User's Guide to Diablo

(Dated: February 11, 2015)

I. GETTING STARTED

This is the 2D parallelisation version of Diablo for channel flow geometries which provides a more efficient implementation of parallel communication and the possibility of using large number of cores (of the order fo 2048 for large problem sizes, see below). The data are not only splitted in vertical chunks, but also along the horizontal (along x or z depending whether we are in physical or spectral space). For this version, it is required to have working MPI and HDF5 libraries, beside the FFTW library generally required for DIABLO. We refer to the UserGuide of the classical version of diablo for further details on these libraries and here we focus on the main modification to the code source and how to compile it.

II. RUNNING THE CODE

The first thing to do is to clone a working version of the code on the local machine by using

```
ed441: anyFolder $  hg clone https://edeusebio85@bitbucket.org/edeusebio85/diablo_2d
```

This will create a folder with all the relevant files with a structure very similar to the original version of diablo. Some of the files do not actually change. The code will take the same input files, grid files and flow fields of the classical version, such that a simulation depending on the need can be run with either the 1d parallelisation or the 2d parallelisation.

However, the procedure to compile the code is slightly different. Assuming that we are in the `mpi` run folder where we have the files `input_chan.dat`, `input.dat`, `grid_def.all` and `grid.h5`. For carrying out the compilation, we advise to use the shell scripts present in the bin folder.Thus, the `PATH` and the `PATH_DIABLO` variables should be set accordingly, with

```
ed441: mpi $ export PATH_DIABLO=path/to/diablo/folder
```

and

```
ed441: mpi $ export PATH=$PATH_DIABLO/bin/:$PATH
```

At this point we can create the `grid_def` file used in the compilation, by

ed441: mpi $ genGridFiles.sh grid_def.all N_p N_y^P grid_def

where N_p is the total number of processes and N_y^P is the number of processes in the y direction. This will also create the file `grid_mpi` which is used in the compilation as well. For instance,

```
ed441: mpi $ genGridFiles.sh grid_def.all 16 2 grid_def
```

will create the files for compiling a code with 16 cores, of which 2 in the vertical and 8 in the horizontal. The code can then be compiled, similarly to the classical version, with

```
ed441: ser $ compDia.sh -Options=''HDF5=TRUE PARALLEL=TRUE'' -Clean=yes
```

and run with

```
ed441: ser $ mpirun -np 16 ./diablo
```

The code will only generate as many statistics files as the number of processes in the vertical direction N_y^P, which have the same structure as the ones generated by the 1d parallelization of the classical version. In a similar way, they can be merged with the python script.

III. MODIFICATION OF THE SOURCE

The general structure of the code is mostly unchanged. The main changes pertain x-z direct and inverse Fourier transforms (which requires communication among processes) and IO operations. However, one point that the user who would like to modify the code should always have in mind is the structure of the data repartition. Each global variable has a size which is

- `(NXP+1,NZ+2,NY+2)` in complex space
- `(NX+2,NZP+2,NY+2)` in physical space

where $NXP = NX/(2{*}N_p^h)$ and $NZP = NZ/(N_p^h)$ are the number of points per process in the horizontal direction. NY is the number of points per process in y, consistent with the 1d parallelisation of diablo. In the `grid_def` file, `NX` and `NZ` are the total number of points in x and z, whereas `NY` is once again the number of points per process in y. Each process has three different communicators active

1. `MPI_COMM_WORLD` Global communicator.
2. `MPI_COMM_Y` Communicator in the y direction.
3. `MPI_COMM_Z` Communicator in the horizontal direction (along x or z depending on whether we are in physical or spectral space).

The implementation of the communication between processes can be found in `dummy_code/mpi.alltoall${VER}.f`, where `VER` represent the version of the `ALLTOALL` communication used. There are several possible implementation for the global communication and depending on the machine and the actual MPI library, one or another may be more efficient. The choice of the version can be done in the compiling phase through the variable `ALL2ALL`,*e.g.*

```
ed441: for $ make HDF5=TRUE PARALLEL=TRUE ALL2ALL=2
```

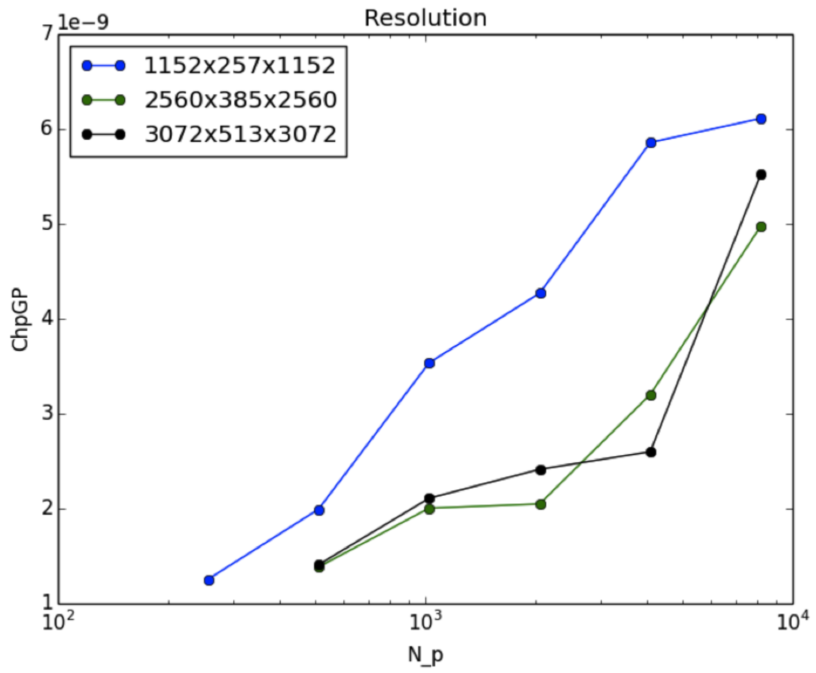

FIG. 1: Scaling test on Archer for different problem sizes. The y-axis given the computing time [in corehours] per grid point.

A. Benchmark

Here we report the results of some benchmarks performed on Archer and other PRACE resources. In figure 1 the total computing time [in corehours] per grid-point (ChpGP) for three different problem sizes (A: 300*106 Gridpoints; B: 2.5*109 GridPoints and C: 5*109 GridPoints) is shown. Runs were carried out on Archer in Nov-Dec 2014 using the Instant Access grant. As expected, increasing the problem size decreases the overall influence of communication and, as a result, the core hours per gridpoint (ChpGP) decreases. For the problem size representative of the simulations to be carried out within this grant (case B), the code scales reasonably well up to 2048 cores, with a drop of efficiency (with respect to the computing time at 256 cores, i.e. the minimum number of process allowed by RAM limitations) to 68% at 2048 cores and 43% at 4096. Such a drop of efficiency is not surprising because of the numerical scheme (pseudo-spectral method) involving a significant number of FFTs with global communications. In this respect, our code shows similar performances with respect to other fluid dynamics codes based on analogous numerical schemes. Figure 2 shows the scaling tests on Archer and on other HPC resources in Europe, where a black line representing a competitor code (the Simson code extensively used in HPC and which has been successfully granted Tier-0 projects on PRACE resources) is shown for comparison. When running on Archer, our code shows a drop of performances at around 1024 cores. This is however largely machine dependent. Further increase at 2048 cores does not significantly change the efficiency which stays around 68%. This is smaller but comparable to the efficiency of 78% achieved by Simson. Given these results, we believe that the parallelisation strategy and the scaling obtained are suitable to use the code on problem sizes of the order of few billions grid points over 2048 cores.

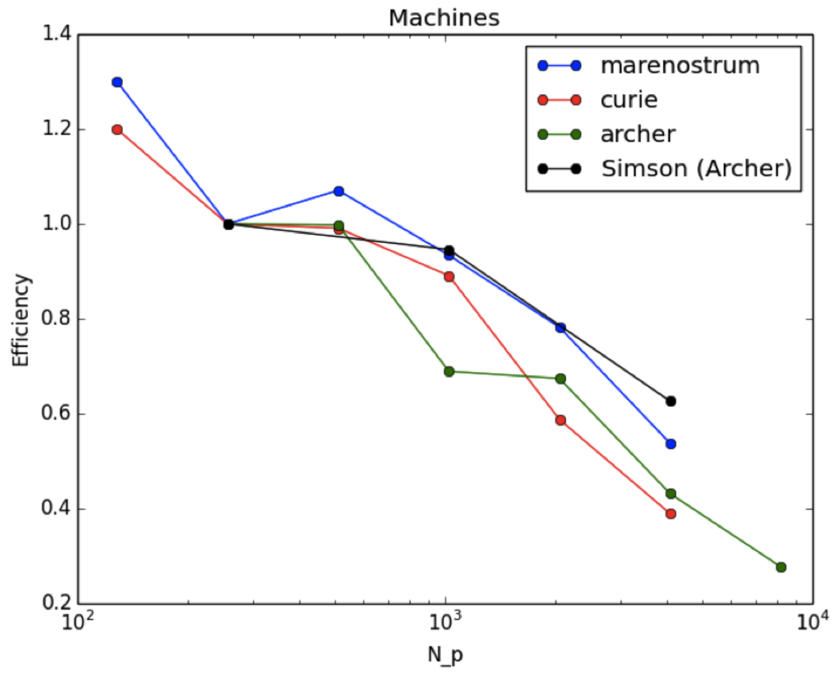

FIG. 2: Efficiency (with respect to the computing time for 256 cores) on different machines: MareNostrum, Curie and Archer. See page 3 for details on the architectures. The black line represents a similar code based on spectral methods which has been extensively used in PRACE projects.