ANL/ALCF/ESP-13/10

# Petascale Simulations of Turbulent Nuclear Combustion

*ALCF-2 Early Science Program Technical Report*

**Argonne Leadership Computing Facility**

ANL/ALCF/ESP-13/10

# Petascale Simulations of Turbulent Nuclear Combustion

*ALCF-2 Early Science Program Technical Report*

prepared by
Christopher Daley
University of Chicago

May 7, 2013

# Petascale Simulations of Turbulent Nuclear Combustion

Christopher Daley

April 3, 2013

## Contents

# 1 Science objectives

Type Ia (thermonuclear-powered) supernovae are important in understanding the origin of elements and are a critical tool in cosmology. The Flash Center is carrying out large-scale, 3D simulations of two key physical processes in Type Ia supernovae, leading to a better understanding of these explosions.

The two key physical situations for which we are carrying out large-scale 3D simulations are: (1) an initially planar flame in a rectilinear domain with constant gravity and nearly constant density in which turbulence due to the buoyancy of the hot ash drives the burning (which we refer to as RTFlame simulations); and (2) an initially spherical or perturbed spherical flame "bubble" inside of a star in which decaying homogeneous isotropic turbulence drives the burning (which we refer to as DDT simulations). Comparison of the results for these two different physical situations will elucidate the ways in which buoyancy-driven turbulent nuclear combustion differs from turbulent nuclear combustion in a homogeneous, isotropic turbulent background and within a stellar environment.

Application of these results will help improve cosmological distances measured using type Ia supernovae which will ultimately shed light on the nature of Dark Energy. Understanding dark energy ranks among the most compelling problems in all of physical science.

# 2 The FLASH code

FLASH is a multiphysics, finite-volume Eulerian code containing capabilities suitable for problems in astrophysics, cosmology, high energy density physics and incompressible fluid dynamics. Key capabilities in the early science simulations are Adaptive Mesh Refinement (AMR) which increases resolution in physically important regions of the computational domain, high-order compressible hydrodynamic solvers in directionally split and unsplit formulations which are able to treat non-ideal equations of state (EOS), a multipole self-gravity solver, a flame model and a nuclear energy release model. The code is written in Fortran 90 and C and is parallelized with MPI and more recently OpenMP. It makes use of parallel I/O capabilities provided by either HDF5 or Parallel-netcdf libraries.

The AMR capability is an important feature which can improve time to solution by orders of magnitude compared to a fixed resolution uniform grid and also allows larger problems to be tackled which would not otherwise fit in memory. At the current time, production FLASH simulations use the Paramesh package [1] to provide a block-structured, oct-tree adaptive grid. All blocks/patches in Paramesh contain the same number of cells consisting of internal cells and additional guard cells which store the solution from neighboring blocks. The explicit solvers in FLASH update the solution in the internal cells of "leaf" blocks and then pass control to Paramesh to exchange guard cells and correct fluxes. A leaf block is a block at the finest resolution in each region of the computational domain. Massive parallelism is possible because different blocks are assigned to different MPI tasks.

# 3 Code changes

We made various code changes so that FLASH early science applications would work well on Mira, the BG/Q platform at the Argonne Leadership Computing Facility (ALCF). The major change of adding hybrid MPI/OpenMP parallelism was part of a longer-term plan and preparation began well in advance of being given access to a BG/Q platform (Section 3.1). Other changes were unplanned and happened after we were given access to BG/Q, but are equally important and necessary to ensure a successful early science program. These include changes for correctness such as altering the source code to avoid issues with the `threadprivate` OpenMP directive on BG/Q (Section 3.2) and creating an unusual FLASH Makefile to obtain expected results (Section 3.3). It also includes changes to improve resource usage such as adding wrapper functions to monitor memory usage (Section 3.4), reducing the amount of communication in a Paramesh initialization subroutine to minimize memory leaks happening in the messaging layer on Mira (Section 3.5) and adding wrapper functions to selectively profile the important parts of FLASH (Section 3.6). The unplanned changes were only possible because of the long early-access period granted by the Early Science Program (ESP).

## 3.1 Architectural

The FLASH computer code has until recently been a MPI-only code. The MPI-only approach works well on BG/P, where we can run efficiently with 1 MPI rank per core (Virtual Node mode) after making a dedicated effort to reduce the memory footprint of FLASH simulations below 512 MB per MPI rank. In contrast, the approach of 1 MPI rank per core is generally a poor choice on BG/Q which has 4 hardware threads per core. On BG/Q it is highly desirable to place multiple MPI ranks or software threads per core to hide memory latency and pipeline stalls. We chose to multithread the FLASH ESP applications. This is a sensible choice because the new capabilities in the FLASH ESP simulations add to the memory footprint, making it even more challenging to fit in 512 MB per MPI rank and so effectively ruling out even 2 MPI ranks per core on BG/Q.

We have added OpenMP directives to code modules in the FLASH ESP applications including the

hydrodynamics solver, turbulence model, flame model, EOS, and multipole solver. The directives exist at both a coarse-grained and fine-grained granularity in the source code. In the coarse-grained case OpenMP threads update the solution in different Paramesh blocks and in the fine-grained case OpenMP threads update the solution in different cells from the same Paramesh block. The coarse-grained approach is hereafter referred to as thread block list and the fine-grained approach is hereafter referred to as thread within block. In general it is simple to assign independent work to the different threads because of stencil-based or point-wise kernels in the code modules. The one exception is the multipole solver where we needed to create a new moment array for each thread to avoid conflicts during frequent multipole moment updates.

### 3.2 Working around broken OpenMP features

We encountered problems with the `threadprivate` OpenMP directive on BG/Q which did not occur on BG/P or x86 architectures with various Fortran compilers. We found that our applications would segfault after accessing `threadprivate` data, but we could never reproduce the issue in a small standalone test problem to submit a simple bug report. Our solution was to remove `threadprivate` directives from FLASH by either rewriting code or removing redundant multithreading. The code that required slight rewrites were the Helmholtz EOS and the Multipole solver. In both cases the code frequently read/wrote `threadprivate` module data in the style of old Fortran codes using common block data. The rewrite involved moving this module data to the stack or heap and, where necessary, passing this data through subroutine argument lists. Although it took a little bit of time, these code changes benefit FLASH because it is now possible to use nested OpenMP parallel regions for the first time, e.g. simultaneous use of thread block list and thread within block FLASH multithreaded strategies. Nested parallelism wouldn't have worked in the older version of FLASH because of the rules regarding the persistence of `threadprivate` data [2].

### 3.3 Creating a custom build for correctness

The biggest issue we had is that we did not get correct FLASH solutions with driver versions before V1R1M2 on Vesta, the test and development BG/Q at ALCF. We found unphysical features in the time history of mass, y-momentum and z-momentum integral quantities. Integral quantities represent the overall state of the simulation and are calculated once per time step by summing over all cells in the computational domain. We found the issues only happened when compiling all FLASH source files with either the OpenMP compiler option or aggressive compiler optimization. The following figures show mass and directional momentum as a function of time in 3 RTFlame test cases run on Vesta BG/Q and Intrepid BG/P. The applications are setup with either the split or unsplit hydrodynamics solver and the parameters which are varied are the flame speed and effective resolution.

Figure 1 shows results from an RTFlame test problem setup with the split hydrodynamics solver and run with a flame speed of 12km/s and an effective resolution of $512^3$. The figure includes results from 3 different FLASH builds on Vesta: an MPI-only build, a regular multithreaded build and a selective multithreaded build in which only the files containing OpenMP directives are compiled with the OpenMP compilation flag.

It is clear that there are numerical issues when using a regular multithreaded build of FLASH with either 1 OpenMP thread or 4 OpenMP threads. It is especially concerning that the numerical issues are different in the 1 OpenMP thread and 4 OpenMP threads case because the threads update the solution in independent cells. Figure 2 shows results from an unsplit hydrodynamics build of FLASH but the same test problem. Once again results are bad for a standard multithreaded build of FLASH, but they are also bad for a selective multithreaded build of FLASH with the `-qhot` compilation option. Finally, Figure 3 shows the same issues from a different RTFlame test with flame speed 9km/s and effective resolution $256^3$ run to a much later time.

In Figure 1 the standard multithreaded experiments are compiled with `-qsmp=omp:noauto -g -O3 -qnohot -qrealsize=8 -qnosave -c -qthreaded` in this order. We specify the `-qsmp` option first because `-qsmp` (without the `noopt` sub-option) instructs the compiler to optimize as well as parallelize, where, the default optimization is equivalent to `-O2 -qhot` in the absence of other optimization options [3]. In our case the explicit `-O3 -qnohot` at the end of the compilation line should override the implicit optimizations and make the standard multithreaded FLASH build equivalent to the selective multithreaded build. It is therefore puzzling that results depend on the choice of multithreaded FLASH build.

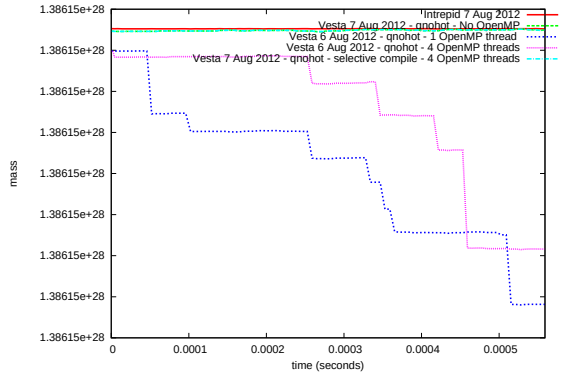
The Vesta results are in good agreement with the Intrepid results when using the selective multithreaded build with `-qnohot` optimization and so we build FLASH in this way for the production early science runs. Recent testing has shown that these issues no longer exist, meaning it is now safe to use `-qsmp=omp:noauto` and `-qhot` on *all* source files. Note that the original issues cannot be reproduced when using the newer V1R1M2 driver but the same May 2012 compiler version.

### 3.4 Monitoring memory usage

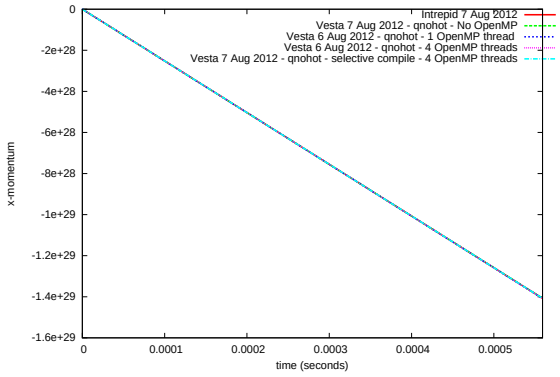
We added wrapper functions around `mallinfo` [4] and `Kernel_GetMemorySize` [5] to monitor FLASH memory usage. This was necessary because the original memory monitoring code in FLASH did not work on BG/Q because of issues with `rusage` [6] on BG/Q. The new memory monitoring code has already allowed us to drill down to a small portion of code in Paramesh which causes a memory leak to happen somewhere in the messaging layer on BG/Q only, see Section 3.5.

### 3.5 Working around a memory leak

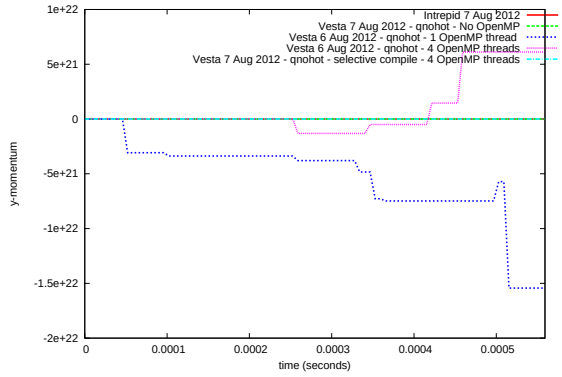
Early runs on Mira revealed an unexpected memory growth during FLASH initialization. We found similar heap memory growth in measurements from both `mallinfo` and `Kernel_GetMemorySize` on Mira BG/Q, but surprisingly no significant memory growth on Intrepid BG/P. The memory growth happens after running a Paramesh subroutine named `find_surrblks` which finds the surrounding block neighbors of every Paramesh block in the domain. It works by passing block metadata around a ring of all MPI ranks in `MPI_Comm_world` using `MPI_Sendrecv_replace` so that each MPI rank can construct a local view of its nearest neighbors in



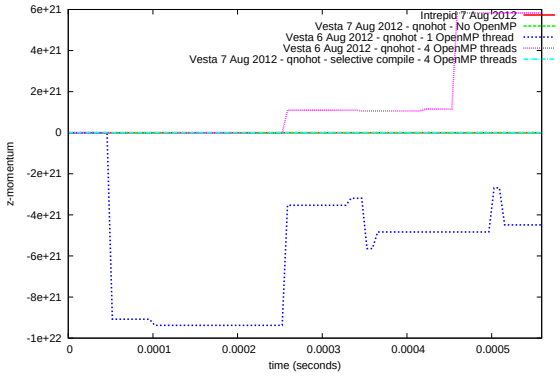

(a) mass




(b) x-momentum

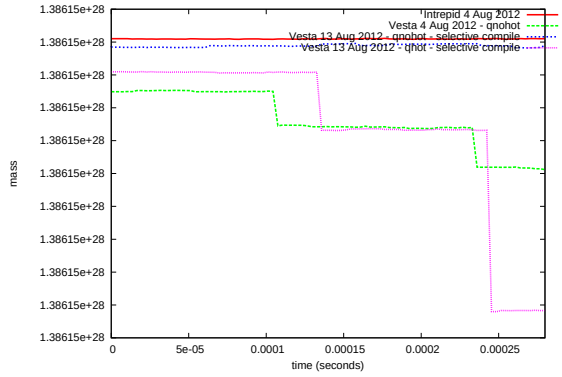



(c) y-momentum

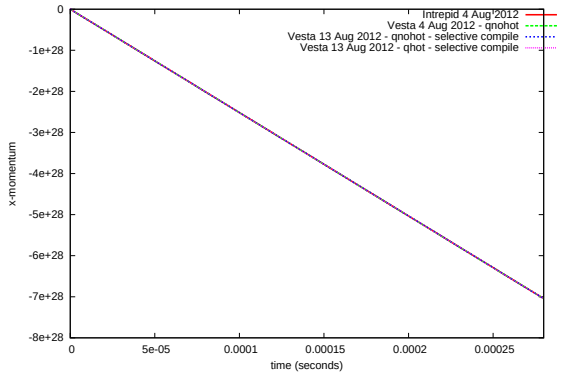



(d) z-momentum

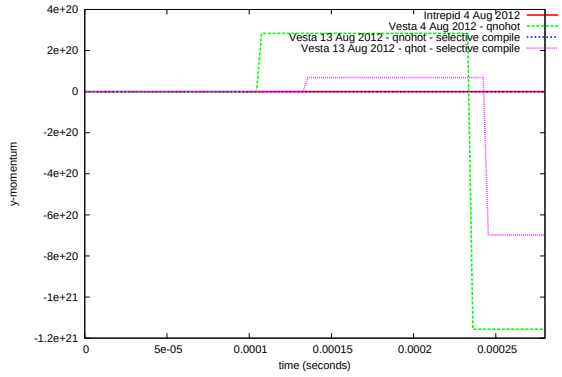
Figure 1: Integrated quantities from a split hydrodynamics RTFlame test problem with flame speed 12km/s and effective resolution $512^3$.



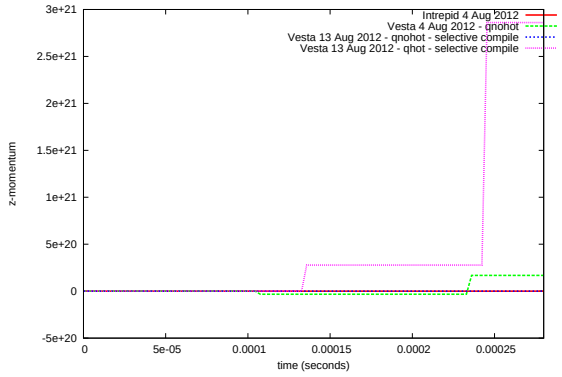

(a) mass




(b) x-momentum

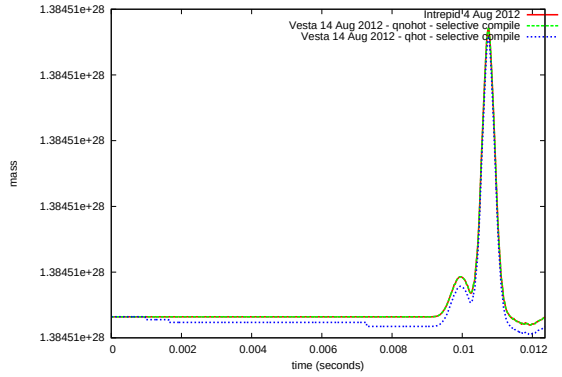



(c) y-momentum

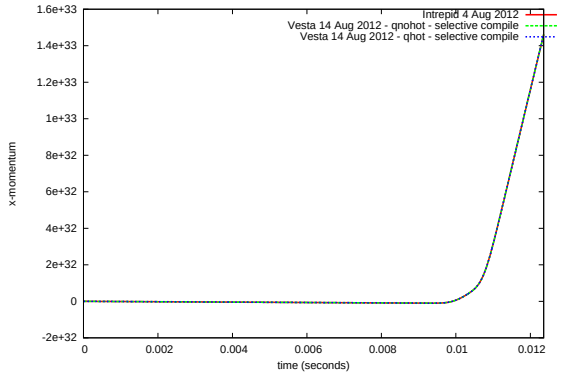



(d) z-momentum

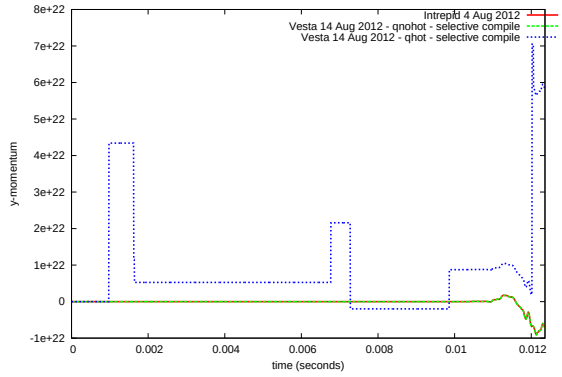
Figure 2: Integrated quantities from an unsplit hydrodynamics RTFlame test problem with flame speed 12km/s and effective resolution $512^3$. All Vesta experiments in this figure are run with 4 OpenMP threads.



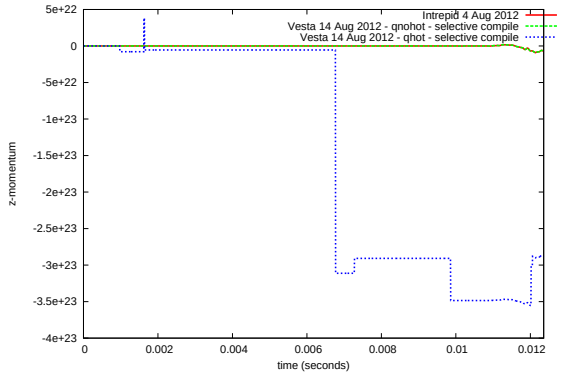

(a) mass




(b) x-momentum




(c) y-momentum

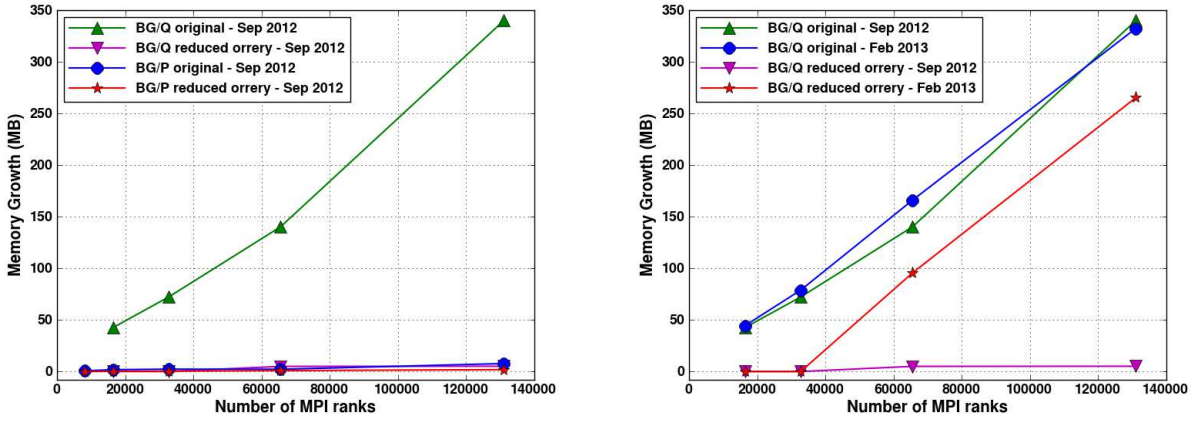



(d) z-momentum

Figure 3: Integrated quantities from an unsplit hydrodynamics RTFlame test problem with flame speed 9km/s and effective resolution $256^3$. All Vesta experiments in this figure are run with 4 OpenMP threads.

the global tree. In the past we used this subroutine during initialization and whenever the grid changed during evolution. It is obviously non-scalable. We implemented a scalable method of updating the local view which enabled larger simulations on Intrepid BG/P [7], but the method only works during evolution because it depends on the initial surrounding block neighbors of the top-level blocks. This means there is still a one-time cost at initialization.

We decided to test an optimized version of this subroutine in which the block metadata is circulated in a reduced MPI communicator consisting of only those MPI ranks that own Paramesh blocks. This idea is only applicable at initialization since it depends on there being many more MPI ranks than Paramesh blocks. Figure 4a shows heap memory growth for the MPI rank with the maximum memory usage after executing the `find_surrblks` subroutine, where, heap memory is obtained from `mallinfo (m.hblkhd + m.uordblks)` [8]. We run the FLASH applications in MPI-only mode with 4 MPI ranks per node on BG/P and 16 MPI ranks per node on BG/Q for both the original and optimized version of the code. It is clear that significant memory growth only happens for the original version of the code on BG/Q. Unfortunately, measurements on Mira in February 2013 show that optimized version of the code memory is now also affected by the memory growth issue, although the optimized version does leak approximately 70 MB less than the original version. Figure 4b compares measurements taken in September 2012 with those in February 2013.




(a) (b)

Figure 4: The increase in memory usage after running the original and optimized `find_surrblks` subroutine. (a) shows results from Intrepid BG/P and Mira BG/Q in September 2012, and (b) shows results from Mira BG/Q in September 2012 and February 2013.

We ran a 4096 node FLASH experiment with the `mtrace` memory debugger [9] monitoring all memory allocations that happen during the `find_surrblks` subroutine on MPI rank 64958. This MPI rank is chosen because it had the maximum memory usage after executing the `find_surrblks` subroutine. The memory leaks detected by `mtrace` are shown in Figure 5. These leaks are not in FLASH.

```
Memory not freed:
-----------------
           Address     Size     Caller
0x0000001f80acc020   0xe800  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/lib/dev/mpich2/src/util/mem/handlemem.c:204
0x0000001f80ada840   0xe800  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/lib/dev/mpich2/src/util/mem/handlemem.c:204
0x0000001f80ae9060   0xe800  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/lib/dev/mpich2/src/util/mem/handlemem.c:204
0x0000001f83a2c740     0x30  at /bgsys/drivers/V1R1M22012_0907_1744/ppc64/toolchain/gnu/gcc-4.4.6/libstdc++-v3/libsupc++/new_op.cc:52
0x0000001f83a2c780     0xc8  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f83a2c860     0xc8  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f83a2ccc0     0x30  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab21e80    0xd00  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab25d40     0x28  at /bgsys/drivers/V1R1M22012_0907_1744/ppc64/toolchain/gnu/gcc-4.4.6/libstdc++-v3/libsupc++/new_op.cc:52
0x0000001f8ab29c20     0x28  at /bgsys/drivers/V1R1M22012_0907_1744/ppc64/toolchain/gnu/gcc-4.4.6/libstdc++-v3/libsupc++/new_op.cc:52
0x0000001f8ab2d380     0xc0  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab2d9e0      0x8  at /bgsys/drivers/V1R1M22012_0907_1744/ppc64/toolchain/gnu/gcc-4.4.6/libstdc++-v3/libsupc++/new_op.cc:52
0x0000001f8ab2da00     0x20  at /bgsys/drivers/V1R1M22012_0907_1744/ppc64/toolchain/gnu/gcc-4.4.6/libstdc++-v3/libsupc++/new_op.cc:52
0x0000001f8ab2da40      0x8  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab2da60      0x8  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab2da80   0x1374  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab2ee00    0x798  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab2f640    0x180  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/sys/buildtools/pami/components/memory/heap/HeapMemoryManager.h:119
0x0000001f8ab37780   0x2000  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/lib/dev/mpich2/src/util/mem/handlemem.c:188
0x0000001f8ab397a0   0xe800  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/lib/dev/mpich2/src/util/mem/handlemem.c:204
0x0000001f8ab4a3e0   0xe800  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/lib/dev/mpich2/src/util/mem/handlemem.c:204
..................
 253 more leaks of   0xe800  at /bgsys/source/srcV1R1M22012_0907_1744.17581/comm/lib/dev/mpich2/src/util/mem/handlemem.c:204
```

Figure 5: Memory leaks detected by `mtrace` after running the `find_surrblks` subroutine on 4096 nodes of Mira BG/Q in September 2012.

### 3.6 Adding selective profiling to FLASH evolution

The performance of FLASH on BG/Q is studied using IBM's High Performance Computing Toolkit (HPCT) which provides statement level profiling through `vprof`, hardware counter summary information and MPI performance data. We use the HPCT API to selectively profile FLASH evolution only, i.e. we exclude initialization, which is important for short performance studies. The initialization is always going to be slightly expensive because the initial adaptive mesh needs to be generated from a small number of root blocks which exist on a subset of MPI ranks. We optimized the most expensive part of initialization, see Section 3.5, however, for short performance studies consisting of a small number of time steps the impact of initialization can skew results. Profilers like `vprof` do not provide call-path profiling and so certain subroutines in the call stack of FLASH initialization appear to be more expensive than they actually are. The selective profiling feature allows us to run shorter performance studies and identify expensive parts of FLASH evolution.

## 4 Optimizations

The early science preparation time has allowed us to increase the level of OpenMP coverage in FLASH and make the multithreading in FLASH truly production ready. In addition to the multithreading we have found opportunities to improve the serial, MPI and parallel I/O performance in FLASH. These improvements were made incrementally during the early science preparation period for both RTFlame and DDT applications. The optimizations for the RTFlame application, which are also usable by the DDT application, are discussed in Section 4.1 and the DDT application specific optimizations are discussed in Section 4.2.

## 4.1 RTFlame

Since it is interesting to see the performance impact of each incremental change, we take the current multithreaded FLASH code and revert all serial and MPI optimizations. We successively apply the ESP optimizations and then profile each transient version of the code. The baseline measurement is obtained from a FLASH binary compiled with `-O3 -qnohot`. These non-aggressive compilation flags were used by Vitali Morozov when he successfully ported FLASH to an early access BG/Q machine. The effect that each optimization has on time to solution is shown in Figure 6 and the description of the optimizations follow. In the figure blue bars indicate optimizations which are currently being used in the early science production runs and red bars indicate optimizations which are not being used. The optimizations that are in-use reduce time to solution by 32.6% in this test problem which is approximately a 1.5x performance improvement.

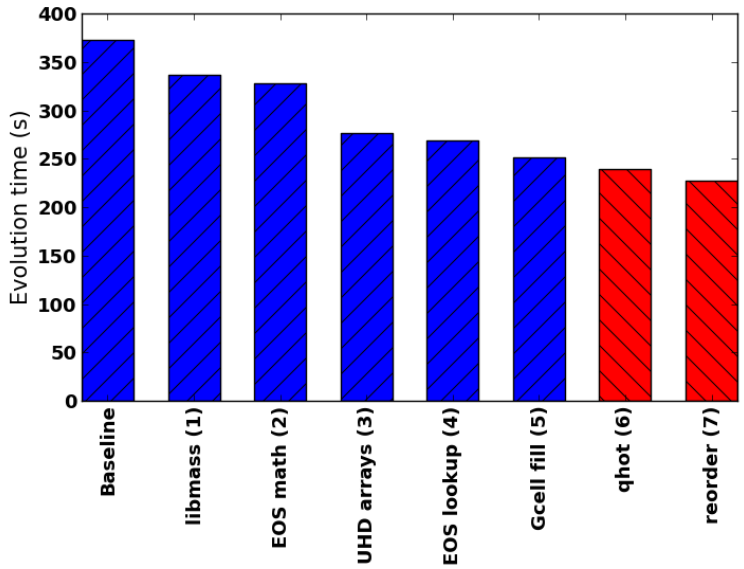



Figure 6: Time to complete 50 steps of an RTFlame test problem on Vesta BG/Q after applying each new optimization. The simulations were run with 16 MPI ranks per node, 4 threads per MPI rank and used the thread within block multithreading strategy.

**Linking against the MASS library - libmass (1)**
The baseline `vprof` profile shows a large number of counts in symbols `__log` and `__ieee754_log` which are glibc log functions. An extremely simple optimization is to use the accelerated log functions in the IBM Mathematical Acceleration Subsystem (MASS) library instead of those in glibc. The calls are inserted automatically when compiling with aggressive optimization [3], however, aggressive flags, such as `-qhot`, originally led to incorrect FLASH answers as described in Section 3.3. Linking against `libmass.a` explicitly allows us to use the accelerated math functions without needing to compile FLASH with aggressive optimization.

**Modifying the Math expressions in EOS - eosmath (2)**
The next change is a contribution from Vitali Morozov which consists of faster math computations in the Helmholtz EOS. He replaced the expression `x4=plasg**(0.25e0)` with `x4=sqrt(sqrt(plasg))` and `inv_lami=s1**(1.0e0/3.0e0)` with `inv_lami=sqrt3(s1)`, where `sqrt3` is a fast cube root function which implements Halley's method [10].

**Changing the array layout in the unsplit hydrodynamics solver - UHD arrays (3)**
The earlier `vprof` profiles show that a large amount of time is spent calling subroutines in the FLASH subroutine `hy_uhd_getRiemannState`. This is understood and happens because the compiler must add code to copy non-contiguous array slices into temporary contiguous arrays for the purpose of the subroutine call. The optimization involves reordering several arrays so that the i, j, k indices are the slowest varying dimensions. A snapshot of the `vprof` profiles for the original and optimized code is shown in Figure 7. On the far left of the figure is the source line number, next to that is the number of clock ticks, where each tick corresponds to 0.01 seconds of CPU time, and on the right is the actual source.

**Improving the table lookup locality in EOS - EOS lookup (4)**
The `vprof` profiles indicate that interpolation of table lookup data in Helmholtz EOS is expensive. This is partly because the lookup quantities exist in separate arrays which hurts the locality. We show just the bicubic hermite polynomial function for electron positron number densities in the before and after `vprof` fragments in Figure 8. In the original code, `eos_xf` is the electron positron number density and `eos_xfd`, `eos_xft`, `eos_xfdt` are various derivatives of this number density. Performance improvement is possible by placing all electron positron number density quantities next to each other in a derived datatype named `eos_tbl` which has fields `xf`, `xft`, `xfd` and `xfdt`. Note that there are many more fields in this derived datatype which are there to improve the access locality in other Helmholtz EOS functions.

**Reducing the number of guard cell fills - Gcell fill (5)**
The next optimization involves reducing the number of MPI synchronization points needed to keep guard cells (ghost cells) updated. Frequent guard cell exchanges happen because FLASH applications consist of multiple physics units called in sequence which update solution data according Strang operator splitting [11]. It is essential that the different physics units access current guard cell data to correctly update the solution data and so the safe convention adopted in FLASH is that any unit which accesses guard cells is responsible for making a call to the mesh package to update guard cells. This approach works well and avoids many bugs, but has the side-effect that guard cells can be exchanged too often. In the case of the RTFlame application there are guard cell exchanges in Hydrodynamics, Flame and Lagrangian Tracer Particles units as well as a simulation specific analysis routine. This results in too many guard cell exchanges because there are no writes to mesh data when updating Lagrangian tracer particles or performing simulation specific analysis. Since the guard cells are still valid we can safely remove two guard cell exchange synchronization points.

**Compiling with aggressive optimization - qhot (6)**
The performance can be improved by using more aggressive compiler optimization, however, this originally resulted in incorrect FLASH answers as discussed in Section 3.3.

**Reordering the Paramesh data arrays - reorder (7)**
The core grid data structure in Paramesh can be reordered so that the fastest varying dimension is no longer

```
 688        !! Left and right Riemann state reconstructions
 689 1003   call hy_uhd_dataReconstOnestep&
 690             (blockID,blkLimitsGC,i,j,k,dt,del,&
 ...              ...
 709              sig    (1:NDIM,1:HY_VARINUMMAX,i,j,k), &
 710              lambda(1:NDIM,i,j,k,1:HY_WAVENUM),   &
 711              leig   (1:NDIM,i,j,k,1:HY_WAVENUM,1:HY_VARINUM),&
 712              reig   (1:NDIM,i,j,k,1:HY_VARINUM,1:HY_WAVENUM) )
 ...
1039        !! ========== x-direction ==========================================
1040        ! YZ cross derivatives for X states
1041   838  call upwindTransverseFlux&
1042           (hy_transOrder,sig(DIR_Z,:,i,j-2:j+2,k),lambda(DIR_Y,i,j,k,:),leig(DIR_Y,i,j,k,:,:),&
1043           reig(DIR_Y,i,j,k,:,:),TransFluxYZ(:))
1044
1045        ! ZY cross derivatives for X states
1046   728  call upwindTransverseFlux&
1047           (hy_transOrder,sig(DIR_Y,:,i,j,k-2:k+2),lambda(DIR_Z,i,j,k,:),leig(DIR_Z,i,j,k,:,:),&
1048           reig(DIR_Z,i,j,k,:,:),TransFluxZY(:))
```

(a) Original code

```
 688        !! Left and right Riemann state reconstructions
 689    72  call hy_uhd_dataReconstOnestep&
 690             (blockID,blkLimitsGC,i,j,k,dt,del,&
 ...              ...
 709              sig    (1,1,i,j,k), &
 710              lambda(1,1,i,j,k),   &
 711              leig   (1,1,1,i,j,k),&
 712              reig   (1,1,1,i,j,k) )
 ...
1039        !! ========== x-direction ==========================================
1040        ! YZ cross derivatives for X states
1041   157  call upwindTransverseFlux&
1042           (hy_transOrder,sig(:,DIR_Z,i,j-2:j+2,k),lambda(1,DIR_Y,i,j,k),leig(1,1,DIR_Y,i,j,k),&
1043           reig(1,1,DIR_Y,i,j,k),TransFluxYZ(:))
1044
1045        ! ZY cross derivatives for X states
1046   174  call upwindTransverseFlux&
1047           (hy_transOrder,sig(:,DIR_Y,i,j,k-2:k+2),lambda(1,DIR_Z,i,j,k),leig(1,1,DIR_Z,i,j,k),&
1048           reig(1,1,DIR_Z,i,j,k),TransFluxZY(:))
```

(b) Optimized code

Figure 7: `vprof` profiles showing the impact of changing the array layout in the unsplit hydrodynamics solver.

the mesh variable. This change is too experimental to be used in the FLASH early science campaign.

## 4.2 DDT

The test DDT simulation originally took over 2 hours to initialize on 8192 MPI ranks on Mira BG/Q. We found that a significant amount of this time was spent reading turbulence field data from a small HDF5 file. This is extremely puzzling because the application code made use of HDF5 parallel I/O and requested collective I/O data transfers. We analyzed core files from a run that exceeded available wall-

```
360          h3x(i,j,w0t,w1t,w0mt,w1mt,w0d,w1d,w0md,w1md) =  &
361                 (eos_xf(i,j)     *w0d   +   eos_xf(i+1,j)     *w0md                    &
362               +  eos_xfd(i,j)    *w1d   +  eos_xfd(i+1,j)    *w1md) *w0t              &
363               + (eos_xf(i,j+1)   *w0d   +   eos_xf(i+1,j+1)  *w0md                    &
364               +  eos_xfd(i,j+1)  *w1d   +  eos_xfd(i+1,j+1)  *w1md) *w0mt             &
365               + (eos_xft(i,j)    *w0d   +  eos_xft(i+1,j)    *w0md                    &
366               +  eos_xfdt(i,j)   *w1d   + eos_xfdt(i+1,j)    *w1md) *w1t              &
367               + (eos_xft(i,j+1)  *w0d   +  eos_xft(i+1,j+1)  *w0md                    &
368               +  eos_xfdt(i,j+1)*w1d    + eos_xfdt(i+1,j+1)  *w1md) *w1mt

630          !!  electron + positron number densities
631   290    xnefer    = h3x(iat,jat, &
632                          si0t,    si1t,    si0mt,    si1mt, &
633                          si0d,    si1d,    si0md,    si1md)
```

(a) Original code

```
356          h3x(i,j,w0t,w1t,w0mt,w1mt,w0d,w1d,w0md,w1md) =  &
357                 (eos_tbl(i,j) % xf     *w0d   +   eos_tbl(i+1,j) % xf     *w0md          &
358               +  eos_tbl(i,j) % xfd    *w1d   +  eos_tbl(i+1,j) % xfd     *w1md) *w0t    &
359               + (eos_tbl(i,j+1) % xf   *w0d   +   eos_tbl(i+1,j+1) % xf   *w0md          &
360               +  eos_tbl(i,j+1) % xfd  *w1d   +  eos_tbl(i+1,j+1) % xfd   *w1md) *w0mt &
361               + (eos_tbl(i,j) % xft    *w0d   +  eos_tbl(i+1,j) % xft    *w0md          &
362               +  eos_tbl(i,j) % xfdt   *w1d   + eos_tbl(i+1,j) % xfdt    *w1md) *w1t    &
363               + (eos_tbl(i,j+1) % xft  *w0d   +  eos_tbl(i+1,j+1) % xft  *w0md          &
364               +  eos_tbl(i,j+1) % xfdt*w1d    + eos_tbl(i+1,j+1) % xfdt  *w1md) *w1mt

588          !!  electron + positron number densities
589    82    xnefer    = h3x(iat,jat, &
590                          si0t,    si1t,    si0mt,    si1mt, &
591                          si0d,    si1d,    si0md,    si1md)
```

(b) Optimized code

Figure 8: `vprof` profiles showing the impact of storing EOS lookup data in a derived datatype.

time and found that some MPI ranks were in the call-stack of the function `MPI_File_read_at` at job end-time. This is an independent MPI-IO function! We created wrapper functions around the HDF5 API functions `H5Pget_mpio_actual_io_mode` (HDF5 >= 1.8.8) and `H5Pget_mpio_no_collective_cause` (HDF5 >= 1.8.10) to help understand the I/O data transfer. The functions returned `H5D_MPIO_NO_COLLECTIVE` and `H5D_MPIO_DATATYPE_CONVERSION` indicating that a datatype conversion prevented a collective data transfer. This conversion happens because the HDF5 dataset contained little-endian data (`H5T_IEEE_F64LE`) but Blue Genes are big-endian machines. Converting the HDF5 dataset in the file to big-endian (`H5T_IEEE_F64BE`) fixed the performance issue. Figure 9 shows how the read time is affected by the endianness of the HDF5 data and the requested data transfer mode. The results clearly show that a collective I/O transfer with big-endian data is approximately 2 orders of magnitude faster than an independent I/O transfer. It is also clear that the failed collective I/O transfer with little-endian data gives nearly identical performance to an independent I/O transfer. For reference, a similar issue happens when converting from single-precision in memory to double-precision in file [12].

A large portion of time was also spent generating the initial positions for the Lagrangian tracer particles. We traced the slow-down to frequent calls to the `random_number` Fortran function, where we found many initial calls followed by a number of additional calls proportional to the MPI rank in order to give different random numbers on each MPI rank. We fixed the performance issue by removing all the unnecessary initial `random_number` calls which is valid because the only requirement in this application is that the initial density

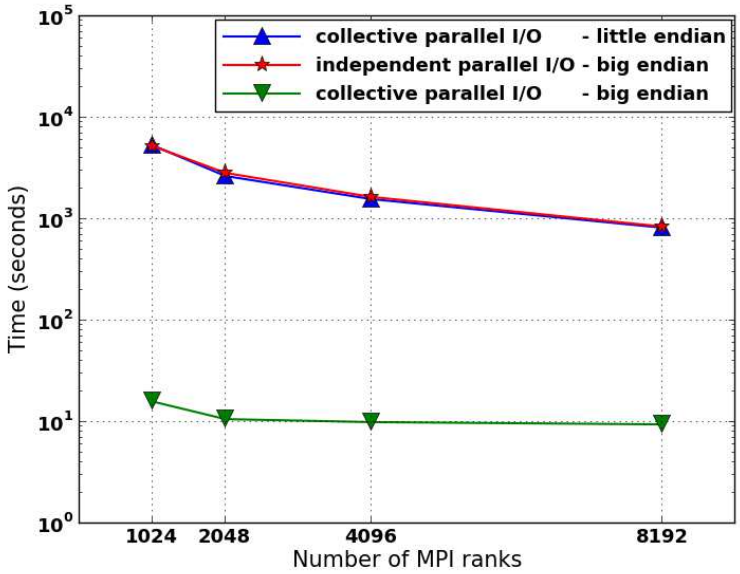



Figure 9: Time to read the turbulence field file on 64, 128, 256 and 512 nodes of Vesta BG/Q. The simulations are run with 16 MPI ranks per node in MPI-only configuration.

of particles is proportional to the gas density. We did not encounter the performance issue on other platforms because those runs used fewer MPI ranks and presumably the `random_number` function was faster. In a brief test we found a single call to `random_number` takes approximately $30ns$ on a x86_64 platform with `gcc-4.4.4` and approximately $4\mu s$ on both BG/P with `xlf-11.1` and BG/Q with `xlf-14.1`. The 2 orders of magnitude performance difference and greater number of MPI ranks explains why this workload was much slower on Blue Gene platforms. Note that this issue was never seen in RTFlame simulations because, here, particles are distributed regularly throughout the computational domain.

# 5 Performance

A crucial first step is to determine the most efficient way to run our applications of interest on the BG/Q architecture. This is quite a large search space because there are many variables such as the number of MPI ranks and OpenMP threads per node and the chosen FLASH multithreaded strategy. Figure 10 shows how the number of MPI ranks per node and OpenMP threads per MPI rank affect the time to solution in a fixed RTFlame problem on 128 nodes of Vesta BG/Q. The chosen RTFlame test problem makes use of AMR and provides an effective resolution of $256^3$ grid points. All data points are from runs using the thread within block multithreaded strategy because this was actually faster (evidence for this will be shown later). The figure shows that it is faster to run FLASH applications on BG/Q in hybrid MPI+OpenMP mode than it is

in MPI-only mode. There are two data points of particular interest in this figure at 32 MPI ranks per node with 2 OpenMP threads per MPI rank and 16 MPI ranks per node with 4 OpenMP threads per MPI rank. The 32 MPI ranks per node with 2 OpenMP threads per MPI rank is the FLASH configuration which gives the fastest time to solution, however, this is not an ideal configuration because it is tricky to fit the FLASH early science applications in 512 MB per MPI rank. The 16 MPI ranks per node with 4 OpenMP threads per MPI rank gives the best compromise between time to solution and memory usage. This configuration is being used for FLASH early science production runs and is used for all performance experiments in the remainder of this document.

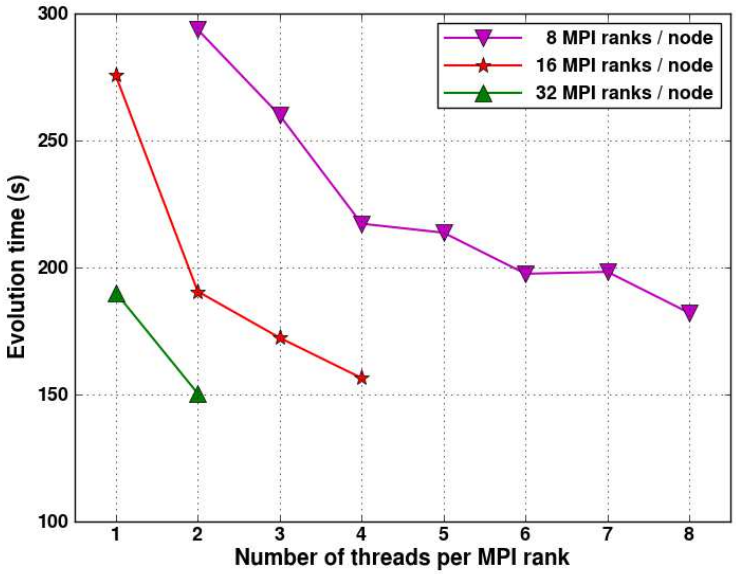



Figure 10: Time to complete 30 steps of a fixed RTFlame test problem on 128 nodes of Vesta BG/Q. The simulations were run with various numbers of MPI ranks and OpenMP threads and used the thread within block multithreading strategy.

The biggest memory consumer in these applications is the new unsplit hydrodynamics solver which requires approximately 2 - 2.5 times the memory of the old split hydrodynamics solver. This is not an inefficient implementation, rather it is the need to save information for all directions. There would be no issue with fitting in 32 MPI ranks per node if using the less accurate split hydrodynamics solver. Moving to 16 MPI ranks per node mode provides 1 GB per MPI rank and makes running FLASH much more comfortable. Many data buffers can be sized larger to accommodate rapid adaptive mesh refinement in regions of the domain and also congregation of many tracer particles on some MPI ranks.

Figure 11 shows how the FLASH multithreading strategy affects the strong scaling of the RTFlame test problem used in Figure 10. The problem initially has 21,462 blocks and 18,786 leaf blocks and is run on 512, 1024, 2048 and 4096 MPI ranks giving approximately 37, 18, 9 and 5 leaf blocks per MPI rank respectively. We do not provide a 8192 MPI rank data point because the load balancing rules in Paramesh

cause 0 leaf blocks, and therefore zero work, to be placed on some of these MPI ranks even though the average is approximately 2 leaf blocks per MPI rank. The figure shows that there is speedup with number of MPI ranks 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. There is better speedup for the finer-grained thread within block strategy. All performance experiments in the remainder of this document use the thread within block strategy.

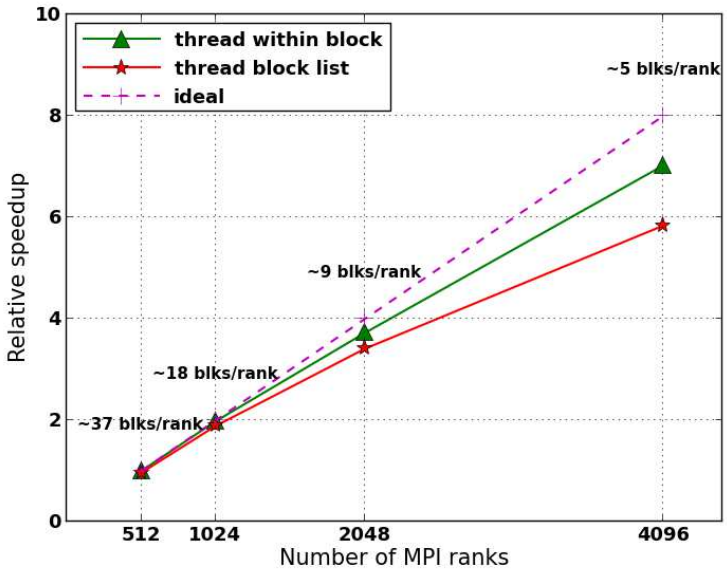



Figure 11: Relative speedup of a fixed RTFlame test problem on Vesta BG/Q. The simulations were run on different numbers of nodes with both FLASH multithreaded strategies but were always run with 16 MPI ranks per node and 4 OpenMP threads per MPI rank.

Figure 12 shows the strong scaling in RTFlame test problems of $256^3$, $512^3$, $1024^3$ and $2048^3$ effective resolution. The high resolution runs are of particular interest because previous production campaigns on BG/P studied configurations having $256^3$ and $512^3$ effective resolution. The largest FLASH run in this figure uses 32,768 nodes and 524,288 MPI ranks with 4 OpenMP threads per MPI rank. The strong scaling is generally good and fits with expectation: performance gets worse when there is less work per MPI rank and also when the resolution increases. The circled data points are from runs with the environment variable settings `PAMI_ALLREDUCE_REUSE_STORAGE=N`, `PAMI_ALLTOALL_PREMALLOC=N`, `PAMI_ALLTOALLV_PREMALLOC=N` and `PAMI_ALLTOALLW_PREMALLOC=N` to reduce the memory overhead. These settings may affect the performance but were necessary to make FLASH run despite an apparent leak of approximately 250 MB from the messaging layer (see Section 3.5). Figure 13 shows weak scaling of the same data. For reference, our early science production runs are using approximately 18 leaf blocks per MPI rank.

The performance advantage of running early science applications on BG/Q compared to BG/P can be summarized by a node-to-node ratio which is given by the expression $(T_p/T_q) \times (N_p/N_q)$, where T is run-

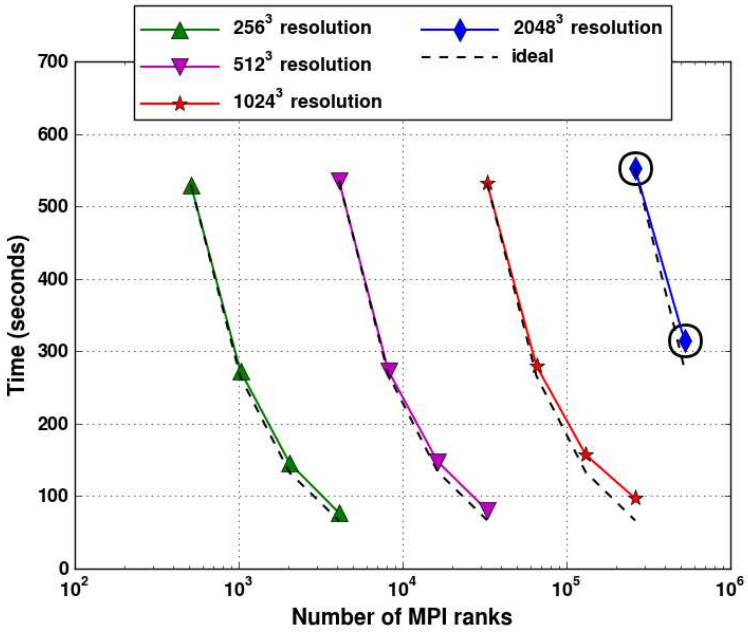



Figure 12: Strong scaling of RTFlame on Mira BG/Q for problems having $256^3$, $512^3$, $1024^3$ and $2048^3$ effective resolution. The simulations were run with 16 MPI ranks per node, 4 OpenMP threads per MPI rank and used the thread within block multithreading strategy.

time, N is the number of nodes, and q and p are subscripts indicating BG/Q and BG/P respectively. We run the fully-optimized early science applications in their most efficient configuration on both platforms, which is MPI-only in Virtual Node mode on BG/P and Hybrid MPI+OpenMP on BG/Q, and record FLASH evolution time as the application run-time. We obtain performance advantages of 8.9x (RTFlame) and 7.9x (DDT) which compares well with the ALCF target of 8x to 10x. Note that the FLASH performance advantage is actually slightly higher than shown because it is unrealistic to use the most efficient FLASH configuration on BG/P (i.e. Virtual Node mode) due to the memory footprint of the early science simulations.

Finally, we show HPCT performance counter information from an RTFlame test problem on 128 nodes of Vesta BG/Q in Figure 14. The FLASH application includes all optimizations up to "qhot(6)" (see Section 4.1) and counts were collected during FLASH evolution only. The performance data shows that the integer/load-/store/branch instructions dominate over floating point instructions (FXU=74.05% vs FPU=25.95%). The average weighted GFLOPS per node is 5.5 GFLOPS out of a possible 204.8 GFLOPS which means that this FLASH application is achieving 2.7% of peak floating point performance. This is quite low, however, there are many properties of FLASH applications which contribute to such a low fraction of peak performance: AMR introduces long-range communication and lots of control and integer code to focus floating point computation where it is needed, table lookups in EOS avoid heavy floating point calculations to improve time to solution, and tracer particles add to the total communication without increasing floating point work. The instructions completed per cycle per core is good (0.545) and there are a high percentage of hits in the L1 data cache (92.86% on node 0). Overall this FLASH application is using the BG/Q cores relatively efficiently.

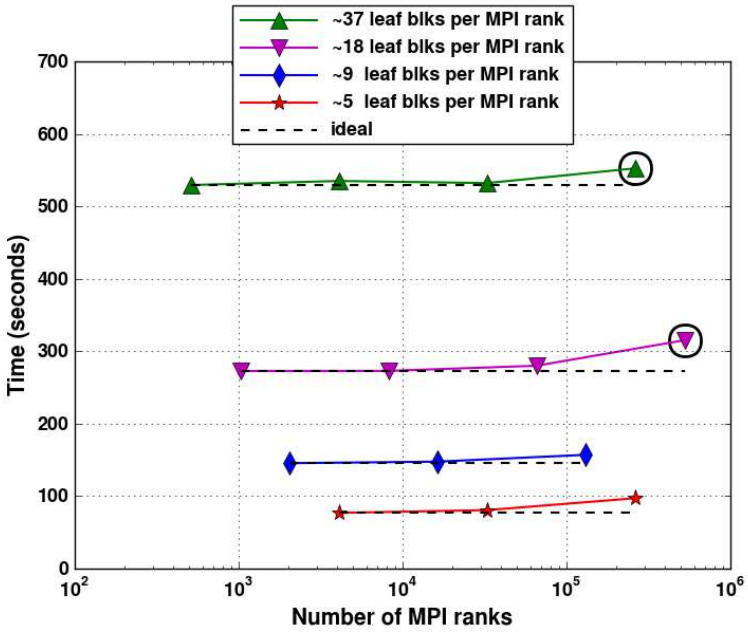



Figure 13: Weak scaling of RTFlame on Mira BG/Q. The number of leaf blocks per MPI rank is kept approximately constant in problems having $256^3$, $512^3$, $1024^3$ and $2048^3$ effective resolution. The simulations were run with 16 MPI ranks per node, 4 OpenMP threads per MPI rank and used the thread within block multithreading strategy.

# Acknowledgments

Thanks to Vitali Morozov for all his help throughout the project. He ported the FLASH RTFlame simulation to BG/Q, provided support for the HPCT libraries, and helped with debugging and performance optimization of FLASH. Thanks to Dean Townsley for providing FLASH RTFlame and DDT simulations and for answering any questions about these simulations. Thanks also to George Jordan for explaining Flash Center science objectives and Anshu Dubey for advising about the layout of this report. The software used in this work was in part developed by the DOE NNSA-ASC OASCR Flash Center at the University of Chicago. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

```
FLASH_evolution, call count = 1, avg cycles = 364213522553, max cycles = 364213557651 :
  -- Counter values for processes in this reporting group ----
    min-value  min-rank    max-value  max-rank    avg-value   label
  3.415637e+09      209   4.935353e+09     1822   4.462556e+09  Committed Load Misses
  5.638317e+10      209   6.779761e+10     1601   6.160317e+10  Committed Cacheable Loads
  2.744730e+09      209   4.035232e+09     1822   3.599715e+09  L1p miss
  1.409032e+11      209   1.583207e+11     1712   1.469828e+11  All XU Instruction Completions
  3.923771e+10      209   6.047476e+10     1822   5.151986e+10  All AXU Instruction Completions
  5.949847e+10      209   9.316478e+10     1822   7.853280e+10  FP Operations Group 1

Histogram of floating-point operation counts:
    flop-bin      #ranks
  5.949847e+10          1
  6.190321e+10          0
  6.430794e+10          0
  6.671268e+10         74
  6.911742e+10         29
  7.152215e+10         13
  7.392689e+10          5
  7.633163e+10       1046
  7.873636e+10        240
  8.114110e+10        116
  8.354584e+10        354
  8.595057e+10         85
  8.835531e+10         56
  9.076005e+10         13
  9.316478e+10         16

Derived metrics for code block "FLASH_evolution" averaged over process(es) in the reporting group
Instruction mix:  FPU = 25.95 %,  FXU = 74.05 %
Instructions per cycle completed per core = 0.5450
Per cent of max issue rate per core = 40.36 %
Total weighted GFlops = 706.553228
```

(a) Summary counts over all nodes

```
FLASH_evolution, call count = 1, avg cycles = 364210026210, max cycles = 364213555879 :
  -- Counter values summed over processes on this node -----
0       71127700508   Committed Load Misses
0      996543943304   Committed Cacheable Loads
0       57759693733   L1p miss
0     2370956187483   All XU Instruction Completions
0      827216441627   All AXU Instruction Completions
0     1261426933713   FP Operations Group 1
  -- L2 counters (shared for the node) ------------------
100    569692088884  L2 Hits
100      7121203216  L2 Misses
100      8169110557  L2 lines loaded from main memory
100      6253069690  L2 lines stored to   main memory


Derived metrics for code block "FLASH_evolution" averaged over process(es) on node <0,0,0,0,0>:
Instruction mix:  FPU = 25.87 %,  FXU = 74.13 %
Instructions per cycle completed per core = 0.5488
Per cent of max issue rate per core = 40.69 %
Total weighted GFlops for this node = 5.541
Loads that hit in L1 d-cache =  92.86 %
                 L1P buffer =   1.34 %
                 L2 cache   =   5.08 %
                 DDR        =   0.71 %
DDR traffic for the node: ld = 2.871, st = 2.198, total = 5.069 (Bytes/cycle)
```

(b) Counts on node 0

Figure 14: Performance counter data from an RTFlame test problem on 128 nodes of Vesta BG/Q. The simulation was run with 16 MPI ranks per node, 4 OpenMP threads per MPI rank and used the thread within block multithreaded strategy. Counts were only collected during FLASH evolution.

# References


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

[2] OpenMP Architecture Review Board. OpenMP application program interface version 3.1. `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`, July 2011.

[3] IBM. *IBM XL Fortran for Blue Gene/Q, V14.1. Optimization and Programming Guide*. IBM, 2012.

[4] Statistics for memory allocation with malloc. `http://www.gnu.org/software/libc/manual/html_node/Statistics-of-Malloc.html`, February 2013.

[5] V. Morozov. Blue Gene/Q Tuning Early Experience. `http://www.alcf.anl.gov/sites/www.alcf.anl.gov/files/morozov-bgqtuning-early.pdf`, March 2012. Slides presented at the ESP March Workshop "Code for Q", Argonne National Laboratory.

[6] The GNU C library. Resource usage. `http://www.gnu.org/software/libc/manual/html_node/Resource-Usage.html`, February 2013.

[7] A. Dubey, A.C. Calder, C. Daley, R.T. Fisher, C. Graziani, G.C. Jordan, D.Q. Lamb, L.B. Reid, D.M. Townsley, and K. Weide. Pragmatic optimizations for better scientific utilization of large supercomputers. *International Journal of High Performance Computing Applications*, to appear. Published online 21 November 2012 `http://hpc.sagepub.com/content/early/2012/11/20/1094342012464404`.

[8] Determining memory use. `https://www.alcf.anl.gov/resource-guides/determining-memory-use`, February 2013.

[9] Allocation debugging. How to install the tracing functionality. `http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc`, February 2013.

[10] L. Killough. Optimizing Single-Node Performance on BlueGene. `http://press.mcs.anl.gov/gswjanuary12/files/2012/01/Optimizing-Single-Node-Performance-on-BlueGene.pdf`, January 2012. Slides presented at the ALCF Winter Workshop, Argonne National Laboratory.

[11] The Flash Center for Computational Science at the University of Chicago. *FLASH Users Guide. Version 4.0.*, September 2012.

[12] 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*, 5(1):015001, 2012.


**Argonne Leadership Computing Facility**
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov