The results are returned in bytes. We now set the maximum number of blocks per processor, `maxblocks`, to 675 for the 256k StB data set. In general, larger data sets require a higher value of `maxblocks`. We get:

```
4281664(.text) + 352096(.data) + 686790736(.bss) + 70186(.loader) +
463914(.debug) + 1110(.except) = 691959706
```

So in this instance the total memory of the executable is about 692 Mb. Note that the segments: `text`, `data`, `bss`, `loader`, `debug` and `except` are explicitly named<sup>12</sup>.

However, in the Improved Virgo Simulation code, memory is also allocated which needs to be taken into account. This additional memory usage, due to memory allocations within the code, is made up of two parts which, according to [34], consist of: the memory required by the FFT and the memory required for the particles which, when added together in addition to the static memory obtained from the `size` utility,

<sup>11</sup> This is a LoadLeveller term that equates, more or less, to the number of processes run on a node – usually set to be no greater than the number of processors in a node.

<sup>12</sup> The `text` segment contains the program instructions and is read only when loaded on to memory. The `data` segment contains the initialized global and static variables, complete with their assigned values. The `bss` (block started by symbol) segment only holds variables that do not have any values yet - it does not take up any actual space in the object file. The `loader` section contains information needed to dynamically load a module into memory for execution. The `debug` section contains the symbolic debugger information and provides symbol attribute information for use by a symbolic debugger. The `except` segment provides information that allows the reason that a trap or exception occurred within an executable object program to be identified.

give an indication of the total memory requirement for the code. The values used for each variable in our example can be found in Table 17. The variables are as defined in [10].

1. For the FFT grid in 3 dimensions, the number of bytes required for each cell is given by:

$$(2 * nxb * 2^{(\max\_pm\_level-1)})^3 * 8 / \#processors$$

where `nxb` is the number of interior cells in each block in the x direction (note that this is cubed to take into account the three dimensions – this implicitly assumes that the number of interior cells is the same in all three Cartesian directions), the 8 signifies the size of a double precision number in bytes on HPCx and `#processors` is the number of processors. Thus the memory requirement for the 256k StB data set with `max_pm_level=8` is 134Mb on 512 or 67Mb on 1024 processors.

Variable	Value
<code>maxblocks</code>	675
<code>max_pm_level</code>	8
<code>nxb</code>	8
<code>int*4 (bytes)</code>	4
<code>Real*8 (bytes)</code>	8
<code>FLASH_NUMBER_OF_INT_PARTICLE_PROPS</code>	3
<code>FLASH_NUMBER_OF_REAL_PARTICLE_PROPS</code>	14
<code>MaxParticlesPerProc</code>	1,800,000

**Table 17: Typical values for the variables used to calculate the memory requirements for the Improved Virgo Simulation code.**

2. Particle data: The memory per particle is determined by the variables `FLASH_NUMBER_OF_INT_PARTICLE_PROPS` and `FLASH_NUMBER_OF_REAL_PARTICLE_PROPS` which specify how many `int*4` and `real*8` attributes per particle there are. These two variables are defined in the file `flash-defines.fh`. The maximum number of particles per processor is set through the variable `MaxParticlesPerProc` in the `flash.par` file. The memory needs for the particle data are given by:

$$\text{MaxParticlesPerProc} * (\text{FLASH\_NUMBER\_OF\_INT\_PARTICLE\_PROPS} * \text{int}4 + \text{FLASH\_NUMBER\_OF\_REAL\_PARTICLE\_PROPS} * \text{real}8)$$

which, in our case is 223.2 Mb.

To summarise, the static memory requirement is 692 Mb determined using the `size` utility, the 3D FFT grid requires 134Mb when the code is run on 512 processors or 67Mb when 1024 processors are used and the particle data require 223 Mb. Adding the two allocatable memory requirements and the static requirement the Improved Virgo Simulation code will need 1.05 Gb per processor when run on 512 processors or 1.45 Gb when run on 128. This shows that, taking into account a `stack_limit` set to 200 Mb and thus not available as direct storage, the code when using the StB

256k data set cannot possibly be run on less than 128 processors with filled nodes, i.e. where 16 tasks are available per node which corresponds to 1.685 Mb of memory per processor. In addition to this, the master processor allocates more memory than all the other processors as, when serial HDF5 is used, it has to coordinate the I/O from the other processors and it will have to allocate additional memory for buffering this information.

The next subsection we see whether this estimate is consistent with actual run time behaviour by using a number of memory profiling tools.

### 10.3 Memory Profiling Tools and Strategies

In order to find out what the actual memory usage is and whether this corresponds to the estimates made in the previous section, it would be good to monitor how memory is being used during a simulation using some kind of memory profiling tool. However, the tools available to investigate memory usage on HPCx are somewhat limited. We nevertheless tried to use the following tools and strategies to gain more insight in to how the memory was actually used at run time:

1. The `vmstat` command.
2. Running for the same number of processors but doubling the number of nodes to get access to twice the amount of memory.
3. The `ulimit` command.
4. The `libhmd.a` library.

The above tools are discussed in the following subsections. However, in the study that follows an incorrect assumption was applied about the total memory requirements previously described – one of the formulas did not take into account all three dimensions. As a consequence of this, the number of processors used for this, i.e. 64, was too small to cater for the allocated sections of memory so the code failed already at the data initialisation stage. There was insufficient time to repeat these experiments. Nevertheless, we can still use the following study to show consistency with the estimated memory requirements.

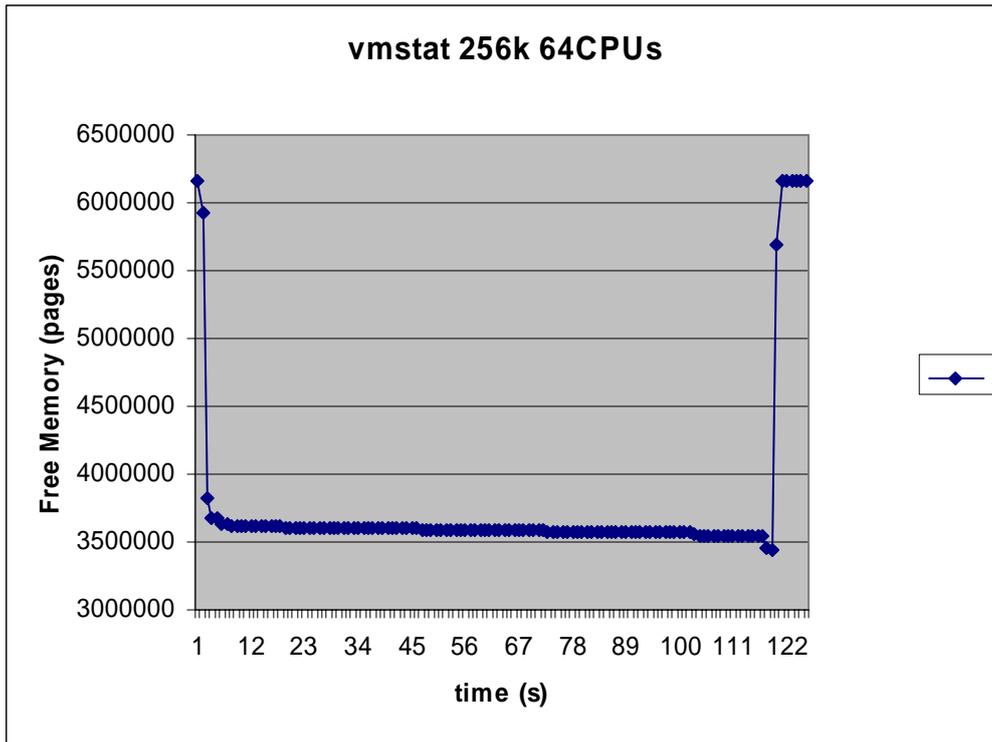
#### 10.3.1 Using `vmstat`

The `vmstat` command reports virtual memory statistics across one node, the master node. It provides information about kernel threads, virtual memory, disks, traps and CPU activity. Although there is no virtual memory available within HPCx we can use this command to determine the amount of free memory available within a node (consisting of 16 processors).

We added a line to the batch script before the `poe` command, used to run MPI jobs on HPCx, is executed:

```
vmstat 2 >> limit.log &
```

The above expression makes the `vmstat` command execute every two seconds for the run time of the batch job and writes the output to a logfile, `limit.log`. We are interested in the free memory left on a node. The result is in units of pages, where one page consists of 4096 bytes. The result for the 256k StB data set on 64 processors on HPCx run until failure is shown in Figure 5.



**Figure 5:** `vmstat` result for the master node for free memory of The Improved Virgo Simulation code with the 256k StB data set run on 64 processors on HPCx.

It can be seen that once the code starts running 2.5 MegaPages (MPages, where 1 MegaPage is  $10^6$  Pages) of memory is used as the free amount of memory drops from about 6.1 MPages to 3.6 MPages.

This amount is gradually going down to 3.4 MPages which suggest that the code has a memory leak. This possible memory leak was not investigated further due to lack of time.

The 6.1 MPages available per node, as seen in the graph as starting point, correspond to 1.58 Gb per processor, whereas the total amount available should be 1.68 Gb. The drop in free memory to 3.6 MPages equals a memory requirement of 634 Mb per processor which corresponds to about the static size of the code (693 Mb). There is no indication that a huge amount of memory is allocated during run time which would account for a memory failure in the code, though. Before the code crashes, the code appears to free all of its used memory and returns to the initial value of 6.1 MPages. There is no further drop beyond 3.4 MPages which would indicate a request of more memory than could be allocated per processor. We have not found the reason for the memory leak. The leak shows slow and gradual leaking which does not seem to be eminent enough to be of concern to provoke code failure.

### 10.3.2 Using half the number of processors available in a node

One strategy to double the amount of memory available to a processor is to use half the number of processors available in a node but it's wasteful in so far that you request more processors than you actually use. To use double the amount of nodes we add to the batch script:

```
#@ cpus = 64
#@ tasks_per_node = 8
```

Then 8 processors –or tasks- per node are used instead of 16 which is the possible upper limit. These 8 processors can access the memory of 16 processors now, i.e.

instead of 1.685 Mb per processor there is 3.37 Mb of memory per processor available.

However, despite the fact that the amount of memory available has been doubled the code still fails, but at a later stage, specifically, just after entering the time evolution loop with an error message which also indicated lack of sufficient memory in the past.

### 10.3.3 Using `ulimit`

The `ulimit` is a command that sets or reports all process resource limits set at the shell level. We use it to check our available memory resources. The command `ulimit -a` reports what the current limits are set to as shown in Table 18. We add this to the batch script as well and get when run on half-full nodes as shown in Table 18:

<code>time(seconds)</code>	<code>unlimited</code>
<code>file(blocks)</code>	<code>unlimited</code>
<code>data(kbytes)</code>	<code>3431302</code>
<code>stack(kbytes)</code>	<code>102400<sup>13</sup></code>
<code>memory(kbytes)</code>	<code>3533702</code>
<code>coredump(blocks)</code>	<code>67108864</code>
<code>nofiles(descriptors)</code>	<code>2000</code>

**Table 18: Result for `ulimit` used on the Improved Virgo Simulation code and the 256k StB dat set run on 64 processors on 8 nodes on HPCx.**

We set the stack limit to 100 Mb<sup>14</sup> in the batch script. This corresponds to the stack given by `ulimit`. When running on half-full nodes the memory per processor should be 3.37 Gb, here it is 3.5 Gb (see Table 18). The sum of stack and data is the total amount of memory available per processor which is a hard limit that cannot be exceeded. Otherwise the code will fail. The `ulimit` output confirms our expectations on available resources.

### 10.3.4 Using `libhmd.a`

The `libhmd.a` library provides debug versions of memory-management routines which provide a summary of the memory that is allocated when a program runs. This will allow us to reconcile the estimates of dynamic memory usage to what is actually used by the program. Calls to the `_dump_allocated_delta()` subroutine (see [38]) from the subroutine `pm_allocate_deallocate` in the file `multigrid.F90` print information to `stderr` about each memory block that is currently allocated or was allocated. A perl script has been developed on HPCx to help analyse the output [39]. The script can only correctly analyze output from a single process. To enable the analysis of parallel jobs export the following environment variables in the batch script.

```
export MP_STDOUTMODE=ordered
export MP_LABELIO=yes
```

<sup>13</sup> A kilobyte is 1,024 bytes.

<sup>14</sup> To be precise, the sizes provided in the LoadLeveller batch script on HPCx are MebiByte rather than MegaByte which is 2<sup>20</sup> bytes. Hence 100 MebiByte correspond to 102,400 KByte or 104,857,600 Byte, respectively.

The output provides a complete list of all arrays that have been allocated and where they have been allocated. The total number of bytes allocated in this program might vary from processor to processor. Listing only the arrays which need 1Mb or over ordered by size, we get different results for processor 0 (see Table 19) and the other processors (checked for processors 1, 12, 23 and 50, see Table 20) which we picked at random. These are results when using only half-filled nodes.

```

Memory Allocations (ordered by size) :

1. 230400000 bytes allocated in :
   checkpoint_wr.F90 : allocate (particlest(MaxParticlesPerProc),
stat=ierr)
   from output_initial.F90
   from init_flash.F90
   from flash.F90

2. 230400000 bytes allocated in :
   InitParticles.F90 : allocate (particles(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90
   from flash.F90

3. 21600000 bytes allocated in :
   InitParticles.F90 : allocate
(part_send_list(MaxParticlesPerProc), stat=ierr)
   from init_flash.F90
   from flash.F90

4. 7200000 bytes allocated in :
   InitParticles.F90 : allocate (is_empty(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90
   from flash.F90

5. 7200000 bytes allocated in :
   InitParticles.F90 : allocate (empty_stack(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90
   from flash.F90

6. 1048600 bytes allocated in :
   h5_write.c : status = H5Dwrite(dataset, H5T_NATIVE_DOUBLE,
memspace, dataspace,
   from checkpoint_wr.F90
   from output_initial.F90

Total Number of Bytes listed above = 497848600

=====
Total Number of Bytes Allocated in Program = 498934031
=====
=====

```

**Table 19: Array allocations over 1Mb for the Improved Virgo Simulation code with the 256k StB data set on HPCx instrumented with the libhmd.a library for processor 0.**

```

Memory Allocations (ordered by size) :
*****

1. 230400000 bytes allocated in :
   InitParticles.F90 : allocate (particles(MaxParticlesPerProc),
stat=ierr)
   from init_flash.
   from flash.F90

2. 21600000 bytes allocated in :
   InitParticles.F90 : allocate
(part_send_list(MaxParticlesPerProc), stat=ierr)
   from init_flash.F90
   from flash.F90

3. 7200000 bytes allocated in :
   InitParticles.F90 : allocate (is_empty(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90
   from flash.F90

4. 7200000 bytes allocated in :
   InitParticles.F90 : allocate (empty_stack(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90
   from flash.F90

Total Number of Bytes listed above = 266400000

=====
Total Number of Bytes Allocated in Program = 266582916
=====
=====

```

**Table 20: Array allocations over 1Mb for The Improved Virgo Simulation code with the 256k StB data set on HPCx instrumented with the libhmd.a library, tested for processors 1, 12, 23 and 50.**

Comparing the main memory allocations between processor 0 and the others, it becomes clear that processor 0 has to handle two more major memory allocations which are related to the HDF5-output files. The code seems to be able to allocate these arrays though, otherwise a built-in check and abort routine would have aborted the code with an appropriate error message. The size of each checkpoint file once written to disk is 5.1 Gb. The code runs successfully for the 256k StB data set on 64 processors with half-filled nodes and a stack limit of 200 Mb when the files responsible for output, i.e. the checkpoint file `checkpoint_wr.F90` and plot file `plotfile.F90` are turned off. This shows clearly that the allocation of temporary arrays, its memory consumption and the impact on successful execution of the code must not be underestimated, especially when dealing with large data sets.

According to our estimation in Section 10.2, the Improved Virgo Simulation code run on 64 processors requires 1.99 Gb. Half-filled nodes offer a memory 3.37 Gb per processor. We set the stack limit to 200 Mb. It is not quite clear why the run on 64 processors with half-filled nodes should fail. But as stated in [10] this is not the total amount of memory used by the code, since fluxes, temporary variables, and coordinate information also consume a large amount of memory. As this study showed above, the amount of memory required for temporary variables linked to HDF5 output already consume 231 Mb on the master processor. This is a very likely explanation for the failure.

FLASH allows using parallel HDF5 output as well. The setup script needs to be run with the option `-with-module=io/amr/hdf5_parallel`. Then a single HDF5 file is created, with each processor writing its data to the same file concurrently.

Employing the parallel version of HDF5 did not stop the code from running out of memory for a run on 64 processors, though. When parallel HDF5 is used for output the memory allocated is distributed evenly across processors, see Table 21, but the code fails again. Now the memory allocated within the file `checkpoint_wr.F90` is 5.12 Kb on each processor instead of 230 Mb on the master processor only. Further, we currently employ HDF5 v1.4.4. Employing v1.6.4 may also solve this memory issue. The last suggested solution is not investigated due to lack of time.

```

Memory Allocations (ordered by size) :
*****

1. 230400000 bytes allocated in :
   InitParticles.F90 : allocate (particles(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90
   from flash.F90

2. 21600000 bytes allocated in :
   InitParticles.F90 : allocate
(part_send_list(MaxParticlesPerProc), stat=ierr)
   from init_flash.F90
   from flash.F90

3. 7200000 bytes allocated in :
   InitParticles.F90 : allocate (is_empty(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90 :
   from flash.F90

4. 7200000 bytes allocated in :
   InitParticles.F90 : allocate (empty_stack(MaxParticlesPerProc),
stat=ierr)
   from init_flash.F90
   from flash.F90

5. 1048600 bytes allocated in :
   h5_parallel_write.c : status = H5Dwrite(dataset,
H5T_NATIVE_DOUBLE, memspace, dataspace
   from checkpoint_wr.F90 : call MPI_SEND(lnblocks, 1,
MPI_INTEGER, 0, &
   from output_initial.F90

Total Number of Bytes listed above = 267448600

=====
Total Number of Bytes Allocated in Program = 268127503
=====

```

**Table 21: Array allocations over 1Mb for the Improved Virgo Simulation code with the 256k StB data set on HPCx instrumented with the `libhmd.a` library, tested for processors 0, 10 and 20 and run with parallel HDF5 for I/O.**

The result from the investigation of the 256k StB data set run on 64 processors on HPCx is that it is too big to run on this number of processors. When we chose 64 processors for our memory investigation our estimate for the code's memory

requirements were wrong, as mentioned earlier. This is in accordance with our result. Unfortunately, there has not been any time left to investigate the run on 128 processors with parallel HDF5 which should allow the Improved Virgo Simulation code to run successfully on HPCx. The Virgo Simulation code did not have the additional requirement for the 3D FFT grid which explains why it can be run on 32 processors since the Improved Virgo Simulation code requires an additional 2.15 Gb of memory for the FFT grid when run on 32 processors.

### 10.3.5 Summary of Findings

The estimate for the memory requirements of the Improved Virgo Simulation code presented in Section 10.2 shows that 128 or more processors would need to be used in order to be able to run the 256k StB data set on HPCx as the memory per processor is limited to 1.685 Gb on fully populated nodes. The Improved Virgo Simulation code requires more memory than the Virgo Simulation code due to the 3D FFT grid employed which is allocated dynamically hence the use of the size utility is no longer sufficient to determine whether the application will run within a given memory limit. However, as the memory is dynamically allocated this additional memory requirement is inversely proportional to the number of processor used.

The Virgo Simulation code can be run on HPCx on 128 processors now, as IBM Regatta MPI run time libraries have been upgraded, with the `perfmom.F90` subroutine turned off and `maxblocks=675`; according to our memory estimate this code requires below 1 Gb of memory per processor. For any number of processors below 128, the code terminates because these choices for `maxblocks` prove to be too small<sup>15</sup>. We also ran the Improved Virgo simulation on 128 processors but it still failed with the same error message as it would for a smaller number of processors. This might be due to the additional memory requirement of 134 Mb for the FFT grid over the Virgo Simulation code, and there are still large temporary arrays which need to be allocated and which become relatively large for large input files, e.g. the serial HDF5 output related temporary arrays on the master processor consume over 230Mb. This is supported by the finding that the Improved Virgo Simulation code does run successfully on 64 processors with half-filled nodes when the output related temporary allocation of arrays is turned off. The Improved Virgo Simulation producing output checkpoint and plot files seems to require a minimum of 256 processors. Another option is to increase the `maxblock` size which is also not straight forward on HPCx due to the limited amount of memory per processor as an increase in this variable increases the memory required per processor. There is a message in the `amr_log` file reporting an error but not terminating the execution of the problem and recommending the value of `maxblocks` to be increased or to use a larger number of processors or to change the refinement criterion<sup>16</sup>. A value of `maxblocks=1000`, increases the static size of the executable from 692 Mb to 1005 Mb and the total memory requirements to a minimum of 1.496Mb for 256 processors. This value will already exceed the available memory per processor, when a stack limit of 200 Mb is set as usual. The total memory requirement decreases to 1.362 Mb for 512 processors.

Due to initially incorrect calculations of the dynamic memory requirements, we investigated the memory issue using 64 processors only which clearly are inadequate. This study was still useful demonstrating the memory requirements of the code and also revealing that the large amount of temporary memory due to output related allocations determines success or failure of a run. The investigation

<sup>15</sup> This error message can be found in the logfile written during program execution.

<sup>16</sup> We use `lrefine_min = 6`, `lrefine_max = 8` as stated before and recommend by [34] for a realistic simulation setup.

with the `vmstat` tool did not help to detect a high drop in available memory which would account for the code calling its internal function to terminate the further execution of the code. The use of the `ulimit` command confirmed that the memory available per processor was as what we expected. The `libhmd.a` library identified the most allocation-intensive calls to dynamically create arrays before program failure. The code fails already at an early stage when the data is being initialized.

Further test runs are needed, including the choice for an optimal value for the maximum number of blocks per processor. To confirm, if 512 processors are used the 256k StB data set with the Improved Virgo Simulation code will run. The current results all indicate that a relatively large number of processors, i.e. 512 or more, are needed in order to allow the simulation to run.

## 11 Discussion of Results and Future Work

The work done by JRA2 enables the cosmologists of the Virgo Consortium to run their simulations on different platforms and the DEISA infrastructure enables them to produce scientific results exceeding those that are possible at their home installation.

Three versions of the cosmological FLASH<sup>17</sup> simulation code FLASH version 2.5 have been ported to various platforms within the DEISA infrastructure. The Virgo Consortium provided three input data sets whose difference is the numbers of particles considered, i.e. 64k, 128k and 256k, representing dark matter only simulations. These data sets are referred to as the Santa Barbara (StB) data sets and are used to test the code.

In addition to what had been achieved and discussed in the first report:

1. The Virgo Simulation code has now been ported to HLRS [27],
2. The Virgo Simulation code has been profiled with various profiling tools on RZG and BSC (and HLRS[27]) to determine the expensive sections of the code,
3. Optimization strategies have been investigated according to the outcome of the profiling studies performed,
4. A new algorithm has been employed to replace the multigrid solver within the Poisson equation by an FFT-based spectral procedure (Improved Virgo Simulation code) based on the outcome of the profiling study.
5. The Improved Virgo Simulation code has been ported to HPCx, RZG, BSC, IDRIS and HLRS and profiled on RZG, BSC and HLRS. NB: the porting and profiling study on HLRS has been executed by [27]. The Profiling study of the Improved Virgo Simulation code showed that the do-loops around line 3497 in the file `Prolong_Block_To_UniformGrid.F90` are very time intensive. We took several multiplications out of this loop without gaining a speed-up on RZG. Probably, the compiler already took care of this procedure by itself.
6. We ran a scaling study for both the Virgo Simulation code and the Improved Virgo Simulation code for both the 64k and the 128k StB data set.
7. Memory usage of the Improved Virgo Simulation code has been investigated on HPCx as the code run with the 256k StB data set turned out to fail during run time due to a lack of sufficient memory. Higher values for the maximum number of blocks per processor, `maxblocks`, and higher numbers of processors need to be tested. Definitely the Improved Virgo Simulation code

<sup>17</sup> The FLASH code as it comes from the Chicago webpage with the Sedov setup, the Virgo Simulation code and the Improved Virgo Simulation code. The latter two being extensions by the Virgo Consortium.

as it stands with its increased memory demands due to the introduction of the FFT based algorithm poses a challenge to systems with a hard limit on available memory per processor, such as HPCx or IDRIS.

8. The code migration discussed in [4] has been successfully implemented.

The profiling study of the Virgo Simulation code revealed two major time intensive parts of the code:

- The most time intensive routine is the Poisson solver. The time intensive lines within the source code cannot be re-written in any obvious way to avoid this but a major re-write of this part of the code and/or a new algorithm has to be envisaged, which motivated the introduction of the FFT version of the code.
- Several MPI communications take up to 50% of the total run time of the code. We tried to replace `MPI_Ssends` by `MPI_sends` which allows the underlying hardware to decide of how to send the messages, i.e. with or without buffering. This strategy did not improve the run time of the code, though, on the platform it was tested on, i.e. RZG.

The Improved Virgo Simulation code does run up to a factor of 8.9 faster than the original Virgo Simulation code (the 128k StB data set on 32 processors on RZG). As for the Virgo Simulation code, HPCx is fastest of all machines used for the Improved Virgo Simulation code with the 128k StB data set on up to 32 processors. For the 64k StB data set RZG is faster, followed by HPCx. HLRS is faster than HPCx for a small number of processors, i.e. up to 8. This suggests that computations are executed faster on HLRS, but communication costs become higher quickly, where HPCx is more efficient.

The FLASH code, including Virgo's extensions and the new algorithm, has been ported to various European HPC platforms. There is an issue, though, with running the Improved Virgo Simulation code for realistic large data sets (256k StB or over) on machines, where no virtual memory can be used, such as HPCx. The Improved Virgo Simulation code cannot be run for large cosmological input data sets on platforms which do not allow for swapping to disk on less than 512 processors or probably even require a higher number of processors as a future study would have to prove.

Future work should look into:

- An investigation why the code produces wrong results on SARA.
- Refactor the MPI communications which take up to 50% of the code's total run time:
  - `MPI_SendRecv` in file `multigrid.F90` line 1403,
  - `MPI_Ssend` in files
    - `amr_guardcell_cc_c_to_f.F90` lines 187 and 194,
    - `amr_refine_derefine.F90` line 434
    - `amr_restrict_cc.F90` line 268.
  - `MPI_Allgather` in
    - `multigrid.F90` lines 451 and 452,
    - `ReDistributeParticles.F90` line 74.

Sets of point-to point communications might benefit by a replacement with global communications. Synchronous sends could, depending on the architectures used, benefit from a replacement with standard sends.

- Consider a re-write time intensive subroutines.
- The memory issues encountered for the Improved Virgo Simulation code with the 256k StB data set on HPCx pose severe restrictions onto the employability of large cosmological simulations on platforms without virtual

memory. One solution could be the usage of parallel HDF5 to avoid the allocation of large temporary arrays. This assumption is supported by the finding that the Improved Virgo Simulation code ran on 64 processors with half-filled nodes once the output file facility was turned off completely.

- Switch to the latest version of HDF5.
- Investigate FFT packages more appropriate for vector machines for HLRS. We were surprised to find how well the Improved Virgo Simulation code performed compared to the other non-vector platforms despite of using FFTW.

The Virgo Simulation code and the Improved Virgo Simulation code were both ported onto four DEISA platforms, i.e. HPCx, RZG, BSC and HLRS.

A good speed-up of up to a factor of 8.9 could be achieved over the Virgo Simulation code by replacing the multi grid based Poisson solver by a spectral FFT based Poisson solver. This result can possibly still be improved if larger data sets are investigated where the benefit of the FFTW would show even better.

It was the first time any FLASH code had been ported to a vector machine, specifically the NEC SX-8 at HLRS. Despite of the FFTW package being specially optimized for cache-based architectures, the Improved Virgo Simulation code competed very well on HLRS compared with the other architectures, and the performance of FFTW was not the limiting factor on the SX-8.

Code migration has been introduced to FLASH, allowing long astrophysical simulations to migrate between DEISA platforms. Large FLASH simulations, which require more time to run than any DEISA site permits within a single batch job, can now run to completion, seamlessly utilising several of the DEISA platforms if required. This code migration is not possible without key elements of the DEISA infrastructure, namely either the UNICORE Workflow manager or the DESHL.

Finally, using code with large StB data sets proved to be a challenge in those DEISA platforms where no virtual memory is available, such as HPCx.

## 12 Appendix: General Machine Information

This Appendix provides an overview of the machines available through DEISA plus the ICC platform that were used for this project. These details were correct for April, 2006.

### 12.1 HPCx

The HPCx [12] service is the UK science community's newest capability computing service. It is operated by EPCC and CCLRC on behalf of the UK Engineering and Physical Sciences Research Council (EPSRC).

The HPCx system consists of 96 IBM eServer 575 LPARs as compute nodes and 2 IBM eServer 575 frames as login and disk I/O nodes. Each eServerframe contains 16 processors, the maximum allowed by the hardware. The service offers a total of 1536 processors for computation.

The eServer 575 compute nodes utilise IBM Power5 processors. The Power5 is a 64-bit RISC processor implementing the PowerPC instruction set architecture. The processors use a 1.5 GHz clock rate. The chips are packaged into a Multi-Chip Module containing 4 chips, i.e. 8 processors. Each eServer node contains 8 chips (16 processors) and has 32 Gbs of main memory.

Inter node communication is provided by an IBM's High Performance Switch (HPS).

### 12.2 SARA

SARA Computing and Networking Services is an advanced ICT service centre that has supplied - for more than 30 years - a complete package of high-performance computing & visualization, high-performance networking and infrastructure services. Among SARA's customers are the business community and scientific, educational, and government institutions.

SARA is an independent organization [32] located in the Netherlands. The platform Aster is an SGI Altix 3700 system, consisting of 416 CPUs (Intel Itanium 2, 1,3 GHz, 3 Mb cache each), 832 Gb of memory and 2.8 Tb of scratch disk space. The total peak performance is 2.2 Teraflop/sec.

The 416 processors are divided over 5 nodes: 4 batch nodes and 1 interactive node. It is a little-endian machine, unlike every other platform in DEISA, which are big-endian machines. Every node in Aster is a CC-NUMA machine. The term CC-NUMA stands for Cache-Coherent Non Uniform Memory Access. In the CC-NUMA model, the system runs one operating system and shows only a single memory image to the user even though the memory is physically distributed over the processors.

### 12.3 IDRIS

Created in 1993, IDRIS (Institute for Development and Resources in Intensive Scientific computing) is CNRS's national supercomputing centre. Together with a second national supercomputing centre - CINES at Montpellier, funded by the Universities Ministry - IDRIS integrates the high-end national supercomputing infrastructures which provide CNRS and Universities research laboratories the ultimate high performance computing resources and services they may need.

In order to cope with the most demanding high performance computing requirements of the French scientific community, IDRIS and CINES deploy advanced

supercomputing environments, and assist scientists through their Users Support teams.

The service at IDRIS [5], zahir, is an IBM eServer p690 (Regatta Power4) consisting of:

- 8 SMP nodes with 32 Power4 P690 processors (256 processors, 832 Gb of memory).
- 12 SMP nodes with 32 Power4 P690+ processors (384 processors, 1,536Tb of memory).
- 6 clusters with 16 nodes with 4 Power4 P655 processors (384 processors, 768 Gb of memory).

Inter node communication is provided by an IBM's High Performance Switch (HPS).

#### **12.4 RZG**

Since 1992 the Rechenzentrum Garching (RZG) is operating as a common computer centre of the Max Planck Institute for Plasma physics and the Max Planck Society, offering services for Max Planck Institutes all over Germany. For four decades now the key tasks provided by the RZG have included: Supercomputing, data management/mass storage and data acquisition.

The RZG [2] has an IBM pSeries Supercomputer, namely the "Regatta" nodes with Power4 processors. There are now 25 compute nodes and 2 I/O nodes, connected by the HPS, or High Performance Switch, with four links per Regatta node:

- Processor clock: 1.3 GHz.
- Peak performance per proc: 5.2 GFlop/s.
- Processors per compute node: 32.
- Main memory of the compute nodes: 23 x 64 GB, 2 x 256 GB.
- Total number of nodes: 27.
- Total number of processors: 812.
- Aggregated total peak performance 4.2 TFlop/s.
- Total main memory: 2 Tb.

#### **12.5 BSC**

BSC (Barcelona Supercomputing Center) is the National Supercomputer Facility in Spain. Established in 2005, it has inherited the tradition of the well-known CEPBA, Institute in Parallel Computing in Europe, also including MareNostrum, the most powerful Supercomputer in Europe at the time of writing this report, and the 8th in the world, according to November 2005 top500 list.

MareNostrum, which consists of 4564 POWER PC 970 processors operating under the IBM Linux OS system with 9Tb of memory space and 233 Tb of disk space. The peak performance of the machine as a whole is 40 Teraflops. There are 2 CPUs per node. Four switch frames with Myrinet, including 10 CLOS 256+256 switches and 2 Spine 1280s and densely bundled Myrinet cabling enables faster parallel processing with less switching hardware.

#### **12.6 HLRS**

The High Performance Computing Center (Höchstleistungsrechenzentrum) Stuttgart (HLRS) of the University of Stuttgart supports users from research and development in the use of leading edge supercomputer technology and its applications. The mission of HLRS is to provide its users with systems, tools, and expertise to achieve top international positions in their research field. Services are

provided in collaboration with other partners in the Center for Competence in High Performance Computing of the State of Baden-Württemberg.

The HLRS has an NEC SX8 vector supercomputer with 576 processors, 9.2 Tb memory space and 180 Tb disk space. The peak performance is 12.67 Teraflops. The backend systems are 72 SX-8 vector nodes, each having 8 CPUs of 16 Gflops peak (2Ghz). Each node has 128Gb, about 124Gb are usable for applications.

The vector systems have a fast interconnect called IXS, which is a central crossbar switch. Each node can send and receive with 16 GB/s in each direction. You can expect an MPI latency of about 5 microseconds for small messages. Each SX-8 CPU can access the main memory with 64 GB/s.

## 12.7 ICC

The Institute for Computational Cosmology (ICC) is a leading international centre for research into the origin and evolution of the universe.

The ICC is also the main UK base of the Virgo Consortium for cosmological simulations, a collaboration of researchers from the UK, Germany, USA and Canada. The ICC is a partner of the Anglo-Australian two-degree Field Galaxy Survey, one of the largest complete surveys to date.

The ICC is based at the University of Durham and is part of the Ogden Centre for Fundamental Physics.

Quintor is one part out of three of the supercomputer COSMA (Cosmology machine) which comprises the machines: titania, CENTAUR and Quintor. Quintor consists of 259 SunFire V210s located at the Institute for Computational Cosmology (ICC) [22] at the University of Durham. Each SunFire comprises of:

- 2 x 1002 MHz UltraSPARC-IIIi processors.
- 1 Mb of cache per processor.
- 2 Gb of RAM (where 32 of the 259 systems have 4 Gb).
- 2 Gbit connections.

There are 8 racks with 32 nodes per rack. The best performance can be obtained when a job is run within the same rack.

## 13 Appendix: Machine specific scripts

In this Appendix we give an overview of the compiler flags which were used for a specific machine as well the appropriate batch scripts.

The batch scripts we use are informed by the user guides provided by each of the sites on their web pages or in the documentation provided electronically.

### 13.1 HPCx

The following optimized compiler flags are used on HPCx:

```
-q64 -O3 -qstrict -qintsize=4 -qrealsize=8 -qzerosize -cpp -c \  
-qsuffix=cpp=F -qtune=pwr4 -qarch=pwr4
```

for Fortran. For F90 code we also use:

```
F90FLAGS = -qsuffix=f=F90:cpp=F90 -qfree
```

Despite HPCx consisting of Power5 processors, the choice of `qtune` and `qarch` as `pwr4` gave better performance and is also recommended in the HPCx User Guide [32].

We do not use `-O4` or any higher level of optimisation on any of the IBM machines as the FLASH User Guide states that `-qipa` or `-qhot` do not work with FLASH. FLASH uses compiler switches to promote REALS to DOUBLE PRECISION [6], which is why `-qrealsize=8` is used. For linking we use `-b64` which links together 64-bit objects. The subroutine `perfmon.F90` we compile with optimization `-O2` instead of `-O3` as recommended in the FLASH User guide.

On HPCx, the default SAVE flags for the `mpxlf90_r` and `mpxlf_r` compilers are `-qnosave` and `-qsave`, respectively. These options are important, as the code may crash during runtime if they are not used.

The batch script is:

```

#@ shell = /bin/sh
#
#@ job_type = parallel
#@ job_name = flash
#
#@ cpus = 32
#@ tasks_per_node = 16
#@ node_usage = not_shared
#
#@ network.MPI = csss,shared,US
#
#@ wall_clock_limit = 01:00:00
#@ account_no = e24-jra2
#
#@ stack_limit = 200Mb
#@ output = $(job_name).$(schedd_host).$(jobid).out
#@ error = $(job_name).$(schedd_host).$(jobid).err
#
#@ notification = never
#
#@ queue

export MP_EAGER_LIMIT=65536
export MP_SHARED_MEMORY=yes
export MEMORY_AFFINITY=MCM
export MP_TASK_AFFINITY=MCM

cat $0

date
echo "my_nodes:" $LOADL_PROCESSOR_LIST

poe ./flash2

```

### 13.2 IDRIS

Before running `gmake` on the IBM Regatta+ cluster, `zahir`, at IDRIS, some modules need to be loaded:

```
module load deisa
module switch fortranreal/4 fortranreal/8
```

The compiler flags are:

```
-q64 -O3 -qstrict -qzerosize -cpp -c -qsuffix=cpp=F -qtune=pwr4 -
qarch=pwr4
```

For F90 code we also use:

```
F90FLAGS = -qsuffix=f=F90:cpp=F90 -qfree
```

We do not use `-O4` or any higher level of optimisation on any of the IBM machines as the FLASH User Guide states that `-qipa` or `-qhot` do not work with FLASH. FLASH uses compiler switches to promote REALS to DOUBLE PRECISION [6], which is why the `module switch fortranreal/4 fortranreal/8` is used (instead of `-qrealsize=8` as on HPCx) is used. For linking we use `-b64` which links together 64-bit objects. The subroutines `perfmon.F90` we compile with optimization `-O2` instead of `-O3` as recommended in the FLASH User guide.

The following batch script is used to run on 32 processors:

```
#!/bin/bash
#@ job_name = flash
### input =
#@ output = $(job_name).$(schedd_host).$(jobid).out
#@ error = $(job_name).$(schedd_host).$(jobid).err
#@ notify_user = never
#@ shell = /bin/bash
#@ requirements = (Feature == "DEISA")
#@ job_type = parallel
#@ total_tasks = 32
#@ cpu_limit = 1:00:00
#@ data_limit = 2Gb
#@ queue

module load deisa

exe=/workdir/deisa/hpx00003/hpx00003/FLASH2.5/object/flash2

cd /workdir/deisa/hpx00003/hpx00003/FLASH2.5/object

$exe
```

### 13.3 RZG

Before running `gmake` on the IBM Regatta+ cluster at RZG, some modules need to be loaded:

```
module load deisa
module switch fortranreal/4 fortranreal/8
module load fftw
```

The compiler flags are:

```
-q64 -O3 -cpp -c -qsuffix=cpp=F -qtune=auto -qarch=pwr4
```

For F90 code we also use:

```
F90FLAGS = -qsuffix=f=F90:cpp=F90 -qfree
```

We do not use `-O4` or any higher level of optimisation on any of the IBM machines as the FLASH User Guide states that `-qipa` or `-qhot` do not work with FLASH. FLASH uses compiler switches to promote REALS to DOUBLE PRECISION [6], which is why the module switch `fortranreal/4` `fortranreal/8` is used (instead of `-qrealsize=8` as on HPCx). For linking we use `-b64` which links together 64-bit objects. The subroutines `perfmon.F90` we compile with optimization `-O2` instead of `-O3` as recommended in the FLASH User guide.

The following batch script is used to run on 32 processors:

```
#sample batch job to run an MPI job
# using the "Federation" switch communication network :
#
#@ output = job.out.$(jobid)
#@ error = job.err.$(jobid)
#@ initialdir=/deisa/rzg/home/hpx00003/hpx00003/FLASH2.5/object
#@ class = lhuge
#@ job_type = parallel
#@ environment= COPY_ALL
#
# 1 node has 32 processors
#@ node = 1
#@ tasks_per_node = 32
#@ stack_limit = 200mb
#
#In case of MPI, you have to set this variable to 1.
#@ resources = ConsumableCpus(1)
#@ network.MPI = sn_all,not_shared,us
#@ queue

#
# run the program
#
poe ./flash2
```

### 13.4 BSC

Before running `gmake` on the IBM Linux cluster, MareNostrum, at BSC, some environment variables need to be exported:

```
export MP_EUILIB = gm
export OBJECT_MODE = 64
export MP_RSH = ssh
```

On MareNostrum the code is compiled with:

```
-O3 -qstrict -qintsize=4 -qrealsize=8 -c -qsuffix=cpp=F -
qtune=ppc970
```

The routine `perfmon.F90` requires slightly different compiler flags to run on BSC:

```
PERFMON_FFLAGS = -O3 -qstrict -qintsize=4 -qrealsize=8 -c -
qsuffix=cpp=F-qtune=ppc970 -qsuffix=f=F90:cpp=F90 -qfree
```

An additional linker flag is necessary to allow for multiple definitions as there are in `abort_flash.F90` and `driverAPI.c`:

```
-Wl,--allow-multiple-definition
```

In addition, the routine `buildstamp.F90` needs to be compiled with `-qfixed`. As we are compiling in 64-bit mode some of the required libraries are in `/usr/lib64` instead of `/usr/lib`.

The batch script is:

```
#!/bin/tcsh
#
#@ job_type = parallel
#@ class = projects
#@ group = hpx69
#@ initialdir = /home/hpx69/hpx69693/FLASH2.5/object
#@ output = $(jobid).$(stepid).out
#@ error = $(jobid).$(stepid).err
#@ restart = no
#@ requirements = (Feature == "myrinet")
#@ node = 4
#@ total_tasks = 8
## time in second
#@ wall_clock_limit = 3600
#@ queue

# environment
setenv MP_EUILIB gm
setenv OBJECT_MODE 64
setenv MP_RSH ssh

setenv MLIST machine_list_${LOADL_STEP_ID}

/opt/ibmll/LoadL/full/bin/ll_get_machine_list > $MLIST

set NPROCS = `cat $MLIST |wc -l`
set OUTDIR = /home/hpx69/hpx69693/FLASH2.5/object

mpirun -s -np ${NPROCS} -machinefile $MLIST ./flash2 >&
${OUTDIR}/ivp$$$.out
```

### 13.5 HLRS

Note that the compiler flag `-Cvopt` is switched on by default, `-Cvsafe` on the other hand specifies to perform only save optimization.

We employ the following compiler flags for all files:

```
-size_t64 -Wf"-A idbl" -c
```

except the files `amr_restrict_unk_fun.F90` and `amr_restrict_work_fun.F90` which need to be compiled with `-Csave`. Otherwise the code crashes during run time [27]. The batch script is:

```
#!/usr/bin/ksh
#PBS -q multi           # queue: dq for <=8 CPUs
#PBS -l cpunum_job=8   # cpus per Node <!!!!!!! NEW
#PBS -b 4              # number of nodes
#PBS -T mpisx         # Job type: mpisx for MPI
#PBS -l elapstim_req=10:00:00 # max wallclock time <!!!!!!! NEW
#PBS -l cputim_job=60:00:00 # max accumulated cputime per job
#PBS -l cputim_prc=60:00:00 # max accumulated cputime per request
#PBS -l memsz_job=100gb # max accumulated memory
#PBS -A s13903        # Your Account code, see login message
#PBS -j o             # join stdout/stderr
#PBS -N cpus32       # job name
#PBS -M name@epcc.ed.ac.uk
                        # you should always specify your email address

export MPIPROGINF=YES # give some performance information
#export MPIMULTITASKMIX=NO # we are hybrid openmp + mpi
#export MPISUSPEND=ON # be nice to others in shared mode,
don't busy wait

#export F_PROGINF=DETAIL # run statistics will be kept in standard
error file
#export F_SETBUF06=0 # disable buffering for standard output file

cd $HOME/outputdir/vanilla/32 # where the results of runs will be
stored

mpirun -np 8 $HOME/FLASHvanilla/FLASH2.5/object/flash2 -par_file
$HOME/FLASHvanilla/FLASH2.5/flash.par
```

Even more details can be found in [27].

## 14 Appendix: Compiler Flag Specifics

This appendix contains a more detailed explanation of what the main compiler flags used within this study do. To find out more use the compiler man pages for each of the sites.

### 14.1 IBM-specific compiler flags

On the IBM machines we use the XL Fortran, version 9.1. The Fortran95 compiler is invoked using `xlf95` and sets `-qfree=f90` (free-form source); the source file extension is assumed to be `.f`:

- Invocations prefixed by `mp` (for example `mpxlf_r`) set the pre-processing option `-I/usr/lpp/ppe.poe/include` as well as the loader option `-bitfini:poe_remote_main`, and are required to link the MPI/MPL libraries
- Invocations suffixed by `_r` additionally set `-qthreaded`, `-D_THREAD_SAFE`, and additionally link to `-lpthreads`; they are the thread-safe versions. This is recommended for MPI codes.

The `-qsuffix` option is used to specify the source file suffix on the command line. For example `mpx1f90_r -qsuffix=f=f90` assumes the source suffix to be `.f90` instead of `.f`.

The compiler option `-qtune=pwr4` produces an object optimized for the POWER4 hardware platforms.

For the XL Fortran compiler the default size of integer or logical entities is 4 bytes. If you specify `-qintsize=8` at compile time, the variable is declared as 64-bit entity, the default is `-qintsize=4`. `-qintsize` effects:

- Integer and logical literals that do not specify kind type parameters.
- Default integer and logical data objects and functions.
- Intrinsic function results of type default integer or logical.
- Typeless constants in integer contexts.

Similar to `-qintsize`, the `-qrealsize` compiler option sets the default size of real and complex entities whose sizes are not explicitly declared. The default in XL Fortran is 4 bytes. The precision of entities of type DOUBLE PRECISION is twice that of the default real size. Thus, if you specify `-qrealsize=8`, entities of type DOUBLE PRECISION have a default of 16 bytes. The DEISA Primer, Sections 4.4.3 and 4.4.4, provides a further discussion about the sizes of Fortran float and integer numbers and the usage of appropriate DEISA modules that is why at IDRIS and RZG we use:

```
module switch fortranreal/4 fortranreal/8
```

instead of setting the `kind` via the compiler flags.

## 14.2 SUN-specific compiler flags

On the SUN we use Fortran90 compiler which comes with SUN Studio 11 on ICC which is the default (status end Jan. 2006).

The compiler option `-fast` provides high performance, among others it selects the highest optimization level, `-O5`. The flag `-fpp` forces pre-processing of input files with `fpp`, the Fortran pre-processor. The cache is defined via the `-xcache` flag. The option `-xtypemap` provides a flexible way to specify the byte sizes for default data types. The allowable data types are REAL, DOUBLE, INTEGER. The data sizes accepted are 32, 64 and 128. This option applies to all variables declared without explicit byte sizes, as in `REAL XYZ`. Specify the target system for the instruction set and optimization via `-xtarget`.

## 14.3 Altix specific compiler flags

On the Altix at SARA we use the `-i8` flag which defines REAL declarations, constants, functions and intrinsics as `DOUBLE PRECISION (REAL*8)`.

## 14.4 NEC specific compiler flags

On the NEC at HLRS we use `-wf`, this flag specifies the option string of the FORTRAN90/SX detailed options.

The option `-A -idbl` specifies the precision expansion as `R(4) → R(8)`, `R(8) → R(16)`, `C(4) → C(8)` and `C(8) → C(16)`, where, for example, `R(4)` stands for a REAL of 4 bytes. However, a variable or a constant with kind type parameter is not applied the precision expansion.

## 15 Appendix: Porting the Virgo Simulation Code to HLRS

Table 22 lists the compilers and methods for accessing HDF5 on HLRS. Code on HLRS is cross compiled on the scalar front-end to be used on the vector backend. This is done because it is much faster to compile on scalar processor than on vector machines.

Fortran90 Compiler	sxmpif90
C Compiler	sxmpicc
HDF5	Version 1.4.4 installed locally
HDF5 wrapper	Installed locally

**Table 22: Features of the most basic tools on HLRS.**

We describe here the porting of the original FLASH code and the Virgo Simulation code (and in Section 6.1.4 on porting the Improved Virgo Simulation onto HLRS) since the issues arising in porting of these codes was not discussed in the interim report [4].

1. For the FLASH code as it comes from the official FLASH distribution site we encountered several platform specific problems:
  - Since the representation of an `INTEGER*8` cannot exceed  $2^{53}$  or, in other words  $10^{15}$ , the integer parameter `big_int` needed to be changed from `SELECTED_INT_KIND(18)` to `SELECTED_INT_KIND(15)` in `perfmon.F90`.
  - Link in the `zlib` library using `-L/SX/usr/lib -lz90sx` required for machine specific cross compilation.
  - Link in the `cpp` library using `-L/SX/usr/lib -lcpp` required for machine specific cross compilation.
  
2. For the Virgo Simulation code:
  - Introduce a new variable `temp_pnum` in the file `SetupMoveParticles.F90` to avoid overwriting in subsequent subroutines. The compiler had complained that the variable `pnum` was redefined since it had not defined properly according to the Fortran standard. For more details, please see [27].
  - A derived data type was not declared according to the way specified in the Fortran standard in the file `InitParticles.F90` hence this was changed from `particles(:).realAttributes` to `particles(:)%realAttributes`.
  - Several variables were declared more than once in various files. The second declaration was commented out. The variables were:
    - `time_unit_in_s` at line 43 in `ReadHeader.F90`,
    - `proc` at line 35 in `InitParticlePositions.F90`,
    - `block_id` at line 35 in `InitParticlePositions.F90`,
    - `omega_m` at line 380 in `init_from_scratch.F90`,
    - `i` at line 599 in `mark_grid_refinement.F90`.
  - Additional continuation marks `'&'` needed to be added in the file `mark_grid_refinement.F90` to solve compilation errors. The compiler expected `'&'` in the beginning of the continuing line instead of in the end of the previous line.

- The code did not link because the function `ran3` in the file `random_numbers.f90` was declared as `external` in the subroutines `gaussian_deviate` and `randg` located in the same file but not used by them. These declarations were removed.

The following was also true for the port onto HLRS, RZG, BSC and IDRIS:

- Write constants as `1E30` and not `1D30` (as done in the file `init_from_scratch.F90` for the solar mass) when extending the original FLASH code by user-written modules. The appropriate compiler switches will promote real numbers to double precision numbers.
- Set logical variables to `.TRUE.` or `.FALSE.`, not `.T.` or `.F.` as done in the file `init_global_parms.F90` when extending the original FLASH code.

This was the first time any FLASH code had been ported to a vector machine. The necessary adaptations will be included into the final version of the code which will include all changes done on HPCx, RZG, BSC and HLRS and constitute the deliverable D-JRA2-3.2.

## 16 Appendix: Implementation of the New Algorithm

The new algorithm will replace the time intensive multigrid solver by a spectral solver which is based on FFTs. The implementation of the new algorithm in the FLASH code and the background information here has been provided by Tom Theuns. In this Appendix we summarize the basic concepts.

The Poisson equation relates density  $\rho$  to gravitational potential  $\phi$  as

$$\nabla^2 \phi = 4\pi G \rho, \quad (1)$$

where  $G$  is Newton's constant. The Laplacian  $\nabla^2$  on a regular mesh with cell-spacing  $\Delta$  in a difference form can be written as:

$$\frac{\phi(i+1, j) + \phi(i-1, j) + \phi(i, j+1) + \phi(i, j-1) - 4\phi(i, j)}{\Delta^2} = 4\pi G \rho(i, j), \quad (2)$$

where  $i$  and  $j$  denote the indices of the cell on the grid. This difference equation needs boundary conditions on the edge of the grid, as does the corresponding differential equation. Periodic boundary conditions will be assumed in what follows.

The density  $\rho(i, j)$  on the mesh will in general be a sum of the density due to the self-gravitating fluid, and that represented by collision-less particles. The way how the density due to particles is computed will not be discussed.

FLASH solves Equation (2) using an iterative scheme. How this works can best be seen by superposing a checker board with red and black squares over the regular grid, and re-writing the equation as:

$$\phi^{n+1}(i, j) = \frac{\phi^n(i+1, j) + \phi^n(i-1, j) + \phi^n(i, j+1) + \phi^n(i, j-1)}{4} - \frac{4\pi G \rho(i, j) \Delta^2}{4}. \quad (3)$$

Initially, only a uniform mesh will be considered.

If  $(i, j)$  is a black square on the board, then notice that the right-hand side of this equation only involves the potential on red squares. This makes it possible to solve the equation by iteration: assume the potential on the red squares is known at iteration level  $n$ , then a better estimate for the potential at iteration level  $n+1$  on the black squares can be obtained by evaluating Equation (3) for each black square. It can be shown that this process converges to the right solution. FLASH uses the potential from the previous time step as a first guess for the solution, and assumes  $\phi = 0$  initially.

This iteration technique has some interesting properties as to how fast errors in the potential decrease at each iteration. One way to think of this is to realise that corrections to the potential move across the grid one cell at a time. This means that small scale errors are quickly corrected, but correcting errors on large scales require many iterations. To see why this is, consider the case where all values of  $\phi$  are off from the correct solution by a constant in some region of the computational domain. Then notice from Equation (2) that this potential is still a solution to the Poisson equation in that region of space, and so the iteration procedure will only correct this erroneous potential at the boundary. But the correction moves just one cell at a time.

This also suggests how the convergence can be sped up. Suppose we consider in addition to our uniform grid another uniform grid at half the resolution, obtained by smoothing  $\rho$ . Errors on large scales on this coarser grid will be corrected twice as fast, as on the finer grid. The fine potential can be corrected for errors on large scales by prolonging the potential from the coarser mesh to the finer one. The multigrid Poisson solver implemented in FLASH uses the whole hierarchy of grids, from the finest mesh to the base mesh. The implementation uses a combination of a series of iterations on a mesh of given refinement, together with corrections coming from the coarser and finer meshes in a fixed sequence. FLASH uses the fact that the coarser blocks are kept to cover each level of refinement with a corresponding coarser mesh to speed up convergence of the iteration. The detailed order of prolongation, restriction, and iteration, can be partly controlled by parameters set in the parameter file.

This algorithm is actually very efficient and is at least as fast as the FFT implementation discussed in the next section. This is because the number of operations per cell is much less than when using FFTs, and it is also very well adapted to an AMR calculation: the number of cells on a given level can be much less than the number of cells on the corresponding uniform grid, used by the FFT calculation.

However, the current implementation does not do very well on a parallel computer. The reason is two-fold. In general, there will be many blocks on the finest level, which are distributed over all processors. Therefore, performing a single iteration of Equation (3) on a fine level could be well parallelised. However, there are of course fewer and fewer blocks on the corresponding coarser levels; there is obviously only one block on the first level, and hence as the iteration progresses fewer and fewer processors are active. Note that levels need to be iterated in lock step, that is you cannot start iterating the next level before the previous one has been processed.

This is not as bad as it may seem at first glance, because there is of course also much less work to be done on the coarser blocks. However, after each iteration on any given level potential values on the surface of each block may change. But these values are needed by the neighbouring block, before it can do its next iteration.

Hence, the algorithm requires that each iteration is followed by a communication, where neighbouring blocks exchange boundary values. Depending on the

interconnect between processors, this communication step may be very expensive. The amount of communication also increases rapidly with number of processors, which makes the code scale poorly.

### 16.1 The FFT Implementation

Define the discrete Fourier transform pair as:

$$\hat{\phi}(k) = \sum \phi(x) \exp(ik \cdot x),$$

$$\phi(x) = \frac{1}{N} \sum \hat{\phi}(k) \exp(-ik \cdot x),$$

where  $N$  is a normalisation factor. Inserting this in Equation (2) gives:

$$(2 \cos(2\pi p \Delta / L) + 2 \cos(2\pi q \Delta / L) - 4) \hat{\phi}(p, q) = \hat{\rho}(p, q). \quad (4)$$

Therefore, one can solve Equation (2) by Fourier transforming  $\rho$  to obtain  $\hat{\rho}$ , dividing by the quantity in brackets (called the Green's function), and inverse transforming the result.

On a uniform grid, most of the calculation time is spent in obtaining the Fourier transforms. Note that the computation of the Green's function can in principle be done just once, and the result stored. There are several parallel three dimensional Fourier transform routines freely available, and computer vendors often provide their own optimised versions. The FFTW routine performs the FFT in parallel, using MPI. It assumes that the uniform grid on which to compute the FFT (for example here the density field) is distributed across the processors along columns (planes in 3D). The FFT is then a combination of parallel 1D FFTs, followed by a communication pattern.

On a uniform grid, the iterative procedure for obtaining  $\phi$  is in general faster than the FFT implementation. This is because the number of operations to be performed is much less. However in parallel, the FFT can be faster, simply because the calculation can be performed in parallel very efficiently. Although there is communication as well, the communication pattern can be executed very efficiently. In practise, Fourier transforms performed with FFTW, scale very well on parallel machines. However, the FFT will only work on a uniform grid, whereas FLASH uses an AMR grid. One could generate a uniform grid at the resolution of the finest grid in use; however that may mean that the size of the FFT grid is so large that it no longer fits in memory. First, suppose this is not the case.

If sufficiently fine FFT grids could be fit in memory, then the density on that grid that corresponds to the AMR density would need to be obtained. This will involve prolongation operations when blocks are coarser than the finest mesh, but also communication because the block and its corresponding FFT plane are in general not on the same processor. Once the potential is obtained, it needs to be restricted back to the AMR grid. Although this involves substantial communication, it needs to be done only once before, and once after performing the FFT. In contrast, the iterative procedure requires communication at *each* iteration. In practise, the FFTs are so fast that they take roughly the same amount of time as the prolongation and restriction operations, together with the communication. Note that prolongation and restriction are distributed calculations, and so scale very well with the number of processors.

Calculations with a large dynamic range may have so many levels of refinement that we cannot fit a sufficiently fine FFT grid in memory, and so we cannot use the previous algorithm. The current FFT implementation will use an FFT mesh as fine as the user allows, and obtain the potential at that level. It will then use the iterative procedure described previously with some minor changes to obtain the potential on the finer grids, using the prolonged potential obtained with FFTs as first guess.

## 17 Appendix: Makefile.h for Various Platforms

In this Appendix we list the `Makefile.h` for RZG, BSC and HPCx as used for the Improved Virgo Simulation code. The appropriate for HLRS can be found in the Appendix of [27].

### 17.1 Makefile for RZG

```
# FLASH makefile definitions for RZG

HDF5_PATH = /deisa/rzg/home/hpx00003/hpx00003/lib/
HDF5_INC = /deisa/rzg/home/hpx00003/hpx00003/include

FFTW = /usr/local/packages/fftw
# double precision fftw libraries required
FFTW_LIB = $(DEISA_LDFLAGS)
INCLUDE_FFTW = $(DEISA_FFLAGS)

# John Helly-s hdf5 wrappers
HDF5_WRAPPER = /deisa/rzg/home/hpx00003/hpx00003/HDF5_Wrapper
MODJCH = -I$(HDF5_WRAPPER)/lib
LIBJCH = -L$(HDF5_WRAPPER)/lib -lhdfwrapper
RLIBJCH = -I$(HDF5_WRAPPER)/src/./lib/
RLIB = $(INCLUDE_FFTW) $(FFTW_LIB) $(RLIBJCH) $(LIBJCH)

PAPI_PATH = /usr/local
PAPI_FLAGS = -c -I$(PAPI_PATH)/include -qsuffix=f=F90:cpp=F90 -qfree
# Compiler and linker commands

F90COMP = mpqlf90_r
FCOMP = mpqlf_r
CCOMP = mpcc_r
CPPCOMP = mpCC_r
LINK = mpqlf_r

# pre-processor flag
PP = -W

# Compilation flags

INCLUDE = -I$(INCLUDE_FFTW) -I$(HDF5_INC)
INCLUDE = $(DEISA_FFLAGS) -I$(HDF5_INC)
MODULES = $(MODJCH)

FFLAGS_OPT = -pg -g -q64 -O3 -cpp -c \
             -qsuffix=cpp=F -qtune=auto -qarch=pwr4 $(INCLUDE)
$(MODULES)

F90FLAGS = -qsuffix=f=F90:cpp=F90 -qfree
```

```

f90FLAGS      = -qsuffix=f=f90:cpp=F90 -qfree

# if we are using HDF5, we need to specify the path to the include
files
CFLAGS_HDF5   = -I/deisa/rzg/home/hpx00003/hpx00003/include -
DNOUNDERSCORE -I/deisa/rzg/home/hpx00003/hpx00003/lib/

CFLAGS_OPT    = -c -O3 -qcache=auto -qtune=auto -DIBM

MDEFS = -WF,

# Linker flags

# Linker flags for optimization
LFLAGS_OPT    = -b64 $(RLIB) $(INCLUDE) -o

LFLAGS_TEST   = -bmaxdata:0x80000000 -o

# Library specific linking

LIB_HDF5      = -L $(HDF5_PATH) -lhdf5_fortran -lhdf5 -L /usr/lib -lz -
lm -lgpfs
LIB_PAPI      = -L$(PAPI_PATH)/lib -lpapi -L/usr/lpp/pmtoolkit/lib -
lpmapi
LIB_MATH      = -lessl

LIB_OPT       = -L/afs/rzg/@sys/lib -lmpitrace -lmpi $(LIBJCH) $(RLIBJCH)
LIB_OPT       = $(LIBJCH) $(RLIBJCH)
LIB_DEBUG     = $(LIBJCH) $(RLIBJCH)

```

## 17.2 Makefile for BSC

```

# FLASH makefile definitions BSC

OBJECT_MODE=64

#-----
# Set the HDF/HDF5 library paths -- these need to be updated for your
system
#-----

HDF5_PATH = /gpfs/apps/HDF5/1.4.4/
HDF5_INC  = /gpfs/apps/HDF5/1.4.4/include

FFTW      = /gpfs/apps/FFTW/2.1.5/64
# double precision fftw libraries required
FFTW_LIB  = -L$(FFTW)/lib/ -ldrfftw_mpi -ldfftw_mpi -ldrfftw -
ldfftw
INCLUDE_FFTW = $(FFTW)/include

# John Helly-s hdf5 wrappers
HDF5_WRAPPER = /home/hpx69/hpx69693/HDF5_Wrapper
MODJCH       = -I$(HDF5_WRAPPER)/src/./lib
LIBJCH       = -L$(HDF5_WRAPPER)/src/./lib -lhdfwrapper
RLIBJCH      = -I$(HDF5_WRAPPER)/src/./lib/
RLIB         = -I$(FFTW)/include $(FFTW_LIB) $(RLIBJCH) $(LIBJCH)

```

```

# Compiler and linker commands

F90COMP = mpif90
FCOMP   = mpif90
CCOMP   = mpicc
CPPCOMP = mpiCC
LINK    = mpif90

# pre-processor flag
PP      = -D

# Compilation flags

INCLUDE      = -I$(HDF5_INC)
MODULES     = $(MODJCH)

FFLAGS_OPT = -O3 -qstrict -qintsize=4 -qrealsize=8 -c -
qsuffix=cpp=F -qtune=ppc970 $(INCLUDE) $(MODULES)

PERFMON_FFLAGS = -O3 -qstrict -qintsize=4 -qrealsize=8 -c -
qsuffix=cpp=F
-qtune=ppc970 -qsuffix=f=F90:cpp=F90 -qfree $(INCLUDE) $(MODULES)

CFLAGS_OPT = -q64 -O3 -D_AIX -DNOUNDERSCORE -c -qtune=ppc970 -
qarch=ppc970 -Wl,--allow-multiple-definition

F90FLAGS = -qsuffix=f=F90:cpp=F90 -qfree
f90FLAGS = -qsuffix=f=f90:cpp=F90 -qfree

# if we are using HDF5, we need to specify the path to the include
files
CFLAGS_HDF5 = -DNOUNDERSCORE -I $(HDF5_PATH)/include

MDEFS = -WF,

# Linker flags

LFLAGS_OPT = -g -Wl,--allow-multiple-definition $(RLIB) $(INCLUDE)

# Library specific linking

LIB_OPT = $(LIBJCH) $(RLIBJCH)

LIB_HDF4 = -L $(HDF4_PATH)/lib -lmfhdf -ldf -ljpeg -lz
LIB_HDF5 = -L $(HDF5_PATH)/lib -lhdf5_fortran -lhdf5 -L /usr/lib64 -
lz -lm -lgpfs

LIB_MATH = -ldfftw -ldrfftw

```

### 17.3 Makefile for HPCx

```

# FLASH makefile definitions for HPCx

HDF5_PATH = /hpcx/home/z001/z001/elena/hdf5interim/lib/
HDF5_INC  = /hpcx/home/z001/z001/elena/hdf5interim/include

FFTW      = /usr/local/packages/fftw

```

```

FFTW_LIB      = -L$(FFTW)/lib/ -ldrfftw_mpi -ldfftw_mpi -ldrfftw -
ldfftw
INCLUDE_FFTW  = $(FFTW)/include

# John Helly-s hdf5 wrappers
HDF5_WRAPPER = /hpcx/home/z001/z001/elena/HDF5_Wrapper
MODJCH       = -I$(HDF5_WRAPPER)/src/./lib
LIBJCH       = -L$(HDF5_WRAPPER)/src/./lib -lhdfwrapper
RLIBJCH      = -I$(HDF5_WRAPPER)/src/./lib/
RLIB         = -I$(FFTW)/include $(FFTW_LIB) $(RLIBJCH) $(LIBJCH)

PAPI_PATH    = /usr/local
PAPI_FLAGS   = -c -I$(PAPI_PATH)/include -qsuffix=f=F90:cpp=F90 -qfree

F90COMP      = mpxlf90_r
FCOMP        = mpxlf_r
CCOMP        = mpcc_r
CPPCOMP      = mpCC_r          LINK      = mpxlf_r

# pre-processor flag
PP           = -W

# Compilation flags

INCLUDE      = -I$(INCLUDE_FFTW) -I$(HDF5_INC)
INCLUDE      = $(DEISA_FFLAGS) -I$(HDF5_INC)
MODULES      = $(MODJCH)

FFLAGS_OPT   = -q64 -O3 -cpp -c -qintsize=4 -qrealsize=8\
               -qsuffix=cpp=F -qtune=auto -qarch=pwr4 $(INCLUDE)
$(MODULES)
F90FLAGS     = -qsuffix=f=F90:cpp=F90 -qfree
f90FLAGS     = -qsuffix=f=f90:cpp=F90 -qfree

# if we are using HDF5, we need to specify the path to the include
files
CFLAGS_HDF5  = -I/hpcx/home/z001/z001/elena/hdf5interim/include -
DNOUNDERSCORE -I/hpcx/home/z001/z001/elena/hdf5interim/lib/

CFLAGS_OPT   = -c -O3 -qcache=auto -qtune=auto -DIBM
MDEFS       = -WF,

# Linker flags# Linker flags for optimization
LFLAGS_OPT   = -b64 $(RLIB) $(INCLUDE) -o

# Library specific linking
#
LIB_HDF5     = -L $(HDF5_PATH) -lhdf5_fortran -lhdf5 -L /usr/lib -lz -
lm -lgpfs
LIB_PAPI     = -L$(PAPI_PATH)/lib -lpapi -L/usr/lpp/pmtoolkit/lib -
lpmapi
LIB_MATH     = -lessl

LIB_OPT      = -lmpi $(LIBJCH) $(RLIBJCH)

```