|epcc|

THE UNIVERSITY OF EDINBURGH

Optimising FLASH for Cosmological Simulations

Chris Daley

August 24, 2007

M.Sc. in High Performance Computing

The University of Edinburgh

Year of Presentation: 2007

Abstract

Runtime performance is a limiting factor in large cosmological simulations. A parallel application code called FLASH, which can be used for cosmological simulations is investigated in this report. FLASH is an Adaptive Mesh Refinement (AMR) code, parallelised using MPI, and incorporating a Particle-Mesh (PM) Poisson solver. Profiles of a cosmological simulation based on FLASH, indicate that the procedure which accumulates particle mass onto the mesh is the major bottleneck. A parallel algorithm devised by the FLASH center may help to overcome this bottleneck. In this project, the algorithm is implemented for the latest beta release of FLASH, which cannot currently perform cosmological simulations. There is interest in FLASH because the adaptive mesh can be refined to resolve shocks in cosmological simulations. In addition, the new version of FLASH is designed with an emphasis on flexible and easy to extend code units.

The communication strategy in the implemented parallel algorithm does not involve guard cell exchanges (halo swaps). As such, it uses less memory than the previous implementation in the previous version of FLASH. Due to time restrictions, the delivered implementation is only compatible with a uniform grid in both uniform and adaptive grid modes. In addition, results indicate it is approximately 3-5 times slower than the previous implementation. This is using a 3D simulation involving 128^3 particles on up to 64 processors. Potential optimisation strategies are discussed in the report.

Contents

1 Introduction **1**
1.1 Overview . . . 1
1.2 The `MapParticlesToMesh` Procedure . . . 3
1.3 Outline of Sections . . . 3

2 Algorithms **5**
2.1 FLASH Geometry . . . 5
2.2 Assigning Particle Mass to the Mesh . . . 7
2.2.1 FLASH 2.5 Algorithm . . . 8
2.2.2 FLASH 3.0 Algorithm . . . 8

3 Implementation **11**
3.1 Accumulating Mass in a Single Block . . . 11
3.2 Accumulating Mass in Multiple Blocks . . . 12
3.3 Communicating the Particle Datatype . . . 15

4 Installation **16**
4.1 The IVS on HPCx . . . 17
4.1.1 Santa-Barbara . . . 17
4.1.2 Pancake . . . 20
4.2 FLASH 3.0 on HPCx . . . 20
4.2.1 Use Cases . . . 20
4.2.2 Sedov . . . 21
4.2.3 Pancake . . . 22

5 Results and Analysis **25**
5.1 Runtime Performance . . . 25
5.2 mpiprof . . . 26
5.3 xprofiler . . . 30
5.4 Paraver . . . 33
5.5 HPCx Specific Optimisations . . . 34

6 Adaptive Mesh Implementation **36**

7 Future Optimisations **39**

7.1 Calculation Optimisations 39
7.2 Communication Optimisations 43

8 Conclusions 44
8.1 Conclusion 44
8.2 Summary of achievements 46
8.3 Future work 46

List of Tables

5.1 Time for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles on various numbers of processors for both IVS and FLASH 3.0 beta 26

5.2 Time and MPI communication information for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles, and a uniform grid on various numbers of processors for both IVS and FLASH 3.0 beta . . 29

List of Figures

2.1 Block structure in a 2D domain. 6
2.2 Child blocks created during a refinement in a 3D domain. 7
2.3 Sieve algorithm pseudo-code 10

3.1 Assigning mass to nearest neighbour grid points in a 2D domain 12
3.2 Accumulating mass across an internal boundary in a 2D domain 13
3.3 Accumulating mass across an external boundary in a 2D domain, with periodic boundary conditions in both directions. 14

4.1 IDL script output from an IVS run using 32 processors with 128k StB dataset, against a Hydra_MPI run with 512k StB dataset 19
4.2 Totalview screenshot showing error during an adaptive grid simulation with `lrefine_min=lrefine_max=4` 23

5.1 Technique used to time mass assignment procedure 25
5.2 Time for 10 iterations of `Grid_mapParticlesToMesh` , when using 128^3 particles on various numbers of processors for both IVS and FLASH 3.0 beta 27
5.3 Totalview screenshot showing error during profiling with mpiprof . . . 28
5.4 Time for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles, and a uniform grid on various numbers of processors for both IVS and FLASH 3.0 beta 29
5.5 xprofiler profile for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles, and a uniform grid on 8 processors 31
5.6 xprofiler line-by-line profile of a section of `maptononhostblock` over 10 iterations, when using 128^3 particles, and a uniform grid on 8 processors 32
5.7 Paraver trace for 1 iteration of `Grid_mapParticlesToMesh`, when using a 128^3 particles, and a uniform grid on 8 processors 33
5.8 Runtime performance using different compilation flags, when using 128^3 particles, and an adaptive grid with `lrefine_min=lrefine_max=3` on various numbers of processors. 35

6.1 Performing prolongation to generate a symmetric mass cloud in a 2D domain 37
6.2 Performing restriction in a 2D domain. 38

Acknowledgements

Firstly, I would like to thank my supervisor Gavin Pringle for all his help with the project. He organised a useful project by coordinating with people at the FLASH center and the Virgo Consortium. In addition, he has always been available to answer any of my questions.

I must also thank key people from the FLASH center. In particular, Anshu Dubey who developed the Sieve algorithm used in this project. Anshu Dubey provided a useful skeleton code to use in the project, and created key code components at short notice. Other members that helped with my project are Klaus Weide, Lynn Reid and John ZuHone. Thanks also to Tom Theuns from the Virgo Consortium.

I would like to thank Elena Breitmoser for helping to install and verify the Improved Virgo Simulation, and also for proof reading this dissertation. Also, thanks to Chris Johnson for explaining the Paraver profiling tool.

A special thanks to my fellow M.Sc. student Bruce Duncan. He has always been available to answer various UNIX questions, and has helped hugely with my understanding of the FLASH application.

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.

Chapter 1

Introduction

1.1 Overview

This project investigates the optimisation of a procedure in the FLASH simulation code [1]. FLASH is a modular Fortran90 code, developed initially to study thermonuclear flashes, and is parallelised using the Message Passing Interface (MPI) [2]. FLASH is an Adaptive Mesh Refinement (AMR) code that uses the PARAMESH package [3] to discretise the computational domain into a block structured adaptive grid. The provision of an adaptive grid is an extremely desirable feature. This is because it enables a high resolution grid to be applied to only interesting regions of the computational domain, so that extra insight can be achieved, without requiring the added computational time and storage of a high resolution uniform grid. The grid is also dynamic, in the sense that the discretisation of the domain changes as the simulation evolves.

Recently, the code was extended with new modules to perform cosmological simulations [4]. This has involved creating modules that add particles to the system which simulate dark matter particles [5]. Current theories believe that the universe contains a large quantity of unseen matter, "dark matter", which interacts with visible matter by gravitational force only. Therefore, inserting the missing mass component into cosmological simulations is critical. One of the attractions of using FLASH for cosmological simulations is that its adaptive mesh can be refined to capture shocks in the simulation. This can be used, for example, to resolve the sudden acceleration caused by an exploding star.

The gravitational force between particles in a simulation is generally evaluated using one of three techniques. These methods are named particle-particle (PP), particle-mesh (PM), and particle-particle-particle-mesh (P^3M) [6]. Although other techniques exist, such as the Barnes-Hut tree algorithm [7]. Originally, cosmological simulations were particle based codes (PP). This means that gravitational force between particles are evaluated as a direct N-Body sum, which is of order $O(N^2)$ work. FLASH, however, is a particle-mesh based code (PM) which approximates the force between particles. This

is of order $O(NlogN)$ work, and is generally faster than the direct summation method for large N.

One research group that is extremely interested in using FLASH for cosmological simulations is the Virgo Consortium [8]. The research areas of Virgo are described on their website as "large-scale distribution of dark matter, the formation of dark matter haloes, the formation and evolution of galaxies and clusters, the physics of the intergalactic medium and the properties of the intracluster gas" [8]. This is challenging research which is performed on supercomputer resources around the world. Machines used include the IBM Regatta system at Max-Planck Rechenzentrum in Garching, hereafter referred to as RZG, the IBM Regatta UK national supercomputer called HPCx, and various Sun SMPs termed "Cosmology Machines" at Durham. Up until now they have only used simulations with P^3M or treecode methods. They are therefore very interested in using FLASH because it is an AMR code which, as such, can resolve shocks.

The extended FLASH simulation code is known to perform poorly for dark matter simulations [5]. In [5], a profile is shown for the 64k Santa-Barbara (StB) input dataset on 32 processors on HPCx. The Santa-Barbara dataset refers to a dataset that is representative of a dark matter only cosmological run, and 64k indicates 64^3 particles. The profile reveals that a large portion of runtime is spent solving Poisson's equation to evaluate the force between dark matter particles. In this implementation, an iterative solver based on a MultiGrid method is used to solve Poisson's equation. Such an algorithm is known to perform well in serial, but does not parallelise easily. In an attempt to improve parallel performance, the solver is replaced by a Fast Fourier Transform (FFT). The FFT solver is implemented by Tom Theuns [9] of the Virgo Consortium and EPCC. The resultant code is described in detail in [5], and is henceforth referred to as the Improved Virgo Simulation (IVS).

The IVS is evaluated in [10]. Profiles reveal that the time to complete the StB simulation for a 64k and 128k data set is greatly reduced. Function level profiles for IVS indicate that the new bottlenecks are the subroutines `MapParticlesToMesh` and `RedistributeParticles`. It is also shown that the code does not scale past 64 processors for a reasonably sized dataset (128k StB). The tests are performed on the machines: The IBM Power PC, MareNostrum, at BSC in Barcelona, hereafter referred to as BSC, the vector NEC-SX8 at Stuttgart, hereafter referred to as HLRS, HPCx and RZG. This is a disappointing result as high application scalability is crucial for tackling larger and more complicated problems. The limited scalability has motivated this project, which will attempt to overcome the largest bottleneck in the application code.

Work in this project is devoted to the optimisation of the `MapParticlesToMesh` subroutine only. This is because there is already a new implementation of `ReDistribute-Particles`. No performance figures are available, but scalabilty is vastly improved [11]. The subroutine is implemented for FLASH 3.0 beta. It should be noted that the IVS is an extended version of FLASH 2.5.

FLASH 3.0 beta is a preliminary version of the FLASH center's next major release, FLASH 3.0. It includes the design enhancements of the full release, but does not con-

tain the same level of functionality as exists in FLASH 2.5. The focus of FLASH 3.0 is about creating code units that are more flexible and easier to extend [13]. This has been achieved by making the design much more object-orientated. The promise of greater usability appeals to many current users of FLASH, including the Virgo Consortium. This prompted the decision to abandon the planned optimisation of the IVS in this project. Therefore, instead of optimising `MapParticlesToMesh` in FLASH 2.5, as in the original plan, it was decided to implement the procedure for FLASH 3.0 beta. A discussion of the risks associated with choosing a project based on FLASH 3.0 beta appears in the author's project preparation report [12].

1.2 The `MapParticlesToMesh` Procedure

To understand the purpose of the `MapParticlesToMesh` procedure, a brief introduction to PM techniques is first provided.

The mesh based approximation technique was developed to evaluate forces between N particles faster. This is necessary in the field of cosmology because realistic simulations typically use $N > 10^7$ particles [15]. This equates to a high computational time to evaluate the gravitational forces between N particles. Such a cost can be prohibitive. One way to reduce this work is to approximate the gravitational forces using a PM technique. The steps performed during a single time-step are described in more detail in [16] [17], and are briefly summarised below.

1. Assign particles' masses to the mesh using an interpolation scheme. Different interpolation schemes are used depending on their accuracy and computational cost. The default scheme in FLASH is Cloud-In-Cell (CIC) [1].
2. Solve Poisson's equation on the mesh in order to obtain the gravitational potential.
3. Perform a finite difference of the gravitational potential to obtain the force at each mesh point.
4. Interpolate the forces back to the particle positions
5. Advance the particles' position and velocity.

`MapParticlesToMesh` performs the task specified in Step 1. It is used to assign the particle mass to grid points on the mesh. An efficient implementation is crucial in the FLASH code, and in any well-designed PM simulation code, for that matter.

1.3 Outline of Sections

Chapter 2 gives an overview of the algorithm implemented in IVS and the proposed algorithm for FLASH 3.0 beta. Also included, is background information which explains

how the the computational domain in a FLASH simulation is decomposed. This helps to explain the key differences between the algorithms.

Chapter 3 describes the implementation of the algorithm. It includes how the implementation is designed to cope with the different boundary condition modes of FLASH.

Chapter 4 explains the installation of actual scientific simulations for IVSand FLASH 3.0 beta. These scientific simulations are used to quantify how output from `Grid_mapParticlesToMesh` compares to `MapParticlesToMesh`. Also discussed are the test cases created to ensure the delivered implementation performs as intended.

Chapter 5 uses a scientific simulation introduced in the previous chapter to assess the performance of `Grid_mapParticlesToMesh`. The performance of `Grid_mapParticlesToMesh` is compared against `MapParticlesToMesh` for a variety of processor counts. Various profiles of `Grid_mapParticlesToMesh` are taken to explain the reasons for the observed poor performance.

Chapter 6 describes the approach taken to accumulate particle mass across blocks which exist at different refinement levels.

Chapter 7 discusses future optimisations based upon the results from the Chapter 5. Then finally, Chapter 8 is a summary of achievements and evaluation of the project.

Chapter 2

Algorithms

Section 2.1 gives an overview of the structure and placement of blocks in FLASH. This is necessary to understand the intricate details of the proposed algorithm described in Section 2.2. The implementation of the Sieve algorithm is described in Chapter 3.

2.1 FLASH Geometry

The computational domain in IVS and FLASH 3.0 beta consists of a number of blocks which are distributed amongst processes. The structure of each block is identical. Here, each block contains `NXB`, `NYB`, `NZB` internal grid points, and `NGUARD` guard grid points at each block boundary. The block structure in a 2-dimensional domain is shown in Figure 2.1. A guard grid point is henceforth referred to as its more common name of guard cell. In other publications, a guard cell is also described as a halo cell. These parameter values and particular problem constraints (e.g. type of boundary conditions) are described in a file named `flash.par`, which is read by the FLASH executable at runtime.

In IVS, the computational domain is decomposed using the PARAMESH package only. This is also available in FLASH 3.0 beta, but there is also the option of decomposing the computational domain with a uniform grid. A uniform grid is appealing as it has no AMR related overhead [13].

In uniform grid mode, only one block per process exists during the entire simulation. This is a logical decision on the basis that refinements do not occur at any time. Since a block always retains the same number of grid points, running a simulation with more processes increases the overall resolution. In real terms, this means that a particular problem can be solved more accurately (in general) by distributing it across more processors.

In adaptive grid mode, the number of blocks per process is *not* fixed. Blocks are created and destroyed during the course of a simulation to change the resolution in different

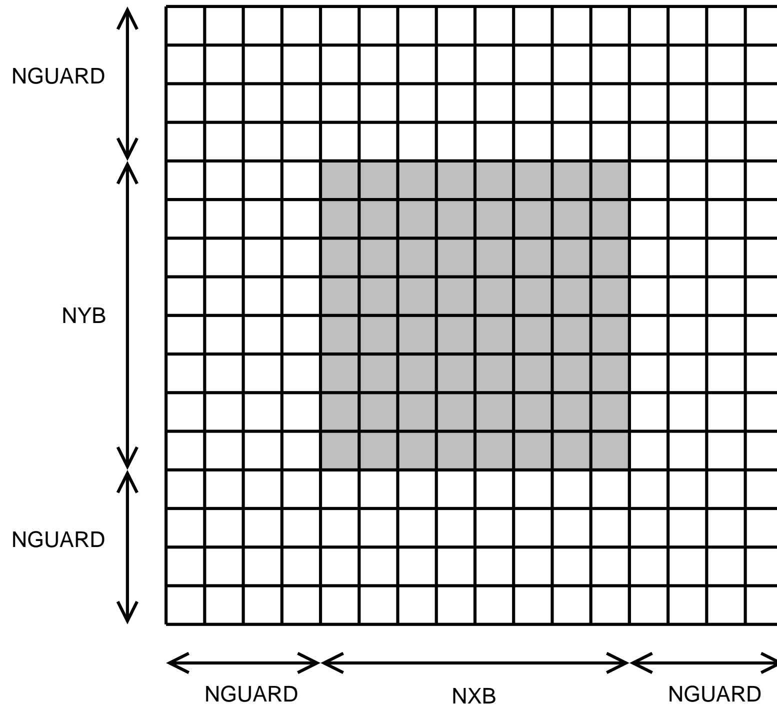

Figure 2.1: Block structure in a 2D domain.
(Reproduced from the FLASH User Guide, version 2.5 [1].)

regions of the domain. The block parameters NXB, NYB, NZB and NGUARD are passed to the PARAMESH package at runtime.

Only certain blocks in the computational domain are able to refine or derefine. These are known as leaf blocks, and are the blocks that represent a particular region of the domain at the highest refinement level. All calculations are performed using the leaf blocks during the simulation. Control over the maximum and minimum refinement levels is possible by setting the parameters lrefine_min and lrefine_max in the input file flash.par. These set the minimum and maximum levels of refinement possible in the computational domain. Note, that setting lrefine_min and lrefine_max generates a uniform grid. Whenever a leaf block is marked for refinement, it will spawn 2^{NDIM} child blocks covering the same region, where NDIM is the number of dimensions. This means each child block is half the size as its parent in each dimension. Figure 2.2 shows a leaf block spawning 8 child blocks in a 3D domain.

Despite spawning child blocks, the parent block is retained in the simulation, but is no longer marked as a leaf block. Whenever a derefinement occurs, the child blocks are destroyed and its parent becomes the new leaf block. In a FLASH simulation each particle is assigned to a single leaf block in the computational domain.

The block distribution specified by the PARAMESH package is generated using a Morton-space filling curve [1]. Here, a Morton-space filling curve provides a way to decompose parallel computing problems between processes in a way that maximises

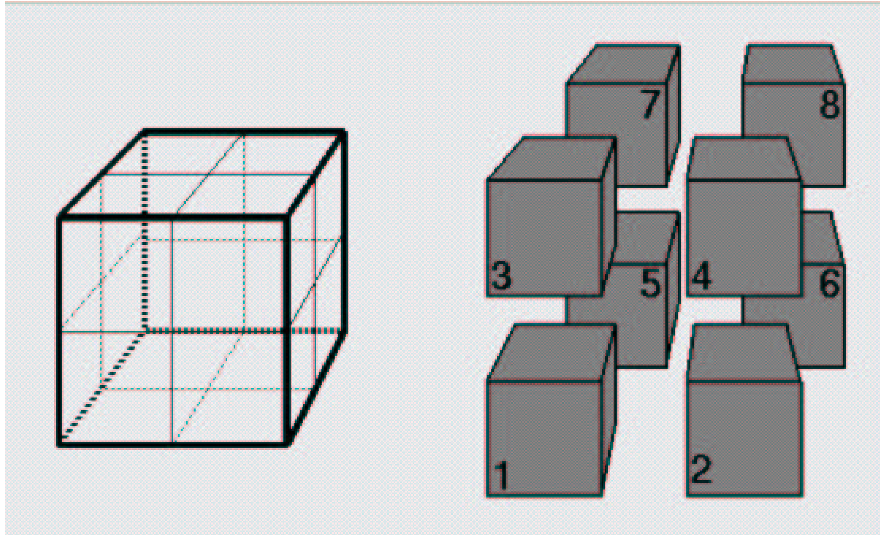

Figure 2.2: Child blocks created during a refinement in a 3D domain.
(Taken from PARAMESH Users Guide [3].)

both load balance and data locality. The criteria used is:

1. The distribution of blocks should maximise the load balance across all processes.
2. Where possible, blocks contained in the same region of the computational domain should be assigned to the same process.
3. Where possible, nearest-neighbour blocks in different processes should exist in nearest-neighbour processes.

Thus, if a particle contributes mass to blocks in different processes, it is likely that the blocks exist in nearby processes. Other space filling curves exist, such as Hilbert and Peano curves [14], and are also well suited to this type of problem.

2.2 Assigning Particle Mass to the Mesh

A dark matter particle in a FLASH simulation exists within a single internal grid cell of only one leaf block. A simple technique for assigning the particle mass to the mesh, is to assign the mass to the single grid point containing the particle. However, such a simple interpolation scheme gives a crude mass distribution, and thus a poor force approximation. Therefore, higher level interpolation schemes exist which "smear" the particle mass across many more grid points. As previously stated, the default interpolation scheme in FLASH is the Cloud-In-Cell (CIC) scheme, which involves 2^{NDIM} grid points.

The higher level interpolation schemes are desirable for increased accuracy, however, they present difficulties for developing a particle mass assignment procedure. This is

because a nearest neighbour grid point may exist in the internal section of a different block. Here, an internal section corresponds to grid points between NGUARD+1 and NGUARD+N[XYZ]B+1 in Figure 2.1, where N[XYZ]B is used to indicate NXB, NYB and NZB. Therefore, the mass of a particle may need to be assigned to grid points in more than one block. To further complicate matters, the block could exist in a different process.

The basic problem is, therefore, assigning particles' mass to blocks in a mesh distributed across multiple processes. Here, processes must coordinate so that a particle's mass can be assigned to any grid point in the computational domain. There are two different approaches, which involve communicating different types of information. The approach adopted in IVS is explained in Section 2.2.1, and the intended approach for FLASH 3.0 beta is explained in Section 2.2.2.

2.2.1 FLASH 2.5 Algorithm

The IVS technique involves utilising guard cells. When particle mass assignment is performed, the guard cell grid points are used to accumulate mass assignment intended for non-internal grid points. After assigning mass from all particles, the potentially updated guard cell grid points are communicated to the neighbouring block.

The communication consists of a message exchange between blocks whose block faces touch. Here, the "halo" message consists of the guard cell grid points associated with the touching block face. If the blocks are local to the current process, then the halo swap is achieved by a direct memory copy. Otherwise, the halo swap involves an MPI send / receive pair between processes. The mass stored in the halo, or guard cells, is used to update the receiving block's internal grid points.

This approach is implemented in the procedure named MapParticlesToMesh. It is possible to analytically calculate the number of message exchanges. This is because the halo swaps occur irrespective of whether any mass is accumulated in the guard cell grid points. In a uniform grid implementation, each block sends $(3^{\text{NDIM}} - 1)$ halo messages. That is, a block sends a message to each block that is a face or corner neighbour. In addition, the same number of messages are received by the block. This equates to a very large number of messages when each block in the computational domain is considered. Most will be direct memory copies however, and not MPI messages, as typically there are many blocks assigned to each processor.

2.2.2 FLASH 3.0 Algorithm

This algorithm, developed by [11], involves accumulating particles' mass in a block's internal grid points only. Mass is never accumulated in guard cell grid points, as FLASH 3.0 will ultimately give the option of using no guard cell.

An extremely desirable feature of the algorithm is that it does not require guard cell grid points. Guard cells take a significant portion of the total memory allocated to blocks. For example, a 3D simulation with NXB=NYB=NZB=50, and NGUARD=4 will use approximately 35% less memory by eliminating guard cells. This is calculated by considering the ratio of internal block data to total block data [3], i.e.

$$\frac{\text{NXB}\times\text{NYB}\times\text{NZB}}{(\text{NXB}+(2\times\text{NGUARD}))\times(\text{NYB}+(2\times\text{NGUARD}))\times(\text{NZB}+(2\times\text{NGUARD}))}$$

Using a reduced memory operation mode is necessary for simulations involving a large number of dark matter particles. For example, IVS simulations using only 256k (256^3) particles were too memory demanding and crashed on HPCx [10].

The first step of the algorithm involves accumulating the mass of a particle in the grid points of the host block. Here, the term host block is used to denote the block which contains the current particle. At first, the position of the grid point containing the particle and its nearest neighbour grid points are calculated. If all the nearest neighbour grid points are found to exist in the internal section of the host block, then all the particle mass is accumulated over grid points within the host block. Alternatively, only a portion of the particle's total mass can be accumulated in the hosts internal grid points. This means that a further portion of particle mass should be assigned to grid point within other blocks, which, in turn, may exist in other processes.

A particle attribute named `currentmass` is used to keep track of the portion of mass not yet assigned to grid points. A search is performed for the nearest neighbour grid points in local blocks. Each time a portion of a particle's mass is accumulated in a nearest neighbour grid point, the value of `currentmass` is reduced accordingly. When the `currentmass` attribute is numerically zero, mass accumulation is complete. If the `currentmass` is non-zero, the particle is communicated to the next process.

Communication involves all processes exchanging buffers containing all partially accumulated particles. If a process has no partially accumulated particles, then it still participates in the communication using an empty buffer. A process attempts to accumulate mass in its internal grid points from each received particle. As before, the value of `currentmass` is reduced accordingly. The algorithm terminates when the `currentmass` attribute of every single particle in all processes is numerically zero.

The communication between processes is a particularly desirable feature of the so-called Sieve algorithm. This is because communication between processes is restricted to a relatively small number of large messages. In contrast, the algorithm in Section 2.2.1 relies on a very large number of small messages. Therefore, the implemented Sieve algorithm should be able to hide the latency of the parallel machine's communication network better.

Another extremely desirable feature is the algorithm's ability to exploit the block distribution generated by the PARAMESH package [3]. Here, blocks are distributed to processes in the computational domain according to several criteria (see Section 2.1). This means it is possible to design a communication pattern based on the PARAMESH

block ordering. In other words, increase the probability of assigning particles' mass in as few moves as possible.

The algorithm, named the `Sieve` algorithm [11] is shown in Figure 2.3.

```
Put_particles_in_sieve
Repeat until (currentmass(all particles)==0)
For each particle i
  For each block j on myPE
    if(i maps on j)
        do mapping
        currentmass(i) = currentmass(i)-mass_mapped_on(j)
    end if
    do global sum of currentmass
    if(currentmass_global_sum /= 0)
       send mySieve to neighbour
       receive sieve from neighbour
    end if
  end For
end For
```

Figure 2.3: Sieve algorithm pseudo-code

The neighbour is determined by:

Receive from process: myPE $+ i * (-1)^{i}$
Send to process: myPE $+ i * (-1)^{i+1}$

Here, `myPE` is the process number, and `i` is the iteration number. `i` is local to each process, and runs from 0 to $P-1$, where `P` is the number of processes. This generates a back-and-forth message exchange motion, which gives rise to the Sieve algorithm name.

Specifying communication between the above processes is designed to assign particles' mass in a minimal number of particle exchanges. In addition, the size of the communication message will drop rapidly with each process visited. A more detailed description of the algorithm can be found in [18].

Chapter 3

Implementation

The implementation of the Sieve algorithm (Figure 2.3) is attempted in two parts. Initially, Grid_mapParticlesToMesh is implemented for the simpler problem of a uniform grid. The procedure is tested for correctness in several sample problems. The second stage of implementation involves extending Grid_mapParticlesToMesh to perform mass assignment when the underlying grid is adaptive. In an adaptive grid, the refinement levels of different blocks must be taken into account to ensure that appropriate grid points receive mass assignment. Fortunately, development is simplified as many code units can be reused for the second stage of implementation. The adaptive mesh work is shown in Chapter 6. This is because FLASH 3.0 beta encapsulates the uniform and adaptive grid implementations behind a common grid interface.

3.1 Accumulating Mass in a Single Block

The first situation to consider is when the grid point containing the particle, and its nearest neighbour cells exist in the internal section of a block. Figure 3.1 illustrates this type of mass accumulation in a 2D domain. Here, the grid point marked with a "P" represents the grid point containing the particle. Next to this grid point are the nearest neighbour grid points, which are coloured black. It should be noted that nearest neighbour cells do not always receive a portion of the mass. Whether they do depends on the accuracy of the mass assignment scheme (e.g. the CIC scheme accumulates mass in two grid points for each dimension).

Some important FLASH parameters are also shown in Figure 3.1. These are the indices ip, jp of the grid point containing the particle. Also shown are fixed parameters, gr_ilo, gr_ihi in the x-direction and gr_jlo, gr_jhi in the y-direction, which represent the edge internal grid points in each dimension. The edge internal grid points are calculated from parameters specified in the flash.par file. For the x-direction, gr_ilo=NGUARD+1 and gr_ihi=NXB+NGUARD+1. The block in Figure 3.1 assumes NXB=8 and NGUARD=4. Note that only a single halo of guard cell grid points

are shown. Also, the implementation functions exactly the same if NGUARD=0.

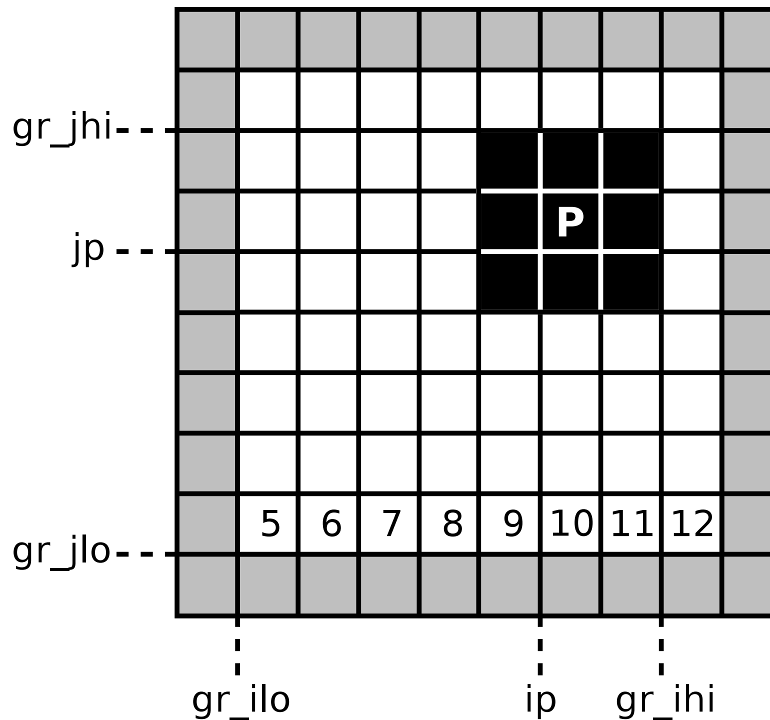

Figure 3.1: Assigning mass to nearest neighbour grid points in a 2D domain

If all nearest neighbour grid points exist in the internal section of a block, the mass assignment can be performed very simply. This is because there is no need to constrain which nearest neighbour grid points receive mass assignment. When this occurs the currentmass attribute of the particle is set immediately to 0.0.

3.2 Accumulating Mass in Multiple Blocks

Mass assignment is more complicated when a particle exists in an edge internal grid point. This is because particle mass may need to be accumulated in grid points in multiple blocks. Figure 3.2 shows how this could occur for a particle P in a 2 dimensional simulation. Here, abutting blocks at position x1 are shown. Note, that they are depicted separately for clarity. The grid cells that receive mass assignment are coloured black.

A simple function is written to ensure the guard cells grid points are never used. It is designed to determine which particles accumulate mass in only the internal section of one block. The purpose is to simplify the mass accumulation procedure for these particles.

For a particle which accumulates mass over multiple blocks, the currentmass attribute is needed to know when mass accumulation is complete. In mathematical terms

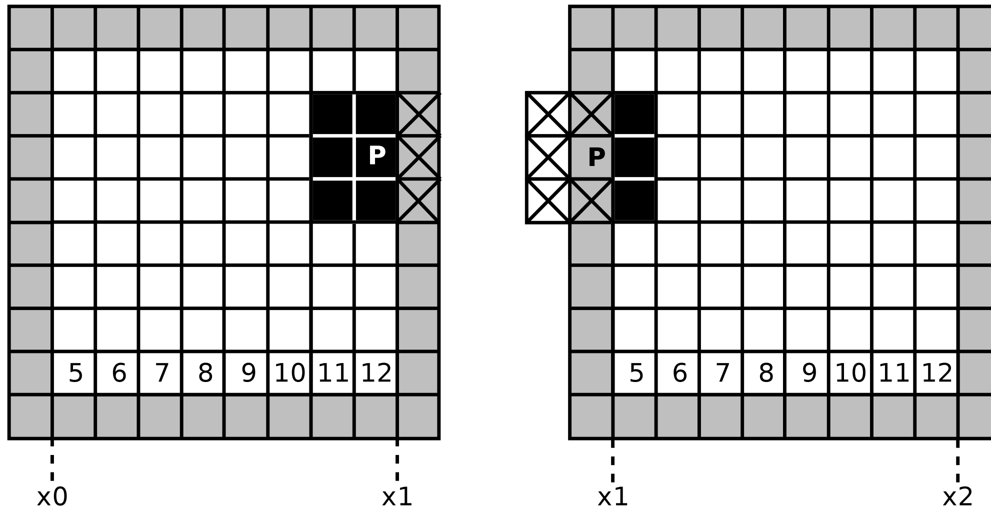

Figure 3.2: Accumulating mass across an internal boundary in a 2D domain

it is appropriate to state that a particle is completely assigned when `currentmass` is numerically zero. However, when using a numbering system with finite precision this may not be the case. Therefore, the particle is considered totally assigned when the relative property remaining is less a tolerance value of `MAPPING_TOLERANCE`. The `MAPPING_TOLERANCE` is currently set to 1.0×10^{-10}. This seems appropriate because the default behaviour in FLASH is to promote all real variables to double precision [1]. It is possible to increase the accuracy of mass accumulation by using a smaller tolerance. However, it is found that using a tolerance of 1.0×10^{-14} sometimes leads to lack of convergence.

It is possible that a particle exists in an edge internal cell that is next to a computational domain boundary. When this happens external boundary conditions must be applied. Several types of boundary conditions are accounted for: reflective, outflow and periodic. When using reflective boundary conditions, the mass assignment intended for a guard cell is reflected back into the internal section of the block. In outflow boundary conditions, the mass assignment is lost if the nearest neighbour cell exists beyond an external boundary. Finally, in periodic boundary conditions, the mass assignment is required in a grid point on the far side of the computational domain.

Care must be taken to ensure that periodic boundary conditions are upheld. Therefore, so-called apparent particle coordinates are employed, which are formed in terms of the system's periodicity. Figure 3.3 illustrates a 2D computational domain with periodic boundary conditions. Here, `P` represents the particle position, and `P'` represents the apparent particle position for each neighbouring block. Notice that apparent particle positions are required to potentially accumulate mass in all nearest neighbour grid points of `P`.

The code fragment required to generate apparent particle positions, is:

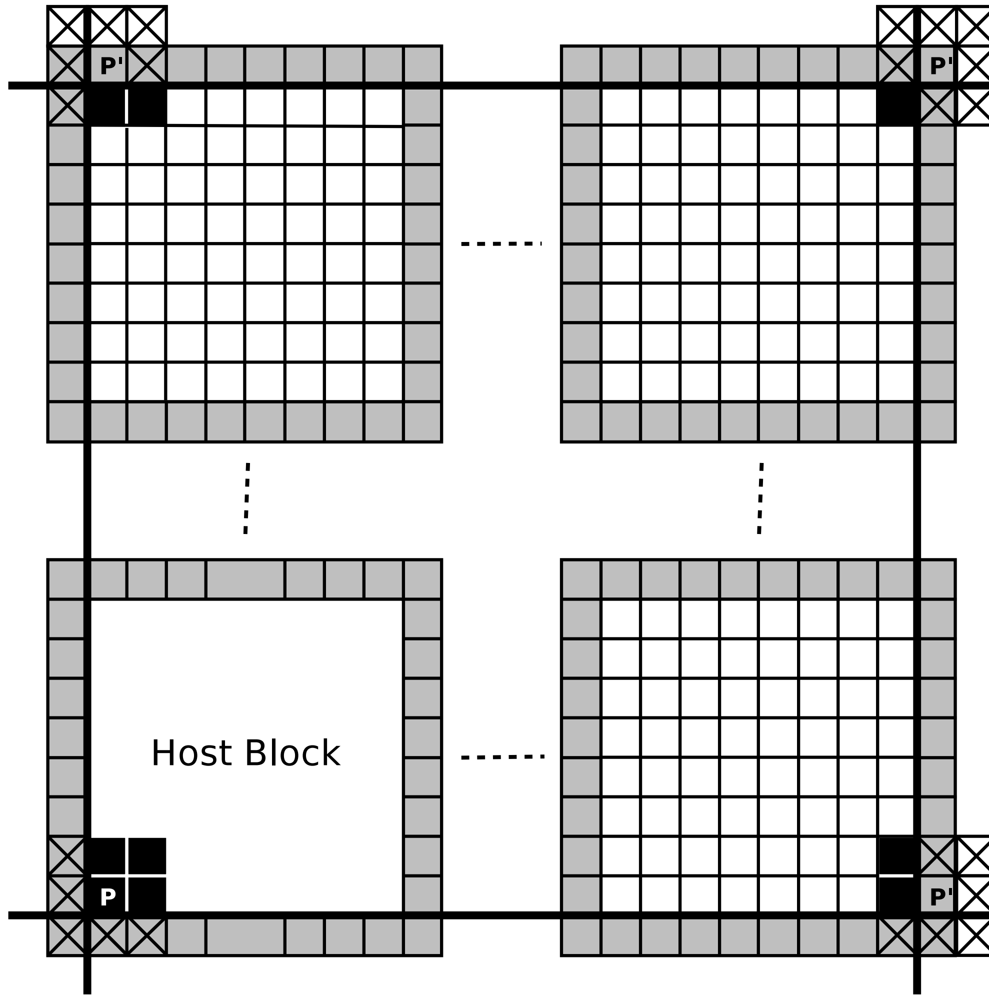

Figure 3.3: Accumulating mass across an external boundary in a 2D domain, with periodic boundary conditions in both directions.

```
ax(1) = particle_x
if(faces(LOW, IAXIS) == PERIODIC) then
  iUpper = 2
  ax(2) = particle_x - (gr_imax - gr_imin)
else if(faces(HIGH, IAXIS) == PERIODIC) then
  iUpper = 2
  ax(2) = particle_x + (gr_imax - gr_imin)
end if
```

Here, the actual particle position in the x-direction is specified as `particle_x`, and the computational domain limits are specified as `gr_imin` and `gr_imax`. The array `ax` is used to store the actual x-coordinate and the apparent x-coordinate. Similar arrays named `ay` and `az` store the same information, but for the y and z dimension respectively. Use of the arrays `ax`, `ay`, `az` enable Fortran90 `do` loops to iterate over each

position. This leads to a maximum of 2^{NDIM} stored particle locations. Each location represents an opportunity for the block to receive mass assignment across a periodic boundary. A potential future optimisation involves reducing the number of particle locations which are checked (see Section 7.1).

3.3 Communicating the Particle Datatype

The situation may arise that a particle only partially contributes mass to grid points owned by the current process. If this happens the mass from the particle needs to be accumulated in grid points that exist in blocks owned by other processes.

The particles are stored in an array of the datatype `particles`. However, the particles that are exchanged between processes have the new datatype `partialParticles`. This is because the `partialParticles` datatype contains the further attribute, namely `currentmass`. It should be noted that the `partialParticles` datatype contains far fewer attributes than the `particles` datatype. This is because it is not used for any physical calculations. It is only used for accumulating mass in grid points.

However, a large array of `partialParticles` may still consume a significant amount of memory. For example, an array of 256^3 `partialParticles` consumes approximately 40MB per processor, when distributed across 32 processors. This assumes the maximum particles that can exist on a processor is $\frac{1.4 \times N}{P}$, where `N` is the number of particles, `P` is the number of processors, and 1.4 is a value indicating the amount of clustering in the particle distribution. The number of `partialParticles` attributes is taken to be 7, and the real datatype is assumed to be 8 bytes.

Allocating extra memory is avoided by storing the `partialParticle` data in a pre-existing array of `particles` named `gr_ptSendBuf`. This is an option because the `partialParticles` datatype contains fewer fields of the same primitive datatype. There is also another array used for receiving particles, named `gr_ptDestBuf`. The partially mapped particles are exchanged between processes using MPI point-to-point `MPI_Sendrecv` communication. Use of `MPI_Sendrecv` is safe during development because it is deadlock free as the receive buffer is provided at the same time as the send.

If all particles currently on one processor are assigned to grid points, then this processor has no particles to communicate to the next processor. When this happens a single element of the data structure is sent to participate in the necessary point-to-point communication. This is because all processors must participate.

Chapter 4

Installation

To verify the FLASH applications perform as expected, several problems with known solutions are installed on HPCx. These are pre-written problems that are provided with the FLASH application. Two sample problems are installed for IVS and FLASH 3.0 beta. These are the Santa-Barbara and Pancake problem for IVS, and the Sedov and Pancake problem for FLASH 3.0 beta. The sample problems are used to check the correctness of `Grid_mapParticlesToMesh`, and also compare the performance of `Grid_mapParticlesToMesh` against `MapParticlesToMesh`. In addition to the sample problems, two extra use-cases are devised for FLASH 3.0 beta, which further test `Grid_mapParticlesToMesh`.

Section 4.1 describes the test problems installed for IVS. Initially, it describes the installation of the Santa-Barbara problem for IVS. This is a dark matter only simulation, which is an ideal problem to use as a test case because of the demand it places on the `MapParticlesToMesh` procedure. Unfortunately, only IVS supports the Hierachical Data Format (HDF) [22] dark matter datasets (HDF5 is a file format for storing scientific data). As such, the test can only be used to verify a correct install of IVS.

A second dark matter simulation, called the Pancake problem is installed. This is because it is a test problem available to both IVS and FLASH 3.0 beta. The Pancake simulation is a common test-case, which is used to verify many astrophysical computer codes. It is a popular test-case because it generates conditions that appear in many astrophysical problems. [15] states that it "provides a good simultaneous test of particle dynamics, Poisson solver, and cosmological expansion.". Another reason for its widespread use is that it has a known analytical solution. This adds weight to the validity of the test, and also enables code accuracy to be evaluated.

Section 4.2 describes the test problems installed for FLASH 3.0 beta. It begins with the installation of two use-cases, which are designed to test `Grid_mapParticlesToMesh` in isolation. It then describes the Sedov problem, which is used to verify the FLASH 3.0 beta installation. The Sedov problem, as a hydrodynamics only simulation, does not use the `Grid_mapParticlesToMesh` procedure. Therefore, the Pancake problem, previously discussed, which uses the `Grid_mapParticlesToMesh` pro-

cedure is installed.

It is found that the Pancake simulation terminates early in FLASH 3.0 beta. The reason for premature termination is explained in Section 4.2.3. This means that results cannot be compared against the analytical solution in FLASH 3.0 beta. Instead, validation is achieved by comparing results from an identical simulations run using IVS. It is assumed that a valid installation of the Santa-Barbara problem for IVS, implies the Pancake problem performs correctly on IVS. As such, comparing results between IVS and FLASH 3.0 beta during the Pancake simulations, verifies that `Grid_mapParticlesToMesh` performs as expected.

In IVS and FLASH 3.0 beta, a Python setup script is used to configure FLASH to solve different problems. It is executed as follow:

```
./setup problem [-arg1 -arg2]
```

Here, `problem` is the simulation to run, and `-arg1` is an example of an optional argument, such the dimensionality of the problem. Running the script copies the appropriate source files into an object directory, and creates a problem specific `Makefile.h` and `Makefile`.

4.1 The IVS on HPCx

4.1.1 Santa-Barbara

The Santa-Barabara test problem is configured as follows:

```
./setup epcc_static -auto -3d -maxblocks=450
```

The resultant Makefile produces an executable, however, the simulation is found to core-dump. This is unexpected, because the installation procedure for HPCx is the same as described in [10]. However, it has since been discovered that several XLF compiler upgrades happened on HPCx during the interim [20].

In order to determine why the program crashed, the core-dump is analysed using the `coretrace` wrapper script to the dbx debugger [21]. This makes it possible to identify the procedure which caused the program to crash. The problem is traced to the file `InitParticlePositions.F90`. When the file is compiled using the `-C` flag, the `coretrace` output displays a `Trace/BPT trap` error. This indicates that the core-dump occurs because an array is accessed out of bounds.

The report [10] describes similar core-dumps occurring on the IBM Regatta+ cluster at RZG. Here, the solution is to compile the problematic Fortran files using the `mpxlf_r` compiler script. As such, the same strategy is adopted to resolve the IVS core-dumps

on HPCx. It is found that the following three Fortran files require special compilation: InitParticlePositions.F90, multigrid.F90 and checkpoint-_wr.F90. This enables the program to run to completion.

The compiler script mpxlf_r prevented the core-dumps because the script contains the flag, qsave. This instructs the compiler to place unsaved local variables in static storage (qsave), and not on the runtime stack (unlike qnosave in mpxlf90_r). In the end it is thought best to compile the three files using the mpxlf90_r compiler script with the qsave flag.

For consistency with [10], results are obtained using a serial installation of HDF5 version 1.4.4. It is required to install this version on HPCx because only HDF5 version 1.6.4 exists on HPCx at the time of investigation. The configuration, make, install lines used are:

```
export OBJECT_MODE=64
export F90="xlf90_r -q64"
export CC="cc_r -q64"
export CFLAGS="-D_LARGE_FILES"
export FFLAGS="-qsuffix=f=f90"
./configure --enable-fortran --disable-shared
--prefix=/hpcx/home/z004/z004/cdaley/HDF5-1.4.4/hdf5-1.4.4
make
make check
make install
```

A simulation using the 128k StB dataset is run on 32 processors, and the HDF5 checkpoint files are evaluated using h5diff, sfocu and IDL. The techniques and results obtained are shown below.

h5diff

Two HDF5 files can be compared using the h5diff [22] utility which is provided with HDF5. This is the easiest way to check that the HDF5 checkpoint files are correct. Here, the h5diff utility is used to compare the checkpoint files from the current installation against the template checkpoint files.

The path to the h5diff utility is
/usr/local/packages/hdf5/hdf5_serial/bin/h5diff.
This installation of HDF5 is used because the command line tool was only introduced in HDF5 version 1.6.0.

The HDF5 files are found to differ. However, this is not surprising as the HPCx compiler has been upgraded, and some different compiler flags are used. Therefore, the checkpoint files are verified using the sfocu tool (Section 4.1.1) and the IDL script (Section 4.1.1).

`sfocu`

The Serial FLASH Output Comparison Utility (`sfocu`) is used to verify the correctness of the gas component in the checkpoint file. It works by quantifying the similarity of data in checkpoint files. The data in the checkpoint file from the HPCx installation is compared against the data in the template checkpoint file. It is found that the maximum magnitude error is $\texttt{1.315}\times\texttt{10}^{-14}$. This indicates correctness of the gas component, as a value of order 10^{-13} to 10^{-15} is acceptable [23].

IDL script

The dark matter component in the simulation is verified using an IDL script. Results are shown for the 128k StB data-set. The simulation is run using 32 processors, and the parameters: lrefine_min=5, lrefine_max=7, pm_level=7. Results shown in Figure 4.1, closely match the 32 processor IVS run on an NEC-SX8 in [23].

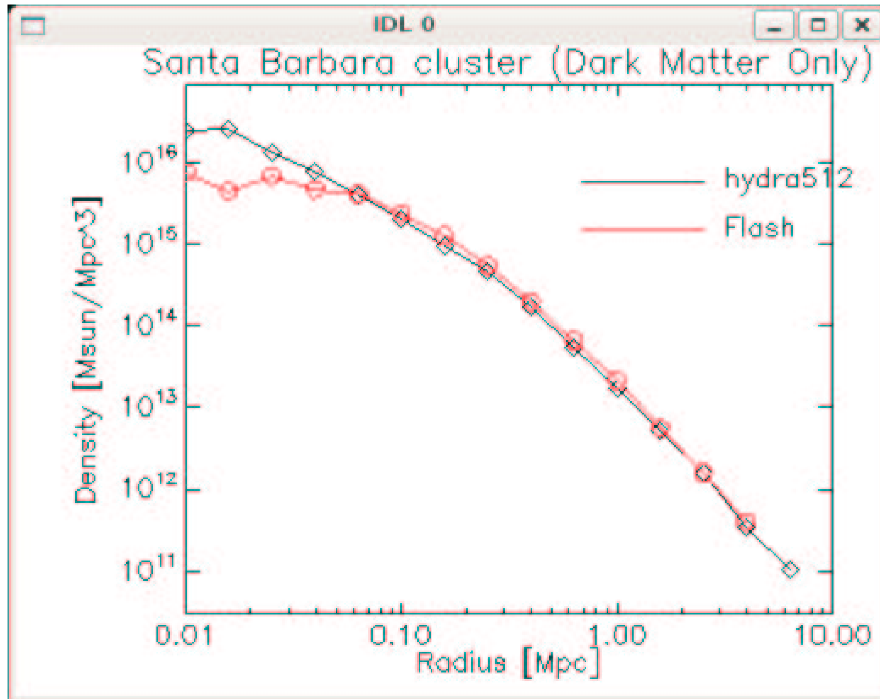

Figure 4.1: IDL script output from an IVS run using 32 processors with 128k StB dataset, against a Hydra_MPI run with 512k StB dataset

This completes the verification of the StB simulation using IVS.

4.1.2 Pancake

The Pancake problem is installed using a so-called MultiPole solver instead of the default MultiGrid solver. This is to be compatible with the FLASH 3.0 beta Pancake installation, which can only use the MultiPole solver. The config file is edited as follows:

```
#REQUIRES gravity/poisson/multigrid
REQUIRES gravity/poisson/multipole
```

The following setup line is used.

```
./setup pancake -auto -3d
```

There is a single error during compilation, which curiously involves a dependency on `multigrid.o` from `MapParticlesToMesh.o`. The entry is simply removed from the Makefile.

Results are compared against the Pancake simulation for FLASH 3.0 beta in Section 4.2.3.

4.2 FLASH 3.0 on HPCx

4.2.1 Use Cases

The quality of the uniform grid implementation of `Grid_mapParticlesToMesh` is ensured by using two different test cases. These test cases were used during development to help locate errors in the implementation. However, the same test cases now form part of a regression test-suite to ensure further development does not introduce new errors.

In FLASH 3.0 beta, new problems can be created by placing all the problem specific source files in an appropriately named directory in `FLASH3/source/Simulation/-SimulationMain`. The Python script, discussed previously, is configured to look in this directory for all FLASH 3.0 beta simulations. By creating an appropriate Makefile, the source files residing in these directories are compiled and linked into the FLASH executable.

For the author's test cases, the problem-specific source files for the `Pancake` simulation are copied into new directories, named `Test1` and `Test2`. This is because the `Pancake` problem in FLASH 3.0 beta is known to use the `Grid_mapParticles-ToMesh` procedure. As such, editing certain files allows control over the data passed to `Grid_mapParticlesToMesh`, which is appropriate for a test case.

Single Particle Mass Assignment

The first test involves placing a single particle at various locations in the computational domain and evaluating the mass assignment to grid points. It is created by editing the procedure `Gravity_potentialListOfBlocks` so that only a single particle is passed to the `Grid_mapParticlesToMesh` procedure. This is a one processor test, so results are checked against a serial implementation of the `Grid_mapParticlesToMesh` procedure. The serial implementation is executed after the actual `Grid_mapParticlesToMesh` procedure, which allows results to be compared. The serial implementation is also written by the author. It is a sensible test because the serial implementation is less complex as all data is known to exist on a single process. This also means there is no need to incorporate parallel communication.

The test is challenging because the particle is placed in grid points next to block boundaries. This is designed to test whether mass assignment occurs as expected across internal block boundaries and external block boundaries. Extremely infrequent particle placements are also investigated, e.g. the particle next to a boundary in each dimension of a 3D domain. In the literature, this type of test is categorised as a white-box test [19]. They are tests that require programmer knowledge of the source code to test different flows of execution though the program. It is found that the mass assignment is the same in the serial and parallel implementation.

Comparison of Serial and Parallel Runs

This test case involves running the application on 1, 2, 3, 4, 5, 6, 7 and 8 processors. Here, the application refers to the test case mentioned above minus the serial `Grid_mapParticlesToMesh` implementation.

Each time the application is run the particle density stored in the mesh is printed to file. The files from parallel runs are compared against the file from the serial run. Each grid point is compared by using a serial Fortran90 program written by the author. Correct parallel performance is assumed when the relative error between the particle density in corresponding grid points is always less than 1×10^{-10}. It is found that the tolerance criterion is satisfied by each test simulation run on different number of processors.

4.2.2 Sedov

To verify installation of FLASH 3.0 beta a test problem named Sedov is used. This is a hydrodynamics problem which has a known analytical solution. It involves tracking the spherical blast waves from a point like explosion in a homogeneous medium. The Sedov problem investigated is a 2D simulation with non-planar symmetry.

The installation involves creating a machine specific Makefile, which contains the path to the `C` and `Fortran90` compilers and the `HDF5` libraries. The following setup script is run to copy the appropriate source files into the object directory.

```
./setup Sedov -auto
```

The `flash.par` input file is provided on the FLASH website. This configures the simulation to run for $t = 0.05$ seconds or 10000 timesteps (whichever occurs first). During the simulation, HDF5 output files are created every $t = 0.01$ seconds. The HDF5 files are analysed using IDL routines provided with the FLASH 3.0 beta source code. The use of the IDL routines is simplified as there is a GUI named `xflash3`. The IDL image is found to match the image on the FLASH website.

For more information, a detailed step-by-step guide can be found in the User Guide on the FLASH website [1].

4.2.3 Pancake

Difficulties are encountered during the installation of the Pancake problem. This is partly because FLASH 3.0 beta has only recently been extended to perform simulations involving dark matter particles. Initially, it is found that the original Pancake problem is limited to one particle in FLASH 3.0 beta. This does not provide a demanding test of `Grid_mapParticlesToMesh`. Upon request, the FLASH team corrected the test to include multiple particles. The updated test required debugging to permit it to function with FLASH 3.0 beta.

The Pancake test problem is configured as follows:

```
./setup -auto -3d -unit=physics/Gravity/GravityMain/Poisson Pancake
```

As mentioned earlier, the Pancake simulation freezes early in FLASH 3.0 beta. This happens irrespective of the values stored in `lrefine_min` and `lrefine_max`. Generally, it is found that for refinement levels of `lrefine_min=lrefine_max`<4, the simulation runs for a few time steps. When the refinement is set to values greater than `lrefine_min=lrefine_max=`3, and *sometimes* when varying the number of processors, the freeze occurs in the first time step.

The bug is investigated by forcing a core dump when the error first occurs. This is achieved by compiling the source files with the `-C` flag. The `-C` option is used to ensure each reference to an array element or array section is within the boundaries of the array. Some FLASH source files cannot be compiled using the `-C` flag. This is because of the presence of zero-sized subscripted arrays, however, this does not necessarily imply an error as zero-sized subscripted arrays are valid in certain circumstances [30]. As such, these files are compiled without the `-C` option.

Analysis of the forced core dump reveals that the application freezes during a PARAMESH routine. It is found that the same error occurs when the source files are compiled with no optimisation flags. The simulation is run inside the Totalview debugger [31]. As the code is compiled with the `-C` option, the debugger runs straight to the problem section, as shown in Figure 4.2.

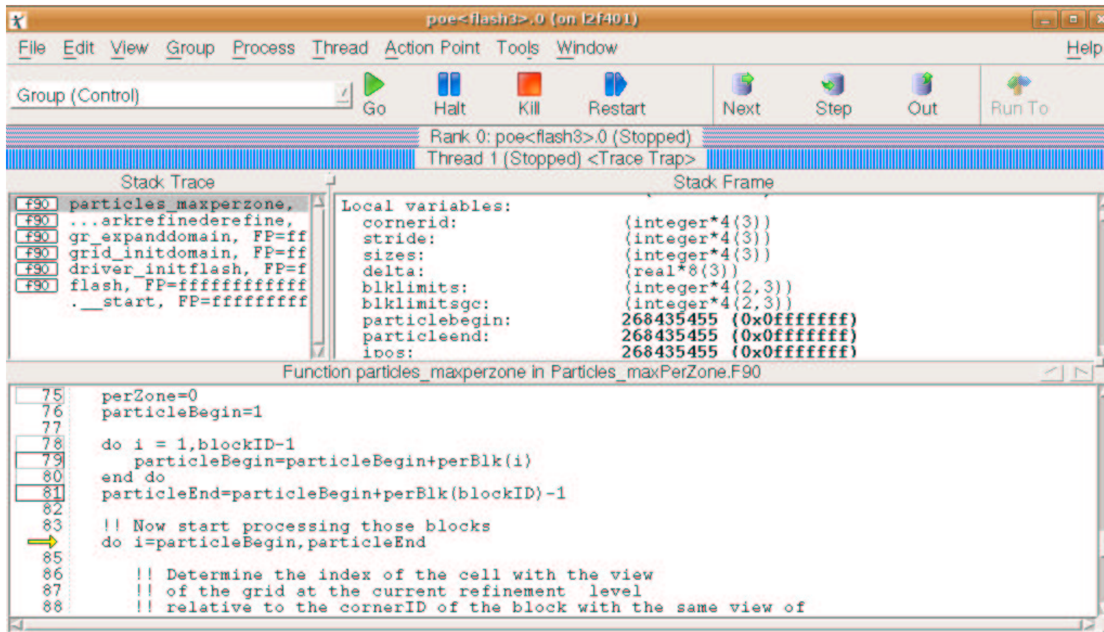

Figure 4.2: Totalview screenshot showing error during an adaptive grid simulation with `lrefine_min=lrefine_max=4`

When the variables are analysed it is discovered that the loop indices `particleBegin` and `particleEnd` are set to unexpected values. This is because the array `perBlk` contains spurious values

The reason for the spurious values is traced back to the `particles` array not being allocated. This is leading to a memory corruption error when values are written to unallocated memory.

The problem arises because the grid is initialised in `Grid_initDomain` before the particles are initialised in `Particles_init`. This order is specified in the program because the initial refinement level of the blocks is determined by the distribution of particles.

An attempt is made to place the particle initialisation routine before the block initialisation routine. This causes a problem because there are no blocks available to associate with each particles. Another attempt is made to move the particle array allocation statement before the block initialisation routine, but this causes other problems. It seems the solution is to split the particle initialisation routine into two parts, as is the case in FLASH 2.5. Unfortunately, there is no further time available to resolve the problem. In summary, the Pancake problem still requires debugging. This is not related to the `Grid_mapParticlesToMesh` implementation.

As mentioned previously, the Pancake simulation generally runs for a few time steps when using refinement levels of `lrefine_min=lrefine_max`< 4. These few iterations allow the correctness of only the `Grid_mapParticlesToMesh` procedure

to be verified against the `MapParticlesToMesh` procedure in IVS.

The same simulation in IVS and FLASH 3.0 beta is run on one processor, and uses a refinement criteria of `lrefine_min=lrefine_max=2`. This enables a comparison between `MapParticlesToMesh` and `Grid_mapParticlesToMesh` over the first 5 timesteps, before the FLASH 3.0 beta termination. Results are obtained by printing the assigned particle density in each grid point of the mesh to file, after executing the `MapParticlesToMesh` and `Grid_mapParticlesToMesh` procedures. Values in corresponding grid points are compared using a serial Fortran90 program written by the author. Results show that the particle density differs on average by a relative error of 1.0×10^{-6}. Here, the particle density relative error for a particular grid point is calculated as:

$$\text{Relative Error} = \frac{|\text{FLASH2} - \text{FLASH3}|}{\text{FLASH2}}$$

where `FLASH2` is the IVS grid point value, and `FLASH3` is the FLASH 3.0 beta grid point value. As calculations are performed in double precision this does not indicate a correct implementation. However, the discrepancy can be explained because the initial particles' position is different in IVS and FLASH 3.0 beta. To make the test fair, the position and block identifier for each particle in IVS is printed to file. This is performed before the call to `MapParticlesToMesh`. These particle positions are then read into the FLASH 3.0 beta simulation before the call to `Grid_mapParticlesToMesh`. The particle density is compared in the same way once `MapParticlesToMesh` and `Grid_mapParticlesToMesh` are executed. Using particle positions from IVS is only valid because the computational domain is decomposed the same in each simulation. It is found that the maximum relative error between mapping in a cell is 1.0×10^{-14}. This is within the tolerance of the rounding error, so the implemented algorithm is performing as expected for this test case.

Chapter 5

Results and Analysis

5.1 Runtime Performance

The runtime of the MapParticlesToMesh procedure is measured using the standard MPI timer MPI_Wtime. This involves inserting a call to MPI_Wtime before and after the MapParticlesToMesh procedure. No clock-tick conversion is necessary because MPI_Wtime returns a double precision value containing elapsed time in seconds. Figure 5.1 shows how the calling procedure Gravity_potentialList-OfBlocks is instrumented to perform the timing.

```
DOUBLE PRECISION :: start_time, end_time, total_time
integer, parameter :: iREPEAT=100
integer :: iRep, ierr

call MPI_Barrier(ierr)
start_time = MPI_Wtime()
do iRep = 1, iREPEAT, 1
  call Grid_MapParticlesToMesh(..)
end do
end_time = MPI_Wtime()
call MPI_Barrier(ierr)
total_time = end_time - start_time
```

Figure 5.1: Technique used to time mass assignment procedure

The Grid_mapParticlesToMesh procedure is executed iREPEAT times to improve timing accuracy. All input/output (I/O) and any calls to the FLASH internal timing system are removed from Grid_mapParticlesToMesh. The FLASH internal timing system is not used for the benchmark because it was found to be unreliable in IVS [10]. Unless otherwise stated, all results are obtained using a 64-bit FLASH 3.0 beta installation, and source files are compiled to -O3 optimisation. The same instrumentation is performed in IVS to obtain timings for MapParticlesToMesh.

The procedure is timed during a 128^3, 3-dimensional simulation on various number of processors. All simulations in this chapter are 3-dimensional. Refinement is fixed during the adaptive grid simulation by setting `lrefine_min=lrefine_max=3`. This divides the computational domain into 64 blocks, which are then distributed amongst processors. As the number of blocks are fixed, the problem size is independent of the number of processors. Thus, the strong scaling properties of `MapParticlesToMesh` and `Grid_mapParticlesToMesh` are investigated, and results are shown in Table 5.1 and Figure 5.2. It should be noted that this is the largest problem that can be investigated. Larger values of `lrefine_min` cause the program to coredump during initialisation, as values are written to the `particles` array before the array is allocated (see Section 4.2.3). As memory is corrupt, the outcome of initialisation is unpredictable, and sometimes the program crashes. Unfortunately, for the same reason, it is not possible to run the simulation on one processor for `lrefine_min=3`.

Number processors	FLASH2 time (s)	FLASH3 time (s)
1	63.8	-
2	20.9	54.3
4	10.9	27.4
8	4.0	14.0
16	2.4	10.0
32	1.5	8.5
64	1.6	8.1

Table 5.1: Time for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles on various numbers of processors for both IVS and FLASH 3.0 beta

For the simulation investigated, results show that the `Grid_mapParticlesToMesh` is consistently slower than the `MapParticlesToMesh`. In summary, `Grid_mapParticlesToMesh` is not as fast as hoped. Therefore, its performance is investigated using the mpiprof, xprofiler and Paraver profilers. This permits a detailed analysis of any bottlenecks that may exist in the code.

5.2 mpiprof

mpiprof is a wrapper library created for IBM platforms that is used to obtain detailed information about MPI communication [24]. It provides elapsed time measurements of each MPI operation, and a call graph giving a procedure level breakdown of the caller of each MPI operation. Ideally, the low overhead mpitrace wrappers would be used to measure MPI communication time. However, results are less useful because no record is made of the procedure executing the MPI operation. Therefore, mpiprof is used during the investigation. Source files must be compiled with the `-g` flag.

It is found that the mpiprof library cannot be used to profile an adaptive grid build of the FLASH simulation. When linked, the simulation terminates during initialisation with

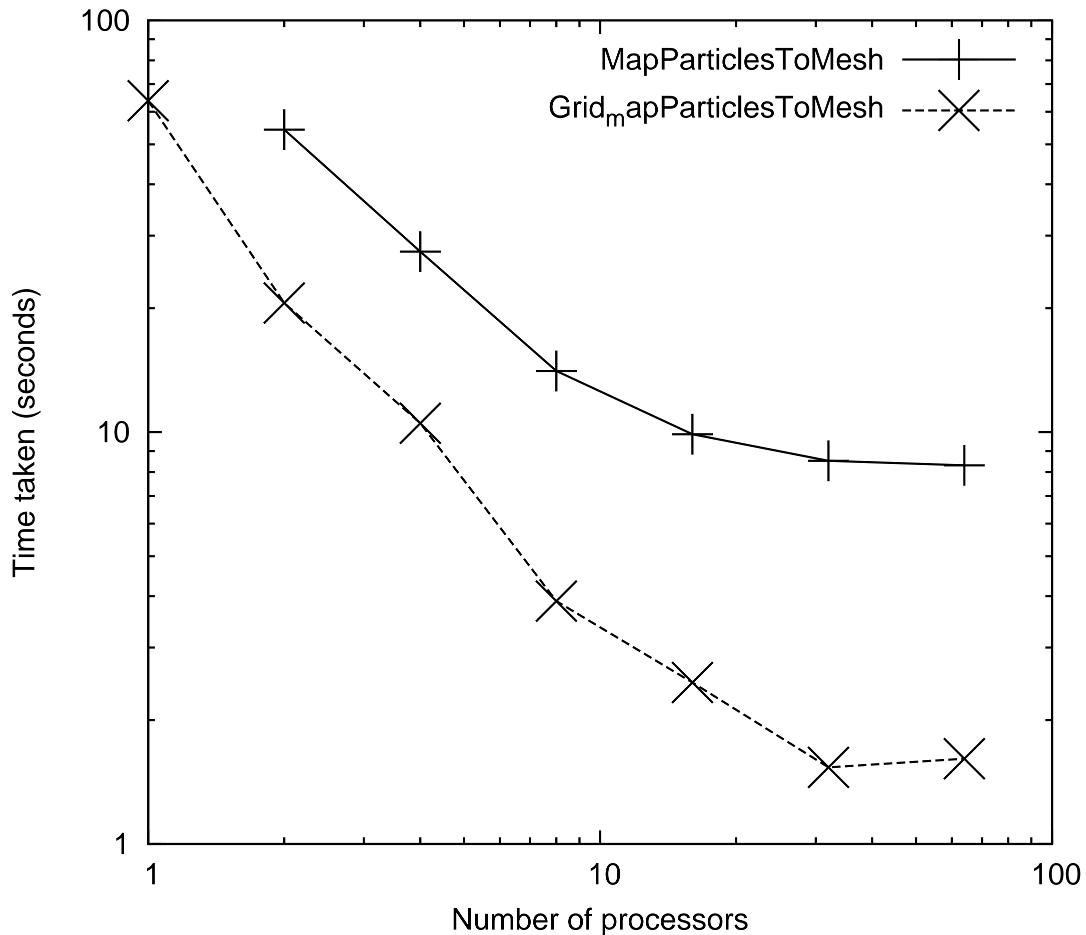

Figure 5.2: Time for 10 iterations of `Grid_mapParticlesToMesh` , when using 128^3 particles on various numbers of processors for both IVS and FLASH 3.0 beta

the following error from each process:

```
ERROR: 0032-121 Invalid rank  (-1) in MPI_Group_translate_ranks, ↩
    task 1
```

The error is raised during execution of the procedure `Fill_Old_Loc`, which is part of the PARAMESH package. It occurs because invalid data is passed to the MPI profiling interface function named `.PMPI_Group_Translate_Ranks`. Figure 5.3 shows the location of the error. Here, the simulation is run inside the Totalview debugger.

The same error occurs when the FLASH application is linked to the mpitrace library. It also occurs in the same procedure during an IVS simulation. In IVS, the procedure is found in the source file `amr_morton.F90`. It is in a different source file because the IVS application uses an older version of the PARAMESH package.

Since the adaptive grid version of the code cannot be profiled, the uniform grid version

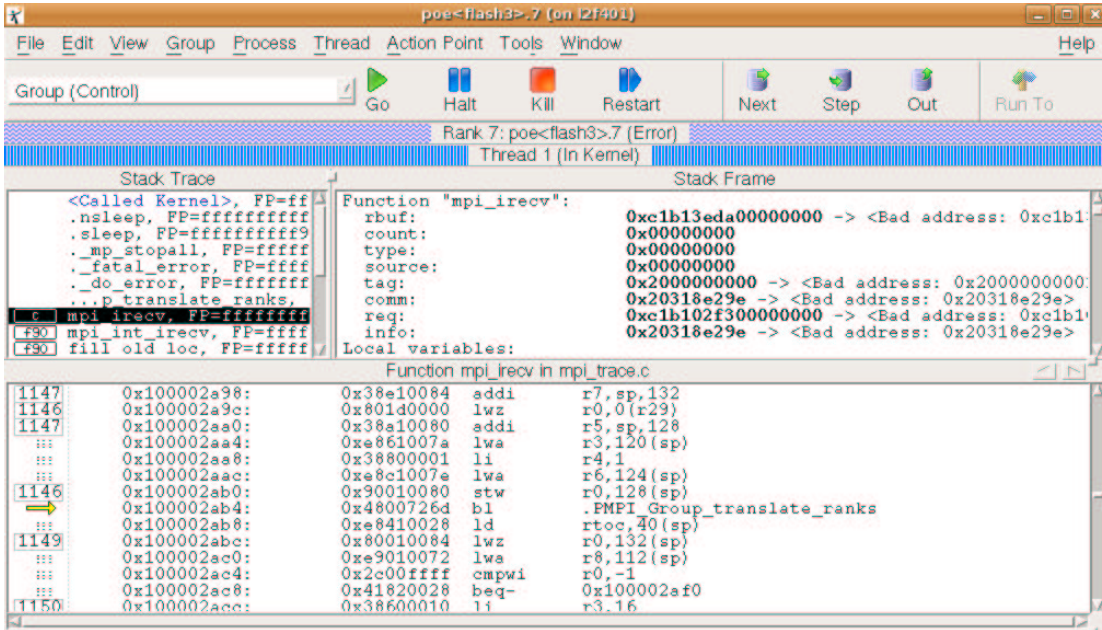

Figure 5.3: Totalview screenshot showing error during profiling with mpiprof

is investigated. A uniform grid installation of FLASH 3.0 beta is obtained using the following setup line.

```
./setup -auto +noio +ug -3d -unit=physics/Gravity/GravityMain/↩
    Poisson Pancake
```

The runtime of `Grid_mapParticlesToMesh` is shown in Figure 5.4 for a simulation using 128^3 particles on 8, 27, 64, 125 processors. By definition, a uniform grid implementation assigns only one block to each process. As such, using these processor counts ensures each block represents a cuboidal section of the computational domain. It should be noted that simulations were compiled using the `-g` flag. Results are not shown for IVS because there is no uniform grid implementation.

In a uniform grid implementation, the resolution of the problem increases linearly with processor count. As such, adding processors to the simulation increases the total number of grid points. Results in Figure 5.4 clearly show that the procedure runs faster for higher processor counts.

Times from an mpiprof profile of the procedure are shown in Table 5.2. Here, the procedure's total MPI communication time, and broken down MPI communication time is shown. The time taken to run the `Grid_mapParticlesToMesh` procedure is displayed in the call graph section of the trace files. It is decided to show the maximum MPI communication time across all trace files in the table. This is because the procedure only executes as fast as the slowest CPU. The first column of the table shows the runtime of the procedure (from Figure 5.4).

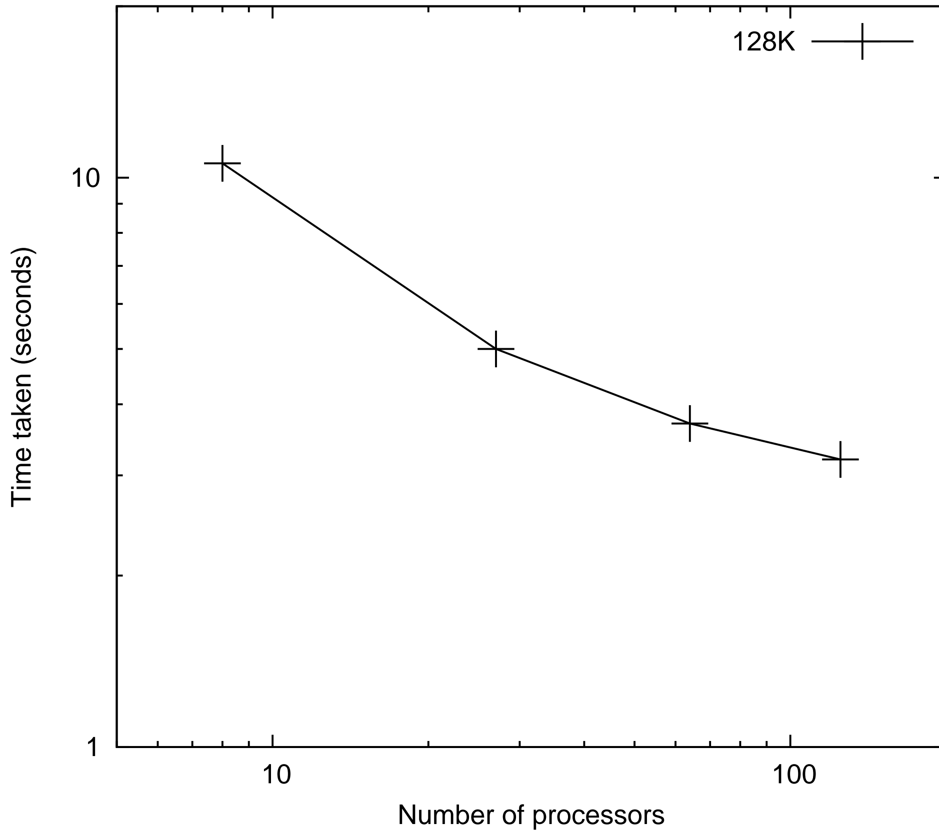

Figure 5.4: Time for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles, and a uniform grid on various numbers of processors for both IVS and FLASH 3.0 beta

Processors	Procedure (s)	Comm (s)	MPI_Sendrecv (s)	MPI_Allreduce (s)
8	10.6	1.5	0.3	1.2
27	5.0	2.0	0.9	1.1
64	3.7	2.2	1.2	1.0
125	3.2	2.2	1.2	1.0

Table 5.2: Time and MPI communication information for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles, and a uniform grid on various numbers of processors for both IVS and FLASH 3.0 beta

The results show that the time spent performing the `MPI_AllReduce` on different number of processors is relatively constant. On the other, the time spent performing the `MPI_Sendrecv` increases with number of processors. It can be seen that the cost of performing the `MPI_Sendrecv` operations relative to the `MPI_AllReduce` operations increases with processor count. This can be understood on the basis, that there are more blocks in the computational domain at higher processor counts. Therefore,

there are fewer particles on each block, so the importance of communication relative to calculation increases. In addition, more blocks in the computational domain mean a particle will visit more blocks on average before being completely assigned to the mesh. This equates to more MPI messages consisting of fewer particles. As such, more messages lead to the impact of `MPI_Sendrecv` increasing because of cumulative message latency.

The time to complete the `MPI_AllReduce` seems to be more problem independent than the `MPI_Sendrecv`. This is because the `MPI_AllReduce` does not involve large quantities of data. It is used simply to determine whether to continue the point-to-point communication. It may be thought that at higher processor counts the time spent waiting for the `MPI_AllReduce` would increase. However, it seems the time spent waiting for more processors to synchronise is offset by the reduced amount of work on each processor. Recall that all results are obtained from simulations using 128^3 particles. This means that as work becomes more distributed, even a lightly loaded processor will remain relatively synchronised with a heavily loaded processor.

Results for eight processors reveal that 10.6 seconds is spent in `Grid_mapParticles-ToMesh`. Of this time, 1.5 seconds is communication, which indicates the `Grid_map-ParticlesToMesh` is dominated by computation at low processor numbers. At higher processor counts, the relative cost of communication increases. An analysis of the computation bottlenecks using eight processors is provided in the following Section.

5.3 xprofiler

xprofiler is an IBM X-Window based profiling tool which is based on the more common gprof tool. These tools are used to determine how long is spent executing each function and subroutine in a program. Profiles are generated by sampling the program counter periodically when the program is run [25]. Source files must be compiled with the `-pg` option to obtain profiling information. xprofiler can also provide a line-by-line profile, provided the source files are compiled with the `-g` flag. Optimisation is left switched on to reduce the overall impact of instrumentation. The profile is recorded over 10 iterations of `Grid_mapParticlesToMesh` procedure.

The xprofiler tool is invoked with the following command line:

```
xprofiler flash3 gmon* -a /source/files/path
```

This loads the FLASH 3.0 beta parallel program executable with the merged statistic file from each process. A profile is exported from the xprofiler tool, and is shown in Figure 5.5. It should be noted that the Figure is edited to show just the key aspects of the profile.

All procedures are represented by their symbolic object name in Figure 5.5. The most expensive procedure in the profile is attributed to `_mcount`. This is a symbol used by

```
%time seconds calls  name
37.9 65.28           .__mcount
 6.6 11.43           ._lapi_shm_dispatcher
 4.9  8.37  12131360 .__mappingmodule_NMOD_mapcarefullytohostblock
 4.5  7.72 237404160 .__mappingmodule_NMOD_biscellcontainedinblock
 4.4   7.5 629696160 .__mappingmodule_NMOD_bmapstointernalcell
 3.8  6.57  11190720 .__mappingmodule_NMOD_maptononhostblock
 3.6  6.16 473123040 .__gr_applybcstohostblock_NMOD_getcellindex
 3.4  5.89           ._lapi_dispatcher
 2.9  4.92  29675520 .__mappingmodule_NMOD_attempttomapblock
 2.7  4.69           ._pxldmod
 2.3     4       640 .__particledecision_NMOD_gr_mapparticlesandfilldestbuf
 1.8  3.16 101294080 .__mappingmodule_NMOD_bpropertycompletelymapped
 1.8  3.12           ._xldmdlo
 1.7  2.94           .__itrunc
 1.5  2.63           ._xldintv
 1.5  2.51           .LAPI__Msgpoll
 1.4  2.37  32162240 .__mappingmodule_NMOD_calculateweightsforinputblock
 1.3  2.16         8 .particles_init
 1.2  2.07           .REG_3stream_store
 1.2  2.02           ._xlidflr
 1.2     2  32162392 .grid_getblkptr
 1.1  1.84           .qincrement
 0.9  1.62  61837800 .grid_getblkboundbox
 0.9  1.48           .__stack_pointer
 0.8  1.38  16984532 ._sin
 0.7  1.21           .qincrement1
 0.6  1.01  36597760 .__particledecision_NMOD_attemptremotemapping@AF5_3
 0.5  0.89  32162240 .pt_assignweights
 0.4  0.72         8 .particles_initpositions
 0.3  0.58   8840160 .__mappingmodule_NMOD_mapnaivelytohostblock
 0.3   0.5           ._is_yield_queue_empty
 0.3  0.45  41806880 .grid_getblkbc
 0.2  0.38  12131360 .__gr_applybcstohostblock_NMOD_applybcs
 0.2  0.28  36597760 .__particledecision_NMOD_attemptremotemapping
 0.2  0.28           ._xldipow
 0.2  0.26         8 .pt_createtag
 0.2  0.26           ._sqrt
 0.1  0.23  20971520 .__mappingmodule_NMOD_bparticlemapsnaivelytoblock
 0.1  0.23  20971520 .__mappingmodule_NMOD_mapparticletohostblock
 0.1  0.22  32162392 .grid_releaseblkptr
 0.1  0.16  50647072 .grid_getdeltas
 0.1  0.15  50647040 .__particledecision_NMOD_lrefine
 0.1  0.12           ._xldmdlo.GL
 0.1   0.1  12131360 .__gr_applybcstohostblock_NMOD_initialisecellindicies
```

Figure 5.5: xprofiler profile for 10 iterations of `Grid_mapParticlesToMesh`, when using 128^3 particles, and a uniform grid on 8 processors

gprof to give an indication of the time associated with instrumentation. Other expensive procedures include `_lapi_dispatcher`, `_lap_shm_dispatcher`, `LAPI-`

__Msgpoll. These cannot not be located in the FLASH object files, so are likely to be MPI communication routines. The IBM MPI implementation is built on the Low-level Application Programming Interface (LAPI) library [5]. There is also a contribution from intrinsic functions and datatype conversions.

A striking thing about the profile is the number of calls to the procedures: biscellcontainedinblock, bmapstointernalcell, getcellindex. These account for approximately 20% of the runtime in the profile. This is a long time to spend in simple functions.

A more in-depth analysis is possible by performing a line-by-line profile. This is investigated for a section of the procedure named maptononhostblock. The resultant profile is shown in Figure 5.6. It is included because it is the focus of possible future optimisations in Section 7. The numbers at the far left of Figure 5.6 show the number of ticks recorded at each source code statement. Generally a tick represents 0.01 seconds [25].

```
1     call Grid_getBlkPtr(mapping_block,solnVec,CENTER)
      EachZ:  do k = -1*K3D, K3D, 1
12      EachY:  do j = -1*K2D, K2D, 1
43        EachX:  do i = -1, 1, 1
309         if(bMapsToInternalCell(ip+i, jp+j, kp+k).eqv..true.) then

291           solnVec(iVar, ip+i, jp+j, kp+k) = &
              solnVec(iVar, ip+i, jp+j, kp+k) + (wt(i, j, k)*rho)

9             particle_currentProperty = particle_currentProperty - &
              (wt(i, j, k)*particle_property)

            end if
5         end do EachX
        end do EachY
28    end do EachZ
17    call Grid_releaseBlkPtr(mapping_block,solnVec,CENTER)
```

Figure 5.6: xprofiler line-by-line profile of a section of maptononhostblock over 10 iterations, when using 128^3 particles, and a uniform grid on 8 processors

Figure 5.6 shows most clock ticks occur at the if statement expression involving a call to bMapsToInternalCell. The update of the array solnVec is also expensive. The solnVec update involves successive writes to non-contiguous memory addresses. Therefore, values are probably being read from main memory for each update.

To obtain further information, the Paraver tool is used to obtain a visual trace of the runtime behaviour of the procedure in Section 5.4. This also enables a much more detailed analysis than mpiprof because individual messages can be analysed. The mpiprof profile is useful for a general *overview*, as analysis is provided at procedure level granularity.

5.4 Paraver

Paraver is a tool created by the European Center for Parallelism of Barcelona, which is used to provide performance visualisation of computer codes [26]. It can be used to analyse multi-threaded / multi-process computer codes. In MPI codes, visual traces are created for each process running the application, and detailed quantitative information is held about the program's performance. It is useful for obtaining insight about the runtime operation of a program, and for locating any performance bottlenecks.

At the time of investigation, the Paraver profiler is not compatible with 64-bit applications on HPCx. As such, FLASH 3.0 beta is compiled in 32-bit mode for the purpose of this study. Paraver does not require instrumentation or special compilation of the source files for analysis.

It is found that Paraver profiles can be obtained during adaptive grid simulations with fixed refinement (i.e. `lrefine_min=lrefine_max`). However, the resultant profiles were unstable, and caused Paraver to crash during analysis. As such, the Paraver profile presented in this report is based on the simulation using a uniform grid.

An 8 processor 128^3 simulation is investigated, and the resultant trace for one iteration of `Grid_mapParticlesToMesh` is shown in Figure 5.7. The trace is obtained by recording information until `Grid_mapParticlesToMesh` completes. Termination of the simulation is forced using an `MPI_Finalize` followed by a Fortran90 `stop`. The `MPI_Finalize` call is required to obtain a `Paraver` profile. The Figure shows a zoomed in section of the FLASH 3.0 beta simulation.

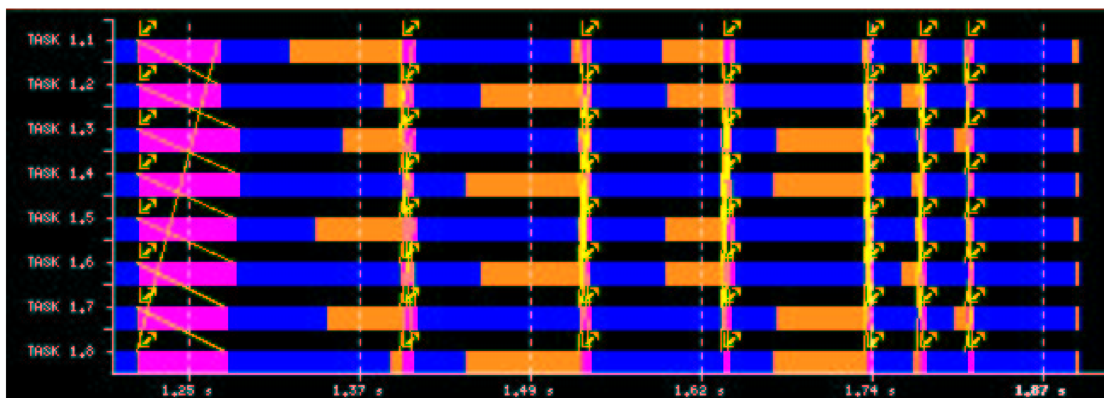

Figure 5.7: Paraver trace for 1 iteration of `Grid_mapParticlesToMesh`, when using a 128^3 particles, and a uniform grid on 8 processors

The trace shows MPI communication occurs in seven rounds. Here, each round of communication is composed of `MPI_Sendrecv` message exchanges between processors, and a single collective `MPI_Allreduce` communication. In the figure, `MPI_Send-recv` messages are marked with a tracer line joining two processors, where the arrow at the start of the tracer line indicates which processor sends the message. The message

before the MPI_Sendrecv is the collective MPI_Allreduce. Finally, the dark regions of the trace (blue if viewed in colour) indicate time spent performing sequential work. As there are seven communication rounds, it means that some particles visit every single process. This is expected because only 8 blocks exist in the computational domain and there are periodic boundary conditions in each dimension.

Interestingly, the time spent in first MPI_Sendrecv is much longer than expected. This is because the MPI communication buffers are allocated during the first iteration. As such, the time spent in the first MPI_Sendrecv operation is slightly misleading because it includes allocation time.

The trace shows a relatively long time is spent performing collective MPI_AllReduce operations. However, no MPI_AllReduce can be seen before the first MPI_Sendrecv operations. In actual fact, the MPI_AllReduce is executed, but it is completed in an extremely short amount of time. This indicates that the processors remain relatively synchronised before the first MPI_AllReduce. This can be understood because the pancake problem has a uniform initial particle distribution. For this reason, each processor, P, is assigned part of the computational domain, which has $\frac{128^3}{P}$ particles.

The time spent executing the large MPI_AllReduce in the second and third round of communication is 0.08 seconds each. This compares to the corresponding MPI-_Sendrecv operations which take approximately 0.01 seconds each. It is clear, form this figure, that the reason for the relatively large time spent in the MPI_AllReduce is load imbalance. This is because the calculation phase takes different lengths of time depending on the quantity of particles that assign of portion of mass to the new block. Also, there will be some loss of synchronisation due to the MPI_Sendrecv operation being exchanged between processors at different rates. Although, not clear from Figure 5.7, the time taken to send point-to-point messages (once synchronised) generally decreases with each particle communication exchange. This is because the number of particles placed in the send buffer generally decreases with each processor visited, and as such there are fewer bytes to communicate.

5.5 HPCx Specific Optimisations

The easiest type of optimisation involves adjusting the compilation and linking flags. By default, every file in the FLASH 3.0 object directory is compiled with
-O3 -qcache=auto -qtune=auto optimisation.

The FLASH User Guide strongly advises against using a higher level of optimisation than -O3 on an IBM platform. However, it is interesting to see how the performance of Grid_mapParticlesToMesh improves at higher optimisation levels. As such, only the files containing Grid_mapParticlesToMesh and its child procedures are compiled at a higher level of optimisation. The flag -O4 is used during compilation, and -qipa is used during compilation and linking.

Results are repeated 3 times and are shown in Figure 5.8. Error bars are displayed, but are very small.

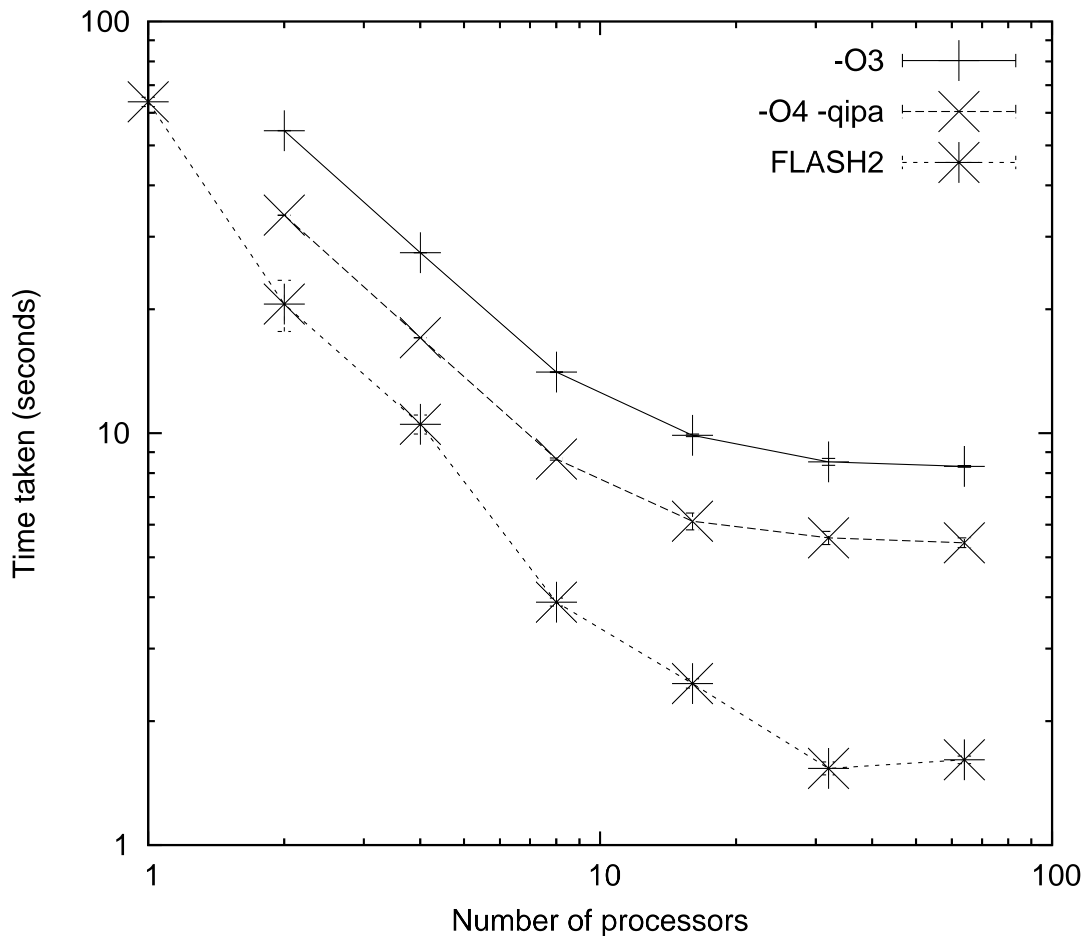

Figure 5.8: Runtime performance using different compilation flags, when using 128^3 particles, and an adaptive grid with `lrefine_min=lrefine_max=3` on various numbers of processors.

It is clear that the overall performance improves when using a higher level of optimisation. Importantly, the correctness of the code is also maintained. The reason for the significant improvement is because of the `-qipa` flag during linking. This causes some small procedures e.g. `bPropertyCompletelyMapped` and `getCellIndex`, which are called by procedures in different source files, to be inlined.

Unfortunately, there is no further time available to perform a more thorough study.

Chapter 6

Adaptive Mesh Implementation

A particularly desirable feature of the FLASH application is the provision of AMR. As such, a version of `Grid_mapParticlesToMesh` is required that is fully compatible with an adaptive mesh. The implementation must be able to perform correct grid point assignment when a portion of a particle's mass is assigned to blocks at different refinement levels.

It is mentioned in Section 2.2 that particle mass may accumulate in each nearest neighbour grid point when using a high level interpolation scheme. This equates to 3^{NDIM} grid points, when the host grid point is also considered.

Figure 6.1 shows a 2D domain, in which a particle assigns a portion of mass to blocks at different refinement level. The grid points which accumulate mass are marked with a cross. Notice, that in order to obtain a symmetric mass cloud, the number of grid points is not equal to 3^{NDIM}. A symmetric mass cloud gives the most appropriate mass distribution. Clearly though, it involves consideration of each block's refinement level.

A proposed intermediate solution involves ignoring refinement levels all together, and simply accumulating mass at the refinement level of the current block [11]. Choosing this option affects the quality of the mass distribution because the accumulated mass in all nearest neighbour grid points is assymmetric in shape. However, as an intermediate step it seems appropriate. Unfortunately, it cannot be used as an intermediate step in `Grid_mapParticlesToMesh` because it is incompatible with the termination criterion. Recall, that termination occurs when the `currentmass` attribute of all particles is numerically zero. When non-symmetric portions of mass are extracted from the particle, the reduced `currentmass` attribute is unlikely to become numerically zero. In other words, mass is not conserved between the particle and the mesh.

Therefore, `Grid_mapParticlesToMesh` must incorporate a way of generating a symmetric mass cloud. For symmetry, mass accumulation must occur in 3^{NDIM} nearest neighbour grid points at the same refinement level. Clearly, this is not possible so "prolongation" and "restriction" techniques are introduced. These are techniques for accumulating mass in the current block, when its grid points are at a different refinement

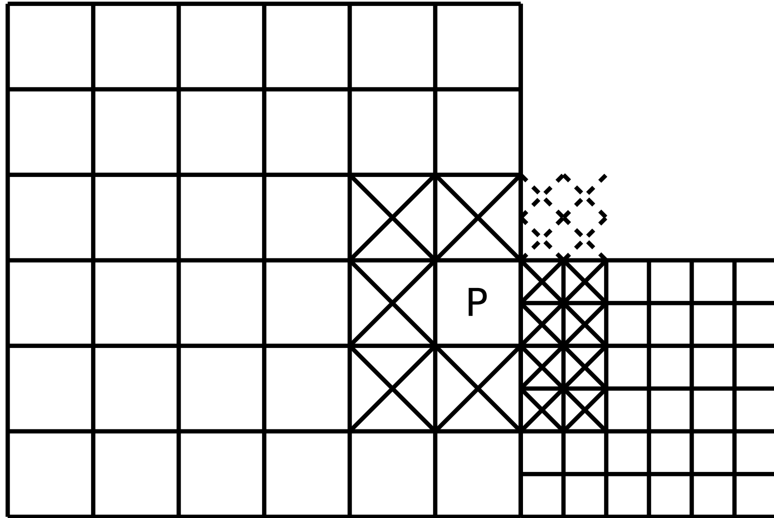

Figure 6.1: Performing prolongation to generate a symmetric mass cloud in a 2D domain

level to the host block. Prolongation is used to describe the interpolation of coarse grid points into finer grid points, and is shown in Figure 6.1. The alternative procedure named restriction, involves accumulating mass in a coarse grid point from the finer grid points in the host block.

The technique adopted in `Grid_mapParticlesToMesh` for prolongation and restriction involves each process calculating the mass assignment in the host block's nearest neighbour grid points. This allows each process to determine the mass that should be accumulated in each region of the physical domain. Further, it ensures each process calculates the same mass accumulation in regions of the physical domain. This technique is the only way to ensure that the `currentmass` attribute in each particle reaches numerical zero.

Each process calculates which of its grid points occupy the same physical region as the host's grid points. Once this is determined it is possible to perform mass accumulation by restriction or prolongation. When restriction is necessary, the calculated mass in each host grid points is copied into grid points occupying the same region of physical space. It is possible that the mass in multiple host grid points is copied into a single grid point of the current block. When prolongation is necessary, the calculated nearest neighbour host grid points must be interpolated. This is attempted by using an interpolation procedure from IVS. Unfortunately, due to lack of time prolongation is not successfully implemented.

At the time of implementation, the only sample problem available is Pancake. For testing, the refinement parameters `lrefine_min` and `lrefine_max` are set to different values and the Pancake problem is run. It is found that blocks in the adaptive grid do not refine to different resolutions. Refinements sometimes happen, but every block

in the computational domain is set to the same resolution. This is because of the uniform particle distribution of the Pancake problem. Therefore, to obtain a computational domain at various resolutions, the source code is modified to fire arbitrary refinements for certain blocks. This enables the restriction and prolongation routines to be tested.

Arbitrary refinements are fired to form the block decomposition shown in Figure 6.2. A particle is placed in grid point P in block 1. Mass accumulation must occur in grid points that correspond to the shaded nearest neighbour grid points. This means mass assignment should occur for grid point 10 and 11 of block 2. This is obtained in `Grid_mapParticlesToMesh`, however, it is found that mass accumulation in `MapParticlesToMesh` occurs in grid point 9 and 10 of block 2. This is a small bug in the FLASH 2.5 implementation which is reported to the FLASH development team. However, it indicates the `Grid_mapParticlesToMesh` implementation in an adaptive grid is working correctly for this test case.

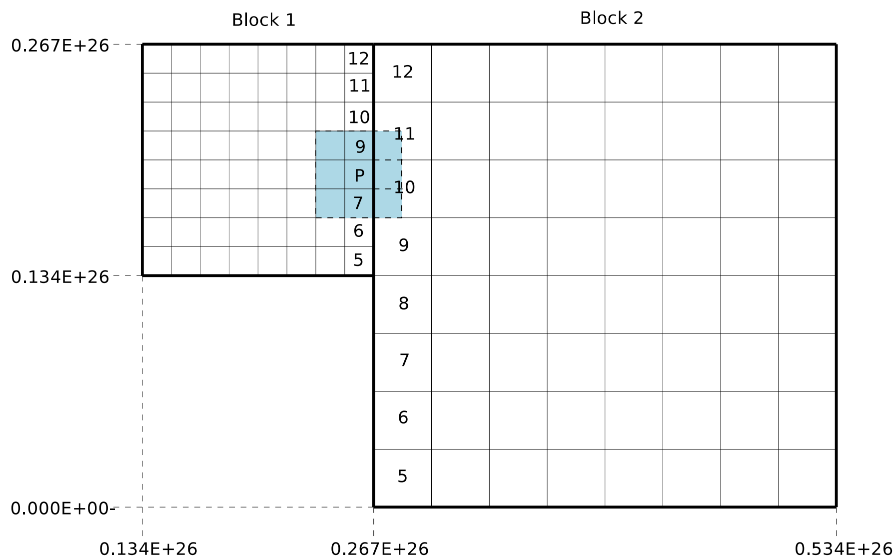

Figure 6.2: Performing restriction in a 2D domain.

Chapter 7

Future Optimisations

The initial profiles reveal that Grid_mapParticlesToMesh runs slower than Map-ParticlesToMesh for 128^3 particles. To improve the runtime performance, some potential optimisations are discussed. These are categorised as either calculation optimisations in Section 7.1 or communication optimisations in Section 7.2.

7.1 Calculation Optimisations

The runtime performance of the implemented algorithm is dominated by significant calculation costs. Therefore, the procedures would benefit from a sequential optimisation.

A line-by-line breakdown of several procedures (e.g. mapCarefullyToHostBlock, bIsCellContainedInBlock, bMapsToInternalCell, etc.) reveal that a large portion of time is spent evaluating if statements. It is possible to replace some of these if statements with pre-processor #ifs, e.g. the dimensionality, NDIM. Although this eliminates the runtime branch, it sacrifices runtime flexibility, because code must be recompiled to change the dimensionality.

A more general solution is to alter the code so that the if statement can be removed. An example where the code should be altered is the if statement in Figure 5.6. This is used to constrain mass assignment to grid points, but is particularly costly because the if statement is enclosed in the centre of nested do loops. Further, the condition tested in the if statement is the return value of the function bMapsToInternalCell, which itself contains a number of if statements.

These if statements within bMapsToInternalCell compare the position of a grid point against the block grid point limits. Here, a logical .true. is returned if the grid point index is greater than or equal to the low grid point index limit, and less than the high grid point limit in each dimension. One way to improve performance is to evaluate all particle positions at a single time. In addition, the integer value 1 could be used in place of a logical .true.. Introducing such a concept enables the mass assignment

expression to be replaced with:

```
EachZ:  do k = -1*K3D, K3D, 1
  EachY:  do j = -1*K2D, K2D, 1
    EachX:  do i = -1, 1, 1
      solnVec(iVar, ip+i, jp+j, kp+k) = &
      solnVec(iVar, ip+i, jp+j, kp+k) + &
      (real(valid(i, j, k))*wt(i, j, k)*rho)
    end do EachX
  end do EachY
end do EachZ
```

Here, `valid(i,j,k)` is an array containing a 0 or 1 in each element. When the array contains a 1, mass assignment will occur. The operation is effectively nullified if the element `i`, `j`, `k` in array `valid(i,j,k)` is 0.

It is actually possible to eliminate all `if` statements in the procedure by creating expressions that return either a 0 or 1. The cryptic expression below, provides this functionality by using a combination of the Fortran90 intrinsic functions `max` and `sign`.

```
do i = -1, 1, 1
  valid_i(i) = max(sign(1,gr_ihi-(ip+i))*sign(1,(ip+i)-gr_ilo),0)
end do
```

Here, a grid point index `(ip+i)` is compared against the high `gr_ihi` and low `gr_ilo` grid point index limits of the block. If the grid point index is within the index limits then each sign expressions will return 1. These signed expressions are multiplied together to give 1, and then the maximum of this value and 0 is returned. In thus case, the value stored in `valid_i(i)` is 1.

On the other hand, the grid point could exist outside the block index limits. Under these circumstances one of the two `sign` expressions will return `-1`. As such, performing the `max` function returns a 0, which indicates no mass assignment. There is no opportunity for both `sign` expressions to return `-1` because `gr_ihi>gr_ilo`.

The proposed code to replace the Figure 5.6 fragment is:

```
subroutine OptimisedMapToSameRefinement()
  implicit none
  integer :: i, j, k
  integer, dimension(-1:1, -1:1, -1:1) :: valid
  integer, dimension(-1:1)  :: valid_i, valid_j, valid_k
  real, dimension(-1:1, -1:1, -1:1) :: property_reduction
  real, pointer :: solnVec(:,:,:,:)

  !Initialise all elements as valid, and then correct later.
  valid_i(:)=1; valid_j(:)=1; valid_k(:)=1;

  !Valid expressions because gr_ihi > gr_ilo.
  do k = -1*K3D, K3D, 1
    valid_k(k) = max(sign(1,gr_khi-(kp+k))*sign(1,(kp+k)-gr_klo),0)↩
       -K3D+1
```

```
  end do

  do j = -1*K2D, K2D, 1
    valid_j(j) = max(sign(1,gr_jhi-(jp+j))*sign(1,(jp+j)-gr_jlo),0)↩
       -K2D+1
  end do

  do i = -1, 1, 1
    valid_i(i) = max(sign(1,gr_ihi-(ip+i))*sign(1,(ip+i)-gr_ilo),0)
  end do

  !valid(i,j,k), property_reduction(i,j,k), wt(i,j,k)
  !enable pipeline to be kept filled as elements are independent.

  call Grid_getBlkPtr(mapping_block,solnVec,CENTER)
  EachZ:  do k = -1*K3D, K3D, 1
    EachY:  do j = -1*K2D, K2D, 1
      EachX:  do i = -1, 1, 1

        !Stores a 0 if not valid, and a 1 if it is valid.
        !This depends on the dimensionality of the problem.

        valid(i, j, k) = valid_i(i)*valid_j(j)*valid_k(k)

        solnVec(iVar, ip+i, jp+j, kp+k) = &
        solnVec(iVar, ip+i, jp+j, kp+k) + &
        (real(valid(i, j, k))*wt(i, j, k)*rho)

        property_reduction(i, j, k) = &
        (real(valid(i, j, k))*wt(i, j, k)*particle_property)

      end do EachX
    end do EachY
  end do EachZ
  call Grid_releaseBlkPtr(mapping_block,solnVec,CENTER)

  particle_currentProperty = particle_currentProperty - &
      sum(property_reduction)

end subroutine OptimisedMapToSameRefinement
```

The removal of the `if` statements gives the compiler more opportunity to optimise the code. Unforunately, there is no further time available to look at hardware counters. However, it is thought that the compiler would be able to schedule instructions to reduce pipeline stalls. If successful, the key ideas presented above could be transferred to the routines `mapcarefullytohostblock` and `bmapstointernalcell`.

The line-by-line breakdown in Figure 5.6 reveals that writing values to the array `solnVec` is particularly expensive. This is because the values accessed in `solnVec` are not stored in contiguous memory addresses. A possible optimisation involves reordering the `solnVec` array so that the grid attribute index, `iVar` corresponds to the last array index. This would result in the loop over `i`, `j`, `k` accessing elements in contiguous

memory locations, which will improve cache usage. However, reordering the indices in such a widely used data structure may be detrimental to the performance of other procedures in the application. For instance, in other procedures, the iVar indice may vary most rapidly if it is necessary to access multiple grid attributes for a single grid point.

Alternatively, the solnVec data structure can be accessed less frequently. The mass accumulation for all particles on a block can be stored in a local array representing internal grid points. Then, when all particles have been processed, the local array can be copied into the non-contiguous elements of solnVec.

The xprofiler profile shows that the function getcellindex is responsible for 3.6% of time and is called 47,312,304 times. This is pure runtime overhead because the function only serves as a data accessor function. As such, it can be safely removed from the program, and the data in the module can be accessed directly.

A significant amount of time is spent in functions that assign particle mass to non-host blocks. However, realistic simulations typically use blocks that have many more internal grid points relative to guard cells. As such, more particles will contribute mass entirely to internal grid points in a single block in a realistic simulation. Therefore, an easy optimisation is to increase the number of internal grid points.

The memory performance of the MapNaivelyToHostBlock and MapCarefully-ToHostBlock is likely to improve if the particles in the initial particles array are sorted in block ID order. This would make it possible to have a single call to the Grid_getBlkPtr API function. The Grid_getBlkPtr API function enables data to be read and written to grid points in a single block. As multiple particles are likely to contribute mass to this block, grid points will potentially remain in cache, and so can be accessed faster than grid points in main memory. In addition, fewer calls to Grid_getBlkPtr will minimise the impact of executing the API function.

The number of calls to bIsCellContainedInBlock can be reduced by considering only relevant particle positions. Recall that apparent particle positions are introduced to remain compatible with periodic boundary conditions. At the current time, every single apparent particle position is tested when attempting to assign mass to blocks next to a computational domain boundary. However, the number of apparent particle positions can be reduced by half, simply by considering whether the block exists on the low or high side of the computational domain.

There is a significant amount of time spent executing FLASH API functions. Grid-_getBoundBox, Grid_getDeltas, Grid_getBlkBC. This can be reduced by storing the results in a look-up table, rather than recalculate the results each time. This does not require a large amount of memory as the number of blocks in a simulation is small relative to the number of particles.

Another optimisation is to replace the search over blocks with a call to the PARAMESH neigh array.

7.2 Communication Optimisations

Results in section 5.2 indicate that the MPI_AllReduce dominates the MPI communication time for 128^3 particles run on 8 processors.

A simple optimisation involves reducing the frequency of calls to MPI_AllReduce [27]. This will help reduce the impact of the MPI_AllReduce, but still provide the possibility of early termination. The optimal frequency is problem dependent, and is therefore hard to propose a suitable value. As such, it may be an idea to create a parameter which controls the frequency of the call to avoid testing everytime.

MPI communication can be further improved by optimising the point-to-point MPI_Sendrecv operation. This is possible by reducing the number of bytes that are communicated between processes. At the moment, an array of type particle is used as the source and destination buffer. This affects performance because it is the partially mass accumulated particles that are communicated. These are stored in the buffer instead of particles. At the time of writing a particle datatype consists of 16 reals, and a partialparticle datatype consists of 7 reals. This means a large number of empty fields are communicated unnecessarily. Further, it is possible to reduce the size of the communicated message without the need for a dedicated partial particles' buffer. This involves using an MPI vector datatype to specify the partial particle attributes in the particle datatype. Effectively, the derived datatype provides a stencil [28] over the contiguous memory attributes of the partial particle. In the FLASH simulation, the first ACTIVE_PART_PROPS reals contain the partial particle attributes. As such, the following derived datatype is appropriate.

```
MPI_Type_vector (1, ACTIVE_PART_PROPS, NPART_PROPS, FLASH_REAL,
ActiveParticleType)

MPI_Type_commit(ActiveParticleType)
```

Here, the ActiveParticleType datatype is constructed using a single block of ACTIVE_PART_PROPS instances of FLASH_REALs, and the pattern repeats every NPART_PROPS FLASH_REALs. This derived datatype may then be used in the MPI_Sendrecv communication. In theory, sending less data should reduce the time to exchange a message. Alternatively, the particle particles' attributes could be manually packed into the destination buffer. This would ensure partial particles attributes are contiguous in memory. However, it is a messy solution which will make the program harder to understand. This is because extra book-keeping will need to be introduced to keep track of the last used memory location of the destination buffer.

Chapter 8

Conclusions

8.1 Conclusion

This report has investigated an algorithm that assigns particles' mass to a mesh. A background of the algorithm is provided, along with a description of its desirable features. The algorithm is implemented in a procedure named `Grid_mapParticlesToMesh` for FLASH 3.0 beta. It is found that the implementation performs in the desired way for uniform grid simulations. This is verified by performing several test cases, and comparing results against `MapParticlesToMesh` in IVS. Currently, the implementation cannot be used with an adaptive grid, in which blocks are at a different refinement level. However, the source code is organised to permit an easy extension.

The implemetation is of interest because it does not use guard cell exchange for parallel communication. This is important because the removal of guard cell grid points reduces memory consumption. It is shown that removal of guard cells in a typical 3D simulation in FLASH 3.0 beta can reduce grid memory consumption by approximately 35%. The implication is that potentially larger systems with more particles can be studied.

The technique of communicating partially accumulated particles between processes is valid. Here, the particle can be passed around many times, and the appropriate process can extract a portion of mass to accumulate in its internal grid points. A single particle attribute named `currentmass` is used to keep a record of the particle's mass yet to be accumulated in grid points.

It is found that the runtime of `Grid_mapParticlesToMesh` is slower over all test cases than `MapParticlesToMesh` in IVS. Typically runtime is 3-5 times slower for a 128^3 particle simulation on processor counts up to 64 on HPCx. This is disappointing because the intention of the project is to overcome the bottlenecks in `MapParticlesToMesh`.

The performance bottlenecks of `Grid_mapParticlesToMesh` are investigated using mpiprof, xprofiler and Paraver profilers. Profile results indicate that both calculation and communication is expensive. In an eight processor simulation, it is shown

that calculation costs dominate. However, the impact of communication becomes more important at higher processor counts.

Calculation is expensive because memory locality, which is composed of spatial and temporal locality is poor. Spatial locality is low because successive references to nearest neighbour grid points are not contiguous in memory. This is because of the way the indices in the FLASH 3.0 beta block data structure are ordered. To improve the spatial locality, it is proposed that either array indices are re-ordered, or a temporary array is introduced. Temporal locality is low because many blocks are referenced when accumulating particles' mass to grid points. This can be reduced by eliminating a complete search over each block existing in a process. Instead, information in PARAMESH global data structures can be used to determine which blocks are neighbours with blocks in different processes. This can be used to reduce the number of blocks that need to be checked for potential mass accumulation. The flow of execution is also impeded by many `if` statements. The large number of `if` statements are used to constrain mass accumulation to internal grid points only. A technique which does not require `if` statements, and still restricts mass accumulation to internal grid points is presented.

Communication is expensive because the strategy forces global synchronisation whenever particles are exchanged between processors. This means particle exchange is delayed until the slowest processor has finished processing each of its particles. Here, a processor's slow completion time may be due to load imbalance. Therefore, performance is sacrificed by forcing the system to work in lock-step. Some techniques are proposed for improving communication performance. This includes reducing the frequency of calls to `MPI_Allreduce`, and using MPI derived datatypes to reduce the amount of data sent during point-to-point communications.

It is hard to comment about the effectiveness of the algorithm until optimisation is attempted. However, perhaps simpler strategies are better suited to this problem. For example, the approach used in `MapParticlesToMesh` for IVS involves exchanging accumulated grid points between blocks. This involves a large quantity of messages between blocks in different processes. However, it is possible to overlap calculation with communication to hide the cumulative message latency, which involves using non-blocking sends and receives. In addition, no expensive collective communication is required to determine when mass accumulation is complete. Therefore, the system is not forced to move in lock-step, and processors can proceed at their natural speed. It also seems like a more appropriate strategy than the Sieve algorithm when the computational domain is spread across a large number of processors. There may be analytically fewer messages in the Sieve algorithm, but messages involve collective communications and therefore forced synchronisation. The total cost of the MPI messages depends on the number of processors a particle visits. Here, the number of visits depends on the quality of the Morton-Space filling curve domain decomposition.

Finally, alternative algorithms should still be considered, and the approach adopted in `MapParticlesToMesh` not discounted. `MapParticlesToMesh` does not involve the expensive collective communication, which will ultimately limit the scalability of `Grid_mapParticlesToMesh`. As such, optimsiation of `MapParticles-`

ToMesh is still valuable. However, the memory required for guard cell exchange in MapParticlesToMesh is not desirable. As such, both algorithms could be implemented in Grid_mapParticlesToMesh, and users given the option of selecting one.

8.2 Summary of achievements

A tested uniform grid implementation of the Sieve algorithm is delivered to the FLASH center. The delivered procedure named Grid_mapParticlesToMesh brings FLASH 3.0 beta a step closer towards performing cosmological simulations. Before this project, only the skeleton to Grid_mapParticlesToMesh existed.

The runtime of the procedure is compared against the MapParticlesToMesh procedure of IVS. In addition, the performance of Grid_mapParticlesToMesh is analysed using various profilers. The analysis reveals the major performance bottlenecks, which are discussed during the report. Some potential optimisations are proposed, which may help improve the runtime of Grid_mapParticlesToMesh. It is possible that some of the optimisation strategies can be used in other procedures in the FLASH 3.0 betasource code. For example, the communication optimisations may be useful for Grid_moveParticles, which implements a similar Sieve algorithm.

The delivered Grid_mapParticlesToMesh procedure is only a uniform grid implementation. However, no major modifications to the code are required to implement particle mass accumulation in an adaptive grid, as the Grid_mapParticlesToMesh was designed with adaptivity in mind.

This project gives useful feedback to the FLASH team about the practical performance of the Sieve algorithm implementation. The descriptions presented in this report could contribute to the FLASH userguide.

The Grid_mapParticlesToMesh procedure is tested using the Pancake problem. Some minor problems with the Pancake were discovered and corrected. The memory allocation bug still exists, however, the procedure which requires modification is specified in the report.

8.3 Future work

The Grid_mapParticlesToMesh procedure needs to be extended to incorporate mass accumulation in grid points owned by blocks at different refinement levels.

The runtime performance of the Grid_mapParticlesToMesh procedure needs to be investigated. This is because the time taken to perform a simulation with 128^3 particles is too long. It would be valuable to measure the effect of some of the proposed optimisations listed in this report.

The performance of the code may improve if MPI-2 single-sided communications (Remote Memory Access (RMA)) are used. Here, a processor is able to use a remote write to accumulate mass in grid points belonging to another processor. Using RMA requires only one processor to participate in the communication, meaning that the synchronisation is reduced.

The `Grid_mapParticlesToMesh` procedure operates without the use of guard cells, and so requires less memory. To take advantage of this the rest of the FLASH source code must be compatible with this reduced memory mode. The FLASH software development team are currently re-writing code units to function without guard cells. Where this is not possible, non-permanent guard cells will be introduced, which refers to the idea of having a single working block per process with guard cell storage [29]. This guard cell storage is then used for all guard cell exchanges. Whenever a block needs to receive data from a remote block, the data in the local block is copied to the single working block with guard cell storage. The solution is advanced on the working block, and then the updated data is copied back to the appropriate local block. This technique is relatively slow, so the approach in `Grid_mapParticlesToMesh`, which avoids guard cell exchange is appropriate.

Bibliography

[1] ASC / Alliance Center for Astrophysical Thermonuclear Flashes,
http://flash.uchicago.edu/website/home, accessed 06/03/07.

[2] The Message Passing Interface (MPI) standard,
http://www-unix.mcs.anl.gov/mpi

[3] PARAMESH V3.4, Parallel Adaptive Mesh Refinement,
http://www.physics.drexel.edu/∼olson/paramesh-doc-/Users_manual/amr.html, accessed 06/03/07.

[4] FLASH Cosmology Home Page,
http://www.astro.uiuc.edu/ pmricker/research/codes/-flashcosmo, accessed 08/08/07.

[5] E. Breitmoser, G. Pringle and M. Antonioletti.
Specifications Document for D-JRA2-3.1 D-JRA2-3.1, 2006, EPCC, University of Edinburgh, UK.

[6] R. W. Hockney, J. W. Eastwood. *Computer Simulation Using Particles* IOP Publishing Ltd. pp18-23.

[7] Treecode Guide,
http://ifa.hawaii.edu/ barnes/treecode/treeguide.html, accessed 19/08/07.

[8] The Virgo Consortium,
http://www.virgo.dur.ac.uk, accessed 06/03/07.

[9] Dr Tom Theuns, ICC, University of Durham, UK. Pers. Comm.

[10] E. Breitmoser, G. J. Pringle, O. Bournas and M. Antonioletti.
Specifications Document for D-JRA2-3.3 D-JRA2-3.3, 2006, EPCC, University of Edinburgh, UK.

[11] Dr A. Dubey, ASC / Alliances Center for Astrophysical Thermonuclear Flashes, University of Chicago, USA. Pers. Comm.

[12] C. Daley. *Optimising the Improved Virgo Simulation*, MSc in High performance Computing, The University of Edinburgh, 2007.

[13] Transitioning from FLASH2 to FLASH3,
flash.uchicago.edu/website/codesupport/-
tutorial_talks/June2006/transition.ppt, accessed 09/08/07

[14] Mapping with Space Filling Surfaces,
http://bmi.osu.edu/resources/techreports/-
ahmedBokhariV21.pdf, accessed 19/08/07.

[15] K. Heitmann, P. M. Ricker, M. S. Warren, S. Habib, *Robustness of Cosmological Simulations. I. Large-Scale Structure* The Astrophysical Journal Supplement Series, 160:28-58, September 2005.

[16] N-Body Simulation Techniques: It's for everybody (Cont.),
supernova.lbl.gov/ evlinder/umass/sumold/com6lx.ps,
accessed 10/07/07

[17] E. Carretti, A. Messina, *Dynamic Work Distribution for PM Algorithm* arXiv:astro-ph/0005512v1

[18] A. Dubey and K. Antypas.
Parallel Algorithms for moving Lagrangian Tracer Particles on Eulerian Meshes 2007, ASC / Alliances Center for Astrophysical Thermonuclear Flashes, University of Chicago, USA.

[19] Software Testing - White Box Testing Strategy,
http://www.buzzle.com/editorials/4-10-2005-68350.asp,
accessed 01/07/07

[20] HPCx Home-Page,
http://www.hpcx.ac.uk, accessed 28/06/07.

[21] Where and why did my program crash?,
http://www.hpcx.ac.uk/support/FAQ/crash/, accessed 07/04/07.

[22] The HDF5 Group,
http://www.hdfgroup.org, accessed 09/03/07.

[23] R. H. Ostrowski. *Porting, Profiling and Optimising an AMR Cosmology Simulation*, M.Sc. in High performance Computing, The University of Edinburgh, 2006.

[24] User's Guide to the HPCx Service (Version 2.02),
http://www.hpcx.ac.uk/support/documentation/-
UserGuide/HPCxuser/HPCxuser.html
accessed 06/03/07.

[25] K. Dowd, *High Performance Computing* O'Reilly & Associates, Inc. pp130.

[26] Official Paraver website,
http://www.cepba.upc.es/Paraver, accessed 14/07/07.

[27] W. Gropp, E. Lusk, A. Skjellum. *Using MPI Portable Parallel Programming with the Message-Passing Interface second edition* The MIT Press. pp96-97

[28] N. MacDonalad, E. Minty, J. Malard, T. Harding, S. Brown, M. Antonioletti. *Writing Message Passing Parallel Programs with MPI. A Two Day Course on MPI Usage. Course Notes, Version 1.8.2*, EPCC, University of Edinburgh, UK.

[29] A Tutorial Without Permanent Guardcells Storage, `http://www.physics.drexel.edu/ olson/paramesh-doc/-Users_manual/amr_tutorial_npgc.html`, accessed 10/07/07

[30] M. Metcalf, J. Reid, M. Cohen, *Fortran 95/2003 explained*, Oxford University Press, pp99.

[31] Totalview Technologies, `http://www.totalviewtech.com/index.htm`, accessed 15/08/07.