Annales UMCS Informatica AI 2 (2004) 291-299

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

FLASH – a tool for surgery of solar plasma

Małgorzata Selwa*, Krzysztof Murawski

Institute of Physics, Maria Curie-Skłodowska University,
Radziszewskiego 10, 20-031 Lublin, Poland

Abstract

The universe has always been object of our fascination. Because the Sun is our nearest star naturally a long time ago it became an object of our extensive observations. Nowadays, investigations of the Sun are carried by means of observations and theoretical modeling. Unfortunately, due to their intrinsic complexity most of theoretical models cannot be solved analytically. As a consequence, we adopt FLASH which is a modular parallel computation, adaptive mesh refinement, hierarchical data format code capable of solving magnetohydrodynamic equations. In this paper the FLASH code is applied to simulate oscillations in coronal loops.

1. Introduction

The FLASH code was originally written in 2000 by Fryxel et al. [1] and then it was improved by the members of the ASCI Center for Astrophysical Thermonuclear Flashes which was founded in 1997 [2]. Due to the contract with the United States Department of Energy it became a part of its Accelerated Strategic Computing Initiative (ASCI). The main purpose of its development is to handle magnetized flow problems found in many astrophysical environments such as thermonuclear flashes on the surfaces of compact stars like neutron stars and white dwarfs.

As the FLASH code is not widely known, the aim of this paper is to present this code to get accustomed quickly with its basic structure and provide its performance for waves in a solar coronal loop.

This paper is organized as follows. The following part contains a brief user guide. Sect. 3 provides an application of the code. This paper is concluded by a short summary.

* Corresponding author: *e-mail address*: mselwa@kft.umcs.lublin.pl

2. A brief user guide

The FLASH code can be run on various Unix-based platforms, e.g. SGI systems (IRIX), Intel – and Alpha-based systems, including clusters (Linux), Cray T3E (UNICOS), ASCI Nirvana machine (built by SGI), IBM SP2 systems, IBM SP4 systems, Sun E10K Starfire Clusters, Compaq Unix Clusters (TRU64) and ASCI Red machine (built by Intel). These platforms are distinguished in the *Makefile.h* file that specifies a platform, hostname, paths and compilation flags.

To run the FLASH code on a platform it is necessary to install Fortran 90 and C compilers. Then, we need to configure an implementation of MPI, e.g. freely available MPICH [3] and HDF libraries which are used for writing the output data [4]. A package like IDL [5] can be a good choice for data visualization purposes. Moreover, the FLASH code requires Python language for the setup script and GNU make utility.

2.1. Installation

We describe the installation of the FLASH code on platforms running Linux, particularly Red Hat 7.* and Slackware 8.* with gcc version 2.9*. We need a Fortran 90 compiler (e.g. a free release of Intel 6.0 [6]) which we unpack and install to e.g. */opt/intel/compiler60/ia32*. We ship over HDF4 library (e.g. HDF4.1r5), configure it with options *CC=/opt/intel/compiler60/ia32/bin/icc FC=/opt/intel/compiler60/ia32/bin/ifc ./configure --prefix=/usr/local* and install (by executing *$make*, *$make test* and *$make install*). We can download and install HDF5 (e.g. hdf5-1.4.4) in a similar way as HDF4 library and compile with options *CC=gcc FC=/opt/intel/compiler60/ia32/bin/ifc ./configure --prefix=/usr/local*. Subsequently we refresh system libraries by adding the path */usr/local/lib* to the */etc/ld.so.conf* file and run *$ldconfig*.

The next component we install is a freely available version of MPI – MPICH-1.2.5 [3]. We install it with the configuration options *CC=gcc CXX=/opt/intel/compiler60/ia32/bin/icc FC=/opt/intel/compiler60/ia32/bin/ifc F90=/opt/intel/compiler60/ia32/bin/ifc F77=/opt/intel/compiler60/ia32/bin/ifc ./configure --prefix=/opt/mpich --with-device=ch_p4 –rsh=ssh*. If MPICH is installed e.g. in the */opt/mpich* directory we modify the paths in the *.bashrc* and *.bash_profile* files as *PATH=$PATH:$HOME/bin:/opt/mpich/bin export PATH* and then log into a system one more time or re-read *.bash** files by executing *$. .bashrc* or *$. .bash_profile*.

After downloading and licensing the FLASH code we adopt it to our platform by making a new directory in *FLASH2.3/source/sites*. This directory name contains hostname and domain name of the platform. Now we copy *Makefile.h* from *FLASH2.3/source/sites* and adjust it to our platform by modifying HDF, MPI, paths and compilers flags.

If we are going to visualize FLASH data by IDL we change HDF and IDL paths in *FLASH2.3/tools/fidlr2/Makefile.linux* and run *$make –f Makefile.linux*. Assuming that IDL 5.6 is installed in */usr/local/rsi* we modify the *.bashrc* and *.bash_profile* files by adding the following:

```
IDL_DIR=/usr/local/rsi/idl_5.6
IDL_PATH=/home/user/FLASH2.3/tools/fidlr2:$IDL_DIR:$IDL_DIR/lib
IDL_STARTUP=~/.idlrc
export IDL_DIR IDL_PATH IDL_STARTUP
XFLASH_DIR=/home/user/FLASH2.3/tools/fidlr2
export XFLASH_DIR
```

Here */home/user/FLASH.3* is the exemplary path to FLASH directory which is different for every user. The *.idlrc* file, which is located in the same directory as *.bashrc* and *.bash_profile*, contains user settings for IDL (e.g. white background that is not supported by default in FLASH routines).

For SGI running Irix or Compaq running TRU64 unix platforms we must install Python which is not delivered by default and use GNU make utility (gmake) instead of make. The *Makefile.h* file must be taken from another Irix or TRU64 directory.

2.2. Running FLASH

A new physical problem requires writing different setup files. There are three essential files that must be created: *Config* which specifies required modules and registers runtime parameters, *flash.par* sets the values of runtime parameters, *init_block.F90* initializes data in each block for the problem.

In order to run FLASH for the first time we create a directory for our problem in *FLASH2.3/setups*. This directory can be called *first_problem*. The name of the directory is arbitrary and is just used as an argument when calling setup script. We put there *Config*, *flash.par* and *init_block.F90*. Then, we execute the command *$./setup first_problem -2d -auto* from the main FLASH directory. The option *-1d*, *-2d* or *-3d* corresponds to the dimensionality of the problem. A useful setup option is *-maxblocks=nr_of_blocks*. It defines maximum number of blocks which will be computed on a single processor.

If we succeed with the above commands we execute *$make*. The *Flash2.3/object* directory is created. This directory should be cleaned before running each job by *$make clean*. If we do not succeed with *setup* or *make* we have to improve *Config* file and repeat these commands. If we make changes in *init_block.F90* we should re-run only *make* (without *setup*). There is no need to run *setup* or *make* after editing the *flash.par* file.

To store data associated with a given problem it is better to create a new directory, not necessarily in the main FLASH directory and copy there *flash.par* with the executable file *FLASH2.3/object/flash2*.

For running problems on a platform with a single processor the *mpirun* command is optional. We can run *flash2* instead. If we have an access to few processors – the command *$mpirun -np nr_of_processors flash2* is required. A job can be run as a background process with the use of the *screen* or *nohup* command. The output data can be found in the directory where the code was run.

After the first run of FLASH we notice such files as: *flash.log* indicates parameter settings, the run and build time, the build machine, it also echoes when the code was refined and output data were stored; *flash.dat* contains the integral quantities as a function of time, e.g. total mass, total energy, total momentum, etc.; *amr_log* contains messages from PARAMESH package. A run of the code results in two different groups of files: **hdf*chk** that are checkpoint files containing dumps of entire simulation at given intervals of time, suitable for restarting the simulation and **hdf*plt_cnt** that are plot files which store only required fluid quantities.

2.3. A structure of the code

The FLASH code is a collection of different modules which can be combined in various ways for specific applications. Each module can be divided into three components: configuration layer, interface layer and algorithm. Some modules contain sub-modules. The configuration layer contains the information about module dependencies, default sub-modules, runtime parameter definitions and library requirements. It is stored in a *Config* file in all module directories. The aim of the interface layer is to provide an access to the data. Wrapper functions communicate directly with FLASH *database* module to access the data. While algorithms are not permitted to query the *database* module directly, they may communicate with database by a formal list of arguments. We can distinguish two classes of interface functions – directly connected with quantities and typical AMR functions.

An abstract representation of the code is shown in Fig. 1. Each box represents a different FLASH module. Typically each component is connected with a different class of solvers. The directory tree of the FLASH code is similar to the abstract representation (Fig. 2). The *doc/* directory contains the user guide, the documents about frequently asked questions, an implementation of the FLASH code on specific platforms and the manual to the package PARAMESH [7]. In the *setups/* directory startup files for several problems are stored. Source codes are stored in *source/*. The *tools/* directory contains routines used for visualization of data. After running *make* we can find linked and executable files in *object/* directory.

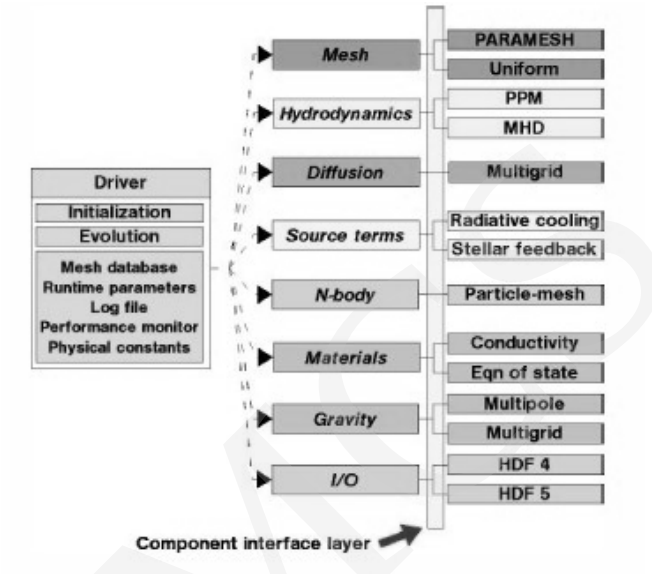


Fig. 1. Abstract representation of FLASH architecture[1]

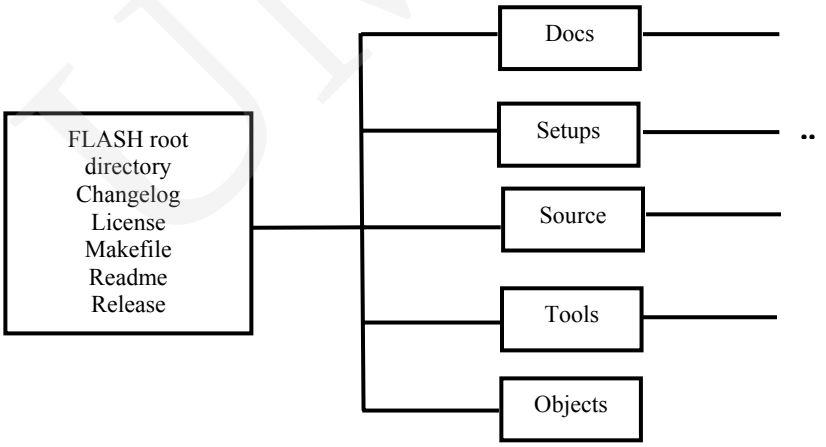


Fig. 2. The directory structure of the FLASH code

The main components we can distinguish in the FLASH code are:

– driver modules among which the default driver controls initialization or evolution and four modules implement different explicit time advancement algorithms: *euler1* – first-order Euler explicit scheme, *rk3* – third-order Runge-Kutta method and second-order accurate splitting method: *strang_state* – algorithm in the state-vector formulation and *strang_delta* – algorithm in the delta formulation,

[1] Thanks to the Center for Astrophysical Thermonuclear Flashes.

- I/O modules control how FLASH data are stored on the platform and the format of output data: *hdf4*, *hdf5 serial* or *hdf5 parallel*,
- a mesh module contains the package *PARAMESH* [7] of subroutines for the parallelization of the code and adaptive mesh refinement (*AMR*),
- hydrodynamic modules: the module which uses piecewise-parabolic method *PPM* [8]; the module based on *Kurganov* [8] schemes; the *MHD* module based on a finite-volume, cell-centered method with truncation-error scheme used to satisfy $\nabla \cdot \boldsymbol{B} = 0$ condition,
- material properties modules contain multiple fluids, different fluid compositions, equations of state: *gamma* implements a perfect gas equation of state: *multigamma* implements a perfect gas equation of state with multiple fluids each of its own adiabatic index; *helmholtz* uses free energy table interpolation and includes radiation pressure and stellar conductivity module: *thermal conductivity*, *viscosity*, *magnetic resistivity* and *magnetic viscosity*,
- source terms modules contain *nuclear burning* module, *stirring* module, *ionization*, *heating* and *cooling* modules,
- gravity modules with gravitational potential or acceleration source terms in the code. The modules: *constant*, *plane parallel* correspond to time constant field that is parallel to one of the coordinate axis; *ptmass* implements field due to a point mass at a fixed location; *Poisson* includes field produced in simulation,
- particle modules incorporate physical particles and Lagrangian mass tracers, *active* and *passive* particles,
- a cosmology module contains *redshift* terms in Euler equations and library of useful cosmological functions, e.g. converting redshifts to times,
- solvers used for ordinary differential equations O*DE*: *multipole Poisson* – for spherical or nearly-spherical mass distributions with isolated boundaries, and *multigrid Poisson* - for general source distributions,
- a visualization module: 2d IDL module with a special widget,
- a utility module consists of the collection of high-level functions simplifying programming in FLASH.

PARAMESH (Parallel Adaptive Mesh Refinement) is a packet of subroutines which handles mesh generation refinement/derefinement, distribution of work to processors, guard cell filling and flux conservation. PARAMESH uses AMR (Figs. 3, 4). PARAMESH works on Cartesian blocks. Each block contains $nxb \times nyb \times nzb$ interior cells and a set of guard cells used to compute boundary conditions (left panel Fig. 5). The refinement/derefinement is achieved by calculating the gradient of tested quantities. The flux is conserved at a jump of refinement. After the refinement/derefinement is complete, the blocks are

redistributed among the processors with the use of work-weighted Morton space-filling curve (right panel of Fig. 5).

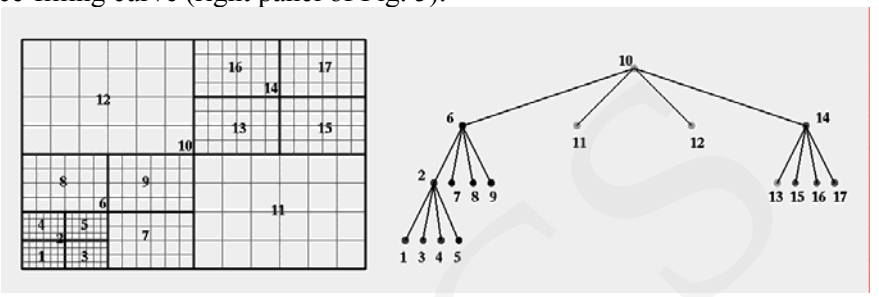


Fig. 3. An idea of grid refinement. The blocks which are surrounded by thick solid lines contain 6×4 numerical cells[2]

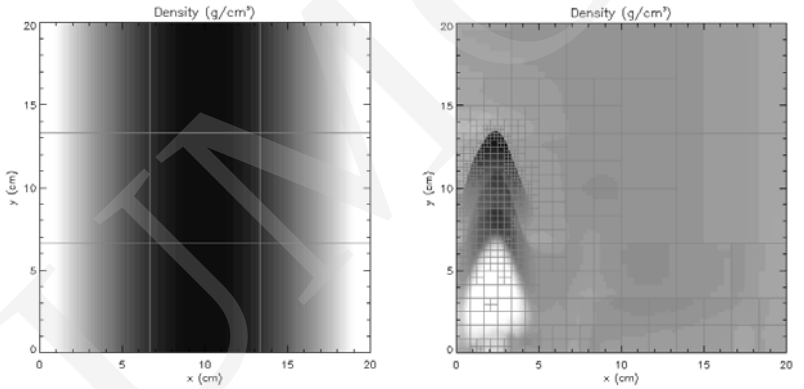
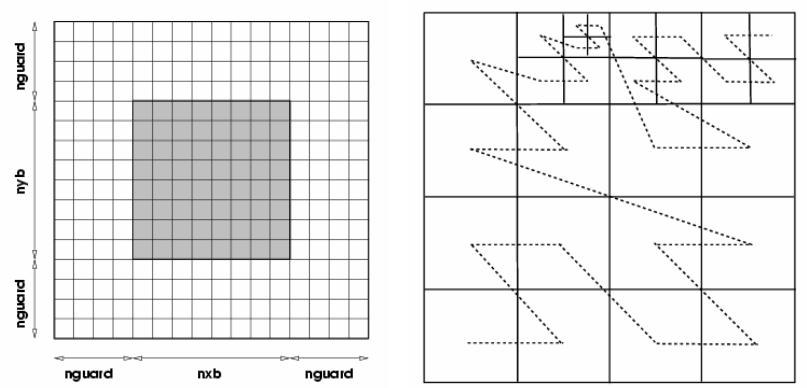


Fig. 4. An example of refinement and derefinement. The left panel shows the initially distributed blocks while the right panel displays the blocks at a later time. Note that the grid is refined at certain places



[2] Redrawn from the NASA Goddard Space Flight Center's Earth and Space Data Computing Division (ESDCD) [8].

Fig. 5. A two-dimensional AMR block with shaded interior 8×8zones and the perimeter of 4 guardcells (left panel). The Morton space-filling curve is displayed in the right panel[3]

The FLASH code uses the HDF library which is a physical file format for storing scientific data. This library and multi-object file format allows transferring of graphical and numerical data between different machines because it is independent of a platform.

The current 2.3 version of the code FLASH supports several types of grid geometry: one-, two- and three-dimensional Cartesian grids, two-dimensional cylindrical grids and one-dimensional spherical grids. No polar geometry is involved yet. The code allows us to use different boundary conditions: periodic or wrap-around, reflecting or non-penetrating, outflow or open, hydrostatic (supports fluid "above" against gravity) and user-defined.

3. An application of the FLASH code to MHD waves in a coronal loop

We show an application of the code to solar plasma which is described by the ideal MHD equations:

$$\begin{aligned}
&\frac{\partial \rho}{\partial t} + \nabla \cdot \left(\rho \mathbf{V}\right) = 0, \\
&\frac{\partial \left(\rho \mathbf{V}\right)}{\partial t} + \nabla \cdot \left[\left(\rho \mathbf{V}\right)\mathbf{V}\right] = -\nabla p + \frac{1}{\mu}\left(\nabla \times \mathbf{B}\right) \times \mathbf{B}, \\
&\frac{\partial p}{\partial t} + \nabla \cdot \left(p \mathbf{V}\right) = -p\left(\gamma - 1\right)\nabla \cdot \mathbf{V}, \\
&\frac{\partial \mathbf{B}}{\partial t} = \nabla \times \left(\mathbf{V} \times \mathbf{B}\right), \nabla \cdot \mathbf{B} = 0,
\end{aligned} \tag{1}$$

where ρ denotes the mass density, $\boldsymbol{V}$ is the velocity of plasma, p is the gas pressure, $\boldsymbol{B}$ is the magnetic field and $\gamma = 5/3$ is the adiabatic index. A coronal loop is modeled by mass density enhancement (Fig. 7):

$$\rho_0(r) = 1 + \frac{2.89}{\cosh^4(r)}, \tag{2}$$

where r is the radial coordinate. The equilibrium magnetic field has only z component:

$$B_z(r) = 1 + \frac{1.97}{\cosh^4(r)}. \tag{3}$$

The total pressure at the equilibrium must be constant. The perturbation was set in x component of velocity:

$$V_x(r,\varphi,z,t=0) = \frac{0.005}{\cosh^2\left(x-10\right)\cosh^2\left(y\right)\cosh^2\left(z\right)}, \quad V_y = V_z = 0. \tag{4}$$

[3] From [1]

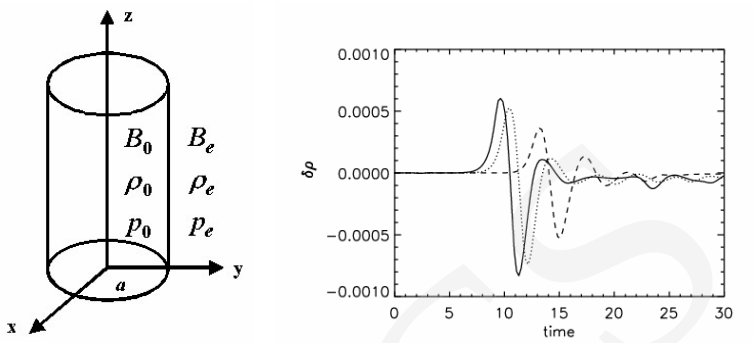


Fig. 7. A simple model of the geometry of a loop and time-signatures which are obtained by measuring the mass density at different detection points: x=0, y=0, z=0 (solid line), x=0, y=0, z=4 (dotted line), x=0, y=0, z=9 (dashed line)

The equations (1) were solved numerically on the $(-10,10)\times(-10,10)\times(-10,10)$ Eulerian box with the open boundary conditions at all boundaries of the simulation region.

The pulse of Eq. [4] excites a packet of waves in which the highest contribution has a fast kink wave that involves displacements with the axis of the loop, resembling a wriggling snake. As it is shown in Fig. 7, the pulse produces complex time-signatures.

4. Summary

This paper contains an introduction into the FLASH code. We explain its structure and describe briefly the main libraries. An example of applicability of the code to solar coronal loop simulation is provided. This paper can be a useful guide for potential users.

This work was financially supported by the grant from the State Committee for Scientific Research Republic of Poland, KBN grant no. 2 PO3D 016 25. The software used in this work was in part developed by the DOE-supported ASCI/Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

References

[1] Fryxell B., et al., ApJS, 131 (2000) 273.
[2] http:// flash.uchicago.edu/flashcode/
[3] http://www-unix.mcs.anl.gov/mpi/
[4] http://hdf.ncsa.uiuc.edu/
[5] http://www.rsinc.com/
[6] http://www.intel.com/software/products/compilers/
[7] http://sdcd.gsfc.nasa.gov/ESS/annual.reports/ess98/paramesh.html/
[8] Murawski K., Analytical and numerical methods for wave propagation in fluids, World Scientific, Singapore, (2002).