Adding shared memory parallelism to FLASH for many-core architectures

Chris Daley, John Bachan, Sean Couch, Anshu Dubey, Milad Fatenejad, Brad Gallagher, Dongwook Lee, Klaus Weide

Flash Center for Computational Science, Astronomy & Astrophysics, University of Chicago, Chicago IL

E-mail: cdaley@flash.uchicago.edu

Abstract. In this paper we discuss evolutionary changes to FLASH to enable enhanced applications to run efficiently on both the current generation BG/P and the next generation BG/Q. We motivate the need for change by discussing current FLASH applications and the challenges they are facing on today's architectures. Our solution to current challenges with a view to the next generation is mixed-mode MPI+OpenMP FLASH applications. We show some preliminary results and discuss next steps.

1. Introduction

FLASH is a collection of code units which can be put together to create custom multiphysics fluid dynamics applications [1]. It contains a huge number of code units which allow simulation of problems from astrophysics [2], high energy density physics (HEDP) and incompressible fluid dynamics. Important infrastructural capabilities are adaptive mesh refinement (AMR) which places additional resolution in important regions of the computational domain, a lagrangian framework and parallel I/O.

At the current time the biggest FLASH simulations are for supernova research and are run on 10,000s to 100,000s of BG/P cores. Enabling FLASH to run efficiently at this scale has involved incremental FLASH changes with each simulation campaign [3, 4]. A new incremental change is necessary to run FLASH efficiently on many-core architectures. This is particularly important for the FLASH code because it is a project which has been given early science time on the soon to arrive BG/Q at Argonne National Laboratory.

In this paper we describe adding shared memory parallelism to FLASH to exploit the increasing on-node parallelism. In Section 2 we give an overview of the most recent FLASH simulation campaigns on BG/P and discuss current and future challenges. This motivates the need for shared memory parallelism as a way to overcome these challenges on current architectures and also enable FLASH to run more effectively on future architectures. We describe the shared memory parallelism added to FLASH in

Section 3, show preliminary performance results in Section 4 and put the findings in context in Section 5.

2. Simulation overview

In the past few years the Flash Center has been awarded a significant computational budget for Type Ia Supernova research on the BG/P at Argonne National Laboratory. We have been able to effectively utilize this time by running massively parallel MPI-only simulations in virtual node (VN) mode. In this configuration there are 4 MPI tasks (1 per core) for each BG/P compute node and the 2GB of memory is divided evenly between the 4 MPI tasks.

The VN configuration imposes a challenging memory constraint which forces the research scientist to make tough trade-offs such as the number of multipole moments vs accuracy. The amount of work per MPI task also needs to be carefully calibrated: too little work and communication costs dominate; too much work and memory usage exceeds physical memory. Estimating the work per MPI task is further complicated because of AMR events during a simulation. It should come as no surprise that FLASH production simulations on BG/P generally use most of the available memory.

The forthcoming Type Ia Supernova simulation campaigns bring a new set of challenges. First, the research scientists want to use an enhanced hydrodynamics solver which is directionally unsplit, and second, some of the simulations will be run on the new BG/Q. The unsplit solver is desirable because it provides greater accuracy and symmetry than the split solver at the expense of a larger memory footprint. An unfortunate consequence of using the unsplit solver is that it becomes even more difficult to run FLASH in VN mode on BG/P. Multithreading FLASH becomes an attractive option because we could use dual mode (with 2 MPI tasks and 2 threads per task) and SMP mode (with 1 MPI task and 4 threads per task) on BG/P without leaving idle cores in each compute node. This is also a good option for BG/Q which has 4 hardware threads per core, 16 computational cores per node and 16GB of main memory.

Another application which has already faced memory limits on BG/P is the Type II Supernova application. A feature of this application is that each MPI task needs to have a copy of a large 300MB nuclear equation of state (EOS) table. This makes running in VN mode extremely difficult unless space-saving optimizations such as storing lower precision tabulated data or fewer data points are made. The multithreaded version of the application saves the scientist from having to make these kinds of compromises which may affect the validity of the simulation.

3. Multithreaded FLASH

We have multithreaded several physics units of FLASH to speed up the solution advancement on the computational mesh. The multithreaded units are split hydrodynamics, unsplit hydrodynamics, flame, EOS (Gamma, Multigamma and

Helmholtz implementations), Multipole Poisson solver (2D cylindrical and 3D Cartesian) and laser energy deposition. In most FLASH production simulations the computational mesh is managed by Paramesh [5]. We give a brief overview of Paramesh before introducing the two OpenMP threading strategies in FLASH.

Paramesh builds a hierarchy of blocks (or patches) as shown in Figure 1 according to the application refinement criteria. Each block contains the some number of internal cells (in the figure each 2d block has 6x4 internal cells) and additional guard-cells which store the solution from neighboring blocks in order to provide a self-contained grid to the application. The application can safely update the solution on different blocks before passing control back to the mesh package for tasks such as guard-cell updates and flux corrections. The provision of self-contained grids enables usage of huge numbers of MPI tasks to simultaneously update the solution on many blocks at the same time. At the current time the largest FLASH Type Ia Supernova simulations have around 1 million blocks where each block contains 16^3 cells per fluid variable.

The first multithreading strategy in FLASH involves assigning different blocks from the list of blocks to different threads. If an MPI rank has 10 blocks and there are two threads per MPI rank then each thread would update 5 blocks. This is a coarse grained threading strategy which we name "thread block list". Such a strategy demands that the list of blocks is much greater than the number of threads to avoid load imbalance resulting from threads being assigned different numbers of blocks.

The second multithreading strategy in FLASH is finer-grained and involves using multiple threads to simultaneously update the solution in different cells of the same block. This involves threading the nested loops in the kernel subroutines of FLASH. We distribute the slowest varying dimension amongst threads to reduce cache conflicts. One nice feature of this strategy, which we name "thread within block", is that it can also be used for a uniform grid in which there is 1 block per MPI task. The work per cell is similar for the hydrodynamic and flame units and so we use an OpenMP static schedule. In future we could pin threads to cores and first-touch the cells assigned to each thread to improve performance on NUMA many-core architectures.

In both strategies the mesh package calls from the application are serialized. This means all threads are idle until a single thread completes operations such as guard cell filling and flux correction.

4. Results and discussion

In this section we show preliminary FLASH multithreaded scaling for the RTFlame application on BG/P (for MPI-only scaling see [4, 6]). This application is used to study the turbulent nuclear combustion phase in Type Ia Supernova. The RTFlame simulation selected for our experiments has a domain with effective resolution of 256^3 cells [7]. It makes use of the Paramesh AMR package and is run with 5 levels of refinement and blocks of 16^3 cells. At initialization the adaptive mesh refines to create 21,462 total blocks and no regridding events happen in the 20 time steps. We choose to run for

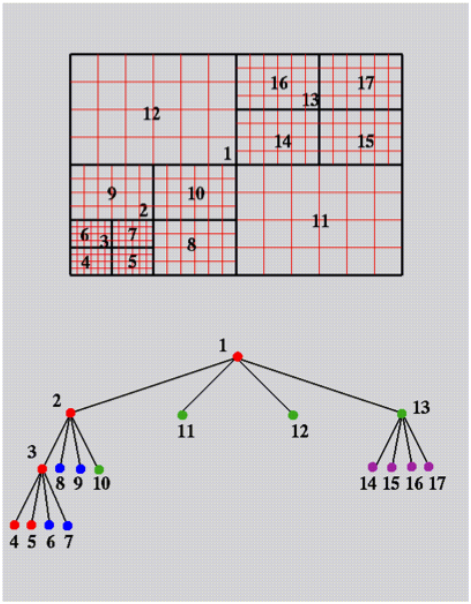

Figure 1. The Paramesh oct-tree. Image taken from `http://www.physics.drexel.edu/~olson/paramesh-doc/Users_manual/amr_intro.html` on 02-13-2012.

only 20 time steps on 512 nodes of BG/P to get a feel for current performance without consuming any significant computational time. The application also excludes I/O and tracer particles.

All experiments are run on BG/P with a single MPI task per node which has access to the entire 2GB of on-node memory (SMP mode). In our experiments this means each MPI task is assigned an average of 42 FLASH blocks. Since each node on BG/P contains 4 cores we expect to get a faster time to solution when we run the multithreaded FLASH code with the same number of MPI tasks and 2,3 and 4 OpenMP threads. We test both the thread block list and thread within block multithreaded configurations and then compare evolution time against the MPI-only version of FLASH. The results are shown in Figure 2.

The figure shows that adding threads improves time to solution for both threading strategies. The speedup is not ideal and a contributing factor to this is that the grid management calls to exchange guard cells and correct fluxes are serialized. We have not yet quantified the time spent in serial sections, OpenMP parallel sections and MPI communication and so this is an important next step. We see slightly better performance for the thread block list strategy which may be because of greater OpenMP parallel region coverage of code sections with non-negligible computational work, but until quantification of coverage and testing on many-core architectures with a larger number of threads it is too early to favor one strategy over another.

It is likely that the speedup would be slightly worse in a production simulation which includes I/O and particles, but as we will discuss this is a small amount currently and should diminish in future. In a recent Type Ia supernova simulation on 65,536 cores of BG/P only 7% of total wallclock time was spent in I/O [3]. Going forward, the

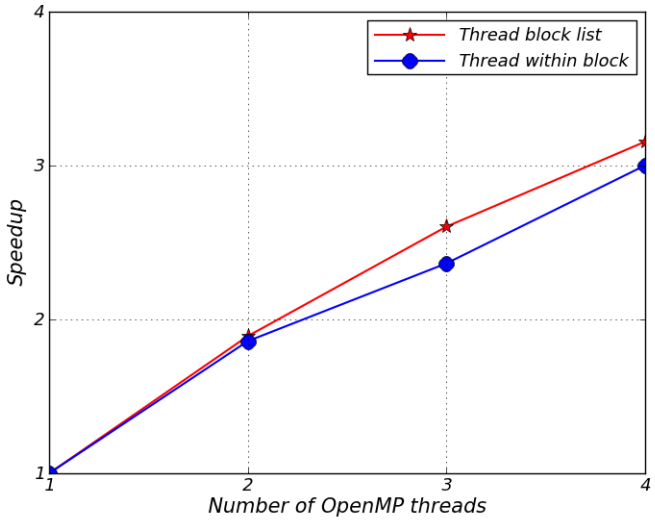

Figure 2. Speedup for RTFlame simulation. Run for 20 time steps on 512 nodes of BG/P in SMP mode with 512 MPI tasks and different numbers of OpenMP threads. I/O and particles are excluded

compute node I/O time is expected to decrease significantly with usage of the Glean framework for quickly moving simulation data to analysis nodes for in-situ analysis and writing data to storage [8]. The analysis nodes could also be used for particle updates because tracer particles have no impact on the dynamics of the simulation and are only in a simulation for post-processing analysis.

Perhaps the most difficult bottleneck to overcome is the serialization of the mesh package calls. We would like to investigate overlapping threads performing computational work with a designated mesh package thread for all MPI communication, but this is a challenging step which would require either or both

(i) restructuring of the adaptive mesh packages to support a non-blocking API which communicates a subset of blocks at a time. In this API the application would notify the mesh package of blocks ready to be communicated and the mesh package would notify the application of blocks ready for solution update.

(ii) restructuring of the kernels in FLASH to update the solution on the cells which will be communicated to other MPI tasks first. A guard cell exchange could then happen in parallel with the solution update on the remaining cells.

5. Conclusion

In this paper we gave an overview of current large-scale FLASH simulations on the BG/P. A major challenge we face is lack of memory per MPI task which makes it difficult to use more sophisticated numerical schemes such as a unsplit hydrodynamics solver, more detailed EOS tables, and more FLASH blocks per MPI task to stay above

strong scaling limits. A new capability in FLASH is support for multiple threads of execution within each MPI task. This allows us to run FLASH in more memory efficient ways which enables our scientists to run higher-fidelity simulations without the memory limitations they are accustomed to on current generation architectures.

We have introduced a coarse-grained and fine-grained threading strategy into FLASH. The coarse-grained strategy distributes the Paramesh blocks assigned to each MPI tasks amongst threads and the fine-grained strategy distributes cells from each block amongst threads. Both strategies show speedup with thread count for a RTFlame application on BG/P. The speedup is not perfect, but some loss of speedup is expected because of serialization of grid calls. Future work will quantify the OpenMP coverage and investigate the most efficient ways to run multithreaded FLASH on many-core architectures.

It is challenging to modify large software like FLASH which contains AMR and multiple physics modules for the next generation of many-core architectures especially since there is still on-going debate about future architectures. We have a large user community and many diverse applications to support and so making revolutionary changes to the code base is extremely difficult. Our approach of inserting OpenMP directives is an evolutionary change necessary to allow a large, highly-capable piece of software to run efficiently on the Blue Gene platforms.

Bibliography

[1] A. Dubey, K. Antypas, M.K. Ganapathy, L.B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[2] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, November 2000.

[3] R. Latham, C. Daley, W.K. Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary. A case study for scientific i/o: improving the flash astrophysics code. *Computational Science and Discovery*, page to appear.

[4] A. Dubey, A. Calder, C. Daley, C. Graziani, R. Fisher, G.C. Jordan, D.Q. Lamb, L.B. Reid, D.M. Townsley, and K. Weide. Pragmatic optimizations for best scientific utilization of large supercomputers. submitted and revised.

[5] P. MacNeice, K.M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.

[6] B.R. de Supinski et al. Modeling the office of science ten year facilities plan: The peri architecture tiger team. *Journal of Physics: Conference Series*, 180(1):012039, 2009.

[7] D. M. Townsley, A. C. Calder, S. M. Asida, I. R. Seitenzahl, F. Peng, N. Vladimirova, D. Q. Lamb, and J. W. Truran. Flame evolution during Type Ia supernovae and the deflagration phase in the gravitationally confined detonation scenario. *The Astrophysical Journal*, 668:1118–1131, 2007.

[8] V. Vishwanath, M. Hereld, M.E. Papka, R. Hudson, G. Jordan, and C. Daley. In situ data analytics and I/O acceleration of FLASH astrophysics simulations on leadership-class systems with GLEAN. In *SciDAC, Journal of Physics: Conference Series*, July 2011.