Pencil FFT Implementation in the FLASH Simulation Code

Bruce Duncan

October 8, 2007

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2007
# Contents

1 Introduction .................................................. 1
   1.1 Background .................................................. 2
       1.1.1 Poisson solver ........................................... 2
       1.1.2 Description of FFTW and PFFT methods ..................... 3
   1.2 Goals .......................................................... 4
   1.3 Outline ........................................................ 4

2 Method .......................................................... 6
   2.1 Discrete Fourier transforms ................................... 6
   2.2 Transformation to the pencil grid ............................. 7
   2.3 Extracting the necessary data from the AMR structure ........... 8

3 Algorithm design and implementation .......................... 12
   3.1 Algorithm .................................................... 12
   3.2 Implementation ............................................... 15
       3.2.1 Sizing the processor and data grids ....................... 18
   3.3 Problems ..................................................... 19

4 Testing and Verification ........................................ 23

5 Conclusions ..................................................... 26
   5.1 Discussion .................................................... 26
   5.2 Critical Evaluation ........................................... 27
   5.3 Future Work .................................................. 27
       5.3.1 Minor changes ............................................. 27
       5.3.2 Non-conformant grid sizes ................................. 28
       5.3.3 Further design and implementation ........................ 28
       5.3.4 Testing and performance measurement ..................... 29
       5.3.5 Summary .................................................. 30

6 Additional Work ................................................ 31

A Code ............................................................ 33
   A.1 Build process ................................................ 33
   A.2 README ....................................................... 33
## List of Figures

2.1 Example 6x4 AMR grid showing the tree structure which represents it. From [4] .................................................. 9

2.2 One two–dimensional FLASH block from [6]. Interior cells are shaded and the surrounding four–deep guard cells are shown. ............. 10

3.1 The single redistribution step required to calculate the FFT in three dimensions using FFTW. Diagram by H. Jagode [7]. ................. 13

3.2 The two redistribution steps required to carry out an FFT in three dimensions with PFFT. Diagram by H. Jagode [7]. ................. 13

3.3 The shape of an example FLASH uniform grid (left) and its equivalent PFFT grid (right). ................................................................. 14

3.4 Flow chart showing the interaction of the various modules which make up the implementation ......................................................... 16

3.5 Definition of the Fortran derived type required to implement an array of pointers. ................................................................. 18

3.6 Makefile.h patch to fix compilation errors and warnings on HPCx. 21

3.7 A loop which causes compilation to fail with the –C flag when SCRATCH_–GRID_VARS_BEGIN = SCRATCH_GRID_VARS_END. From source/–IO/IOMain/hdf5/serial/PM/io_writeData.F90 ..... 22
Abstract

FLASH 3 beta is the latest release of a parallel Adaptive Mesh Refinement (AMR) code for simulating thermonuclear flashes. It can be easily adapted with other modules as required to solve various different physical problems.

Poisson’s equation is a second–order partial differential equation which is used to calculate the gravitational force on a system of particles. It can be solved efficiently using Fast Fourier Transform (FFT) algorithms. This project aimed to allow non–uniform grid simulations to use the Pencil FFT (PFFT) library, which requires a uniform grid. This is achieved by remapping the grid into a uniform one which can be used directly by the PFFT code.

This project produced an implementation of the algorithm in uniform grid mode and the basic structure required to implement the AMR mode, but did not solve the problems associated with the latter. The project did not produce the library calls required for a working parallel FFT nor did it produce any FFT test cases, although both are discussed.
Acknowledgements

Most importantly I would like to thank Anshu Dubey and Klaus Weide of the University of Chicago FLASH Center, without whom this project would not have been possible. I am also grateful to Tom Theuns of the Virgo consortium whose guidance helped steer the project on many occasions. My supervisor, Elena Breitmoser, provided invaluable input and helped keep me sane throughout the project.

Having the help of another MSc student, Chris Daley, also working on the FLASH code was invaluable. On more than one occasion, Chris helped me to solve bugs that I had already spent too much time tracking down and which were missed by the FLASH team. Also everyone at EPCC who were always willing to help with my seemingly stupid and trivial problems.

To my girlfriend, Amy, who has been so patient with me in the past few months: Thank you.

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

FLASH [2] was written to solve the problem of thermonuclear flashes associated with supernovae and neutron stars. FLASH is a Message Passing Interface (MPI) [3], Adaptive Mesh Refinement (AMR) [4] Fortran 90 code. It comes with a number of modules, some of which can use Fast Fourier Transforms (FFTs) for the efficient solution of linear equations. For example, it is used to solve cosmological dark matter simulations for the Virgo consortium [5], and it contains modules for wind tunnel simulations and molecular dynamics, among others [6]. FLASH is developed by the University of Chicago FLASH Center under contract from the United States Department of Energy.

FLASH uses PARAMESH [4] to implement an AMR grid. The computational volume is divided into blocks containing a fixed number of cells. PARAMESH places blocks of finer resolution, with smaller cell size, as specified by the system being simulated. This allows FLASH to concentrate computational resources on parts of the problem which require more detailed simulation and hence are more computationally intensive. FLASH 3 is currently in the beta phase and has a wide variety of new features. Like FLASH 2.5, it is extensible, allowing individual modules to be overridden and tailored to a specific simulation. We make use of this modularity to implement the algorithm described in this report.

It should be clearly noted that the implementation simply performs the necessary communications such that the PFFT library may be called with relative ease in the future. There is no implementation for the PFFT library calls themselves, nor for the AMR-aware transformation. Hence, there are no tests or performance measurements for the adaptive parallel FFT. The data redistribution was implemented in Uniform Grid mode, along with a set of tests.

The simplifying assumption has been made that the dimensions of the data grid are each a multiple of the corresponding dimension of the processor grid. This assumption made the implementation easier, but the algorithm does not depend on it in general. This limited the range of tests which could be run, but the tests which were available represented the expected usage in practice.
1.1 Background

Here we present the methods used to solve the equations of motion of a gravitational system, in both iterative and direct spectral schemes. We outline the steps required to make use of the Pencil FFT (PFFT) library, especially the interface with PARAMESH, and how it differs from the FFTW method used by the Virgo consortium.

1.1.1 Poisson solver

Cosmology simulations such as the Virgo simulation need to solve Poisson’s equation to compute the force on a self–gravitating fluid, for example dark matter. The equation relates the density, \( \rho \), of a system to the potential energy, \( \phi \), by

\[
\nabla^2 \phi = 4\pi \rho. \tag{1.1}
\]

In Cartesian co–ordinates in three dimensions, the Laplace operator is given by

\[
\nabla^2 = \left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right). \tag{1.2}
\]

It takes a different form in general curvilinear co–ordinates, for example, polar co–ordinate systems, but we do not consider these, as the FLASH grid can transform any co–ordinate frame into Cartesian form.

Equation (1.1) may be written in finite difference form,

\[
\nabla^2 \phi(x, y, z) =
\frac{1}{h^2} (\phi(x + h, y, z) + \phi(x - h, y, z) \\
+ \phi(x, y + h, z) + \phi(x, y - h, z) \\
+ \phi(x, y, z + h) + \phi(x, y, z - h) \\
- 6\phi(x, y, z)),
\]

where \( h \) is the cell spacing. This is the equation which the FLASH multigrid solver solves iteratively.

Equation (1.1) can also be solved by convolution with a Green’s function which can be computed once and stored for future timesteps:

\[
\phi(x) = \int_{-\infty}^{\infty} dx' G(x - x') \rho(x'). \tag{1.3}
\]

In Fourier space, this is simply a multiplication. Fourier transforms are quite computationally expensive and, in serial, there is a lot more work than iterating (1.1). However, in parallel the time spent communicating to perform the forward and backward Fourier
transformations once is much less than the overhead of communicating nearest neighbours at each iteration.

For this reason, the iterative multigrid solver used by the Virgo simulation was replaced with the non–iterative spectral method to create the improved Virgo simulation [8]. A speed–up factor of 8.9 was achieved even on small data sets, but the implementation failed to handle large data sets on machines which do not swap to disk and hence have no virtual memory, such as HPCx [18].

1.1.2 Description of FFTW and PFFT methods

The modularity of FLASH is an important factor in its flexibility. FLASH 3 adds some new modules which are especially important for this project: the inclusion of the Pencil FFT (PFFT) library [9] and the availability of a uniform grid (UG) mode. The UG greatly simplified the initial implementation of the algorithm. The PFFT library provides the motivation for this project and gives some exciting opportunities to experiment with the simulation code. Although this was not done due to time constraints, we can compare the performance (and especially the performance scaling) of the PFFT method (with three one–dimensional FFTs) with the FFTW [11] method implemented in the improved Virgo simulation (with two two–dimensional FFTs). PFFT is a fast Fourier transform library using fttpack, “a package of Fortran subprograms for the fast Fourier transform of periodic and other symmetric sequences” [10] written in Fortran 77, largely between 1973 and 1979. PFFT computes the Fourier transform based on “pencils”: these are long cubic–based columns of \( N_x \times N_y \times N_z \) cells, with a column covering the whole computational volume, and hence \( N_x \leq N_y \ll N_z \). The columns are distributed over the available processors by the PFFT algorithm, as shown in Figure 3.3 on page 14.

In the FFTW algorithm the problem is decomposed in one dimension, hence the data distributed in two–dimensional “slabs”, as shown in Figure 3.1 on page 13. However, as shown by Breitmoser et al. [8], the parallel scaling of this method is not readily determined. This is because there was not enough data to distribute amongst large numbers of processors due to the limited problem size which was tractable on HPCx. Good scaling is expected with larger problem sizes, and the one–dimensional decomposition is especially suited to large problems. Additionally, in order to create the uniform FFT grid from an AMR grid, special care must be taken to avoid allocating memory which is never used. The FFTW algorithm could have been adapted to better managed memory in this regard, but it was decided that a new PFFT algorithm would provide more interesting performance data. This over–allocation of memory was one of the reasons for the failure of large data sets on HPCx.

By using a two–dimensional decomposition, we avoid the need to do a global all–to–all communication and also avoid allocating large slabs of uniform FFT grid. More importantly, the two–dimensional decomposition allows scaling to much larger processor counts. The library to compute the FFT using a two–dimensional decomposition, in
this case PFFT, handles the two redistribution steps. A method is required to convert the FLASH grid into a PFFT grid, call the PFFT routines and then perform the reverse mapping. An example of the FLASH and PFFT grids which must be mapped to each other is shown in Figure 3.3 on page 14.

1.2 Goals

The first pre-requisite of the project was to understand the FLASH Application Programming Interface (API) and which functions would be needed to interface with the AMR structures. Next, a clear understanding of the layout of the FLASH grid and the requirements of the PFFT library were needed. With this knowledge in hand, the ultimate goal of the project was to create a set of routines to allow a FLASH user to call the PFFT library on an arbitrarily refined AMR grid. The intermediate goals of the project were to create a simplified implementation of the code which worked with the new uniform grid mode of FLASH 3 and one which worked with an AMR grid at a constant refinement level. Overall, the algorithm to be implemented is as follows:

1. For each cell (a real element in the data array), determine which column covers the same region of space. Find which processors contain the column.
2. For each pair of processors, exchange the required pieces of data.
3. For each received chunk of data, insert the density field onto the local column. This may require prolonging or restricting the density to the appropriate refinement level (see Section 2.3).
4. Given the density on all columns, obtain the gravitational potential.
5. Invert steps. Copy the potential from the column to the blocks that cover this column.

The aim of the project is to implement this new PFFT algorithm. This should improve the performance of cosmological simulations in FLASH as well as providing the ability to use Fourier transforms to solve equations other than the Poisson equation.

1.3 Outline

This report is divided into five chapters. In the next chapter, we discuss the methods used to perform a discrete Fourier transform, the steps required to transform the FLASH data into a form which can be used by the FFT library and explain how to interface with the FLASH data structures. Chapter 3 is concerned with the design and implementation of the code and also explains problems encountered during the implementation process. Chapter 4 details the testing methods which were implemented and also discusses some testing methods which could be implemented in a future version. The final chapter
summarises the project and provides some suggestions for future improvements and some work which must be done to meet all of the project goals. Finally, Appendix A shows the README file and Makefiles which describe the build process, as well as giving an example of the implementation.
Chapter 2

Method

In order to solve Poisson’s equation for gravity, a Green’s function may be convolved in Fourier space. In this chapter, the Fourier transform and the steps which need to be taken in order to prepare the simulation data for processing by the PFFT library are presented. The problem solved by this project is this data preparation. As shown in Section 2.3 interfacing the PFFT library with the AMR grid of FLASH in parallel is not straightforward.

2.1 Discrete Fourier transforms

The Fourier transform is widely used in science, from signal processing to the solution of partial differential equations. Mathematically, a Fourier transform is performed on a continuous function and is the generalisation of the infinite Fourier series. A Fourier series is the infinite sum of sines and cosines which is the exact representation of a periodic, continuous function. We replace the sum by an integral so that in one dimension, the forward transformation is defined by

\[
F(u) = \mathcal{F}[f(x)](u) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i xu} dx, \quad (2.1)
\]

and the backwards transformation by

\[
f(x) = \mathcal{F}^{-1}[F(u)](x) = \int_{-\infty}^{\infty} F(u) e^{2\pi i xu} du. \quad (2.2)
\]

This continuous transform is not representable on a computer. Numerically, the Discrete Fourier Transform (DFT) is formed on an interval \([0, N]\) by taking \(u = k\Delta\) where \(k = 0, \ldots, N - 1\) and \(\Delta\) is a real number which is \(\frac{1}{N}\) of the range of \(x\). From this and Equation (2.1), we get

\[
F(n) = \sum_{k=0}^{N-1} f(k) e^{-2\pi i n k}, \quad (2.3)
\]
and, from Equation (2.2),

$$f(k) = \frac{1}{N} \sum_{n=0}^{N-1} F(n)e^{2\pi ik \frac{n}{N}}. \tag{2.4}$$

These operations are easily implemented on a computer, although their execution time scales as $\Theta(N^2)$, where $\Theta$ means asymptotically tight bound [1]. Thankfully, FFT algorithms exist which are able to compute the same problem in $\Theta(N \log N)$ time.

The function $F(n)$ is a spectral representation of $f(k)$. In other words, the frequencies of $f$ may be extracted from $F$. Certain operations become much simpler in Fourier space, for example differentiating a continuous transform is a simple multiplication by $i\omega$, where $\omega = 2\pi k$. Convolution, defined by

$$h = f \odot g = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau, \tag{2.5}$$

can easily be calculated in Fourier space by a simple multiplication of the two arguments,

$$\mathcal{F}[h] = H = FG = \mathcal{F}[f] \mathcal{F}[g]. \tag{2.6}$$

Equation (2.5) is of the same form as the Green’s function in equation (1.3) that we require to solve Poisson’s equation (1.1). Hence, we can solve Poisson’s equation in Fourier space very easily by making use of Equation (2.6).

The two–dimensional DFT is a simple generalisation of the one–dimensional case,

$$F(n, m) = \sum_{k=0}^{N-1} \sum_{j=0}^{M-1} f(k, j)e^{-2\pi i(n \frac{k}{N} + m \frac{j}{M})}. \tag{2.7}$$

The backward transform is similarly formed, except that the normalisation factor is now $\frac{1}{N M}$.

These details are handled by the PFFT library, which simply requires that the data is presented in a three–dimensional real array.

### 2.2 Transformation to the pencil grid

The PFFT library requires the data to be distributed in three–dimensional cuboids, with all the data required to compute the FFT in a single dimension on one processor. In order to achieve this layout, some redistribution of the data must occur.

On the FLASH uniform grid, the data is distributed in three–dimensional cuboids on an arbitrary three–dimensional grid of processors. There is no constraint that any dimension must be contained entirely within one processor. To propagate the required data from the FLASH grid to the PFFT grid, we must determine which parts of the FLASH grid on each processor overlap with the PFFT grid on each other processor.
On the PFFT grid, the number of processors in one of the dimensions (say, \(x\)) must be one. This way, the PFFT library can compute the FFT in the first dimension on every processor without performing any communications. The FLASH processor layout will generally be as close to a cube as is possible, although the layout may be specified arbitrarily in uniform grid mode. In the uniform grid mode, every processor can determine the extent of the FLASH and PFFT grids on every other processor. From there, it can decide easily how much data it will need to send to and receive from every processor.

### 2.3 Extracting the necessary data from the AMR structure

The simulation data in FLASH are stored in a five–dimensional real array named \(\text{unk}\). There are three dimensions for the simulation data in the physical \(x, y, z\) dimensions, a dimension for the computational variable to be stored (such as density, pressure or temperature) and a final dimension which represents the PARAMESH block (indexed from one to \(\text{MAXBLOCKS}\)). FLASH 2.5 required the user to query a database for the constants which indexed a particular variable, but FLASH 3 simplifies this procedure by declaring the variable indices as preprocessor \#defines.

A typical grid structure is illustrated in Figure 2.1. This shows some of the key features of PARAMESH. Firstly, note that all of the blocks are the same size (6 \(\times\) 4 cells), only the size of a cell varies. Secondly, notice that the refinement level never changes by more than one at any boundary. Finally, see that even at full refinement, the data array still exists at a higher level in the tree. This means that, if we choose to, we can operate on the data grid at a level of refinement that is lower than the highest available. This will be useful later.

The action of “restriction” means to move up the tree, averaging the values of each nearest cell. To “prolong” means to move down the tree, interpolating the data to increase the resolution of the grid. FLASH allows the user to define up to four variables, along with upper and lower threshold limits, whose values are monitored to trigger refinement or derefinement upon crossing one of the thresholds. The size of a refined block is always \(\frac{1}{2^d}\) of its parent, where \(d\) is the dimensionality of the problem. Hence, in two dimensions, each block will be cut into 4 pieces upon refinement. In our simulations, each block will be cut into 8 pieces in three–dimensions.

FLASH provides abstraction routines to access the simulation data, such as \texttt{Grid\_getBlkData}. As arguments, this routine takes the block ID, the variable to be retrieved and the extent of the region to be retrieved. It returns a three–dimensional array containing the requested data. This data may or may not include the guard cells of a block. Guard cells, which have been copied from neighbouring blocks, are provided so that a simulation does not need to worry about communicating data between processors in order to advance the simulation and hence it does not need to know about the shape of the grid except on its local block. As shown in Figure 2.2, a block can be configured...
such that it is surrounded by guard cells (in this case, four in each dimension), which are copied from the neighbour blocks at each iteration. Although the PFFT mapping algorithm does not use guard cells, it is important to be aware of their presence.

The complementary routine, Grid_putBlkData allows write access to the data grid in a similar fashion.

FLASH also provides some routines to control the restriction and prolongation of the grid. The routine which will be used by this algorithm is Grid_restrictAll-Levels. This routine synchronises the tree, passing data on the leaves to their parents. In this way, the data in the higher levels of the tree can be accessed without needing to do any work to manually restrict the data. Apart from this routine, most of the refinement is automatic. Specific prolongation and restriction routines are required to handle the
mapping to and from the PFFT data grid.

The global grid is indexed as if the grid were fully refined. For example, a one-dimensional problem with a block size of eight and maximum refinement level of three will have a maximum x co-ordinate of $8 \times 2^{3-1} = 32$, since if the grid were fully refined there would be $2^{3-1} = 4$ blocks each of size eight.

The extent of the entire computational volume is given by $\text{Grid\_getGlobalIndexLimits}$. This implementation uses the extent to determine which processors will contain which data. In each dimension, the extent of the grid is divided by the number of processors in that dimension and each processor is given a contiguous piece of the grid of that size. If the size of the grid is not divisible by the number of processors, the highest numbered processor has less data than the others.

In order to identify a block uniquely, we use the $\text{Grid\_getBlkCornerID}$ subroutine. This returns a pair of tuples (for example, two triplets in three dimensions) as arrays, containing the corner ID and the stride. The corner ID contains the co-ordinates of the block on the global grid. The stride, $s$, is related to the refinement level, $n$, of the
block by

\[ s = 2^{\text{lrefine\_max}} - n, \]

(2.8)

where \( 1 \leq \text{lrefine\_max} \leq n \), and hence to obtain the corner ID of the next block in the positive \( x \), \( y \) or \( z \) dimensions, one must multiply the size of the current block by its stride and add the corner ID.
Chapter 3

Algorithm design and implementation

Now we will present the computational steps taken to solve the remapping problem. Then, the actual implementation will be summarised and explained. Finally, the problems which arose during this implementation will be discussed.

3.1 Algorithm

In order to solve Poisson’s equation in Fourier space, FLASH must redistribute the data into a shape which can be used by an FFT library. The improved Virgo simulation used FFTW which required the data to be distributed into two-dimensional “slabs”, as shown in Figure 3.1. We wish to investigate the performance of the PFFT library. For this we require to distribute the data into one-dimensional “pencils”, as demonstrated in Figure 3.2.

FLASH distributes the work data amongst the processors in a possibly arbitrary way. For example, when using UG mode the user is required to specify the number of processors in each dimension manually. The PFFT library requires that the data be distributed such that all of the data in a single dimension, say $x$, be on one processor. That is, the number of processors in the $x$ dimension is one. Of course, in the case that the FLASH grid has processors distributed in the same way, no work needs to be done, but when there is a processor grid where some data must be moved, we must do so using MPI communications.

Once the data are suitably distributed, we call the PFFT library to make the forward transformation into Fourier space. This involves two redistribution steps as shown in Figure 3.2. Then, some work will be done in Fourier space, after which the data must be retransformed back into real space. This also involves two redistribution steps, such that when the data is returned into real space, it is in the same shape as before the transformation. Therefore, to transform the data into the original FLASH shape is simply the inverse of the first process.
The uniform grid essentially removes the FLASH abstraction of a block (see Section 2.3), which is an $n$–dimensional array with fixed extent containing the computational variables. In UG, each processor has exactly one FLASH block of the same size as every other processor’s block. This greatly simplifies the process of redistributing the data. Without needing to communicate with any other processor, every processor
knows how much data it will need to send to and receive from every other processor. The extent of each block is calculated from the FLASH grid and compared to each block from the PFFT grid. Then, it may be determined which parts of each grid which overlap and which therefore must be sent or received.

The algorithm is as follows:

1. Work out the arrangement of processors which gives the most square two–dimensional grid,
2. Get the sizes of the local block and the global problem,
3. Calculate the co–ordinates of every processor on the FLASH and PFFT grids,
4. From these co–ordinates, calculate and store the limits of each block for each processor on the FLASH and PFFT grids,
5. Calculate how much of each other processor’s FLASH grid we must receive, allocate enough space for that data and post an asynchronous receive[1],
6. Similarly, calculate how much data we need to send to each other processor’s PFFT grid, retrieve that data from the FLASH Grid module and send it,
7. Wait for the receive operation to complete and store the data at the appropriate point in the PFFT grid array,
8. Do some work on the data (including a Fourier transform),

[1] Asynchronous receives are a way to avoid deadlock with MPI. The processor which is waiting for data will return from the MPI library immediately, allowing the program to continue and execute the corresponding send operations.
9. Repeat steps 5 to 7, except from the PFFT grid to the FLASH grid, to restore the FLASH grid.

This algorithm was implemented in Fortran 90, using the FLASH Grid interface and MPI, as described in the next section.

3.2 Implementation

FLASH’s modular structure provides a set of high-level modules, called: Grid, IO, Multispecies, PhysicalConstants, RuntimeParameters (these five make up the FLASH core infrastructure), physics and Simulation (user code is expected to override parts of these). The new PFFT algorithm is best placed in the Grid module, which manages the physical structure of the computational mesh. Hence, the module and subroutine names added will be prefixed with Grid_.

The algorithm was largely implemented in a pair of subroutines called Grid_mapFLASHtoPFFTgrid and, its inverse, Grid_mapPFFTtoFLASHgrid. Various support routines provided the glue for these routines to be used in user code, namely Grid_pfftInit, Grid_pfftDump and a module called Grid_pfftData to store all of the persistent data. This modularity potentially allows for more efficient operation, since the persistent data need only be calculated and allocated once, even when the FFT proper may be executed many hundreds of times in a given simulation.

Initially, the code resided in one subroutine, Grid_mapToPFFTGrid, which would do the forward and backward transformations. This was useful for testing, since both operations are very similar and could be displayed simultaneously to spot trivial bugs, but it was soon realised that this monolithic approach would be unwieldy in a practical context. Splitting up the code provides modularity and encapsulation which would be lost if the simulation code had to be placed within the remapping routines.

The following paragraphs describe in detail the function of each subroutine. Their interactions are shown in Figure 3.4. Note that this is not a call tree, each subroutine is called by the simulation. In this case, Grid_unitTest calls each subroutine in turn. Grid_pfftMapData is not shown, since it simply encapsulates data storage and does not contain any executable code. “Other processing” should be taken to include calling the PFFT library.

Grid_pfftMapInit calculates the shape of the PFFT processor grid. It aims for the most square two-dimensional grid possible. Then, it fills the flash_other and pfft_other arrays with the co-ordinates of each processor on the FLASH and PFFT grids, respectively. It allocates enough space to store the PFFT data grid and records the extent of the grids in the gridLimits array.

Grid_mapFLASHtoPFFTgrid performs the transformation to the PFFT grid. There are two algorithms in this subroutine which are selected based on the FLASH_GRID_-PARAMESH macro. The first is the UG algorithm. It calculates how much of the FLASH
Figure 3.4: Flow chart showing the interaction of the various modules which make up the implementation
grid on each processor overlaps with the PFFT grid on this processor and allocates enough memory to store the communicated data. It then sets up an asynchronous MPI receive operation. Next, it calculates the overlap of the each processor’s PFFT grid with the FLASH grid on this processor and sends that chunk of data, obtained with Grid_getBlkData, to the appropriate processor. The algorithm then waits for the receive operations to complete before storing the data in the PFFT data grid.

The AMR version of Grid_mapPFFTtoFLASHgrid is slightly different. It does not calculate the extent of the FLASH grid in advance, but instead employs a loop over all blocks for each processor. That is, for each processor, it checks how much data from this block should be sent. It stores this information in an integer array of size $p$ and performs a global all–to–all operation. This informs every processor of how much data it should expect to receive from each other processor. In order to know where to place the data on the PFFT data grid, the receiving processor must be informed of the co-ordinates of each block and its refinement level. These data are sent to each processor using the same asynchronous receive – synchronous send – wait algorithm as the grid simulation data. The last step in the implementation is to communicate the grid data. An asynchronous receive, a synchronous send and a wait are used. Then, each block is placed at the appropriate point on the PFFT data grid.

This is the forward mapping, from the FLASH grid to the uniform PFFT grid. As an example of the code, Grid_mapPFFTtoFLASHgrid is given in Appendix A.

Grid_pfftMapDump is used to check the validity of the remapping. In UG mode, the FLASH grid can be dumped with Grid_dump, as described in Section 4. In order to dump the PFFT data grid, we use MPI-IO to set up a file named “PF0000”, create a subarray with the size of the local PFFT data grid and instruct all processors to write their data grid to the file in parallel.

Grid_mapPFFTtoFLASHgrid is very similar to the forward mapping. The UG mode performs the reverse overlapping calculations to determine how much data will be sent and received by this processor. It then performs an asynchronous receive, a synchronous send and a wait, before placing the received data back onto the FLASH grid with Grid_putBlkData.

The AMR implementation does almost the same thing as the forward mapping, except now it must iterate over the list of blocks upon receiving the PFFT data in order to determine which block will hold the data.

Both Grid_mapFLASHtoPFFTgrid and Grid_mapPFFTtoFLASHgrid take var as a parameter. This parameter corresponds to the variable in the FLASH unk array which will be Fourier transformed. The variable, usually DENS_VAR for density, need not be specified at initialisation and so the result of the reverse mapping may be placed in a different variable from the source. This also allows one to perform an FFT on a different data set of the same size (i.e. a different variable stored in unk).

Grid_pfftMapFinalise simply deallocates the arrays allocated in Grid_pfftMapInit.
Grid_pfftMapData stores all of the persistent data required to implement the above routines. It defines the four–dimensional integer gridLimits array, the three–dimensional real pfft_data_grid array and three Fortran derived types given in Figure 3.5. The types are required to implement an array of pointers. Since the size of the data to be communicated is not known at compile time, an array must be instantiated with \(p\) pointers to a three–dimensional real array. This avoids the need to allocate a square array of size \((p, \text{max}(\text{commSize}(p)))\), which would waste a huge amount of space. The recvGrid array is a size \(p\) array of type grid_pointer, which stores the pointer to the communications buffer and also the MPI request object, which tags an asynchronous receive so that it can later be waited on.

```fortran
    type grid_pointer
      real, dimension(:,,:,:), pointer :: grid
      integer :: request
    end type grid_pointer
```

Figure 3.5: Definition of the Fortran derived type required to implement an array of pointers.

In the AMR incarnation of Grid_pfftMapData, the grid_pointer type is subtly different. It requires a second level of indirection to store the list of blocks on each processor.

These subroutines were placed in a Sod [13] setup. The Sod problem is one of the available FLASH test codes which models “an essentially one-dimensional flow discontinuity problem which provides a good test of a compressible code’s ability to capture shocks and contact discontinuities”. We only used the problem to set up the computational domain. Immediately after the first Fourier transform, FLASH terminated successfully, having completed all tests (detailed in Chapter 4).

### 3.2.1 Sizing the processor and data grids

A critical part of the algorithm is storing the location of each processor on the two–dimensional PFFT grid and the FLASH grid, which can have dimensionality from one to three. We also need to determine the extent of both data grids on each processor.

FLASH uses MPI_Split_comm to create a processor grid with independent MPI communicators. The code used for the PFFT mapping implementation used the grid sizes generated with these communicators to calculate processor co–ordinates. In three dimensions, the following code is used to calculate the co–ordinates of processor \(i\) in each dimension:
do  i = 0, dr_numProcs - 1
    flash_other(i, IAXIS) = mod(i, procGrid(IAXIS))
    flash_other(i, JAXIS) = mod(i / procGrid(IAXIS), procGrid(JAXIS))
    flash_other(i, KAXIS) = i / (procGrid(IAXIS)*procGrid(JAXIS))
...
end do

procGrid stores the number of processors in each dimension. Similarly for the PFFT grid, except that since this grid is two–dimensional, the x co–ordinate of every processor is 0. The code which generates the PFFT grid co–ordinates is in the same do–loop as the above FLASH code:

pfft_other(i, JAXIS) = mod(i, pfft_grid(JAXIS))
pfft_other(i, KAXIS) = i / pfft_grid(JAXIS)

In this case, pfft_other stores the processor co–ordinates and pfft_grid stores the number of processors in each dimension.

In the same loop, the limits of the FLASH and PFFT data grids on each processor are calculated. These will be used later when calculating where the grids overlap. As an example, to calculate the lower extent of the FLASH data grid on processor 0, we take that processors co–ordinates and multiply them by the global grid size divided by the number of processors in each dimension. To adhere to the Fortran array–indexing standard, we then add one. The process is similar for the upper extent. The code to perform this calculation also takes place in the above do–loop. After this code is completed, the values of pfft_other and flash_other are no longer used and the arrays are deallocated automatically. It is shown here with the PFFT grid limits calculation code to find the grid limits on processor i:

gridLimits(i, FLASH, LOW, :) = flash_other(i, :) * ←
globalIndexLimits / procGrid + 1
gridLimits(i, FLASH, HIGH, :) = (flash_other(i, :) + 1) * ←
globalIndexLimits / procGrid
gridLimits(i, PFFT, LOW, :) = pfft_other(i, :) * ←
globalIndexLimits / pfft_grid + 1
gridLimits(i, PFFT, HIGH, :) = (pfft_other(i, :) + 1) * ←
globalIndexLimits / pfft_grid

Here, the FLASH and PFFT constants index the array for the FLASH and PFFT grid limits, respectively. Similarly, LOW and HIGH elements store the lower and upper limits of each grid. globalIndexLimits is a three–dimensional array which stores the global problem size, as obtained from Grid_getGlobalIndexLimits.

3.3 Problems

A number of problems were encountered when implementing the algorithm. They are detailed here along with their solutions, where available.
Firstly, a misunderstanding of the FLASH “block” abstraction (described in Section 2.3) led to an incorrect implementation of the algorithm. The code was based on the assumption that a block could be retrieved and sent to the correct processor based on its global co-ordinates. This turned out to be invalid, since a block is generally much larger than the array sections which have to be exchanged. At any rate, they never overlap exactly with the processor grid in the general case. This problem was solved by moving the focus of the implementation from the “block” data structure to the data contained within a block, which is simply a three-dimensional real array.

When initially compiling the code, it was noted that FLASH did not compile the code properly on HPCx. FLASH ships a number of customised Makefile.h files for specific machines and broader Makefiles for different machine types. For example, HPCx uses the default AIX Makefile.h file. This file failed to compile FLASH for the following reasons:

1. mpxlf90 failed to compile umap.F because, by default, it compiles free-form Fortran. Adding -qfixed to the FFLAGS variables allowed it to compile fixed-form code without errors.

2. The compiler also gave a warning that the -qfree option had been superseded by -qfree=f90:

   COMMAND LINE 1520-022 (W) The FREE option has been replaced by the option FREE(F90).

   Making this change silenced the warning.

3. Finally, even though the Makefile defined CFLAGS_HDF5, which specifies the location of the HDF5 library to the C compiler, it later undefined the value. The latter definition was removed, allowing the program to compile with the HDF5 library.

   These problems were fixed with the patch given in Figure 3.6.

As is often the case with AMR simulations in FLASH, first attempts to run a simulation failed due to the value of MAXBLOCKS being too small. MAXBLOCKS specifies the maximum number of PARAMESH blocks which can reside on a processor. In initial tests, the parameters lrefine_min and lrefine_max were both set to 6. The lrefine parameters dictate how many levels of refinement are used. If the minimum and maximum are set to the same value, the grid always remains at that refinement level. As described in Section 2.3, each level of refinement requires 8 new blocks to be allocated in three-dimensions. Hence, with lrefine_min = 6 the program was attempting to allocate space for $8^6 = 262144$ blocks. This caused the simulation to fail almost immediately, complaining that the refinement required would exceed the maximum number of blocks available. The values of lrefine_min and lrefine_max were reduced to four, and the value of MAXBLOCKS increased to 4096.

FLASH failed at run time because of a missing use Grid_interface line in the main Grid_mapFLASHtoPFFTgrid routine. This caused the compiler to miscom-
Figure 3.6: Makefile.h patch to fix compilation errors and warnings on HPCx.

pile the call to Grid_getBlkIndexLimits, which has one optional argument to specify the computational variable for which to get the limits. The subroutine then attempted to access the argument and aborted with this error:

called index limits with invalid gridDataStruct

The missing use line was inserted at the top of the subroutines which use this function, solving the problem.

Some important compiler flags were unusable with the FLASH code. -C, which checks each reference to an array element, array section, or character substring for correctness [14] does not work with the FLASH code because it also creates errors due to illegal array accesses which are detected at compile time:

"Grid_getBlkData.F90", line 480.28: 1516-152 (S)
Zero-sized arrays must not be subscripted.
In this example, the access is to the scratch array, which is defined to be of zero size because no scratch variables have been specified for this problem.

The problem is that these errors are simply warnings without -C and do not cause the compilation to fail. These errors were caused by accesses to the scratch array, used to store temporary simulation variables, and loops of the form shown in Figure 3.7. When the NSCRATCH_GRID_VARS macro is zero, as is the case with the simulation setup used in this project, these loops never run and the arrays are never accessed. However, in this case the scratch array is also defined to have zero size, which the XLF compiler notices and flags as an error.

```fortran
  do i = SCRATCH_GRID_VARS_BEGIN, SCRATCH_GRID_VARS_END
    scratchBuf(:, :, :, i, :localNumBlockst) = scratch(:, :, :, :)
  end do
```

Figure 3.7: A loop which causes compilation to fail with the -C flag when SCRATCH_GRID_VARS_BEGIN = SCRATCH_GRID_VARS_END. From source/IO/IOMain/hdf5/serial/PM/io_writeData.F90

Also, -qlanglvl, which checks for nonconformance against various Fortran standards returned copious inappropriate warnings. FLASH uses the C preprocessor extensively, but no Fortran standard includes a preprocessor. Any useful warnings were swamped by many pages of very similar warnings, the first of which was:

```
  "Grid_getBlkData.F90", line 249.1: 1518-234 (L)
  The #line directive is not permitted by the Fortran 90 standard.
```

It would have been possible to use -qlanglvl by preprocessing the files in advance with the -d flag and running the compiler with -qlanglvl on the processed files instead. This was deemed to be a lot of work and was not attempted. Instead, the preprocessor-induced errors were ignored using a simple filter script.

In FLASH, all reals are defined to be of size 8, or double precision, by the command line option -qrealsize=8. When MPI communications were made with the MPI_REAL data type, this caused random memory corruption and only half of the data was actually transferred. Once the correct, FLASH-specific, data type FLASH_REAL was used instead, this final problem was fixed with the help of the dbx debugger.

We have presented the algorithm to map the FLASH data to the PFFT grid and back again. We have outlined the implementation which does this and given some of the problems encountered along with their solution.
Chapter 4

Testing and Verification

The ultimate test of the new PFFT–based algorithm is a direct comparison of the results of the simulation in FLASH 3 with both that of the multigrid and FFTW methods from FLASH 2.5. The latter is not possible, since the improved Virgo simulation has not been ported to FLASH 3. However, some tests of the mapping algorithm can be performed. Here, we outline these testing methods, give some results and discuss the testing which could be performed in future.

With a suitable reference simulation in FLASH 3, such as the improved Virgo simulation, a test routine would have made use of the FLASH HDF5 [15] checkpoint files and the Serial FLASH Output Comparison Utility (sfocu) [16], although this may require transforming the particle data into a mesh structure, since sfocu does not work with particles. Since it is unlikely that the simulations will be perfectly identical (the multigrid solver is iterative, for example), it would have been prudent to plot the magnitude of the error (defined as the absolute difference between the two data files) against simulation time. This would have required a qualitative interpretation of the results.

However, it is still possible to investigate the correctness of the mapping code. Uniform grid mode provides a convenient opportunity to engineer a test case where the size and shape of the FLASH and PFFT grids are the same. In this case, it is simple to show that there is no communication. Therefore, the computed size of the communications is zero and each processor merely copies data to it’s own PFFT data grid. In this case, the code was instrumented and the communication pattern was manually inspected. Each processor successfully calculated that it would be required to send and receive the same number of elements to and from itself. This does not constitute a test, but provided valuable information to aid in debugging deadlocks.

The FLASH 3 UG mode also provides a new feature which enables further testing. The Grid_dump subroutine uses MPI-IO to write the values of a chosen set of variables to a file one at a time. The data could be visualised using IDL, The Data Visualization & Analysis Platform [17], but this was not necessary. In order to test the validity of the mapping routine, another routine named Grid_pfftMapDump was written. This routine was mostly a simplified version of Grid_dump. It takes the data stored in
pfft_data_grid and dumps it to a file using MPI-IO. It does not need to handle multiple variables in the same way that Grid_dump does, nor does it have to worry about whether or not to dump guard cells, since the PFFT data grid has none. This test is very simple to run and was included in every UG test simulation. Both grids were dumped between the forward and backward mappings and the files were compared for binary equality using the diff utility.

A second, less reliable, test was to store the state of the FLASH UG simulation before and after the PFFT mapping. While this only served to show that the forward and backwards mappings were the inverse of each other, the code passed and it still proved to be useful for debugging. For example, this test provided the final piece of information required to solve the FLASH_REAL bug discussed earlier. It was noted that the grid data was undefined for exactly half of the computational domain, which corresponded to the fact that MPI was communicating half of the data, as expected, as MPI_REAL variables are half the size of FLASH_REAL variables.

For the test data, the Sod [13] problem was used. This initialised a unit cube with two regions of gas at different density and pressure separated by a diaphragm which is taken to break at \( t = 0 \). The angle which the diaphragm made with the \( x \)-axis was varied to provide more thorough testing. \( \Theta_x = 0, 45 \) and 90 degrees were used. The angle to the \( y \)-axis was also varied, using \( \Theta_y = 0, 45 \) and 90. Therefore, the plane representing the diaphragm was rotated in all three dimensions. The diaphragm was also offset in the \( x \) direction such that it crossed the \( x \) axis at 1.0, as this gave slightly less uniform data.

The tests which were carried out are outlined in Table 4.1. These were run on the national supercomputer, HPCx [18]. The Makefile and Makefile.h files which control the build process are given in Appendix A. A wall clock limit of thirty seconds was used to avoid wasting CPU time when debugging deadlocks and the maximum size of a core file was increased from zero to 300 megabytes. The code was revision controlled by Subversion [19] on flash.uchicago.edu.

The Sod problem provided a convenient framework to develop the code but the data at \( t = 0 \) are rather trivial. We work around the triviality of the problem by varying the angle of the diaphragm to the \( x \) and \( y \) axes, but more thorough test problems, such as the ones outlined in Chapter 5.3, are desirable.

With the availability of the \( -C \) compile flag, these tests provide strong evidence that the data grids allocated on each processor are at least large enough to hold the communicated data. Also, since the tests do not deadlock, significant evidence is provided that the communications pattern is correct. If the size of the data being sent did not match the expected size at the receiving end, the program would deadlock. Again using \( -C \) we can demonstrate that the origin of an array slice is correct, since the access would likely exceed the bounds of the array on the boundary cells.

Most importantly, the tests would fail if data was being sent to the wrong destination. The MPI-IO test confirms that the data is being redistributed correctly. If, for example, the ordering of processors in the target grid was reversed in one dimension, the test would fail.
Table 4.1: A summary of the test results for the UG implementation, including tests which were expected to fail. The tests were carried out for $\Theta_x$ and $\Theta_y = 0, 45, 90$. A block size of $NXB = NYB = NZB = 8$ was used.

All available testing methods have been outlined, which provide adequate empirical evidence, in the absence of a compatible port of the FLASH 2.5 simulations, that the code works. A comprehensive test suite has been implemented where this is possible and the code passes all tests. The tests which pass are:

1. Manual comparison of the communications pattern with what was expected,
2. Compare *Grid_dump* output with that of *Grid_pfftMapDump*, and
3. Compare the output of the reverse mapping with the input to the forward mapping routines.

The tests which should be implemented in future are given again in Section 5.3.
Chapter 5

Conclusions

The improved Virgo simulation is a dark matter simulation built on the modular FLASH 2.5 simulation code. It uses a Fast Fourier Transform, instead of the FLASH built-in multigrid solver, to calculate the force on a self-gravitating fluid. Instead of the FFTW method employed in FLASH 2.5, the improved Virgo simulation could utilise the PFFT library. In order to make use of and test the performance of the PFFT library included with the latest beta of FLASH 3, interface code was required. This should allow simulations to efficiently calculate FFTs in parallel on many thousands of processors.

This project produced a usable PFFT mapping routine to allow any uniform grid simulation to interface with PFFT. The code also provides a base on which to implement an AMR interface which would allow cosmological simulations to harness the full power of FLASH on an adaptive grid.

The implementation has been shown to conform to the Fortran standards and to pass all available tests. As demonstrated in Section 3.2, the memory usage is bounded and depends only on the size of the problem and the number of AMR blocks in use. The memory usage may become excessive on large numbers of processors, but in Section 5.3, mechanisms for mitigating this cost are discussed.

5.1 Discussion

The power of PFFT lies in its ability to scale to many more processors than a comparable one-dimensional decomposition. The performance exceeds that of the one-dimensional decomposition only where in the latter case there is not enough data to distribute over all processors.

According to H. Jagode [7], the performance of the PFFT-style method is worse than that of the two-dimensional FFTW method. This was attributed to a poor placement of MPI tasks on the BlueGene network but mostly the increased latency of the two-dimensional decomposition. Both methods displayed super linear speedup, which is likely to
be due to the vastly improved cache use on hundreds of processors. For a cubic problem of size $n^3$, where $n$ is the length of the data set in each dimension, the maximum number of processors which can be used is $n$ for the one-dimensional FFTW decomposition, but $n^2$ for the two-dimensional PFFT method. This allows PFFT to scale to much larger processor counts.

It is likely that this work described in [7] also applies to HPCx, however the actual performance scaling of FFTW vs. PFFT cannot be determined at this time for the reasons given in Section 5.3.

5.2 Critical Evaluation

The experience of working on such a large piece of software as FLASH was a great opportunity. The code is well-documented and the online API reference was invaluable, especially when trying to understand the AMR aspect. The code written for this project should be easy to incorporate into the final FLASH 3 release. Much of the time available to the project was spent in learning the FLASH interfaces.

I had the honour of collaborating with the FLASH Center team in Chicago. Working with such team on a different continent can be very demanding. Questions which take seconds to answer can have a latency of a few hours. However, having the expertise of the FLASH team on hand was crucial to the continuous productivity of the project. Without the input of those who work with the code every day, simple misunderstandings could have stalled the project for days.

As was anticipated, the project started late due to a holiday. The time lost was made up in the following weeks, but the stall allowed for the project to take a new and much more useful tack. While the initial project might have been easier and have had fewer risks, this project will be of much greater value to both the FLASH team and to the Virgo consortium.

5.3 Future Work

In this section, we will outline various changes which would improve various aspects of the implementation and also discuss the next steps which should be taken to fulfil the project goals.

5.3.1 Minor changes

Currently, the algorithm persistently stores less data than it could. For example, the FLASH to PFFT mapping algorithm sends the block metadata (corner ID, stride and index limits) to the receiving processor, but the PFFT to FLASH mapping routine also
sends the same information. It would be possible to simply save the block metadata and avoid communicating it again for the reverse mapping. Additionally, the reverse mapping has to do some work to decide which parts of the PFFT grid overlap with which FLASH blocks by looping over all local blocks. It would save a lot of effort to simply send the block ID back to the source processor, however more memory would be required. On the other hand, the amount of memory would be modest, merely 4 bytes per block. We already store much more than that for each block, as will be shown in the next section. The benefits of this change greatly outweigh the minimal memory usage and its implementation is strongly encouraged.

5.3.2 Non-conformant grid sizes

Currently, the code does not gracefully handle data grids which are not a multiple of the processor grid. The program crashes due to attempting to access an array out of bounds. The size of the data grid on each processor is determined by integer division, which always rounds towards zero (i.e. it truncates). For this reason, the size of the local PFFT grid is too small on the highest numbered processor (in a given dimension) if the FLASH grid does not divide perfectly into the chosen PFFT processor grid. For example, with 24 processors in a PFFT grid of shape $1 \times 4 \times 6$ distributing a $32 \times 32 \times 32$ FLASH grid, processor 24 calculates that it needs a local grid of shape $32 \times 8 \times 5\frac{1}{2}$. The last dimension is rounded down to 5 which causes the code to crash when it receives data from its neighbours.

This could easily be fixed by rounding up the PFFT data grid calculation. In this way, each processor, taking the previous example again, would allocate a grid of shape $32 \times 8 \times 6$ (the grid is only defined for $32 \times 8 \times 2$ for the highest-numbered processor). This is the most efficient shape, since no special code is required to handle the array at the edge of the computational domain. Also, it should be faster to process than a grid of shape $32 \times 8 \times 5$, since the highest-numbered processor then requires a grid of shape $32 \times 8 \times 7$, and the run time is determined by the greatest computational load.

This problem was solved late in the testing stage, as described in Chapter 6.

5.3.3 Further design and implementation

The most important step which will make the current code useful is to integrate it with the PFFT library. A test driver exists in the source/Simulation/Simulation-Main/unitTest/Pfft directory in the FLASH source which contains the basic code required to perform a distributed pencil FFT with a three-dimensional real array. It consists of code to transform a three-dimensional real array into a one-dimensional array of the same size, to gather the necessary metadata to perform the FFT and to call the initialisation and finalisation routines in PFFT.
Once the PFFT library has been interfaced, the next crucial step is to investigate the performance scaling. This simply requires that a large enough problem be simulated on a suitably large number of processors and the performance be compared to the FFTW algorithm present in the improved Virgo simulation. It is expected that the PFFT algorithm will exhibit worse performance on smaller numbers of processors, but that the FFTW algorithm will fall behind on larger processor numbers. The reasons for this assertion are outlined in Section 5.1. An intermediate testing step could be to fill the simulation data with an easily determined pattern, for example storing the co–ordinates of each cell in that cell would allow easy testing of the mapping algorithm.

The AMR implementation allocates a potentially large amount of memory at run time in order to avoid having to calculate or communicate metadata more than once. For example, it allocates three three–dimensional integer arrays of total size $12p^2b_p$ bytes where $p$ is the number of processors, $b_p$ is the number of blocks on the current processor and an integer is 4 bytes on HPCx. This is to buffer the metadata for each block to be sent to the appropriate destination processor. This could be very expensive on thousands of processors and is inefficient, especially since most entries will be empty where the grids do not overlap. The current implementation also allocates buffers to store incoming blocks before they are written to the PFFT grid array. This is necessary because, in general, MPI cannot handle Fortran array sections which are not contiguous in memory. Although MPI derived data types could have been used to work around this problem, this solution was rejected as the size of an array section can be different on each processor. This could have required as many MPI data types as processors and would have been more difficult to implement.

Similarly, the UG implementation buffers metadata, allocating space to store the extent of the grid on every processor in an integer array of size $48p$ bytes. It also stores the co–ordinates of every processor on the FLASH and PFFT processor grids, using an integer array of size $24p$ bytes. Combined, these persistent data take up $72p$ bytes of storage. This is insignificant, even on thousands of processors, but it could be avoided by simply moving the calculations to where they are needed. The arrays could be replaced with accessor functions without changing the main code by hiding the arrays and creating an interface of the same type. However, due to the small size and small asymptotic growth of this memory usage, avoiding it is not justified.

In general, it is not clear whether CPU time or memory is more valuable. Some more investigation would be required to decide which course of action is the correct one on a particular architecture. At the very least, any implementation must store the same amount of data in the PFFT data grid as exists in the FLASH grid, thus roughly doubling the memory requirements.

5.3.4 Testing and performance measurement

There are a variety of tests which could be implemented to demonstrate the correctness of the PFFT method. Some of these were outlined in Chapter 4. They are:
1. Fill the computational domain with a known pattern (for example, fill each cell with its cell ID, or fill the grid with a periodic function) and examine the grid at various points in the algorithm. Placing the cell ID in each cell allows a quick test after the grid has been transformed back into real space, and the periodic function (say, a sine or cosine) allows a quick test in Fourier space.

2. Compare the output of the improved Virgo simulation using FLASH 2.5 and the new PFFT method using FLASH 3 using sfocu (with the particles to mesh mapping) or IDL.

There is the potential for PFFT to outperform FFTW. PFFT avoids global communication and hence global synchronisation. However, PFFT sends smaller messages and therefore is more sensitive to latency. PFFT also has the potential to require less memory than FFTW. Since it sends smaller messages, MPI requires less memory to buffer the messages before sending. Neither of these advantages can be proven without a thorough test of both algorithms.

5.3.5 Summary

The most important next step is to implement the interface with the PFFT library. Then, a simple test can be written, preferably based on the improved Virgo simulation to allow a direct comparison for correctness. This requires that the improved Virgo simulation be ported to FLASH 3, since FLASH 3 cannot currently perform cosmological simulations. Once this has been done, an AMR simulation at a constant refinement level can be used to directly compare the new algorithm against the improved Virgo simulation in terms of performance. A number of small improvements could bring savings in time or memory to the existing code, as well.

Also, it would be useful to confirm whether PFFT enables datasets larger than $256^3$ particles to be simulated on HPCx, solving the problems noted by E. Breitmoser et al. [8].
Chapter 6

Additional Work

Late in the testing stage, it was decided to adapt the code to remove the restriction that the FLASH data grid must be a multiple of the PFFT processor grid. Previously, a FLASH grid such as $24 \times 40 \times 8$ (represented by a FLASH processor grid of $3 \times 5 \times 1$ with a cubic blocksize of 8) could not be transformed by the PFFT mapping code, since it involved neighbouring processors having a different block size. The PFFT mapping code determined that it should use a $1 \times 3 \times 5$ processor grid which resulted in allocating too little space for the transformation. This problem was solved.

The implementation determines the size of the PFFT data grid on each processor by assigning an equal sized chunk of the FLASH grid, based on the processor’s co-ordinates. Taking the above example, processor 14 (with co-ordinates $(0, 2, 4)$ is given a block of size $24 \times 13 \times 1$ (since integer arithmetic truncates the result). In order to cope with the non-conformant grid, an extension was written which additionally takes the modulus of the FLASH grid with the PFFT processor grid. In this example, this gives $(0, 1, 3)$, meaning that one extra cell had to be placed on the second dimension, and three extra cells had to be placed on the third dimension. In order to acheive this, extra code was written so that low-numbered processors in each dimension would be given one extra cell (by increasing their HIGH grid limit by one). This also involved increasing both the LOW and HIGH grid limits of each higher-numbered processor in that dimension by one.

In order to confirm that the new algorithm worked correctly, the expected grid decomposition was written out by hand. For the above problem, it was determined that the decomposition in the second dimension should be $(14, 13, 13)$ and in the third dimension should be $(2, 2, 2, 1, 1)$. This was confirmed by adding print statements to the code. Also, the test suite was re-run for every combination of processor count, FLASH grid and PFFT grid previously tested. All tested passed. Accordingly, Table 4.1 on Page 25 has been updated. The new test results are given in Table 6.1. Every row now passes both tests. Previously, rows in which the PFFT grid was not a multiple of the FLASH grid failed. Specifically, the test using 15 processors failed.

A minor change to the Grid_pfftMapDump routine was also required. Previously,
the code assumed that multiplying the local grid size by the processor grid would give the global grid size. Since the local grid is no longer the same on every processor, this did not give the expected result. The code was changed to use the HIGH grid limits on the highest-numbered processor, which gives the size of the global grid.

This is a major improvement to the code. The restriction that the FLASH grid must be a multiple of the PFFT processor grid has been removed.

<table>
<thead>
<tr>
<th>( N_{\text{proc}} )</th>
<th>FLASH grid dimensions (processors)</th>
<th>PFFT grid dimensions (processors)</th>
<th>MPI-IO test result</th>
<th>Backward / Forward test result</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>( 4 \times 3 \times 1 )</td>
<td>( 1 \times 3 \times 4 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>15</td>
<td>( 3 \times 5 \times 1 )</td>
<td>( 1 \times 3 \times 5 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>16</td>
<td>( 1 \times 4 \times 4 )</td>
<td>( 1 \times 4 \times 4 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>16</td>
<td>( 16 \times 1 \times 1 )</td>
<td>( 1 \times 4 \times 4 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>16</td>
<td>( 4 \times 2 \times 2 )</td>
<td>( 1 \times 4 \times 4 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>16</td>
<td>( 4 \times 2 \times 2 )</td>
<td>( 1 \times 16 \times 1 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>24</td>
<td>( 2 \times 2 \times 6 )</td>
<td>( 1 \times 4 \times 6 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>24</td>
<td>( 8 \times 1 \times 3 )</td>
<td>( 1 \times 4 \times 6 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>24</td>
<td>( 6 \times 2 \times 2 )</td>
<td>( 1 \times 4 \times 6 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>24</td>
<td>( 12 \times 1 \times 2 )</td>
<td>( 1 \times 4 \times 6 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>32</td>
<td>( 4 \times 4 \times 2 )</td>
<td>( 1 \times 4 \times 8 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>32</td>
<td>( 4 \times 8 \times 1 )</td>
<td>( 1 \times 4 \times 8 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>32</td>
<td>( 2 \times 2 \times 8 )</td>
<td>( 1 \times 4 \times 8 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>48</td>
<td>( 4 \times 4 \times 3 )</td>
<td>( 1 \times 6 \times 8 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>48</td>
<td>( 4 \times 3 \times 4 )</td>
<td>( 1 \times 6 \times 8 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>64</td>
<td>( 4 \times 4 \times 4 )</td>
<td>( 1 \times 8 \times 8 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>64</td>
<td>( 16 \times 2 \times 2 )</td>
<td>( 1 \times 8 \times 8 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
<tr>
<td>128</td>
<td>( 8 \times 4 \times 4 )</td>
<td>( 1 \times 8 \times 16 )</td>
<td>Passed</td>
<td>Passed</td>
</tr>
</tbody>
</table>

Table 6.1: A summary of the test results for the UG implementation. The tests were carried out for \( \Theta_x \) and \( \Theta_y = 0, 45, 90 \). A block size of \( NXB = NYB = NZB = 8 \) was used.
Appendix A

Code

A.1 Build process

The files given below form the FLASH build process. They are included for completeness. Makefile.h is included in the top-level Makefile and contains machinespecific build flags and compiler definitions. The top level Makefile contains the proper targets to build the flash3 executable. Both of these files are generated automatically by the ./setup script as described in the README.

Finally, the code for the Grid_mapFLASHtoPFFTgrid subroutine is given as an example of the code.

A.2 README

PFFT interface driver code README
=================================

Author: Bruce Duncan
Date: July 2007
Copyright: (C) Bruce Duncan 2007

This code was written as part of an MSc project at EPCC, University of Edinburgh, to allow the FLASH simulation code, version 3.0 beta, to make use of the PFFT library. This library computes a distributed FFT of real or complex data using the fftpack library. This interface code provides a way to transform the data in the FLASH AMR grid into a uniform real array which can be used by PFFT. As such, the code is specific to FLASH. This file explains how to build the code and use it in a simulation.

The code requires FLASH to run. You can obtain FLASH (after signing the license agreement) by downloading it from:
http://flash.uchicago.edu/

Unpack the source with the usual "tar zxf" command and then read the user’s guide to learn how to create your own simulation. To speed up this process, I have summarised the basic process below.

This code runs with the "Sod" simulation setup. Copy the PFFT interface code into the source/Simulation/SimulationMan/Sod directory. The Grid_unitTest.F90 file overrides the FLASH default and links the rest of the source files together. The Makefile snippet overwrites the one in the Sod directory so that the new source files are linked into the the executable. Next, run the following script to initialise the build system:

```
./setup Sod -auto +ug -3d
```

This instructs FLASH to create the build system to compile the Sod simulation in Uniform Grid mode in three dimensions. Next, ‘cd’ to the object directory which has just been created and run ‘make’. On HPCx, the following command is required to make everything go smoothly:

```
OBJECT_MODE=64 HDF5_PATH=/usr/local/packages/hdf5-1.6.5 gmake -e
```

After some time, the executable object/flash3 will be created. On HPCx, I created a LoadLeveler batch script from the HPCx user’s guide and ran the code with:

```
poe ./flash3 -llfile ~/inter.ll
```

The simulation puts its output in the object directory. PF0000 is the result of Grid_pfftMapDump. It contains a dump of the PFFT data array, an array composed of FLASH_REAL variables (floating-point numbers of size 8 bytes each). It could be visualised using various data plotting tools, for example IDL or AVS/Express.

### A.3 EPCC/Makefile

```
Simulation += Simulation_data.o
Simulation_init.o : Simulation_data.o
Simulation_initBlock.o : Simulation_data.o
Grid += Grid_pfftMapData.o Grid_unitTest.o Grid_pfftMapInit.o
    Grid_mapFLASHtoPFFTgrid.o Grid_mapPFFTtoFLASHgrid.o
    Grid_pfftMapDump.o Grid_pfftMapFinalise.o
```

### A.4 Top level Makefile
# Makefile for FLASH code. Automatically generated by setup script.
# Do not edit!

MAKEDISPLAY = 1
REORDERFLAG = 0

ifeq ("$(MAKEDISPLAY)","1")
MAKEFLAGS =
else
MAKEFLAGS = -s
endif

EXE = flash3

.PHONY: default
default: $(EXE)

# Machine-dependent include file

CDEFINES = -DMAXBLOCKS=2000 -DNXB=8 -DNYB=8 -DNZB=8 -DN_DIM=3
FDEFINES = $(MDEFS)-DMAXBLOCKS=2000 $(MDEFS)-DNXB=8 $(MDEFS)-DNYB=8 $(MDEFS)-DNZB=8 $(MDEFS)-DN_DIM=3

# print the compiler flags to file in FLASH/object/ (files name are also hard-coded into make_bstats)

PRINT_F77_FLAGS = @(ECHO) F77: $(FCOMP) $(FFLAGS) $(FDEFINES)
PRINT_F90_FLAGS = @(ECHO) F90: $(FCOMP) $(FFLAGS) $(F90FLAGS) $(FDEFINES)
PRINT_C_FLAGS = @(ECHO) C: $(CCOMP) $(CFLAGS) $(CDEFINES)
PRINT_L_FLAGS = @(ECHO) Linker: $(LINK) $(LFLAGS) $(EXE) $(Object files of units) $(LIB) $(CONFIG_LIB)

PRINT_FLAGS_FILE = setup_flags
SUCCESS_FILE = .success

include Makefile.h

# define touch and mv command if not already defined in Makefile.h
TOUCH ?= touch
MV ?= mv

FFLAGS := $(FFLAGS_DEBUG)
LFLAGS := $(LFLAGS_DEBUG)
LIB := $(LIB_HDF5) $(LIB_MPI) $(LIB_DEBUG)
CFLAGS := $(CFLAGS_HDF5) $(CFLAGS_DEBUG)

ifeq ("$(MAKEDISPLAY)","0")
define ECHO-COMPILING
@$(ECHO) Compiling $<
endif
else
define ECHO-COMPILING
endif

ifeq ("$(MAKEDISPLAY)","0")
define ECHO-PROCESSING
@$(ECHO) Processing $<
endif
else
define ECHO-PROCESSING
endif

#   Compile

.SUFFIXES:

.SUFFIXES: .f .F .f90 .F90 .c .C .o

%.o : %.f
 $(ECHO-COMPILING)
 $(FCOMP) $(FFLAGS) $(FDEFINES) $<
%.o : %.F
 $(ECHO-COMPILING)
 $(FCOMP) $(FFLAGS) $(FDEFINES) $<
%.o : %.f90
 $(ECHO-COMPILING)
 $(FCOMP) $(FFLAGS) $(f90FLAGS) $(FDEFINES) $<
%.o : %.F90
 $(ECHO-COMPILING)
 $(FCOMP) $(FFLAGS) $(F90FLAGS) $(FDEFINES) $<
%.o : %.c
 $(ECHO-COMPILING)
 $(CCOMP) $(CFLAGS) $(CDEFINES) $<
%.o : %.C
 $(ECHO-COMPILING)
 $(CPPCOMP) $(CFLAGS) $(CDEFINES) $<
%API.c %API.h %API-bridges.F90: %.int
 $(ECHO-PROCESSING)
 ./int2API.py $<

.PRECIOUS: %API.c %API.h %API-bridges.F90
# What to do in case we are unable to build an object file

MKFILES = $(shell grep -l $(1) Makefile.* | egrep -v "Makefile.→
Depend")
UNITNAMES = $(patsubst Makefile.%,%,$(1))
SRCFILES = $(shell ls | grep -i $(basename $(1)).)

.DEFAULT:
  @$(ECHO) Sorry, I am unable to build $@
  mention in $(call MKFILES, $@)
  @$(ECHO) Related directory contents: $(call SRCFILES, $@)
  @$(ECHO) If you do not see an appropriate source file, $@ is probably mentioned in the
  @$(ECHO) wrong Makefile and should probably go deeper inside the $(call UNITNAMES,$(call MKFILES, $@)) unit
  @$(RM) -f $(SUCCESS_FILE)
  @exit 1

# Setup buildstamp routine

.PHONY: setup_buildstamp.F90
setup_buildstamp.F90:
  @$(ECHO) Generating Buildstamp
  ./make_bstamp

# Release routine

.PHONY: setup_flashRelease.F90
setup_flashRelease.F90:
  @$(ECHO) Generating Flash Release
  ./make_release

# Build statistics routine

.PHONY: setup_buildstats.c
setup_buildstats.c:
  @$(ECHO) Generating Build Statistics
  ./make_bstats

# Make the Makefile.Depend
Makefile.Depend:
  @$(ECHO) Calculating dependencies
  ./setup_depends.py $(FFLAGS) $(F90FLAGS) $(CFLAGS) *.f *.→
  f90 *.F90 *.F

.PHONY: reorder
# run reorder script or kill the script if we dont need it
ifeq ("$(REORDERFLAG)","1")
  reorder: Makefile.Depend
    bash reorder.sh
else

reorder:
    rm -f reorder.sh

endif

.PHONY: reorderclean
ifeq ("$(REORDERFLAG)","1")
    bash reorder.sh --clean
endif

#    FLASH Units routine

#    include Dependency information
#    the -include supresses the warning if Makefile.Depend is not found

-include Makefile.Depend

#    Unit Makefiles

MISC_OBJS = $(MACHOBJ) setup_buildstamp.o setup_flashRelease.o →
            setup_buildstats.o setup_getFlashUnits.o

include Makefile.Burn
include Makefile.Conductivity
include Makefile.Cool
include Makefile.Cosmology
include Makefile.Diffuse
include Makefile.Driver
include Makefile.Eos
include Makefile.Flame
include Makefile.Gravity
include Makefile.Grid
include Makefile.Heat
include Makefile.Hydro
include Makefile.IO
include Makefile.Ionize
include Makefile.Logfile
include Makefile.MHD
include Makefile.MagneticResistivity
include Makefile.MassDiffusivity
include Makefile.Multispecies
include Makefile.Particles
include Makefile.PhysicalConstants
include Makefile.Profiler
include Makefile.RuntimeParameters
include Makefile.Simulation
include Makefile.Stir
include Makefile.Timers
include Makefile.Viscosity
include Makefile.flashUtilities
ALL_OBJ_FILES = $(Burn) $(Conductivity) $(Cool) $(Cosmology) $(Diffuse) $(Driver) $(Eos) $(Flame) $(Gravity) $(Grid) $(Heat) $(Hydro) $(IO) $(Ionize) $(Logfile) $(MHD) $(MagneticResistivity) $(MassDiffusivity) $(Multispecies) $(Particles) $(PhysicalConstants) $(Profiler) $(RuntimeParameters) $(Simulation) $(Stir) $(Timers) $(Viscosity) $(flashUtilities) $(MISC_OBJS)

# Remove duplicate names (side effect is sorting)
ALL_OBJS = $(sort $(ALL_OBJ_FILES))

# DATA_OBJS contains all object files which define modules, rest are CODE_OBJS
# CODE_OBJS = ALL_OBJS - DATA_OBJS
CODE_OBJS = $(filter-out $(DATA_OBJS),$(ALL_OBJS))

# Link
.PHONY: printflags

printflags:
ifeq ("$(MAKEDISPLAY)","0")
  @$($(ECHO) Compiler and Flags Info follows
  @$($(PRINT_F77_FLAGS)
  @$($(PRINT_F90_FLAGS)
  @$($(PRINT_C_FLAGS)
  @$($(PRINT_L_FLAGS)
endif

@$($(RM) $(PRINT_FLAGS_FILE)
@$($(RM) -f $(SUCCESS_FILE)
@$($(TOUCH) $(PRINT_FLAGS_FILE)
@$($(PRINT_F77_FLAGS) >> $(PRINT_FLAGS_FILE)
@$($(ECHO) >> $(PRINT_FLAGS_FILE)

39
$($(PRINT_F90_FLAGS)) >> $(PRINT_FLAGS_FILE)
@$(ECHO) >> $(PRINT_FLAGS_FILE)
$($(PRINT_C_FLAGS)) >> $(PRINT_FLAGS_FILE)
@$(ECHO) >> $(PRINT_FLAGS_FILE)
$($(PRINT_L_FLAGS)) >> $(PRINT_FLAGS_FILE)
@$(ECHO) >> $(PRINT_FLAGS_FILE)
@$(ECHO) Data Object files: >> $(PRINT_FLAGS_FILE)
@$(ECHO) $(DATA_OBJS) >> $(PRINT_FLAGS_FILE)
@$(ECHO) >> $(PRINT_FLAGS_FILE)
@$(ECHO) Code Object files: >> $(PRINT_FLAGS_FILE)
@$(ECHO) $(CODE_OBJS) >> $(PRINT_FLAGS_FILE)

.PHONY: data code clean distclean
data: $(DATA_OBJS)
code: data $(CODE_OBJS)

$(EXE) : printflags reorder data code
  @$(ECHO) Linking in Units and Libraries
  $(LINK) $(LFLAGS) $(EXE) $(ALL_OBJS) $(LIB) $(CONFIG_LIB)
  @$(ECHO) SUCCESS
  @$ $(SUCCESS_FILE)

# Clean up object directory
clean:

distclean: clean reorderclean
  $(RM) Makefile.Depend

precision_test: precision_test.o
  $(LINK) $(LFLAGS) precision_test precision_test.o

A.5 Makefile.h

# Set the HDF/HDF5 and PAPI library paths
# -- these need to be updated for your system
# If PAPI doesn’t exist on your system, comment them out
HDF5_PATH = /usr/local/hdf5/hdf5-1.4.2/parallel
PAPI_PATH = /usr/local
PAPI_FLAGS = -c -I$(PAPI_PATH)/include -qsuffix=f=F90:cpp=F90 -qfree

ZLIB_PATH =
NCMPI_PATH =
MPE_PATH =

# Compiler and linker commands
#
# Use the MPICH wrappers around the compilers -- these will automatically
# load the proper libraries and include files.
#
FCOMP = mpixlf90_r
CCOMP = mpcc_r
CPPCOMP = mpCC_r
LINK = mpixlf90_r

# pre-processor flag
PP = -D

# Compilation flags
#
# Three sets of compilation/linking flags are defined: one for optimized code
# code ("-opt"), one for debugging ("-debug"), and one for testing ("-test").
# Passing these flags to the setup script will cause the value associated with
# the corresponding keys (i.e. those ending in ",_OPT", ",_DEBUG", or ",_TEST") to
# be incorporated into the final Makefile. For example, passing ",-opt" to the
# setup script will cause the flags following "FFLAGS_OPT" to be assigned to
# "FFLAGS" in the final Makefile. If none of these flags are passed, the default
# behavior will match that of the ",-opt" flag.
# In general, ",-opt" is meant to optimize compilation and linking.

# should enable runtime bounds checking, debugger symbols, and other compiler-
# specific debugging options. "-test" is useful for testing different
# combinations of compiler flags particular to your individual system.
#
FFLAGS_OPT = -O3 -qintsize=4 -qrealsize=8 -cpp -c \ -qxlf90=autodealloc \ -qsuffix=cpp=F -qtune=auto -qfixed

# qhot causes problems with 2.0
# -qhot -qcache=auto
FFLAGS_DEBUG = -g -qxlf90=autodealloc -qsuffix=cpp=F -qintsize=4 -qrealsize=8 -cpp -c -qfixed -qlanglvl=90pure -C

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

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

CFLAGS_OPT = -c -O3 -qcache=auto -qtune=auto -DIBM
CFLAGS_DEBUG = -g -c -DIBM
CFLAGS_TEST =
CFLAGS_NCMPI =

MDEFS = -WF,
#
# Linker flags
# There is a separate version of the linker flags for each of the _OPT, _DEBUG, and _TEST cases.
#
# Linker flags for optimization
LFLAGS_OPT = -bmaxdata:0x80000000 -o

# Linker flags for debugging
LFLAGS_DEBUG = -bmaxdata:0x80000000 -o
LFLAGS_TEST = -bmaxdata:0x80000000 -o

# Library specific linking
#
# If a FLASH module has a 'LIBRARY xxx' line in its Config file, we need to
# create a macro in this Makefile.h for LIB_xxx, which will be added to the
# link line when FLASH is built. This allows us to switch between different
# (incompatible) libraries. We also create a _OPT, _DEBUG, and _TEST
# library macro to add any performance-minded libraries (like fast math),
# depending on how FLASH was setup.
#
LIB_HDF5 = -L $(HDF5_PATH)/lib -lhdf5 -L /usr/local/lib -lz
LIB_PAPI = -L$(PAPI_PATH)/lib -lpapi -L/usr/lpp/pmtoolkit/lib -lpmaipi
LIB_MATH = -lessl
LIB_OPT =
LIB_DEBUG =
LIB_TEST =
LIB_NCMPI =
LIB_MPE =
LIB_MPI =

# Additional machine-dependent object files
#
# Add any machine specific files here -- they will be compiled and linked
# when FLASH is built.
#
MACHOBJ =

A.6 Grid_mapFLASHtoPFFTgrid.F90

```fortran
!!****f* source/Grid/Grid_mapFLASHtoPFFTgrid
!!
!! NAME
!! Grid_mapFLASHtoPFFTgrid
!!
!! SYNOPSIS
!!
!! Grid_mapFLASHtoPFFTgrid(integer :: var)
!!
!! DESCRIPTION
!!
!! Fills the pfft_data_grid array with the data suitable for passing to the PFFT library. The FLASH data are distributed amongst processors such that the PFFT data for the x-dimension are entirely contained within one processor. The subroutine uses MPI to redistribute the data.
!!
!! There are two versions of this program, one for uniform grid mode and the other for PARAMESH’ AMR mode. The appropriate code is selected at compile time. The AMR code is essentially the same as the UG code, except that it communicates (using MPI) the amount of data to be transferred and the block metadata in advance.
!!
!! Most of the data on which this subroutine operates is dynamically allocated. For the AMR code, of the static data, there are
!! sizeof(integer)*3*MDIM = 36
!! bytes of working array and
!! sizeof(integer)*2*dr_numProcs = 8 * dr_numProcs
!! bytes of communications metadata.
```
Of the dynamic data, there are
sizeof(integer) \times (3 \times \text{MDIM} \times \text{dr\_numProcs} \times \text{numBlocks} + \text{numBlocks})
= (36 \times \text{dr\_numProcs} + 4) \times \text{numBlocks}
bytes of block metadata. There are also some transient dynamic data, whose
size depends on the size of a block and the amount of data which overlap
with the two data grids. Essentially, the code must allocate space for the
entire \text{pfft\_data\_grid} again, in order to buffer MPI communications.

The UG code only allocates space for the data which overlap with the two
 grids. Again, this amounts to allocating space for the entire \text{pfft\_data\_grid}, for MPI buffering.

ARGUMENTS

var - Gives the variable which will be extracted from the unk array and
      placed on the PFFT data grid.

EXAMPLE

Given in \text{Grid\_unitTest}:
call Grid\_mapFLASHtoPFFTgrid(DENS\_VAR)

NOTES

Requires that \text{Grid\_pfftMapInit} be called beforehand.

SEE ALSO

\text{Grid\_pfftMapInit}
\text{Grid\_mapPFFTtoFLASHgrid}
\text{Grid\_pfftMapFinalise}
\text{Grid\_unitTest}

COPYRIGHT

(c) 2007 by Bruce Duncan

***

subroutine Grid\_mapFLASHtoPFFTgrid(var)

use Grid\_pfftMapData
use Driver\_data, ONLY: dr\_myPE, dr\_numProcs
use Grid\_interface, ONLY: Grid\_getBlkData, Grid\_getLocalNumBlks,
    &
  Grid\_getListOfBlocks, Grid\_getBlkCornerID, Grid\_getBlkIndexLimits
implicit none

#include "Flash.h"
#include "Flash_mpi.h"
#include "constants.h"

! Define array indices for the coordinates of other processors on the flash and pfft grids
#ifdef FLASH_GRID_PARAMESH
#define PFFT 1
#else
#define FLASH 1
#define PFFT 2
#endif
#define MSIZE PFFT

integer, intent(in) :: var

#ifdef FLASH_GRID_PARAMESH
! Variables for iterating over the list of blocks
integer :: j, blockID, numBlocks
integer, dimension(:), allocatable :: listOfBlocks

! Variables to store the return value of Grid_getBlkCornerId
integer, dimension(MDIM) :: cornerID, stride

! And to store the return value of Grid_getBlkLimits
integer, dimension(2, MDIM) :: blkLimits, blkLimitsGC

! Variables to store how much data will be sent/received by each processor.
integer, dimension(0:dr_numProcs-1) :: commPieces, recvSizes

! Variables to store the metadata about blocks before they are communicated.
integer, dimension(:,::), allocatable :: cornerID_list, stride_list, &
blkLimits_list

! Array to store the cornerID, stride and blkLimits of each block
type(block_metadata), dimension(0:dr_numProcs-1) :: blkMetaData

! A buffer to store 2D integer data being sent. This is necessary because the
! data in a useful shape cannot be used by MPI since it is discontiguous in
! memory.
integer, dimension(:,::), allocatable :: sendDataBlock
#endif

! General processor loop counter and MPI error code (always ignored)
integer :: i, ierr
! Temporary buffer to hold data for MPI calls. Could perhaps be → removed with
! the use of MPI vector types.
real, dimension(:,,:,:), allocatable :: datablock

! Arrays to store the amount of data to be communicated and the ← offset into
! the pfft_data_grid array for the current chunk.
integer, dimension(MDIM) :: commSize, pfft_data_grid_origin
#ifdef FLASH_GRID_PARAMESH
! An array of pointers to a list of blocks
  type(block_pointer), dimension(0:dr_numProcs-1) :: recvGrid
#else
! An array of pointers to the MPI buffer for each processor.
  type(grid_pointer), dimension(0:dr_numProcs-1) :: recvGrid
#endif

#ifdef FLASH_GRID_PARAMESH
  call Grid_getLocalNumBlks(numBlocks)
  allocate(listOfBlocks(numBlocks))
  call Grid_getListOfBlocks(LEAF, listOfBlocks, numBlocks)
  commPieces = 0
  allocate(cornerID_list(MDIM, 0:dr_numProcs-1, numBlocks))
  allocate(stride_list(MDIM, 0:dr_numProcs-1, numBlocks))
  allocate(blkLimits_list(MDIM, 0:dr_numProcs-1, numBlocks))
  cornerID_list = 0
  stride_list = 0
  blkLimits_list = 0
#endif

! For each block, find the overlap of this block with the PFFT ←
grid on each other processor
do i = 0, dr_numProcs-1
! TODO j could be replaced with commPieces(i)
j = 0
  do blockID = 1, numBlocks
    call Grid_getBlkCornerID(listOfBlocks(blockId), cornerID, →
                          stride)
    call Grid_getBlkIndexLimits(listOfBlocks(blockID), blkLimits←
                         , blkLimitsGC)
    blkLimits(HIGH,:) = blkLimits(HIGH,:) - blkLimits(LOW,:)
    commSize = min(cornerID + stride*blkLimits(HIGH,:), →
                   gridLimits(i, PFFT, HIGH, :) ) - &
                   max(cornerID, gridLimits(i, PFFT, LOW, :) ) + 1
    if (all(commSize > 0)) then
      ! There will be data to send. Note this fact, and store ←
      ! important
      ! metadata.
      commPieces(i) = commPieces(i) + 1
      j = j + 1
      cornerID_list(:,i,j) = cornerID
      stride_list(:,i,j) = stride
      blkLimits_list(:,i,j) = commSize
  end if
! Notify each processor of how much it can expect to receive.
call MPI_Alltoall(commPieces, 1, FLASH_INTEGER, recvSizes, 1, ←
FLASH_INTEGER, MPI_COMM_WORLD, ierr)

! TODO would this be more efficient as an all-to-all? Do we have enough
! information about the size of the data to be communicated? The problem is
! that commPieces is different on each processor.

do i = 0, dr_numProcs - 1
allocate(blkMetaData(i)%cornerID(MDIM, recvSizes(i)))
alocate(blkMetaData(i)%stride(MDIM, recvSizes(i)))
allocate(blkMetaData(i)%blkLimits(MDIM, recvSizes(i)))
! Recv the cornerID, stride and blkLimits of each block we will recv from each
! processor. There should be 9*recvSizes(i) of these
call MPI_Irecv(blkMetaData(i)%cornerID, size(blkMetaData(i)%←
cornerID), &
FLASH_INTEGER, i, 0, MPI_COMM_WORLD, blkMetaData(i)%← request_cornerID, ierr)
call MPI_Irecv(blkMetaData(i)%stride, size(blkMetaData(i)%←
stride), &
FLASH_INTEGER, i, 0, MPI_COMM_WORLD, blkMetaData(i)%← request_stride, ierr)
call MPI_Irecv(blkMetaData(i)%blkLimits, size(blkMetaData(i)%←
blkLimits), &
FLASH_INTEGER, i, 0, MPI_COMM_WORLD, blkMetaData(i)%← request_blkLimits, ierr)
end do

do i = 0, dr_numProcs - 1
! Pack the cornerID, stride and blkLimits into contiguous ←
! arrays and send ! them to the relevant processor.
allocate(sendDataBlock(MDIM, commPieces(i)))
sendDataBlock = cornerID_list(:, i, 1:commPieces(i))
call MPI_Ssend(sendDataBlock, size(sendDataBlock), &
FLASH_INTEGER, i, 0, MPI_COMM_WORLD, ierr)
sendDataBlock = stride_list(:, i, 1:commPieces(i))
call MPI_Ssend(sendDataBlock, size(sendDataBlock), &
FLASH_INTEGER, i, 0, MPI_COMM_WORLD, ierr)
sendDataBlock = blkLimits_list(:, i, 1:commPieces(i))
call MPI_Ssend(sendDataBlock, size(sendDataBlock), &
FLASH_INTEGER, i, 0, MPI_COMM_WORLD, ierr)
deallocate(sendDataBlock)
end do

! Wait for all the receive operations to complete.
call MPI_Waitall(dr_numProcs, blkMetaData%request_cornerID, ←
MPI_STATUSES_IGNORE, ierr)
call MPI_Waitall(dr_numProcs, blkMetaData%request_stride, ←
MPI_STATUSES_IGNORE, ierr)
call MPI_Waitall(dr_numProcs, blkMetaData%request_blkLimits, ←
MPI_STATUSES_IGNORE, ierr)

! Next, receive the parts of the grid that we need.
do i = 0, dr_numProcs-1
  allocate(recvGrid(i)%block(recvSizes(i)))
  recvGrid(i)%block%request = MPI_REQUEST_NULL
  do j = 1, recvSizes(i)
    allocate(recvGrid(i)%block(j)%grid(blkMetaData(i)%blkLimits(←
      IAXIS, j), &
      blkMetaData(i)%blkLimits(←
      JAXIS, j), &
      blkMetaData(i)%blkLimits(←
      KAXIS, j)))
    call MPI_Irecv(recvGrid(i)%block(j)%grid,
      size(recvGrid(i)%→
      block(j)%grid), &
      FLASH_REAL, i, 0, MPI_COMM_WORLD, recvGrid(i)%block(j)%←
      request, ierr)
  end do
end do
end do

! Now send the corresponding data.
do i = 0, dr_numProcs-1
  j = 1
  do blockId = 1, numBlocks
    ! Find the overlap of this block with the PFFT grid on each ←
    ! other processor
    call Grid_getBlkCornerID(listOfBlocks(blockId), cornerID, ←
      stride)
    call Grid_getBlkIndexLimits(listOfBlocks(blockId), blkLimits←
      , blkLimitsGC)
    blkLimits(HIGH,:) = blkLimits(HIGH,:) - blkLimits(LOW,:)
    commSize = min(cornerID + stride*blkLimits(HIGH,:), ←
      gridLimits(i,PFFT, HIGH, :) - &
      max(cornerID, gridLimits(i,PFFT, LOW, :) + 1
    if (all(commSize > 0)) then
      allocate(datablock(commSize(IAXIS), commSize(JAXIS), ←
        commSize(KAXIS)))
      call Grid_getBlkData(listOfBlocks(blockId), CENTER, var, ←
        INTERIOR, &
        max(cornerID, gridLimits(i,PFFT, LOW, :) - cornerID + 1←
        , datablock, commSize)
      ! TODO restrict data before sending. Example code.
      ! do j = 1, ilog2(stride) - ilog2(target_stride)
      !   call restrict(datablock, listOfBlocks(blockId))
      ! end do
      call MPI_Ssend(datablock, size(datablock), FLASH_REAL, i, ←
        0, &
        MPI_COMM_WORLD, ierr)
      deallocate(datablock)
    end if
  end do
end do
end do

! Wait for the receives to complete (recvGrid%block%request doesn’t work)
do i = 0, dr_numProcs - 1
call MPI_Waitall(recvSizes(i), recvGrid(i)%block%request, MPI_STATUSES_IGNORE, ierr)
end do

! Place all the data at the appropriate point on the grid, calling the prolong routines where necessary
do i = 0, dr_numProcs - 1
  do j = 1, recvSizes(i)
    ! TODO prolong the data, where required.
    !call prolong(recvGrid(i)%block(j))
cornerID = blkMetaData(i)%cornerID(:,j)
commSize = blkMetaData(i)%blkLimits(:,j)
pfft_data_grid_origin = max(cornerID, gridLimits(dr_myPE, PFFT, LOW, :) - &
  gridLimits(dr_myPE, PFFT, LOW, :) + 1
pfft_data_grid(pfft_data_grid_origin(IAXIS):pfft_data_grid_origin(IAXIS)+commSize(IAXIS)-1, &
pfft_data_grid_origin(JAXIS):pfft_data_grid_origin(JAXIS)+commSize(JAXIS)-1, &
pfft_data_grid_origin(KAXIS):pfft_data_grid_origin(KAXIS)+commSize(KAXIS)-1) &
  = recvGrid(i)%block(j)%grid
deallocate(recvGrid(i)%block(j)%grid)
  end do
deallocate(recvGrid(i)%block)
deallocate(blkMetaData(i)%cornerID)
deallocate(blkMetaData(i)%stride)
deallocate(blkMetaData(i)%blkLimits)
end do

! Clean up
deallocate(cornerID_list)
deallocate(stride_list)
deallocate(blkLimits_list)
deallocate(listOfBlocks)

#else recvGrid%request = MPI_REQUEST_NULL
do i = 0, dr_numProcs - 1
  ! The amount of data we receive from each processor is the product in each dimension
  ! of the fraction of it’s FLASH grid which corresponds to our PFFT grid
  ! The size of the overlap (intersection) in each dimension is the difference of the
! minimum of the lowest indices and the maximum of the highest ← indices
commSize = min(gridLimits(i, FLASH, HIGH, :), gridLimits(dr_myPE, PFFT, HIGH, :) – &
    max(gridLimits(i, FLASH, LOW, :), gridLimits(dr_myPE, PFFT, LOW, :)) + 1
if (all(commSize > 0)) then !This processor has some data we ← want
    ! Allocate enough space for the receive. This will be freed ← after the data
    ! has been placed on the PFFT data grid. This allocation ← could be avoided
    ! by using an MPI vector type, but it would be useless for ← the AMR
    ! allocation, since the size could be arbitrary.
    allocate(recvGrid(i)%grid(commSize(IAXIS), commSize(JAXIS), ←
        commSize(KAXIS)))
    ! Post a receive for the data, will be completed by the ←
    MPI_Waitall call.
    call MPI_Irecv(recvGrid(i)%grid, size(recvGrid(i)%grid), ←
        FLASH_REAL, &
        i, 0, MPI_COMM_WORLD, recvGrid(i)%request, ierr)
end if
end do

! Wait on the recv requests.
call MPI_Waitall(dr_numProcs, recvGrid%request, ←
    MPI_STATUSES_IGNORE, ierr)

! Now go through each processor again and save the data to the ←
    pfft_data_grid
do i = 0, dr_numProcs-1
commSize = min(gridLimits(i, FLASH, HIGH, :), gridLimits(dr_myPE, PFFT, HIGH, :)) - &
  max(gridLimits(i, FLASH, LOW, :), gridLimits(dr_myPE, PFFT, LOW, :)) + 1
if (all(commSize > 0)) then !This processor has some data we just received
  pfft_data_grid_origin = max(gridLimits(i, FLASH, LOW, :), gridLimits(dr_myPE, PFFT, LOW, :)) - &
    gridLimits(dr_myPE, PFFT, LOW, :)) + 1
  pfft_data_grid(pfft_data_grid_origin(IAXIS):pfft_data_grid_origin(IAXIS)+commSize(IAXIS)-1, &
    pfft_data_grid_origin(JAXIS):pfft_data_grid_origin(JAXIS)+commSize(JAXIS)-1, &
    pfft_data_grid_origin(KAXIS):pfft_data_grid_origin(KAXIS)+commSize(KAXIS)-1) = recvGrid(i)%grid
  deallocate(recvGrid(i)%grid)
end if
end do
endif !FLASH_GRID_PARAMESH

! This function is only required for the restriction operation, which has not yet been implemented.
#if 0 contains
integer function ilog2(n_in)
! Function to calculate the log (base 2) of an integer quickly.
! Returns the integer which is the exponent of 2 which is not greater than n
! Returns -1 when asked to take the logarithm of 0. Throws away the imaginary part when asked to take the logarithm of a negative number.
implicit none
integer, intent(in) :: n_in
integer :: n
n = n_in
ilog2 = -1
do i = 1, bit_size(n)
  if (n == 0) then
    exit
  end if
  n = n / 2
  ilog2 = ilog2 + 1
end do
end function
#endif
end subroutine
Bibliography

http://www.nist.gov/dads/HTML/theta.html

[2] ASC Center for Astrophysical Thermonuclear Flashes,
http://flash.uchicago.edu/

http://www.mpi-forum.org/

[4] AMR,

[5] The Virgo Consortium,
http://www.virgo.dur.ac.uk/

http://flash.uchicago.edu/website/codesupport/flash3_ug_beta/


[9] PFFT : A Parallel Portable Library for Computing FFT’s,
http://flash.uchicago.edu/~dubey/pfft/

[10] FFTPACK,
http://www.netlib.org/fftpack/

[11] FFTW Home Page,
http://www.fftw.org/


http://www.astro.uiuc.edu/~pmricker/research/codes/flash/test_gallery/sod.html

53
[14] IBM XLF compiler manual page

[15] HDF5 Home Page,
http://hdf.ncsa.uiuc.edu/HDF5

[16] Serial FLASH Output Comparison Utility (sfocu),
http://flash.uchicago.edu/website/codesupport/flash3_
ug_beta/node32.html

[17] IDL, The Data Visualization & Analysis Platform,
http://ittvis.com/idl

[18] HPCx Service,
http://www.hpcx.ac.uk/

[19] Subversion,
http://subversion.tigris.org/