Profiling Studies of FLASH StirTurb on the HPCC

Jasmin Shin

December 12, 2016

Abstract

FLASH is a fully parallel multi physics simulation code written in FORTRAN90 and C and using MPI and HDF5, a parallel I/O. FLASH comes with various test problems one of which is a hydrodynamics code called StirTurb which simulates a homogeneous, isotropic, and weakly compressible turbulence. Turbulence is driven in order to be sustained against its dissipative nature. Through a profiling study of StirTurb on the MSU HPCC, it was found that I/O routines became a significant percentage of the runtime with increasing problem size and processor count.

1 Introduction

FLASH is a fully parallel multi physics simulation code written in FORTRAN90 and C that comes with various test problems one of which is StirTurb. StirTurb is a hydrodynamics code which simulates a homogeneous, isotropic, and weakly compressible turbulence that must be driven in order to be sustained against its dissipative nature [1]. The focus of this study is to profile a FLASH test problem StirTurb on the HPCC in search for possible bottlenecks and inefficiencies.

HPCToolkit is an application analysis program that works by taking statistical data of performance counters, doing a binary analysis of code to obtain information about code structure, and then finally connecting the sampling measurements to the structure elements such as loops and subroutines. HPCToolkit is ideal for analyzing the performance the StirTurb test problem because it is a toolkit designed to analyze parallelized codes with support for MPI and Open-MP on anywhere from multicore computes to large supercomputers [4].

The AMR grid used for this simulation is called PARAMESH4.0. Paramesh is a collection of Fortran90 subroutines that allow the application developer to easily implement parallelization and AMR capabilities to their serial UG code. It was built in accordance with the trends of large scale simulations being run on distribute memory systems.

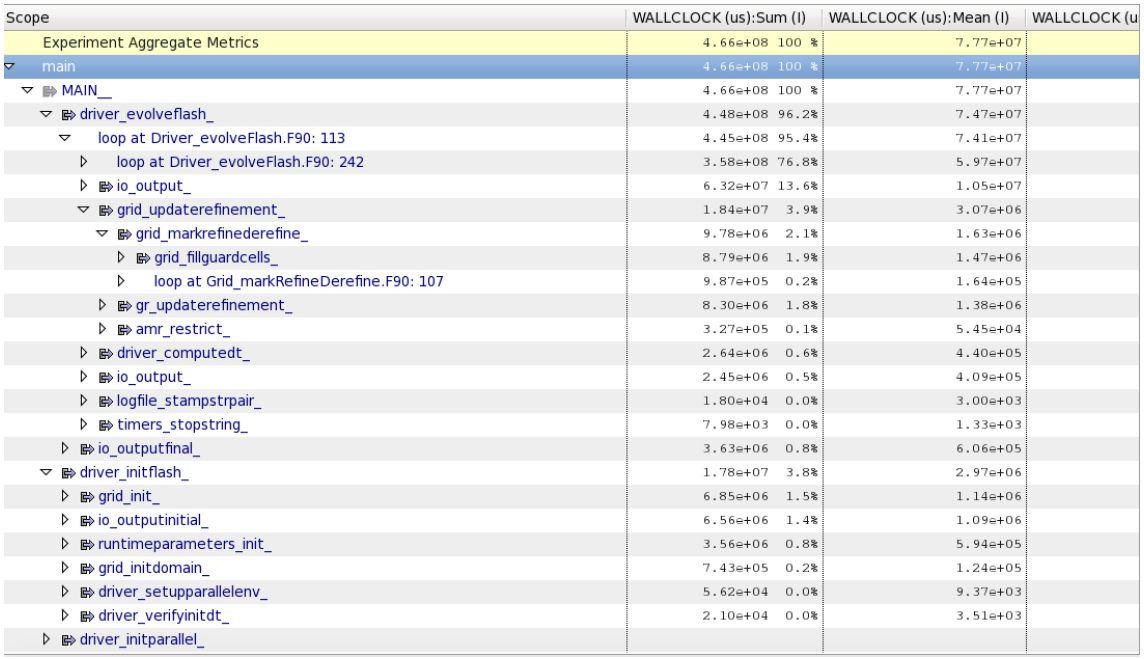
Paramesh works by splitting the computational domain into smaller and smaller sub-grid blocks where each refinement level splits the parent grid blocks into predefined divisions set in a parameters file and where each dimensional length of the sub-grid block is half the dimensional length of the parent grid block. This division of grid block cells can be organized in a tree data structure. At the edge of each sub-grid block, there are guard cells that communicate with the guard cells of other sub-grid blocks. Sub-grid blocks on the edge of the computational domain will abide by boundary conditions set by the developer. The data structure is defined a compile time. The compiler will distribute grid blocks to each processor such that data locality is accounted for and processor-processor communication is minimized [3].

2 Methods

For the Flash code, the size of the computational domain is determined by the variables xmin, xmax, ymin, ymax, zmin, and zmax. The minimum of the variables have been set to 0, and the maximum have been set to 1. The computational domain is then split by block sizes defined but the parameters (Nblockx, Nblocky, Nblockz). Each block contains computational cells defined by nxb, nyb, and nyz which have each been set to 8 for a resolution of 8^3. Since changing the Nblock parameters will effectively change the problem size it is one of the controls of the scaling study discussed [1].

The other control of this study is number of MPI processors. This value is set with the hpcrun and mpirun commands and have been varied based on 2^n. 16 was the maximum number of processors used in order to test within on node of dev-intel16 that contains a maximum of 28 processors.

3 Results



Scope	WALLCLOCK (us):Sum (I)	WALLCLOCK (us):Mean (I)	WALLCLOCK (u
Experiment Aggregate Metrics	4.66e+08 100 %	7.77e+07	
main	4.66e+08 100 %	7.77e+07	
MAIN__	4.66e+08 100 %	7.77e+07	
driver_evolveflash_	4.48e+08 96.2%	7.47e+07	
loop at Driver_evolveFlash.F90: 113	4.45e+08 95.4%	7.41e+07	
loop at Driver_evolveFlash.F90: 242	3.58e+08 76.8%	5.97e+07	
io_output_	6.32e+07 13.6%	1.05e+07	
grid_updaterefinement_	1.84e+07 3.9%	3.07e+06	
grid_markrefinederefine_	9.78e+06 2.1%	1.63e+06	
grid_fillguardcells_	8.79e+06 1.9%	1.47e+06	
loop at Grid_markRefineDerefine.F90: 107	9.87e+05 0.2%	1.64e+05	
gr_updaterefinement_	8.30e+06 1.8%	1.38e+06	
amr_restrict_	3.27e+05 0.1%	5.45e+04	
driver_computedt_	2.64e+06 0.6%	4.40e+05	
io_output_	2.45e+06 0.5%	4.09e+05	
logfile_stampstrpair_	1.80e+04 0.0%	3.00e+03	
timers_stopstring_	7.98e+03 0.0%	1.33e+03	
io_outputfinal_	3.63e+06 0.8%	6.06e+05	
driver_initflash_	1.78e+07 3.8%	2.97e+06	
grid_init_	6.85e+06 1.5%	1.14e+06	
io_outputinitial_	6.56e+06 1.4%	1.09e+06	
runtimeparameters_init_	3.56e+06 0.8%	5.94e+05	
grid_initdomain_	7.43e+05 0.2%	1.24e+05	
driver_setupparallelenv_	5.62e+04 0.0%	9.37e+03	
driver_verifyinitdt_	2.10e+04 0.0%	3.51e+03	
driver_initparallel_			

Figure 1: Hpcviewer on the StirTurb test problem showing the names of most loops and routines involved.

Figure 1 lists the the routines used in the simulation. There are four driver routines, Driver_initParallel(), Driver_initFlash(), Driver_evolveFlash(), and Driver_finalizeFlash() that are called in that order. Driver_initParallel initializes the parallel environment, and Driver_finalizeFlash finalizes the routines for each unit, deallocates memory, and "cleans up" what is necessary. For the most part, those two routines did not take up a significant percent of the runtime. Typically, most of the runtime was spent between Driver_initFlash which initializes the routines, creates the computational domain, and reads the runtime parameters and Driver_evolveflash which controls the time stepping of the simulation and the termination of the simulation based on given parameters, tmax, nend, and zfinal set in the flash.par (flash parameters) file. Within Driver_evolveflash the "loop at Driver_evolveFlash.F90" could be thought of as doing the majority of the computationally intense work of the simulation while the other routines focus on initialization and communication [1].

Theoretically, (1,2,1) and (1,1,2) should have equivalent results if the problem is symmetrical because the total number of divisions of the computational domains should be the same in both cases. However, the results of the HPCToolkit (Table 1) proved otherwise. For this reason, only runs were Nblockx=Nblocky=Nblockz were considered going forward.

(Nblockx, Nblocky, Nblockz)	# of processors	driver_initflash_(%)	Total Runtime (μs)
(1,1,2)	no mpirun	90.1	4.04e+06
""	2	45.8	1.36e+06
""	4	51.0	2.65e+06
(1,2,1)	no mpirun	18.9	5.54e+05
""	2	77.6	2.62e+6
""	4	80.3	6.85e+06

Table 1: Runtime and percent of runtime spent in driver_initflash for (1,1,2) and (1,2,1)

(Nblockx, Nblocky, Nblockz)	# of processors	runtime (μs)	driver_initflash_%
(1,1,1)	no mpirun	9.73e+06	95.2
""	2	3.57e+06	83.3
""	4	1.29e+07	70.5
""	8	6.23e+07	72.2
""	16	2.45e+08	50.2
(2,2,2)	no mpirun	5.23e+06	78.6
""	2	5.59e+06	65.6
""	4	2.68e+07	62.9
""	8	7.97e+07	59.9
""	16	2.94e+08	47.2
(3,3,3)	no mpirun	8.72e+06	4.0
""	2	9.25e+06	56.2
""	4	2.89e+07	56.8
""	8	1.02e+08	50.9
""	16	3.49e+08	47.3

Table 2: Results of the hpcrun for (Nblockx,Nblocky,Nblockz)=(1,1,1),(2,2,2),and (3,3,3) running with varying number of processors.

Hpctoolkit results show similar trends when a comparing problem size and increasing processor count. Generally, increasing the number of processors increased runtime. Table 2 shows that for a low processor count, driver_initflash took the majority of the runtime percentage. However, as shown in Figure 2, as processor count increased, driver_evolveflash became increasingly prominent in the percent of runtime eventually, for problem size (3,3,3) and processor count=16, it overtook initflash. Within initflash there were two important subroutines: Grid_initdomain and io_outputinit both of which took up and increasingly larger percentage of the runtime with increasing processor count overtaking the runtime parameters subroutine and loop_evolve routine that were prominent for lower processor counts.

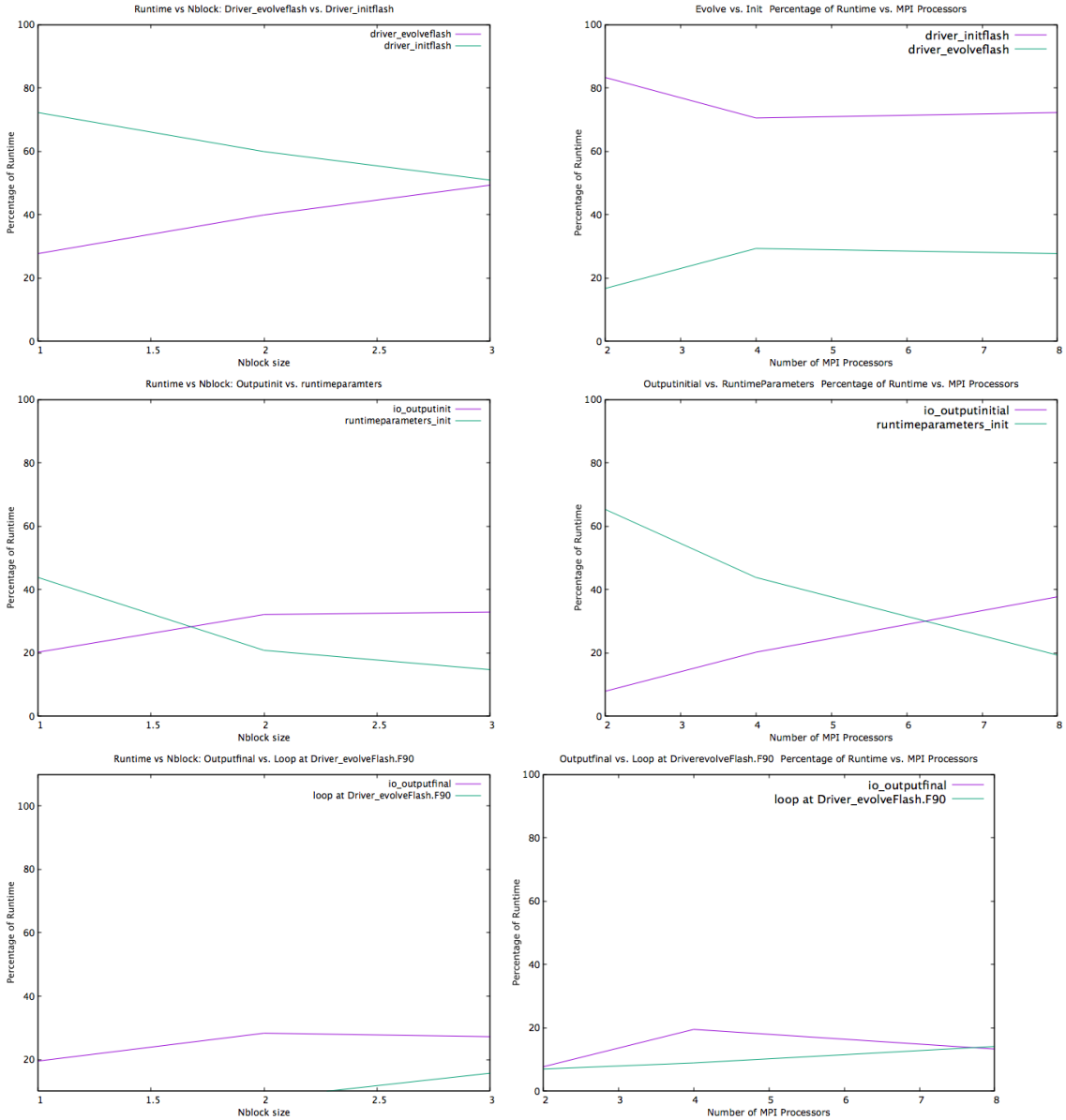


Figure 2: The graphs on the left show the relationship between runtime percentage and Nblock dimension for the various routines and subroutines of this simulation for. The IO vs. Runtime and IO vs. Evolveflash loop graphs came from 4 processor runs and the Evolveflash and Initflash graph came from an 8 processor run. The graphs on the right show the corresponding relationship with the number of processors for (1,1,1).

Similar results can be seen when comparing problem sizes for constant processor counts (Figure 2). As one increases the problem size from (1,1,1) to (3,3,3) several patterns emerge. Runtime parameters and loop runs became an increasingly smaller percentage of the runtime while input output routines of both the initflash and evolveflash subroutines became more prominent. For 2, 4, and 8 processors the IO subroutines were not always the dominant subroutines within initflash and initdomain. However, at 16 processors, the IO subroutines were the most dominant subroutines in all 3 problem sizes.

With increasing processor count, the IO subroutines init and final became prominent proportions of the their respective parent routines. This was also the case for increasing problem size. The reason could be because communication overhead has increased. When there are more processors involved, there were more processors that needed to be communicated with. Also, when the number of blocks was increased, there were more boundaries and therefore more guard cells that needed to communicate with each other. Communication overhead could be one of the bottlenecks of this test problem. If the trends observed above continue, with continually increasing problem

size, the performance of the simulation will be limited by how fast or efficiently it can communicate.

4 Conclusion

HPCToolkit profiling of the StirTurb test problem revealed that with increasing processor count and with increasing problem size, the IO subroutines, io_outputinitial and io_outputfinal, of the parent routines, driver_initflash and driver_evolveflash, respectively became an increasingly significant proportion of the runtime and also that the evolveflash routine became a more significant proportion compared to the initflash routine. This suggests that data communication is the main reason for the bottleneck as IO routines have overtaken other routines concerned with grid initialization and parameter setup as the most resource intensive routines. Fortunately, HPCToolkit is not an instrumentation type profiler, so no significant overhead was incurred. However, being a statistical sampling type of profiler can still introduce uncertainty in short runs [2]. The 2 MPI processor runs were done in much less than a few minutes. Future strategies for improving this study could be to run HPCToolkit for bigger problem sizes for a few hours to improve the accuracy of the study.

References

[1] Flash Center for Computational Science: University of Chicago. *FLASH User's Guide*, 4.3 edition, July 2015.

[2] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Boca Raton, FL, first edition edition, 2011.

[3] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda de Fainchtein, and Charles Packer. Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126, August 1999.

[4] John Mellor-Crummey, Laksono Adhianto, Mike Fagan, Mark Krentel, and Nathan Tallent. *HPCToolkit User's Manual*. Rice University, 5.3.2 edition, December 2015.