Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

I/O Optimization Strategies in the PLUTO Code

A. Mignone[a,*], G. Muscianisi[b], M. Rivi[b], G. Bodo[c]

[a]Dipartimento di Fisica Generale, Universitá di Torino, via Pietro Giuria 1, 10125 Torino, Italy
[b]Consorzio Interuniversitario CINECA, via Magnanelli, 6/3, 40033 Casalecchio di Reno (Bologna), Italy
[c]INAF, Osservatorio Astronomico di Torino, Strada Osservatorio 20, Pino Torinese, Italy

Abstract

PLUTO is a modular and multi-purpose numerical code for astrophysical fluid dynamics targeting highly supersonic and magnetized flows. As astrophysical applications are becoming increasingly demanding in terms of grid resolution and I/O, efforts have been spent to overcome the main bottlenecks of the code mainly related to an obsolete and no longer maintained library providing parallel functionality. Successful achievements have been pursued in The Partnership for Advanced Computing in Europe First Implementation Phase Project (PRACE-1IP) and are described in the present white-paper.

Project ID: PRPC04

1. Introduction

PLUTO is a Godunov-type modular code for the solution of hyperbolic/parabolic systems of conservation laws, providing both finite volume and finite difference techniques, see [1] and [2] for a comprehensive description. The code is well suited for supersonic and super-fast magneto-sonic flows in multiple spatial dimensions and provides a modular structure whereby different integration schemes can be combined together to treat diverse physical regimes including classical or relativistic magnetohydrodynamics (MHD), ideal/dissipative effects, Cartesian or curvilinear geometries, heating/cooling processes, body forces and so forth. The code is developed at the University of Torino in a joint effort with the Astronomical Observatory of Torino and it is mainly used by the astrophysical community for state-of-the-art numerical simulations of plasma in the MHD approximation limit. PLUTO is freely distributed at `http://plutocode.ph.unito.it`.

Written in the C programming language, PLUTO is built upon a systematic approach commonly employed by high-resolution shock-capturing (HRSC) schemes [4]. Most of the HRSC methodology is based on a quite general sequence of steps whereby volume averages are first reconstructed inside each computational cell using piece-wise monotonic interpolants, a Riemann problem is then solved at each interface with discontinuous left and right states, and the solution is finally evolved to the next time level in a conservative, time explicit fashion. PLUTO can run on either single processor machines or distributed parallel systems. Parallelization is achieved by domain decomposition, i.e. the global computational box is divided into sub-domains and each of them is assigned to a processor. For this purpose makes extensive use of ArrayLib [3], a library that supports parallel finite difference computations on block structured meshes, based on the Message Passing Interface (MPI), originally developed by A. Malagoli at the University of Chicago. ArrayLib aims at providing an abstraction for distributed array objects, and simple interfaces to the underlying MPI routines. The parallelization model adopted in ArrayLib is the usual one of distributed arrays augmented with guard cells (ghost points) to deal with boundary conditions. In particular, it supports cell-centered meshes providing basic functionality to define distributed arrays, update the guard cells on each processor and provide conversion routine between local and global addressing of the arrays.

The main bottlenecks of PLUTO were related to parts handled by ArrayLib, which is no longer maintained since 2001. In fact, this library suffered from a number of flaws and implementation bugs which could severely limit the code performance on Petascale systems and make future additional extensions rather difficult to implement. Moreover, the standard procedure for raw binary I/O operations was implemented through collective and blocking I/O calls where every processor accessed independently the same file. In configurations with very large number of processing units and grid sizes this approach has been found, on some system, to lead to execution hangs and/or considerable slow down and efficiency loss.

*Corresponding author.
tel. +0-000-000-0000 fax. +0-000-000-0000 e-mail. mignone@ph.unito.it

At present, PLUTO is widely used by a large number of institutions worldwide for different astrophysical applications, e.g., stellar/extragalactic jets, shock wave dynamics, magnetized turbulence, accretion flows, stellar winds and so forth. Among these, the problem of angular momentum transport in accretion disks is certainly one of the most challenging application and can be tackled only by high-resolution numerical simulations of global magnetized disks requiring intensive petascale HPC resources.

In this perspective, we have successfully improved several aspects in the parallelization strategy as well as in the I/O performance, by a number of actions performed on both the ArrayLib and the PLUTO code which are presented in Section 2. In particular:

1. ArrayLib has been largely debugged, upgraded and simplified resulting in a more compact set of routines. The major achievement concerns the correct implementation of the distributed array descriptor handling staggered mesh arrays;
2. modification of writing of raw binary data in both single and double precision by using an asynchronous and split collective approach, available in the MPI-2 I/O standard;
3. implementation of the HDF5 file format (previously available only in the adaptive grid version of PLUTO) in the static grid version of the code.

Test and benchmark results on JUGENE Tier-0 system are presented in Section 3. General comments and conclusions are written in Section 4.

2. PLUTO optimizations

In its original implementation, PLUTO starts the execution by performing a number of initialization operations that include, among others, parallel domain decomposition, memory allocation and assignment of initial conditions. The main integration loop is then commenced and it is comprised of the following steps:

- first, the main variables are written on a single file or multiple files by using 'blocking' and 'synchronous' MPI calls at fixed time steps;
- then the actual integration is performed and the time step is updated;
- a number of collective MPI operations useful for diagnostic purposes and not involving the main dataset is performed. These operations are based on extensive usage of the '`MPI_Allreduce`' function in order to retrieve relevant quantities such as the maximum flow velocity, the maximum number of iterations encountered or the minimum time scales for different physical processes.

At the end of the integration loop, the main variables in the dataset are dumped to disk and the MPI tasks are finalized.

Our optimizations were mainly focused on the I/O operations (see Sections 2.1. and 2.2.), without changing the structure of the code but postponing the writing of the binary files after the integration step within the main loop. Furthermore, it was also fixed a bug in handling staggered arrays in the ArrayLib. In particular, the conversion routines between local and global addressing of the arrays was analyzed and conveniently modified.

2.1. Raw Binary I/O

As mentioned before, PLUTO performed binary I/O operations at specific times during which each processor gained independent access to the file and wrote each variable through blocking and collective calls from within an iteration loop. This step was followed by a number of collective MPI communications not involving the main integration dataset. By aiming at improving the performance of reading/writing raw binary data in both single and double precision, the ArrayLib has been modified by conveniently replacing the previous I/O calls with 'non-blocking' and 'split collective' calls, available in the MPI-2 I/O standard. We remember that a blocking I/O call will not return until the I/O request is completed, while a non-blocking I/O call initiates an I/O operation, but does not wait for it completion. Given suitable hardware, this allows the transfer of data out/in the user's buffer to proceed concurrently with computation. A separate request complete call is needed to complete the I/O request, i.e., to confirm that the data has been read or written and that it is safe for the user to reuse the buffer. This condition forced us to move the writing of the file at the end of the iteration.

As a result, variables are now dumped to disk all together by setting a unique view of the whole file and by building a global sub-array describing how the data of each process has to be written in the file. Between the begin and the end of the I/O operations, the collective MPI operations (for diagnostic purposes) are performed in order to overlap computation with the I/O operations.

In the following there is a sketch of the integration loop, in which the I/O operations are performed by using 'non-blocking' and 'split collective' calls.

```
for t = 1,..., N

    integration time step t

    if (binary asynchronous I/O has to be performed)
        definition of the global sub-array for the view of the file
```

```
        call MPI_File_set_view
        definition of the global sub-array for the asynchronous write
        call MPI_File_write_all_begin
    else
        continue the loop
    end if

    MPI_Allreduce calls (diagnostic)

    if (binary asynchronous I/O has to be performed)
        call MPI_File_write_all_end
        update of the log file: dbl.out/flt.out
    end if

end for
```

2.2. HDF5 I/O

Following the parallelization strategy implemented in PLUTO, the usage of HDF5 library has been extended to the static grid version of the code. In implementing HDF5 output, we set two different 'property list', one for creating the file and one for accessing (in reading/writing) the dataset. Variables are sequentially written to the same file as different HDF5 datasets. For each variable, two dataspaces (using 'hyperslab' selections) are created: one specifies the shape of the data in each processor's memory and another provides the layout of the data in the file. Moreover, we added a group containing information about the computational grid, useful for visualization purposes.

Two HDF5 available file drivers have been tested: MPI-POSIX and MPI-I/O, the latter using both the 'independent' and the 'collective' access. The benchmarks have shown that on the JUGENE system the usage of the MPI-I/O file driver with a collective access to the dataset yields the best performance.

3. I/O benchmark results

The enabling process started with the porting of the PLUTO code on the JUGENE system. Subsequently, a detailed profiling of both the communication and the I/O parts handled by the ArrayLib has been done.

JUGENE is an IBM BlueGene/P system, hosted by the Gauss Centre for Supercomputing (GCS) at the Forschungszentrum Jülich (FZJ) in Germany, which has a massively parallel supercomputer architecture with different types of nodes and networks. In total JUGENE has 72 racks and contains 73.728 compute nodes or 294.912 cores. One rack contains 1024 compute nodes, or 4.096 cores, organized in 2 midplanes each containing 16 node cards.

In what follows, the test cases used and the benchmark results for both the raw binary and HDF5 I/O are presented.

3.1. Benchmark results for Raw Binary I/O

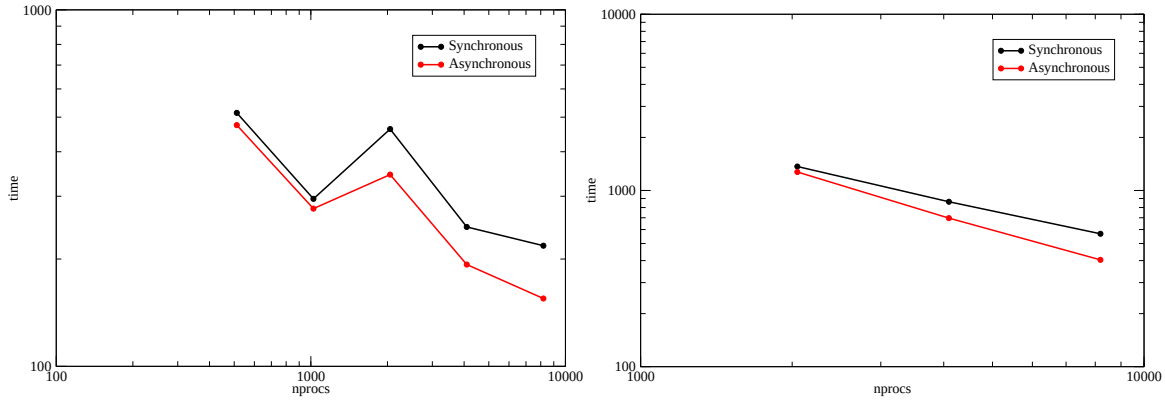

Fig. 1. Plot of wall clock time with grid sizes of $512 \times 1024 \times 512$ (on the left) and $512 \times 4096 \times 512$ (on the right).

The test case used for benchmarking raw binary I/O consists of a supersonic jet on a 3D Cartesian domain with size $14 \times 70 \times 14$, in units of the jet radius, with 6 variables in double precision written to the same file for a total of 20 files per run. We considered two grids of different sizes along the y-direction: the first is made of $512 \times 1024 \times 512$ points, while the second has $512 \times 4096 \times 512$ points corresponding to output sizes of 12GByte and 48GByte respectively. These benchmarks, involving more than 4096 MPI processes and intensive I/O (i.e. output files written at each step of the integration loop), have shown that the new non-blocking version of the

Table 1. Total running time with different resolutions, $512 \times 1024 \times 512$ (columns 2-4) and $512 \times 4096 \times 512$ (columns 5-7).

	$512 \times 1024 \times 512$			$512 \times 4096 \times 512$		
Nprocs	Synchronous time [sec]	Asynchronous time [sec]	Gain [%]	Synchronous time [sec]	Asynchronous time [sec]	Gain [%]
512	512	475	7.5	-	-	-
1024	295	277	6	-	-	-
2048	463	345	25	1368	1273	7
4096	246	193	21.5	863	697	19.2
8192	218	155	29	568	404	29

code is able to decrease the writing time with respect to the previous blocking version. The gain obtained, starting with 512 MPI tasks, increases with the number of MPI tasks involved up to 20% for 4096 tasks and 30% for 8192 tasks (see Table 1 and plots in Figure 1).

Notice, from the first plot in Figure 1, that at 2048 processors both the times of synchronous and asynchronous runs increase with respect to the times obtained for 1024 MPI tasks. This is due to the configuration of the JUGENE cluster, because 71 out of the 72 JUGENE racks have a fixed ratio of 1 I/O node per 128 compute nodes (i.e. 4 I/O nodes per midplane and a total of 8 I/O nodes), whereas there is deviant rack (named R87) which has a much richer ratio: 1 I/O node per 32 compute nodes. Only the two simulations involved 512 and 1024 MPI processors run on rack R87, by using respectively 8 and 16 I/O nodes. The other simulations, conversely, run on some of the other 71 'standard' racks, thus they used a smaller number of I/O nodes with respect to the number of cores involved in the computation. The racks used in the simulations are different, due to the settings of the LoadLeveler classes available on JUGENE.

The same test case has been used to perform weak scaling, in which each MPI task has a fixed grid sizes of $64 \times 128 \times 64$. The values in Table 2 show that a linear scaling is achieved, because the gain of the asynchronous version stays constant while the workload is increased in direct proportion to the number of processors.

Table 2. Weak scaling of binary I/O.

Nprocs	Synchronous time [sec]	Asynchronous time [sec]	Gain [%]	Total grid size
512	514	475	7	$512 \times 1024 \times 512$
2048	1368	1273	7	$512 \times 4096 \times 512$

3.2. Benchmark results for HDF5 I/O

The benchmarking configuration for HDF5 I/O consists of a vertically stratified accretion disk in 3D cylindrical coordinates (r, ϕ, z) with a domain extent given by $1 < r < 4$, $0 < \phi < 2\pi$, $-0.4 < z < 0.4$, covered with $480 \times 1920 \times 128$ zones. The ideal MHD equations are solved with an adiabatic equation of state using a third-order Runge-Kutta time stepping with piece-wise parabolic spatial reconstruction and staggered mesh constrained transport evolution of the magnetic field to ensure the divergence-free condition. User-defined boundary conditions are adopted in the vertical and radial direction while periodicity is assumed along the azimuthal direction. Since the numerical scheme requires 10 variables to be solved for and written to disk, each output file has (approximately) size of 8.8GByte (binary dataset) and 11GByte (HDF5 dataset).

Figure 2 shows the time needed to write a single file versus the number of MPI tasks obtained by averaging ≈ 30 files per simulation (binary files were written by using the blocking version of the code).

Notice that at 2048 processors the time needed to write the binary file increase with respect to the time obtained for 1024 MPI tasks, due to the configuration of the JUGENE cluster, as explained in the previous subsection.

The results indicate that HDF5 I/O performs worse than the synchronous binary I/O. This is probably due to the structure overhead of HDF5 and the additional information provided. The compatibility between the internal parameters of this format and the configuration of the underlying file system (GPFS) must also be considered.

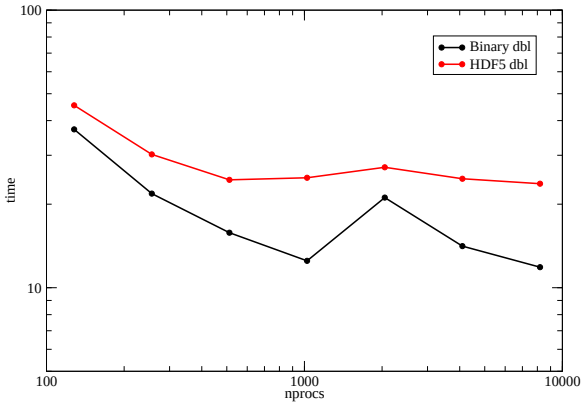

Fig. 2. Time needed to write a single file versus the number of MPI tasks obtained by averaging ≈ 30 files per simulation.

4. Conclusions

The granted preparatory access and the joint efforts with the PRACE experts has allowed to achieve high-quality results which have greatly expanded the code capabilities in terms of i) flexibility, ii) enhanced I/O features and performances and iii) portability.

ArrayLib is now able to handle both cell-centered and staggered array in a correct and efficient way. This is an important improvement to the code as it provides a more manageable environment in the treatment of complex boundary conditions involving magnetic field. The introduction of the HDF5 file format for static grid represents an improvement for PLUTO both in term of portability and also for post-processing and visualization purposes. Finally the implementation of asynchronous binary I/O allows a net performance improvement on very large systems like JUGENE.

These optimizations warrant a major release of the PLUTO code, from 3.1.1 to 4.0, which will be made available to the astrophysical community within the next 6 months.

The results and improved techniques achieved during this work give strong and encouraging indications that global disk simulations on Petascale computing systems should now be feasible with the PLUTO code, provided enough computational resources are allocated. This will open for potential scientific innovation in the field of accretion flows and angular momentum transport in disks through high-resolution numerical simulations.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work is achieved using the PRACE Research Infrastructure resources [insert here machine names and the corresponding sites and countries].

References

1. Mignone, A., Bodo, G., Massaglia, S. et al., Astrophys. J. Suppl. S. 170 (2007) 228.
2. Mignone, A., Zanni, C., Tzeferacos, P. et al., Astrophys. J. Suppl. S. 198 (2012) 7.
3. FLASH Report (1999), ASCI/Alliances Center for Astrophysical Thermonuclear Flashe, University of Chicago
4. Toro, E. F., Riemann Solvers and Numerical Methods for Fluid Dynamics, Springer, Berlin, 1997