



CONTRACT NUMBER 508830

**DEISA**

**DISTRIBUTED EUROPEAN INFRASTRUCTURE FOR  
SUPERCOMPUTING APPLICATIONS**

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

Specifications Document for D-JRA2-3.2

Deliverable ID: D-JRA2-3.1

Due date: April, 30, 2006

Actual delivery date: May 17, 2006

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

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

Duration: 4 years

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

## Table of Content

Table of Content.....	1
1. Introduction.....	3
1.1 Motivation and project goals.....	3
1.2 Background.....	3
1.3 Intended Audience.....	4
1.4 Outline.....	4
1.5 Conventions.....	5
1.6 Acknowledgements.....	5
1.7 References.....	5
1.8 Document Amendment Procedure.....	6
1.9 List of Acronyms and Abbreviations.....	6
2. Details of the DEISA platforms employed.....	7
2.1 Porting process on each individual platform.....	7
3. Hydrodynamical FLASH simulation: porting of FLASH using the Sedov problem.....	8
4. Dark matter FLASH simulation: porting and profiling of FLASH with Santa Barbara initial conditions.....	9
4.1 Porting to HPCx.....	10
4.2 Porting to SARA.....	11
4.3 Porting to IDRIS.....	11
4.4 Porting to RZG.....	11
4.5 Porting to BSC.....	12
4.6 Porting to HLRS.....	12
4.7 The code on ICC.....	12
5. Ensuring portability.....	12
5.1 Problems with the routine <code>perfmon.F90</code> .....	13
6. The HDF5 wrapper.....	13
7. Scaling study.....	14
7.1 SCALING RESULTS.....	14
7.2 Start up costs.....	17
8. Profiling Study.....	18
8.1 On HPCx with <code>gprof</code> .....	18
8.2 FLASH internal profiling results.....	19
9. Code Migration.....	21
9.1 General Description.....	21
9.2 DEISA Specifics.....	22
9.2.1 Stage 1: Submission to the Batch System.....	22
9.2.2 Stage 2: Code Aware of Elapsed Time.....	22
9.2.3 Stage 3: Automatic Code Resubmission.....	22
9.2.4 Stage 4: Portable Restart Files.....	22
10. Discussion of results and future work.....	23
10.1 Delays and difficulties.....	23
10.2 Review of work done.....	23
10.3 Future work.....	24
11. Appendix: General Machine Information.....	25
11.1 HPCx.....	25
11.2 SARA.....	25
11.3 IDRIS.....	26
11.4 RZG.....	26
11.5 BSC.....	26
11.6 HLRS.....	27
11.7 ICC.....	27
12. Appendix: Machine specific scripts.....	28

12.1	HPCx.....	28
12.2	SARA.....	29
12.3	IDRIS.....	30
12.4	RZG.....	31
12.5	BSC.....	32
12.6	HLRS.....	33
12.7	ICC.....	34
13.	Appendix: Compiler Flag Specifics.....	35
13.1	IBM-specific compiler flags.....	35
13.2	SUN-specific compiler flags.....	35
13.3	Altix specific compiler flags.....	36
13.4	NEC specific compiler flags.....	36
14.	Appendix: Validation of results and the <code>sfocu</code> tool.....	36
15.	Appendix: Usage of the debugger on BSC.....	37

## 1. Introduction

This is the specifications document for D-JRA2-3.2 and, as such, includes a description of the work which will be carried out over the next 6 months. This description is contained in Sections 9 and 10.3.

### 1.1 *Motivation and project goals*

This work package consists of porting FLASH [9], a simulation code used to perform cosmological simulations by members of the Virgo Consortium [25], to use the various different platforms within the DEISA infrastructure. Subsequent to this, platform-independent optimisations will be introduced and some provision will be made for code migration.

Code migration will allow Virgo to run currently prohibitively long FLASH simulations, as a simulation which has been terminated due to a batch system restriction, can continue via self-submission. Moreover, code migration will allow for a faster time to solution, as it allows effective use of the DEISA batch system. To ensure an efficient use of code migration, we must port the code to a number of different platforms, and to ensure an efficient use of each platform, we shall examine platform-independent optimisations to Virgo's FLASH simulation.

This interim report summarises what has been done within the project and outlines what is expected to be done during the remainder of the project.

### 1.2 *Background*

FLASH is a modular, parallel<sup>1</sup>, Fortran90 Adaptive Mesh Refinement (AMR) simulation code which is available to the research community subject to licensing conditions. The code was initially developed for the US government to model thermonuclear flashes, but is now widely used by a number of different scientific communities. FLASH cannot be freely re-distributed and a licence agreement must be signed in order to be able to download and use the source code<sup>2</sup>. The code models general compressible flow in astrophysical simulations. The objects of interest are separated by large regions of empty space and the mesh refinements can be placed where the objects of interest are, thus less resource is used to model the void regions. The AMR algorithm uses a block-structured adaptive grid, placing resolution elements only where they are needed most.

From the start, the hydrodynamics of FLASH have been developed to address problems in astrophysics such as thermonuclear flashes on surfaces of compact stars, in particular X-ray bursts, type Ia supernovae and novae. The code is written using highly modular Fortran90, making the code easily extensible allowing it to be applied to new domains. Particles have been added to model a non-fluid component, such as dark matter, within a simulation. These particles act as Lagrangian mass tracers, which may or may not contribute to the dynamics of the system. For the type of simulation considered in this document, these particles *do* contribute to the dynamics of the system. FLASH has been parallelised using MPI. The hydrodynamics of the parallel version have been found to scale well, but when particles, dark matter particles in this case, are included in the simulation the code scales less well [21].

---

<sup>1</sup> It uses the Message-Passing Interface (MPI) library.

<sup>2</sup> See [flash.uchicago.edu/site/coderequest](http://flash.uchicago.edu/site/coderequest) for more details.

The code has been extended by Virgo to allow HDF5 [20] files to be used for input – previously FLASH only allowed output files, e.g. check-pointed files, to use HDF5. The Virgo Consortium then hope to be able to employ FLASH to investigate the early structure formation of the universe. This will require large amounts of computational power as is available in the DEISA infrastructure.

The goal for this project is to port FLASH to the various DEISA platforms and then investigate the performance of the code across the various architectures. Using various profilers will help to identify the time-intensive parts of the code for the types of simulation that Virgo Consortium hope to carry out in order to identify any possible platform-independent optimisations that might be made to improve the code's performance.

The simulations under investigation for WP3 of JRA2 run using the Santa-Barbara data sets [18] as its initial conditions.

### 1.3 *Intended Audience*

The intended audience for this document are:

- **The DEISA sites we port the code to.** The porting of a large code which relies on widely distributed libraries, like HDF5, on to various DEISA platforms will inform the site administrators about the issues and challenges related to supporting such a project within the DEISA infrastructure.
- **The FLASH developers.** The use of a variety of architecturally different machines, some of which FLASH has never been run on such as the NEC [6] platform, will help to highlight code portability issues and inform them about the way the code performs across these systems.
- **The Virgo Consortium and other researchers that use FLASH.** These are the users who apply the code to solve real scientific problems. It will be highly beneficial for them to see how the code can be ported to these architectures and how the code scales on these machines. In addition, we will hopefully be able to point out any major bottlenecks in the code and be able to reduce these as well as trying to provide other generic performance improvements. Using these architectures efficiently is a major goal for the potentially long-running cosmological simulations of the Virgo Consortium.

### 1.4 *Outline*

In the remainder of this document Section 2 describes the DEISA platforms that are employed in this work package. Section 3 introduces the hydrodynamical-only FLASH simulation which is used to test FLASH in the DEISA platforms. Section 4 describes the FLASH Virgo simulation used, which includes dark matter and thus corresponds to the type of simulation type that are used by Virgo, and how the code was adapted locally for each of the platform employed. Section 5 summarizes any special porting issues encountered. Section 6 contains a discussion of the HDF5 wrapper, as supplied by the Virgo Consortium and which is required to perform the Virgo simulation.

The following two Sections contain the execution time results: Section 7 describes how the code scales with processor numbers on each platform and Section 8 attempts to list the most computationally expensive routines across all the platforms employed.

Section 9 describes the current plan to introduce code-migration, which will allow the Virgo simulation to start on one DEISA platform and finish on another.

Finally, Section 10 contains the discussion of results and plans for the remaining six months of this work package.

The major differences between all the platforms investigated, including the machine specific compilers, flags, environments and batch systems are listed in Appendices 11 and 12. Some specific compiler flags are explained in more detail in Appendix 13. The `sfocu` tool used to validate the hydrodynamical simulation results is discussed in Appendix 14. The last Appendix demonstrates how to use the debugger on BSC.

## 1.5 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.

## 1.6 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 and, in particular: the HDF5 wrapper was written and provided by John Helly and the HDF5 subroutines responsible for handling the reading of the Santa Barbara [18] input data was provided by Tom Theuns. Thanks are also due to Craig Booth, Richard Bowers and Carlos Frenk.

We thank Wim Rijks (SARA), Stefan Haberhauer (NEC) and, particularly, David Vicente (BSC) for their help with porting the code.

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

## 1.7 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](http://www.bsc.es).
- [4] Cactus-G: [www.cactuscode.org](http://www.cactuscode.org).
- [5] Condor: [www.cs.wisc.edu/condor](http://www.cs.wisc.edu/condor).

- [6] DEISA Primer, [www.deisa.org/userscorner/primer.php](http://www.deisa.org/userscorner/primer.php).
- [7] “DESHL v3.0”, draft DEISA JRA7 Report, DEISA-JRA7-3.10, April, 2006
- [8] EuroVO : [www.euro-vo.org](http://www.euro-vo.org).
- [9] 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).
- [10] FLASH user mailing list,  
[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).
- [11] Gridlab: [www.gridlab.org](http://www.gridlab.org)
- [12] High Performance Computing Center Stuttgart,  
[www.hlrs.de/hw-access/platforms/sx8](http://www.hlrs.de/hw-access/platforms/sx8).
- [13] HPCx Service, [www.hpcx.ac.uk](http://www.hpcx.ac.uk).
- [14] 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).
- [15] John Helly, Virgo Consortium, Private Communication.
- [16] Institut du Développement et des Ressources en Informatique Scientifique,  
[www.idris.fr](http://www.idris.fr).
- [17] 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).
- [18] Frenk, C. S., et al., The Astrophysical Journal, 525:554-582, 1999.
- [19] Max-Planck-Gesellschaft Rechenzentrum Garching der Max-Planck-Gesellschaft und des IPP, [www.rzg.mpg.de/computing](http://www.rzg.mpg.de/computing).
- [20] NCSA HDF Home Page, [hdf.ncsa.uiuc.edu](http://hdf.ncsa.uiuc.edu).
- [21] Paul Ricker, FLASH Center, Private Communication.
- [22] RealityGrid: [www.realitygrid.org](http://www.realitygrid.org).
- [23] 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).
- [24] Sedov, 1959.
- [25] Tom Theuns, Virgo Consortium, Private Communication.
- [26] UNICORE, [www.unicore.org](http://www.unicore.org).
- [27] Virgo Consortium, [www.virgo.dur.ac.uk](http://www.virgo.dur.ac.uk).
- [28] VirtU: [star-www.dur.ac.uk/~csf/virtU/virtU-final.pdf](http://star-www.dur.ac.uk/~csf/virtU/virtU-final.pdf)
- [29] VOTable: [www.us-vo.org/VOTable/index.html](http://www.us-vo.org/VOTable/index.html).

## 1.8 Document Amendment Procedure

Intentionally left blank

## 1.9 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>CC-NUMA</b>	<b>C</b> ache- <b>C</b> oherent <b>N</b> on <b>U</b> niform <b>M</b> emory <b>A</b> ccess.
<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> eistungs <b>R</b> echen <b>Z</b> entrum <b>S</b> tuttgart
<b>HPS</b>	<b>H</b> igh <b>P</b> erformance <b>S</b> witch
<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> éveloppement 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

**SARA**            Stichting Academisch Rekencentrum Amsterdam  
**StB**             Santa Barbara

## 2. Details of the DEISA platforms employed

The following machines were made available as part of the DEISA infrastructure<sup>3</sup> plus Quintor, a Sun Beowulf cluster, operated by the Virgo Consortium at the ICC [17], which is not part of DEISA:

- The IBM at HPCx (hpcx), see Appendix 11.1 for more details.
- The SGI at SARA (Aster), see Appendix 11.2 for more details.
- The IBM at IDRIS (zahir), see Appendix 11.3 for more details.
- The IBM at RZG (psi), see Appendix 11.4 for more details.
- The IBM at BSC (MareNostrum), see Appendix 11.5 for more details.
- The NEC at HLRS (SX-8/576M72), see Appendix 11.6 for more details.
- The Sun at ICC (Quintor), see Appendix 11.7 for more details.

The general characteristics for each of these machines are given in the table below.

Owner Platform	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	128 GB per node (8 CPUs)	2GB

**Table 1: The DEISA platforms employed, including hardware available at the ICC.**

For the sake of simplicity, we shall hereafter refer to each platform by the name of its owner, thus Quintor will be called ICC, etc.

### 2.1 Porting process on each individual platform

For the porting process we always used one of two tar balls<sup>4</sup>, either the standard FLASH tar ball, as downloaded from the FLASH website [9] for the hydrodynamical-only test case, or the Virgo Simulation tar ball which included extensions to run the Santa Barbara (StB) dark matter data sets, which includes FLASH, as made available by the Virgo Consortium [25]. The hydrodynamical test case was run at HPCx, IDRIS, RZG, BSC and SARA (see Chapter 3. ). The Virgo simulation using the FLASH code was run at HPCx, IDRIS, RZG and BSC to perform a scaling study (see Chapter 4. ). SARA and HLRS still need to be included in a future investigation as there has not been enough time to complete the porting process of the basic FLASH code onto HLRS nor the Virgo simulation onto SARA or HLRS.

<sup>3</sup> The IBM Power systems at CINECA, ECMWF and FZJ were not considered since they provide no additional insight.

<sup>4</sup> A tar ball is a collection of files, which can include source files, object files, executables, data files, etc., which are all collected, or balled up, using the Unix command tar, such that the so called tar ball resides in a file called filename.tar, for instance.

In general, the step required to port FLASH on to a DEISA platform are as follows:

- install HDF5 version 1.4.4,
- install the HDF5 wrapper for FLASH for the Virgo simulation (this is not always straight-forward as it requires cross-linking a C and Fortran code which generally requires changes for each platform),
- run the Python-based FLASH setup script,
- adapt the Makefile.h produced accordingly for each of the machine specific compilers, flags and library paths,
- compile and link the code via `gmake` (this might reveal other platform dependent problems which require attention) ,
- setup a batch script for the corresponding machine,
- run the code with the Sedov setup (Chapter 3. ),
- validate results using the `sfocu` tool (Appendix 0),
- port the Virgo simulation to use the smallest of the three data sets included in the StB simulation data in order to have a short run time (Chapter 4. ),
- validate the results obtained either using the `h5diff` tool [20] or Tom Theuns IDL script [25],
- to undertake a scaling study choose one of the StB data set input files, adapt the corresponding parameters in the `flash.par` input file,
- submit the batch script for various data sets and numbers of processors.

The `flash.par` input file contains two parameters, `lrefine_min` and `lrefine_max`, which set the minimum and maximum level of adaptive mesh refinement (AMR) that the user wants to employ for the code. The values for `lrefine_min` and `lrefine_max` were generally constrained to lie between four and eight.

As a default FLASH uses the serial version of HDF5. We also use serial HDF5 for the I/O to obtain all the result presented in this report.

We use 64-bit addressing as it increases the amount of addressable memory that a program can use and removes some of the restrictions on shared memory segments. Another benefit from doing this is that the 64-bit MPI library has some additional optimisations.

A profiling tool can give valuable information about any bottlenecks in a code. The `gprof` tool, available on HPCx, also allows one to have a line-by-line breakdown of the code with indications of where exactly most of the computing time is used. In this report we use the internal profiling results provided by FLASH itself (see Chapter 8.2) and the `gprof` and `xprofiler` tools available on HPCx (see Chapter 8.1).

### **3. Hydrodynamical FLASH simulation: porting of FLASH using the Sedov problem**

To start the porting exercise FLASH version 2.5 is downloaded from the FLASH website [9].

The process of porting FLASH to each of the DEISA platforms is described in more detail in the following Section.

FLASH is configured to run the so-called Sedov explosion problem [23]. The Sedov problem is one of the test cases included within the FLASH distribution. It models a hydrodynamical component only, using the default hydrodynamic solver, equation of state, mesh package and serial HDF for the I/O. The Sedov explosion problem is a purely hydrodynamical test which checks the code's ability to deal with strong shocks in a non-planar symmetry. The problem involves the self-similar evolution of a blast wave from a delta-function initial pressure perturbation in an otherwise homogeneous medium. This allows the code to be tested without requiring any changes to be made to the code.

The runs are configured using the FLASH setup script, which requires an instance of the Python interpreter to be present :

```
./setup sedov -auto
```

This automatically produces a list of the modules required to run this problem with the FLASH code and produced the corresponding `Makefile.h` and `Makefile` to compile the code. Usually, the `Makefile.h` needs to be edited and the compilers and linkers, their associated flags, as well as machine specific library paths, all need to be adjusted. The code can then be compiled using `gmake`.

The code compiled and runs successfully at: HPCx, RZG, IDRIS, ICC, BSC and SARA. At the time of writing, the porting of the simulation to HLRS has not been finished due to lack of time. On HLRS there are problems during the link stage of the compilation which still needs to be investigated. The code compiles and runs on HLRS if all HDF5-related parts are left out (run the setup script with `-with-module=io/null`), in other words no checkpoint or plot output files are produced.

The results obtained from these runs were validated using the `sfocu` application as described in Appendix 0. The results obtained on the same machine for a given number of processors are compared against the results obtained from the same run but using double the number of processors, usually 8 and 16 processors. In addition, the results obtained from different machines are also compared. In the worst cases the so-called *mag error*<sup>5</sup> was found to vary between  $10^{-13}$  to  $10^{-15}$ . This result was deemed to indicate a successful porting of the code for the hydrodynamical simulations.

#### 4. Dark matter FLASH simulation: porting and profiling of FLASH with Santa Barbara initial conditions

In this Section we describe the porting of the FLASH code, extended by Virgo with the ability to read in HDF5 files and adapted to run a dark matter simulation - the Santa Barbara (StB) data set [18]. This requires the cosmology modules simulations for dark matter (DM) only as provided by Virgo.

The Virgo simulation relies on HDF5 for both data input and data output. The input files employed throughout this work package are Virgo's StB data sets and these are all in HDF5 format thus requiring an HDF wrapper to be read in. This HDF5 wrapper, developed by John Helly [15], must be installed on each machine (see Section 6). We always use the serial component of the Parallel HDF5 distribution version 1.4.4 and install it locally on each machine. The reason for using this particular version of

<sup>5</sup> The FLASH User Guide defines the mag error as  $\text{mag error} = \sup |a-b| / \max(\sup |a|, \sup |b|, 1e-99)$ , where  $\sup$  is the supremum.

HDF5 is that this is the version installed and used on Virgo's machine Quintor at the ICC. Note that the StB data sets do not intrinsically require a *parallel* HDF library. We use the FLASH HDF5 module based on serial HDF5 which is the default. Serial HDF5 is a subset of parallel HDF5.

The FLASH Python script is then invoked using the following:

```
./setup_durham_cosmology -auto -3d -maxblocks=#,
```

where the “#” represents the maximum number of blocks allowed on a single processor and is set accordingly (we choose a value of 450 for the 64k and 128k data sets and also for the single 256k run on RZG on 128 processors) .

There are several StB data sets which come with different numbers of particles. These data sets are called 64k, 128k and 256k, where  $64^3$ ,  $128^3$  and  $256^3$  particles are included in each data set, respectively, and each data set is increasingly more computationally demanding. The scaling and profiling work with these data sets is detailed in Sections 7. and 8.

To validate the results the HDF5-tool `h5diff` [20] can be used when the results are obtained on machines from the same vendor, i.e. results from HPCx, IDRIS and RZG can be compared this way as they are all IBM platforms. Comparison across different machines is difficult because of differences produced in the numbers due to round off errors and the like. Otherwise, an IDL-script needs to be used to compare variables and properties relevant to a dark matter simulation.

The next few subsections contain details regarding the porting of the code on to the different DEISA platforms. The code was either the standard FLASH tar ball, as downloaded from the FLASH website [9], or the Virgo Simulation tar ball, which includes FLASH, as obtained from the Virgo Consortium [25].

NB: FLASH must be run in 64-bit mode in order to be able to use the amount of memory required and all `REAL` variables must be cast to 8 bytes which is a FLASH requirement.

The code was ported from the Sun version, on Quintor at ICC, to the following systems.

#### 4.1 Porting to HPCx

The following table lists the compilers and methods of accessing HDF5

Fortran90 Compiler	mpxf90_r, version 9.1.0.3
C Compiler	mpcc_r, version 7.0.0.4
HDF5	Version 1.4.4 installed locally
HDF5 Wrapper	Installed locally

**Table 2: Features of most basic tools used on HPCx.**

As a test case we run the code on 32 processors with the StB 64k data set and set the minimum refinement level to the same value as the maximum refinement level, `lrefine_min=lrefine_max=3` in the input file `flash.par`. This example will only take a few minutes to run, allowing for a rapid test of the porting process. In this case, the default stack size on HPCx, `@#stack_limit=20mb`, needs to be increased otherwise there are references to bad addresses in the file `poison_mg_residual.F90` which will cause the program to fail due to a segmentation fault.

We use `#@stack_limit=70mb` for the 64k and 128k StB data sets. Note that the `RSS_limit` is the maximum amount of physical memory per instance (884MB, on HPCx), where RSS stands for the Resident Stack Size. The `data_limit` is the `RSS_limit-stack_limit`. If the stack limit is too high (e.g. 300MB) the code will fail when attempting to allocate memory during runtime, as there will be insufficient heap memory.

## 4.2 Porting to SARA

The platform at SARA is an SGI called Aster. We access SARA via the UNICORE ssh-plugin [23].

The default object mode on SARA is 64-bit, which is our required object mode. 64-bit addressing increases the amount of memory that a program can use and removes some of the restrictions on shared memory segments so the 64-bit MPI library has some additional optimisations. The following table lists the compilers and methods of accessing HDF5

Fortran90 Compiler	ifort, version 8.1
C Compiler	icc
HDF5	module load hdf5-1.4.4pp (interactive)
	module load hdf5/1.4.4pp (batch)
HDF5 wrapper	Installed locally

**Table 3: Features of most basic tools used on SARA.**

The porting of the Virgo simulation to SARA has not been finished yet. The 64k data set produced results which were not in agreement with those obtained on the other machines when compared using the IDL script by Tom Theuns. In addition, the run tried took many fewer time steps to reach the simulation end when compared to the other machines used, where it took the same number of simulation time steps to reach the end.

## 4.3 Porting to IDRIS

The issues concerning the stack limit as discussed for HPCx in Section 4.1 also apply to IDRIS. At IDRIS it is the data limit which needs to be increased from the default value of 1.3 GB. We chose `#@data_limit=2gb` in the batch script (see the batch script in Appendix 12.3) when using up to 64 processors (see [16]). For more than 64 processors the maximum data limit allowed is 1.3GB. The following table lists the compilers and methods of accessing HDF5.

Fortran90 Compiler	mpxf90_r, version 9.1.0.2
C Compiler	mpcc_r, version 7.0.0.1
HDF5	Installed locally
HDF5 wrapper	Installed locally

**Table 4: Features of the most basic tools used at IDRIS (zahir).**

## 4.4 Porting to RZG

The following table lists the compilers and methods of accessing HDF5.

Fortran90 Compiler	mpxf90_r, version 9.1.0.4
C Compiler	mpcc_r, version 7.0.0.3
HDF5	Installed locally

HDF5 wrapper	Installed locally
--------------	-------------------

**Table 5: Features of the most basic tools used at RZG.**

#### 4.5 Porting to BSC

The following table lists the compilers and methods of accessing HDF5.

Fortran90 Compiler	mpif90, version 9.1
C Compiler	mpicc, version 7.0
HDF5	/gpfs/apps/HDF5/1.4.4/
HDF5 wrapper	Installed locally

**Table 6: Features of the most basic tools used on BSC.**

#### 4.6 Porting to HLRS

The following table lists the compilers and methods of accessing HDF5.

Fortran90 Compiler	sxmpif90, version 2.0 Rev. 313
C Compiler	sxmpicc, version Rev. 068
HDF5	Installed locally
HDF5 wrapper	Not installed yet

**Table 7: Features of the most basic tools on HLRS.**

Since we have not ported the FLASH code with the Sedov setup yet, there are no results for the dark matter Virgo simulation available either.

#### 4.7 The code on ICC

We did not port the code to ICC as it is the canonical platform used by Virgo. We took the Virgo simulation from there and ported this version to all other machines.

Fortran90 Compiler	mpf90, Studio11
C Compiler	mpcc, Studio11
HDF5	/opt/local/hdf5/64/1.4.4/
HDF5 wrapper	/home/jch/HDF5/Wrapper/64/previous/

**Table 8: Features of the most basic tools on ICC.**

## 5. Ensuring portability

This Section contains a number of observations regarding the portability of FLASH and the Virgo Simulation in general.

In the directory *FLASH2.5/setups/durham\_cosmology/Config* at ICC (Quintor), where the Virgo simulation is stored, the two logical variables, `gadget_verbose` and `gadget_cool`, need to be set to `.TRUE.` and `.FALSE.`, respectively. Originally they were set to `.T.` and `.F.` which is not part of the Fortran90 language and can result in fatal compiler errors.

In FLASH, v2.5, the variables are all declared as `REAL` instead of `DOUBLE PRECISION` or `REAL*8`, and constants are written as `1.e30` instead of `1.d30`, the Makefiles for the different platforms use compiler switches to promote the `REALS` to `DOUBLE PRECISION`. A compilation error will occur on the IBMs when a constant is written as `1.d30`. This has to be taken into account when a user adds any new code.

In addition, the line

```
read_parameters.o :
    $(F90COMP) $(FFLAGS_TEST) $(F90FLAGS) $(FDEFINES)
read_parameters.F90,
```

is added to the Makefile as recommend by the FLASH User Guide for IBMs; where the environment variable FFLAGS has been replaced with FFLAGS\_TEST, where the only difference between FFLAGS and FFLAGS\_TEST is the exchange of -O3 with -O2.

### 5.1 Problems with the routine `perfmon.F90`

The Fortran90 routine `perfmon.F90` caused problems during the porting process on two machines, namely HPCx and BSC (MareNostrum).

On HPCx problems arise when the compiler flag `-qinitauto=FF` is used to initialise all un-initialised variables which helps locate variables that are referenced before being defined. In this case the code crashes in the subroutine `h5_write.c`, line 67:

```
full_timer_data[i].t_stacks...
```

with a segmentation fault. That usually happens once the code attempts to write the first checkpoint file. This appears to be a problem with this particular FLASH routine (see the FLASH user mailing list [10]). This issue is circumvented by simply not initialising all un-initialised variables via a compiler option.

On the BSC platform, the code crashes with the error message:

```
perfmon: ran out of space on timer call stack.
```

This happens when `perfmon.F90` is compiled with either -O3 or -O2. It transpires that this routine needs to be compiled with compiler flags different from the ones we use for the rest of the Fortran90 code. The correct compiler flags on BSC are:

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

## 6. The HDF5 wrapper

The HDF5 wrapper was written by John Helly [15] allowing FLASH to read the StB HDF5 input files. The `config.AIX` file must be created for the IBMs and the Makefile has to be adapted accordingly. To allow the code to compile, we removed the underscores at the end of the C-routines on all the IBM platforms, namely HPCx, IDRIS, RZG, when C-routines are called from Fortran90.

The `HDF5_wrapper` Fortran90 code provides separate versions of each routine for REAL and DOUBLE PRECISION data, but if REAL and DOUBLE PRECISION are the same size on a system there will be two indistinguishable versions of each subroutine. One possible solution is to provide REAL\*4 and REAL\*8 versions of the routines instead.

We compile the HDF5 wrapper with HDF5-v.1.4.4. Note that, in contrast to the FLASH code, the wrapper is not compiled with compiler flags which set all REALs to 8-bytes.

## 7. Scaling study

In this Section, we investigate the execution times of the Virgo simulation for various sizes of StB initial conditions for various processor counts. We refer to this process as the Scaling Study.

The Virgo simulation using the FLASH code is run at HPCx, IDRIS, RZG and BSC for this scaling study. SARA and HLRS still need to be included in a future investigation. Unfortunately, there has not been enough time to complete the porting process of the basic FLASH code onto HLRS nor the Virgo simulation onto SARA. The simulation on SARA needs much less time steps than on all the other machines ( $n=57$  instead of  $n=159$  for the 64k data set) and the resultant matter distribution does not agree with the distribution found on all other platforms. However, the simulation does appear realistic. We shall continue to investigate the reason for this disagreement.

For this scaling study, we employ the three different StB input data files, named 64k, 128k and 256k. Within the input file `flash.par` different minimum and maximum levels of mesh refinement are chosen depending on the input data set used. The refinement levels instruct FLASH of up to how many levels of adaptive mesh refinement to use [9]. These changes are applied to simulate more realistic scientific calculations. The values employed, [25], are given in Table 9 below.

	<code>lrefine_min</code>	<code>lrefine_max</code>
64k	4	6
128k	5	7
256k	6	8

**Table 9: Minimum and maximum refinement level for the 64k, 128k and 256k StB data sets.**

### 7.1 SCALING RESULTS

It was decided that, since there were three data sets and 7 platforms and each platforms had a range of processor counts that, to achieve a high turn around of jobs, we would limit the total wall clock time of the simulation to 1 hour. It is then possible to determine the performance of each platform by which time step each simulation reaches within 1 hour. A one hour run time might not be a reasonable choice for the larger data sets (see below for the 128k data set on BSC), for example the 256k data set only reaches the fourth time step after one hour run time. Unfortunately, due to lack of time, we were unable to profile every possible combination of data set and platform. We hope to provide further results will be made available in D-JRA2-3.2.

Figure 1 shows how the 64k StB data set scales at HPCx, RZG and IDRIS up to time step  $n=41$ . Figure 1 also shows the results for the 128k data set up to time step  $n=48$  on HPCx, RZG and BSC.

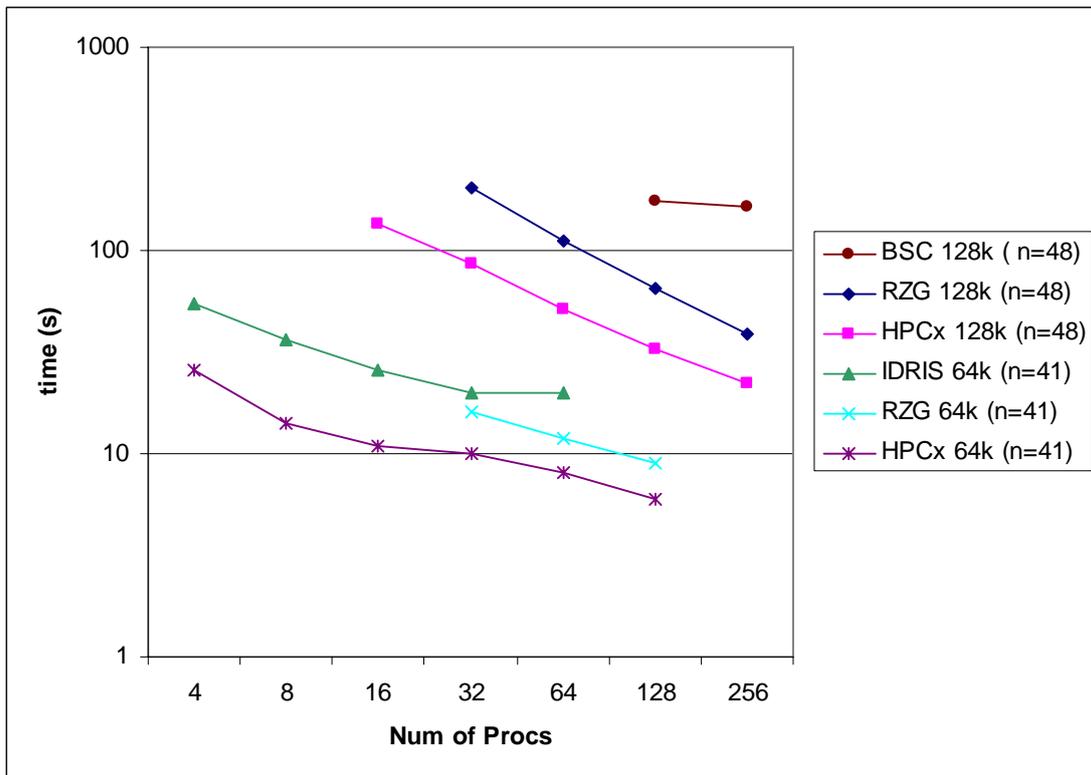
On BSC we only have results up to time step  $n=15$  for the 64k data set. Since the simulation is still in a very initial stage at that time step we compare the result on BSC with the ones obtained on the other machines in Figure 2 separately.

These results and additional ones, including a run at ICC (Quintor), can be found in Table 10 to Table 17. For RZG we only have results on 32 processors or more as it is

not permitted to use less than a complete node (i.e. 32 processors). The 128k data set was only run on 64 processors or more and in the 3-hour-queue on BSC as the supposedly small and fast 64k data set had performed relatively poorly.

The three rather similar machines HPCx, IDRIS and RZG still show that HPCx is faster than RZG which in turn is faster than IDRIS. The 64k data set would not run to completion within one hour as did the jobs on HPCx and RZG. Anyway, the 64k run does not scale very well. Load imbalance can be expected on 64 or more processors regarding the particle number we start with.

HPCx is also fastest for the 128k data set compared to RZG or the even slower BSC, though it scales better than the smaller data set.



**Figure 1: Results for the 64k StB data set up to time step  $n=41$  and the 128k StB data set up to time step  $n=48$  at HPCx, IDRIS (64k only), RZG and BSC (128k only).**

On the following page, Figure 2 shows the time needed for the 64k data set to reach time step  $n=15$  at HPCx, IDRIS, RZG and BSC. Again, HPCx is the fastest machine, followed by RZG, IDRIS and finally BSC. As discussed in the following Section, time step 15 is very likely to still include many start-up costs such as I/O and initial set-up of the mesh and is not very representative for the overall scaling of the code. Nonetheless, the basic patterns found for the 64k run up to time step  $n=41$  are already visible.

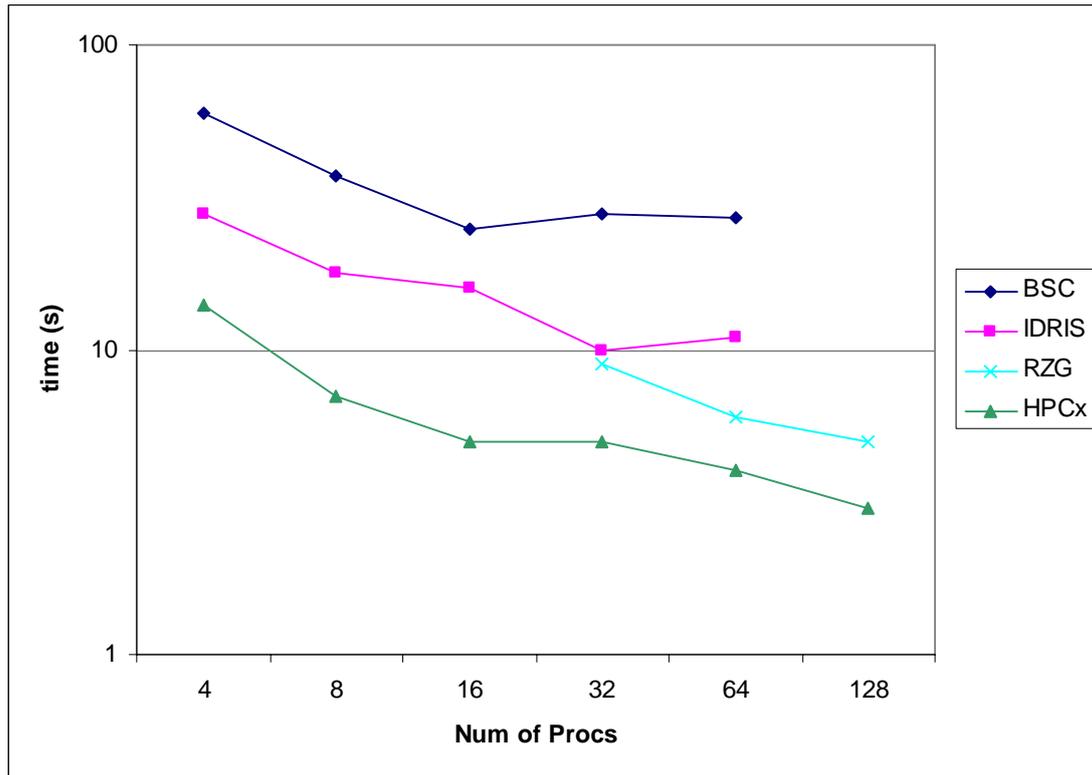


Figure 2: Results on HPCx, IDRIS, RZG and BSC for 4, 8, 16, 32, 64 and 128 processors and run up to time step n=15.

	n=15	n=41	n=159 (completion)
<b>4</b>	14	26	-
<b>8</b>	7	14	41
<b>16</b>	5	11	32
<b>32</b>	5	10	28
<b>64</b>	4	8	22
<b>128</b>	3	6	18

Table 10: Time in minutes needed on HPCx to reach certain time step n for 4, 8, 16, 32, 64 and 128 processors for the 64k StB data set.

	n=15	n=41
<b>4</b>	28	55
<b>8</b>	18	36
<b>16</b>	16	26
<b>32</b>	10	20
<b>64</b>	11	20

Table 11: Time in minutes needed on IDRIS to reach certain time step n for 4, 8, 16, 32 and 64 processors for the 64k StB data set.

	n=15	n=41	n=159 (completion)
<b>32</b>	9	16	47
<b>64</b>	6	12	34
<b>128</b>	5	9	27

Table 12: Time in minutes needed on RZG to reach certain time step n for 32, 64 and 128 processors for the 64k StB data set.

	n=15
<b>4</b>	60
<b>8</b>	37

16	25
32	28
64	27

**Table 13: Time in minutes needed on BSC to reach time step n=15 for 4, 8, 16, 32, and 64 processors for the 64k StB data set.**

	<b>n=159 (completion)</b>
<b>16</b>	1293

**Table 14: Time in minutes needed at ICC to reach time step n=159 for 16 processors for the 64k StB data set.**

The 64k StB data set runs for 159 time steps to complete. This run takes 32 minutes on HPCx on 16 processors but 1293 minutes at ICC (see Table 14). This proves the great benefit for Virgo to run on a machine like HPCx.

	HPCx	RZG	BSC	ICC
<b>16</b>	135	-	-	3115
<b>32</b>	86	205	-	-
<b>64</b>	51	112	-	-
<b>128</b>	33	65	175	-
<b>256</b>	22	-	164	-

**Table 15 Time in minutes needed to reach time step n=48 for the 128 StB data set at HPCx, RZG, BSC and ICC on 16, 32, 64, 128 and 256 processors.**

	<b>n=341 (completion)</b>
<b>64</b>	483
<b>128</b>	318

**Table 16 Time in minutes needed to complete at time step n=341 for the 128k StB data set on RZG.**

The 128k StB data set needs 341 time steps to run to completion as shown on RZG (see Table 16).

	<b>n=94</b>
<b>128</b>	1101

**Table 17 Time in minutes needed to reach time step n=94 for the 256k StB data set on RZG.**

We only have one result for the 256k StB data set so far because there were problems to find the right stack limit on HPCx. The job usually crashed with an error message which had previously always been linked to a too big value of the stack limit. In addition, the turn-around time for the 128-processor-3-hour queue is very low. Consequently, we only have one result on RZG for 128 processors and up to time step n=94 (see Table 1).

## 7.2 Start up costs

The time taken to initialise the simulation, such as reading in all the input data and setting up associated meshes, has been investigated for runs with the 64k data set on different numbers on processors on HPCx and for 16 processors at ICC.

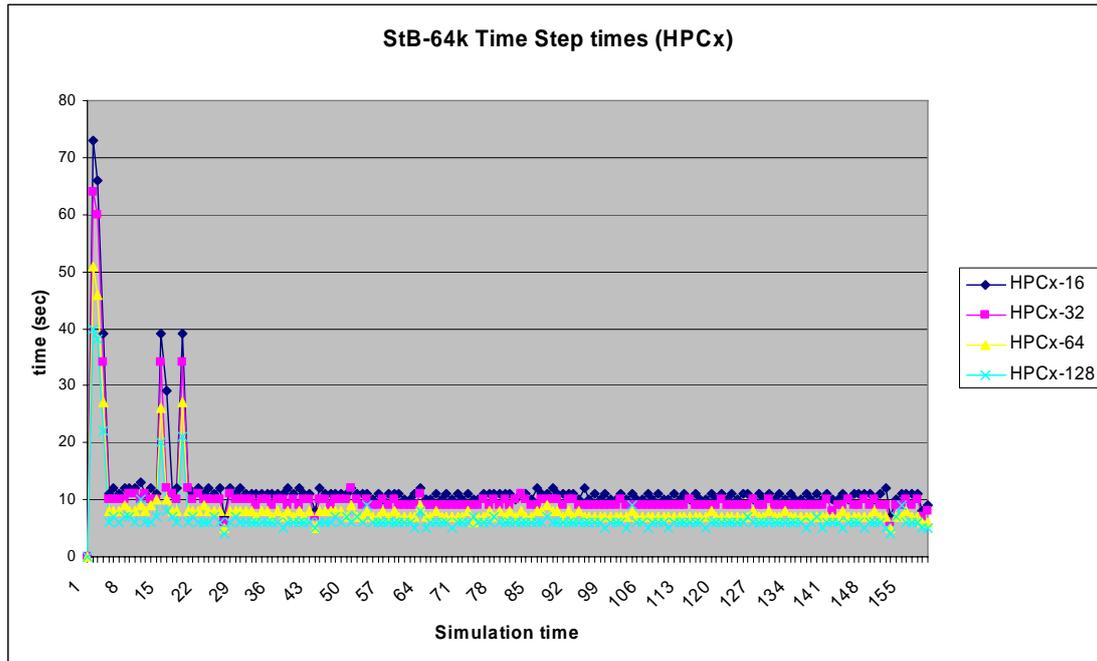


Figure 3 Shows how much time is taken for each time step from the very beginning to the end of a complete run ( $n=1, \dots, 159$ ).

It is noted that the first three time steps take at least three times as long as any other time step, and there are two to three other more time intensive steps at time steps 15, 16 and 19. This is also in agreement with the result at ICC

## 8. Profiling Study

Whilst the intention was to profile FLASH on a number of DEISA platforms, it was considered by Virgo [25] and the author of the FLASH code which is exercised most by the FLASH simulation, [21], that the profile of FLASH would not vary greatly from platform to platform for the Virgo simulation, hence the code was only profiled to any great depth on HPCx.

### 8.1 On HPCx with gprof

Initially, we did a profiling study on HPCx. For this purpose we use the tools `gprof` and `xprofiler`. The code needs to be instrumented with the additional compiler flags `-pg` and `-g`. The latter allows the user to identify the line of source code which is most time consuming.

Our first profiling case is run for the 64k StB input data set on 32 processors, with `lrefine_min=4`, `lrefine_max=6`, `maxblocks=450` and `stack_limit=100mb`. The ten most expensive routines, as determined by `gprof`, are given on the following page.

The order of these ten most time consuming routines varies slightly depending on which MPI rank we probe, as the processor with rank 0 is the master process and, as such, behaves differently from the others. The following list gives the ten most expensive routines on a non-master processor:

%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
21.5	301.40	301.40	5190648	0.06	0.06	.amr_prolong_gen_work_fun
11.2	458.09	156.69				._stripe_hal_newpkts
9.3	589.30	131.21				._lapi_dispatcher
8.4	707.55	118.25				._lapi_shm_dispatcher
4.3	767.34	59.79	112044	0.53	1.24	.poisson_mg_residual
3.6	817.75	50.41				._tag_waiting
3.3	864.27	46.52				.icopy
3.0	906.41	42.14				._cntr_waiting
2.9	947.25	40.84	320	127.62	132.08	._modulemapparticlestomesh_NMOD_mapparticlestomesh
2.9	987.82	40.57				._ptrgl

The FLASH routines are `amr_prolong_gen_work_fun.F90`, `poisson_mg_residual.F90` and the Fortran90 module `modulemapparticlestomesh.mod`, which take 21%, 4.3% and 2.9% of the total run time, respectively. It is the poisson-multigrid routines (like `poisson_mg_*.F90`) which call `amr_prolong_gen_work_fun.F90` in the end, via multiple other routines.

The remaining routines are likely to be MPI communication routines (IBM's MPI is built on top of LAPI). From this table, it would appear that at around 40% of the total time is spent communicating. The next step is to profile the code using IBM libraries which can measure the amount of time spent communicating using particular MPI routines.

## 8.2 FLASH internal profiling results

The FLASH code itself also includes a routine which reports execution times for code sections, rather than individual subroutines (which can be extended by the user by adding timers as required).

Once a FLASH run finishes to completion there is a profile attached to the end of the log file which is generated with each run. It displays the most time consuming sections of the code. We show the percentage relative to the total run time for the most time consuming routines. The indented names are subroutines of the previously named one.

The actual time in seconds for the initialization is the same on HPCx, independent from the number of processors used.

We have complete runs for the 64k data set for HPCx and RZG.

Run on HPCx for 64k StB data set on 64 processors.  
Total run time 1091 s.

Routine	Percentage of total run time
Initialization	1%
IO	0.2%
Evolution	98.7%
Particles	1.4%
Redistribution	1.0%
Particle forces	0.3%
Gravity	96.5%
Guardcell internal	0.3%
Gc_srl	0.3%
IO	0.6%
Update refinement	0.3%

Run on RZG for 64k StB data set on 64 processors.  
Total run time 2054 s.

Routine	Percentage of total run time
Initialization	1.5%
IO	0.2%
Evolution	98.4%
Particles	1.5%
Redistribution	0.9%
Particle forces	0.6%
Gravity	96.1%
Guardcell internal	0.6%
Gc_srl	0.3%
IO	0.3%
Update refinement	0.5%

The comparison between HPCx and RZG shows that the time consuming sections of the code are the same places for both machines. It has to be kept in mind, though, that these two machines are rather similar. Hence, more results on architecturally different machines would be of interest to investigate in the future.

For the 128k data set we have results for the code to have run to completion on RZG.

Run on RZG for 128k StB data set on 128 processors.  
Total run time 19215 s.

Routine	Percentage of total run time
Initialization	0.8%
IO	0.2%
Evolution	99.1%
Particles	1.9%
Redistribution	1.4%
Gravity	95.9%
guardcell internal	0.8%
gc_srl	0.7%

The comparison between the most time expensive routines shows that for the 128k data set `Particles` and, within `Particles`, the `Redistribution` takes more time of the total run time than for a smaller data set.

## 9. Code Migration

In the context of this document, we refer to Code Migration as the ability for a code to move from one HPC platform to another, during the simulation process. In other contexts, Code Migration has been referred to as the process of moving a job from one batch queue to another, before the job has started. Within DEISA, this is referred to as Job Migration and is catered for as part of the DEISA infrastructure.

The process of Code Migration is beneficial as it allows scientists to run very long simulations which would otherwise be prohibited due to time restrictions that simulations can run for at one particular site. Moreover, allowing simulations to migrate from one site to another means a faster turn-around time for a single simulation.

A survey of the State of the Art was carried out, investigating Cactus [4], Condor [5], Gridlab [11] and RealityGrid [22]. There is no explicit mechanism for Code Migration in RealityGrid. In Cactus, Condor and Gridlab, Code Migration is provided, given that the Grid in question is homogeneous or, at least, each platform is binary compatible. The executable is then staged on the next platform, where staging an executable (or, indeed, any kind of file) means to ensure that the executable (or file) is present on the target platform before the executable (or file) is utilised. Condor is more powerful in that the entire state of a running code is saved and restarted on another binary-compatible machine.

DEISA is, by construction, a more general system than those described for Cactus, Condor and Gridlab, since DEISA is a heterogeneous environment, and the platforms are, in general, not binary compatible, therefore the executables cannot, in general, be staged.

### 9.1 General Description

The general plan to migrate FLASH during a simulation on DEISA is as follows.

- **Stage 1:**  
Submission to Batch System: FLASH is submitted to the DEISA batch system and the simulation begins running on a particular platform, given that FLASH has already been ported to that platform. The platform is either specified by the user in advance or chosen by a brokering system.
- **Stage 2:**  
Code Aware of Elapsed Time: The FLASH code detects that there is only a predefined amount of time left before the job is terminated, it stops the simulation and dumps a set of 'restart' files.
- **Stage 3:**  
Automatic Code Resubmission: A script then resubmits the simulation to the DEISA batch system. This requires that each potential target host already has a version of the executable which is compiled, installed and optimised for that site. Thus this process does not require a homogeneous cluster.
- **Stage 4:**  
Portable Restart Files: The code then begins running on a platform, using the restart files as input.
- **Stage 5:**  
Repeat Until Completion: The process is repeated until the simulation has completed.

## 9.2 DEISA Specifics

### 9.2.1 Stage 1: Submission to the Batch System

For Stage 1 to function, the DEISA batch system must contain a list of platforms where FLASH has been ported to in advance and has been given approval to be executed. The FLASH simulation will only be run on platforms within this list. The choice of platform will be made by the user, as DEISA does not currently include a brokering system.

### 9.2.2 Stage 2: Code Aware of Elapsed Time

Stage 2 is already possible, since FLASH is already able to perform this function. It does this by taking note of the batch script's hard wall clock limit and notes how long the code has been running for.

### 9.2.3 Stage 3: Automatic Code Resubmission

Stage 3 can be achieved using DEISA's UNICORE workflow [26]. It might also be achieved using the DESHL [7], in conjunction with a high level script, such as a Unix script, however the DESHL does not currently support UNICORE's workflow features. (The DESHL is a standards-based command line interface to a subset of UNICORE features).

There are a number of potential stumbling blocks with using either UNICORE or the DESHL. Within the UNICORE workflow, the initial and each subsequent platform must be specified in advance, which theoretically means the simulation will take longer to perform since the first available platform is not used. This restriction is shared with the DESHL. One solution might be to use the brokering service offered by UNICORE, where a machine is nominated automatically, given that the machine is available to be used and that the simulation has already been ported to that machine. However, UNICORE's brokering service is currently not available on DEISA (nor via the DESHL).

Another issue may be in coping with exit status codes. The simulation may stop on a particular platform because it has hit the clock limit or it has completed, or it has failed. The failure may prove difficult to detect, since UNICORE and, hence, the DESHL, differentiate between exit status codes, however, FLASH's own self-management system, as described in 9.2.2, may alleviate this issue.

### 9.2.4 Stage 4: Portable Restart Files

The particular simulation under consideration in this Workpackage only reads and outputs HDF5 which is a binary, portable data format.

HDF5 was investigated as part of JRA2's Workpackage 2, where it was found that HDF5 outperformed all the other binary, portable data formats tested. It has since been adopted as the portable, binary data format of choice by VirtU [28] and Virgo [26], especially since it is compatible with VOTable [29], the data format of the European Virtual Observatory community [8].

Here, we have two choices:

1. We can either leave the restart files to reside on the platform on which they were written, and then once FLASH restarts on another platform, these restart files will be available via the DEISA MC-GPFS. However, the time to access these files will be determined by how busy the network is at the point when the files are read, and this delay will be paid for as part of the batch job.

2. The restart files can be staged automatically at the forthcoming platform, when using the workflow feature of UNICORE and/or the DESHL, irrespective of whether the forthcoming platform is part of the MC-GPFS or not.

## 10. Discussion of results and future work

### 10.1 Delays and difficulties

Porting the code proved to be problematic, as the Virgo Consortium updated their modified version of the FLASH code a number of times with algorithmic changes, bug fixes and switching the underlying version of FLASH employed from FLASH 2.4 to 2.5. This made porting a time consuming process. This process was made more difficult as the method of ensuring the port was correct though initially straightforward for similar HPC platforms proved to be far from straightforward for dissimilar platforms. Non-particle setups can use the `sfoctu` tool, which comes with FLASH, to ensure the correctness of the results. However, this tool does not currently provide output for particle simulations and currently there is no other tool available to check particle-related results in this manner.

A final definitive version of the Virgo Simulation was made available by the beginning of March 2006 and this code has now been ported to HPCx, IDRIS, RZG and BSC. The process of porting the simulation to SARA and to HLRS has begun, however, the simulation does not currently produce what are considered to be “correct” results, i.e. the simulation runs to completion, however the resultant distribution of matter, whilst reflecting reality, does not agree with the distribution of matter found using HPCx, IDRIS, RZG and BSC. We shall endeavour to port the code to both SARA and HLRS and the results will be reported in D-JRA2-3.2.

Further delays were also incurred as the UNICORE ssh-plugin was found to be unstable. (Whilst UNICORE can be used for submitting batch jobs, we required to be able to log into each DEISA platform in order to port the code to that platform. In the case of SARA, such access was only available via the UNICORE ssh-plugin).

Finally, there are the typical issues of running HPC codes on national services, namely a slow throughput of jobs, services being taken offline for maintenance, and delays caused through general ignorance of local variations in working practices. Thankfully, DEISA is working towards providing a feel of uniform accessibility.

### 10.2 Review of work done

We ported the code, FLASH2.5 for the Sedov setup onto HPCx, IDRIS, RZG, BSC and SARA. In addition, we ran it on ICC to have a baseline against which numerical comparisons could be made, to ensure each port was correct.

Then we ported the Virgo Simulation, which includes FLASH2.5 as its engine, to HPCx, IDRIS, RZG and BSC and tested the simulation for three differently sized StB input files, containing  $64^3$ ,  $128^3$  and  $256^3$  particles.

At present, the results on SARA runs to completion but do not agree with the results from any the other platforms. This will be investigated further and reported upon in D-JRA2-3.2.

Unfortunately, we have yet to successfully port the basic FLASH2.5 code to run the Sedov setup, when run with the HDF5-related FLASH modules enabled. This will be investigated further and reported on in D-JRA2-3.2

We ran a scaling study for the Virgo simulation. For both the 64k and the 128k StB data set the fastest machine is always HPCx followed by RZG, IDRIS and BSC. The speed-up for the 64k data set turned out to be rather poor. This is due to the fact that the data set is not large enough to scale to many processors. The 128k data set shows better speed-up. It is difficult to run the 256k data set, as it requires a high number of processors, a very large total execution time and is memory demanding. The memory issue requires some experimenting with the stack limit on HPCx or the data limit on IDRIS:- we only have a result for the 256k data set on RZG so far. The poor scaling associated with the 64k data set, and the issues raised for the 256k data set, make the 128k data set the obvious choice for the best initial conditions file to use for all remaining investigations. The simulation using the 128k data set exhibits about 70% parallel efficiency<sup>6</sup> on HPCx when run on between 32 and 64 processors (78% and 66%, respectively).

A detailed breakdown of the time taken for each time step of the simulation showed consistent behaviour, independent of both the number of processors employed and the platform used, such as HPCx (IBM) or ICC (Sun). It is the initial time steps (the first three and then another three peaks at around time step 15) which take considerably more time. This is due to the initialization processes within the code, I/O, the mesh setup, etc. Compared to the overall runtime, the initialization takes less than 2% of the total time as confirmed by the FLASH internal profiling result.

On HPCx, for example, there are also MPI trace tools available which provide detailed information about MPI performance, such as `gprof`, which permits a line-by-line breakdown of the source code to identify the bottlenecks in the code. The availability and potential of the profiling tools available on each of the platforms will be investigated for D-JRA2-3.2.

We also identified the ten most time consuming routines via `gpof` on HPCx. The most time consuming routines were found in `amr_prolong_gen_work_fun.F90` with 21% of the total run time, followed by `poisson_mg_residual.F90` (4.3%) and the module `modulemapparticlестomesh.mod` (2.9%). These will have to be investigated in more detail.

It is thought that the MPI performance may be poor within the dark matter gravity computation, and this is supported by the `gprof` profile on HPCx, where a considerable amount of time is spent in communication routines. There may be some fundamental bottlenecks, such as the relaxation requiring boundary transfers to synchronize the processors. More communication is also required at the fine grain scales. It is suggested that the performance from employing the basic point-to-point MPI communications routines may be improved by replacing these with collective communication routines.

### 10.3 Future work

This subsection lists, in no particular order, a number of tasks which will be investigated and reported on in D-JRA2-3.2

1. A final attempt to port the Virgo Simulation to both SARA and HLRS will be made.
2. The 128k StB data set will be employed on all platforms where the simulation has been shown to be realistic.

---

<sup>6</sup> The parallel efficiency is defined as the time needed for the sequential run for a given problem size divided by time needed for a run on P processors for the same problem size and divided by the number of processors P.

3. The simulation will be profiled in greatest detail on HPCx, where particular subroutines will be considered for optimisation. These subroutines are likely to include the so-called Multi-Grid solver and the MPI communication routines.
4. Platform-independent optimisations will then be attempted upon the most time consuming sections of the simulation.
5. Platform-dependent optimisations may also be considered, if time permits, and the platform in question performs particularly poorly.
6. The design for code migration, presented here, will be tested and incorporated in FLASH if the tests are successful.
7. A method of introducing multi-time stepping will be investigated.

## 11. Appendix: General Machine Information

This Appendix provides an overview of the machines available through DEISA plus the ICC platform that were used.

### 11.1 HPCx

The HPCx [18] 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 GBytes of main memory.

Inter node communication is provided by an IBM's HPS.

### 11.2 SARA

SARA Computing and Networking Services is an advanced ICT service center that supplies - since 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 [23] 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 TFlops.

The 416 processors are divided over 5 nodes: 4 batch nodes and 1 interactive node. It is a little-endian machine. Every node in Aster is a CC-NUMA machine. 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.

### **11.3 IDRIS**

Created in 1993, IDRIS (Institute for Development and Resources in Intensive Scientific computing) is CNRS's national supercomputing center. Together with a second national supercomputing center - 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 highly efficient Users Support teams.

The service at IDRIS [6], *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 IBM's HPS.

### **11.4 RZG**

Since 1992 the Rechenzentrum Garching (RZG) is operating as a common computer center of the Max Planck Institute for Plasmaphysics and the Max Planck Society, offering services for Max Planck Institutes all over Germany. Since four decades now key tasks provided by the RZG include: 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, with four links per Regatta node:

- Processor clock: 1.3 GHz
- Peak performance per proc: 5.2 GFlops
- 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 TFlops
- Total main memory: 2 TB.

### **11.5 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, and the 8th in the world, according to November 2005 top500 list.

MareNostrum currently consists of 4,812 2.2 GHz IBM PowerPC 970FX processors operating under the IBM Linux OS system with 9TB of memory in total and 233 TB of disk space. The peak performance of the machine as a whole is 42.35 TFlops. There are 2 CPUs per node, where each node has 4GB of main memory. 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.

### **11.6 HLRS**

The High Performance Computing Center (Höchstleistungsrechenzentrum) Stuttgart (HLRS) of the University of Stuttgart, Germany, 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 SX-8 vector supercomputer with 576 processors, 9.2 TB memory space and 180 TB disk space. The peak performance is 12.67 TFlops. The backend systems are 72 SX-8 vector nodes, each having 8 CPUs of 16 Gflops peak (2 GHz). The CPU can access the main memory with 64 GB/s. Each node has 128GB of memory, with about 124GB available for applications.

The vector nodes 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. MPI latency of around 5 microseconds for small messages can be expected.

### **11.7 ICC**

The Institute for Computational Cosmology (ICC) [17] 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) [17] 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.

## 12. Appendix: Machine specific scripts

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

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

### 12.1 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 [23].

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 requires the use of compiler switches to promote `REALS` to `DOUBLE PRECISION` [9]; here, we employ `-qrealsize=8`. 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 otherwise.

The batch script is:

```
#!/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 = 70MB
#@ 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

cd
/hpcx/work/e24/e24/elena/Deisa/durham_flash/FLASH2.5/object

poe ./flash2
```

## 12.2 SARA

On the SGI, Aster, at SARA, the Fortran90 compiler flags used are:

```
-O3 -c -i8 -i4
```

The batch script used is:

```

# MPI job
#BSUB -W 2:30          #Job takes 2 hours and 30 minutes
#BSUB -M 8388608      #Job needs 8 Gigabyte
#BSUB -J flash        #Job is called flash
#BSUB -e stderr.%J    #Job sends stderr output to stderr.job_id
#BSUB -o stdout.%J    #Job sends stdout output to stdout.job_id
#BSUB -n 4            #Job uses 4 CPUs.

. /opt/modules/init/ksh
module load hdf5/1.4.4pp

cd path/to/my/stuff
mpirun -np 4 flash2

```

### 12.3 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 [9], 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:

```

#@ 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

```

## 12.4 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

```

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 [9], 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:

```

#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
#
#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

```

## 12.5 BSC

Before running `gmake` on the IBM Beowulf, MareNostrum, at the 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_FUILLIB 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
```

## 12.6 HLRS

Note that the compiler flag `-Cvopt` is switched on by default. The flag `-Cdebug`, on the other hand, suppresses optimization, automatic vectorization and automatic parallelisation.

We employ the following compiler flags:

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

## 12.7 ICC

The following optimized compiler flags are used on the Sun Beowulf, Quintor, at the ICC:

```
-fast -xtarget=ultra3i -xarch=v9b -fpp -xtypemap=real:64,integer:32 -  
xcache=64/32/4:1024/64/4
```

The batch script is:

```
#!/bin/tcsh  
#  
# (c) 2000 Sun Microsystems, Inc.  
#  
# -----  
# User needs to customize the following items  
# enclosed by <>  
#  
#$ -N CB-SBGasHiRes  
#$ -S /bin/csh  
# $ -o <filename of your choice for std output>  
# $ -e <filename of your choice for std error>  
# -----  
# no mail should be sent  
#$ -m n  
# -----  
#  
# Execute the job from the current working directory  
#$ -cwd  
#  
# Parallel environment request  
# -----  
#$ -l cre  
#  
# -----B  
# parallel environment:  
#  
#$ -pe hpc-1 16  
# -----  
#  
# all resources are defined here  
#  
limit coredumpsize 0  
unlimit stacksize  
#  
/opt/SUNWhpc/bin/mprun -x sge -np $NSLOTS ./flash2 > fla.out  
# -----
```

## 13. Appendix: Compiler Flag Specifics

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

### 13.1 IBM-specific compiler flags

On the IBM machines we use the XL Fortran, version 9.1. Xlf95 invokes the Fortran90 compiler and sets `-qfree=f90` (free-form source); the source file suffix 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 `-lpthreads`; they are the thread-safe versions. It is also recommended for MPI codes.

The `-qsuffix` option is used to specify the source file suffix on the command line. For example `mpxlf90_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
- Type-less 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 [6], (Chapters 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 fortrareal/4 fortranreal/8
```

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

### 13.2 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 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`.

### 13.3 *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).

### 13.4 *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 detailed option `-A idbl` is used for 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, variables or constants with KIND type parameter are not expanded.

## 14. Appendix: Validation of results and the `sfocu` tool

DEISA-JRA2-WP3 needed a tool to ensure consistency of results on the same machine when using different numbers of processors and in order to compare results between the different machines in the DEISA infrastructure. If the resulting difference obtained in the results from these runs it would be deemed that the porting of the code had been done successfully on to the given machine. As the main simulation platform at this point used by the Virgo Consortium was ICC this was deemed to provide the canonical set of results against which the initial comparisons would be made.

Paul Ricker [21] suggested using `sfocu`, the serial FLASH output comparison utility, which comes with the FLASH distribution, located in the `tools/sfocu` directory. As input, `sfocu` takes the names of two FLASH HDF5 files as command line arguments and compares the files block by block and variable by variable. Four different error measures (min, max, abs and mag) are used and the results for each of these are reported. If two output files differ by more than round off error then the pattern of errors can tell you something about which solvers are failing. Differences for the same setup and parameter setting run on two different machines should be no greater than 1E-10 if double-precision arithmetic is used. Errors like 1E-3, or errors of order unity, call out for explanation<sup>7</sup>. At the time of writing `sfocu` does not allow to check particle data, in other words, it does not check the results obtained from DM-simulations. When comparing results obtained on the same platform, or for architecturally nearly identical platforms such as HPCx, IDRIS and RZG, it is possible to compare the results with `h5diff`. This is part of the HDF5 library. Otherwise Tom Theuns IDL script to compare the results needs to be employed.

The `sfocu` utility reports a failure even if the numbers are a little different, this is because the FLASH developers usually run `sfocu` on the same machine to report failures in their test suite. When running the code on two different machines, you do not expect the numbers to be exactly identical because round off policies on different machines are generally different and compiler optimizations will introduce quantization differences which will be reflected in the output produced. The key is to look at the column indicating the mag error. These

<sup>7</sup> Email by Paul Ricker from 27<sup>th</sup> January 2006.

errors are well within an acceptable range due to quantization differences when they are about the order  $10^{-13}$  to  $10^{-15}$  **Erreur ! Source du renvoi introuvable..**

## 15. Appendix: Usage of the debugger on BSC

At this moment there is no parallel debugger on BSC. If a code needs to be debugged the only debuggers available on MareNostrum are GDB and DDD (located in `/gpfs/apps/DDD/bin`). These debuggers can be used with parallel codes by attaching `gdb` to one of the parallel processes and then using `gdb` like a serial program. We usually use a script like `debug.sh` to attach `gdb` to the master process of the execution:

```
EXE=$@

if [ "$GMPI_ID" = 0 ]; then
xterm -e gdb $@
else
$EXE
fi
#####
```

To run the `gdb` debugger the following steps need to be executed:

- `ssh -X loginname@mn3.bsc.es`
- `echo "localhost">a`
- `mpirun DISPLAY=$DISPLAY -np 2 -machinefile a ./debug.sh ./flash2`