LA-UR-13-23339
Approved for public release; distribution is unlimited.

Title:	PENNANT: An Unstructured Mesh Mini-App for Advanced Architecture Research
Author(s):	Ferenbaugh, Charles R.
Intended for:	Next Generation Platforms Working Group telecon with LLNL, 5/14/2013
Issued:	2013-05-08

PENNANT: An Unstructured Mesh Mini-App for Advanced Architecture Research

Charles R. Ferenbaugh
Scientific Software Engineering, HPC-1

Next Generation Platforms Working Group
May 14, 2013
LA-UR-13-XXXXX

Motivation

- HPC is undergoing a major transition
  - New architectures: multi-core, GPU, Intel MIC, BlueGene, ...
  - Designed for massively parallel processing
  - “Memory access is expensive, flops are free”
  - Application codes must be redesigned, reimplemented
- One tool for understanding new architectures: *mini-apps*
  - Small, standalone apps that capture the algorithms, memory patterns in a problem domain of interest
  - Can be used to model performance, test optimization ideas for larger apps
  - Can be easily adapted to new hardware types, programming models, ..., because of their small size

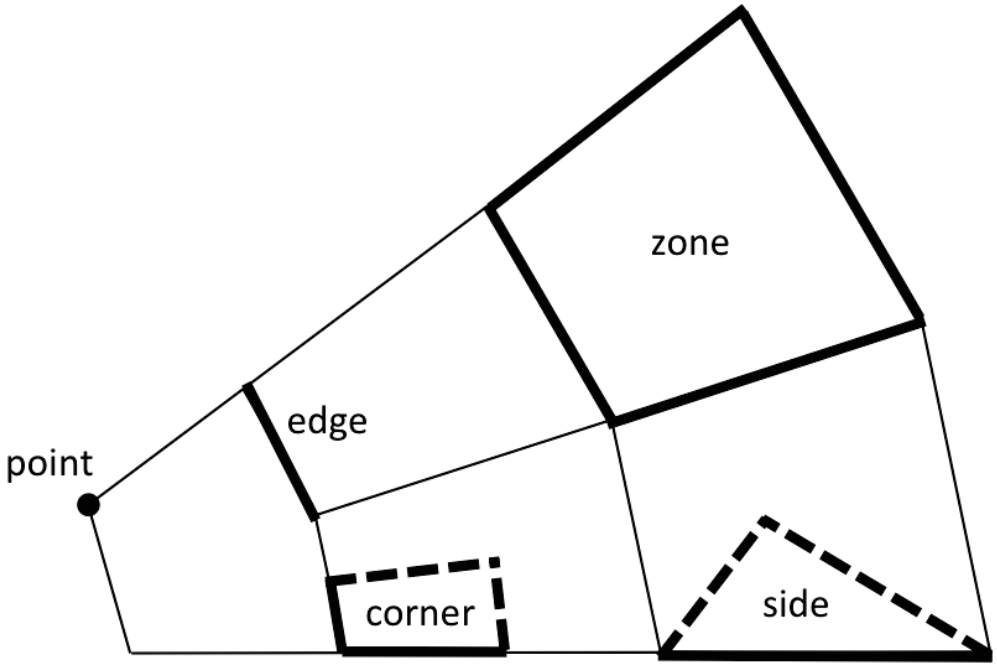
Motivation (2)

- Unstructured mesh apps usually aren't well represented in studies of new architectures
  - Roadrunner demonstration apps (2006-8) implemented five typical apps
  - All were structured mesh or particle apps; none used unstructured meshes
  - Similarly for Titan at Oak Ridge (2011-12)
- But unstructured mesh codes play an important role at LANL and elsewhere
  - FLAG at LANL, KULL and ALE3D at LLNL, ...
- So we need to study unstructured mesh codes earlier in the design/deployment process for new architectures
- An unstructured mesh mini-app might help

PENNANT overview

- Implements a small subset of FLAG physics
- Includes simple input parser, mesh input, viz output
- Uses 2D cylindrical geometry
- Operates on general unstructured meshes (arbitrary polygons)
- Contains just over 2200 lines of C++ source code
- Has complete serial, multicore (OpenMP), and GPU (CUDA) implementations
- Internal release at LANL: April 2012
- Open-source release to wider community: January 2013

Mesh data structures



- **Points, edges, zones** are mesh elements in 0, 1, and 2 dimensions respectively
- **Sides and corners** are subregions within a zone

Mesh data structures (2)

- Associations between points, zones, etc., are irregular; must be given by explicit connectivity arrays
- Because of the irregular connectivity, unstructured mesh physics can be hard to optimize
  - Mesh is hard to divide into independent chunks for parallel processing
  - Memory systems are sometimes not optimized for non-contiguous access
  - Efficient array lookup instructions (e.g., vector gather/scatter) are not always available in hardware

Physics details

- PENNANT implements the following from FLAG:
  - Compatible Lagrangian staggered grid hydrodynamics (SGH)
  - Single material, gamma-law gas equation of state
  - Temporary Triangular Subzoning (TTS) for subzonal pressures
  - Campbell-Shashkov tensor artificial viscosity
- This is just enough to run a few interesting test problems
- To keep PENNANT small, I didn't include:
  - arbitrary Lagrangian-Eulerian (ALE) methods
  - multiple material models, mixtures
  - radiation diffusion

Physics details (2)

- PENNANT physics algorithms are *staggered-grid*
  - Mesh positions, velocities are stored on points
  - Most state variables (density, pressure, ...) are stored on zones
  - So we must frequently use values of zone-based variables to compute point-based results, or vice versa
    - This is done by computing some intermediate values on sides and corners
  - This means we will be using the connectivity arrays frequently (irregular access!)

Optimizations: SIMD vectorization

- On most modern architectures, need some form of SIMD vectorization to achieve peak performance
- FLAG typically has inner loops over dimension:

```
real*8 pos(dim, numpts), pos0(dim, numpts),
       vel(dim, numpts)
do p = 1, numpts
  do d = 1, dim
    pos(d, p) = pos0(d, p) + vel(d, p) * dt
  end do
end do
```

- This keeps the outer `p` loop from vectorizing!

Optimizations: SIMD vectorization (2)

- If `dim` is fixed, we can unroll the inner loop by hand
- For example, if `dim` is 2:

```
real*8 pos(dim, numpts), pos0(dim, numpts),
       vel(dim, numpts)
do p = 1, numpts
  pos(1, p) = pos0(1, p) + vel(1, p) * dt
  pos(2, p) = pos0(2, p) + vel(2, p) * dt
end do
```

- This loop will vectorize

Optimizations: SIMD vectorization (3)

- PENNANT handles this by defining a `double2` structure
- `double2` has inline operators for $=$, $+$, $\times$, ...
- So PENNANT code would look like this:

```
double2 *pos, *pos0, *vel;
for (int p = 0; p < numpts; ++p) {
  pos[p] = pos0[p] + vel[p] * dt
}
```

- To the compiler, this looks about the same as the code on the previous page, so it vectorizes too
- As a bonus, the code is shorter and easier to read

Optimizations: High-level parallelism

- Divide point, side, and zone lists into (nearly) independent chunks
- For each stage of the main hydro algorithm, process chunks (point, side, or zone) in parallel
  - OpenMP uses a `parallel for` loop
  - CUDA uses a grid with one grid block per chunk
- Add special handling for places where chunks aren't quite independent
  - Summing corners to points (force calculation)
  - Taking a global minimum over sides (timestep)

Optimizations: Memory locality

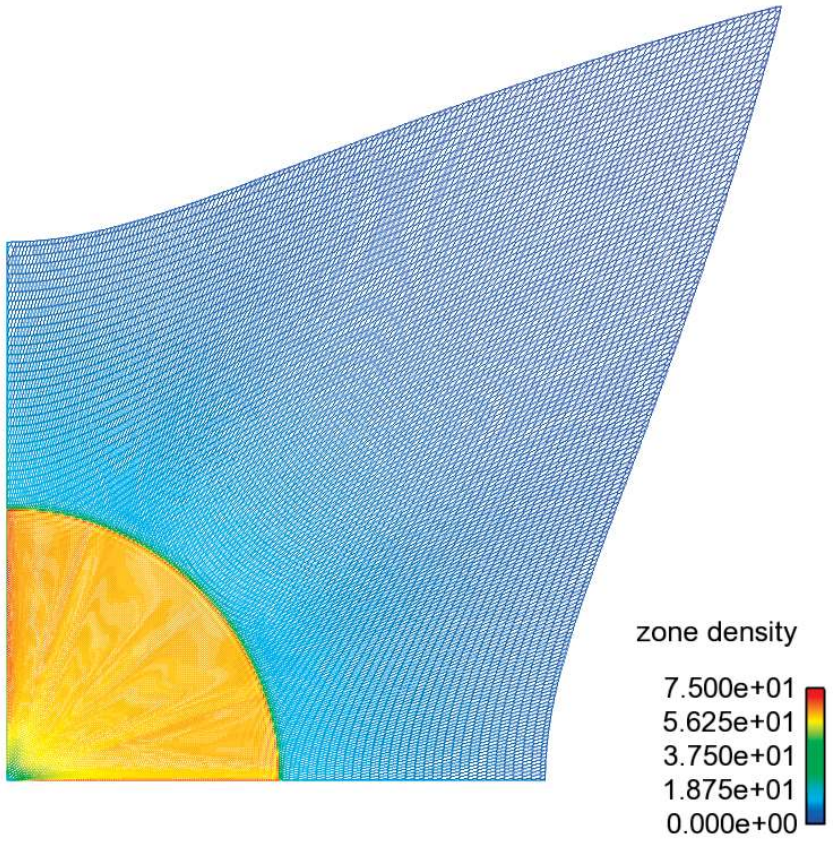
- For best performance, we'd like to have most data accesses from a fast local memory, rather than slower global memory
- On CPU, L1 or L2 cache will do this for us
- Newer NVIDIA GPUs have similar caches
- PENNANT relies on cache to improve memory performance
  - Each processor operates on a separate chunk, so cache data is (mostly) independent
  - Cache use helps minimize the penalty for irregular access

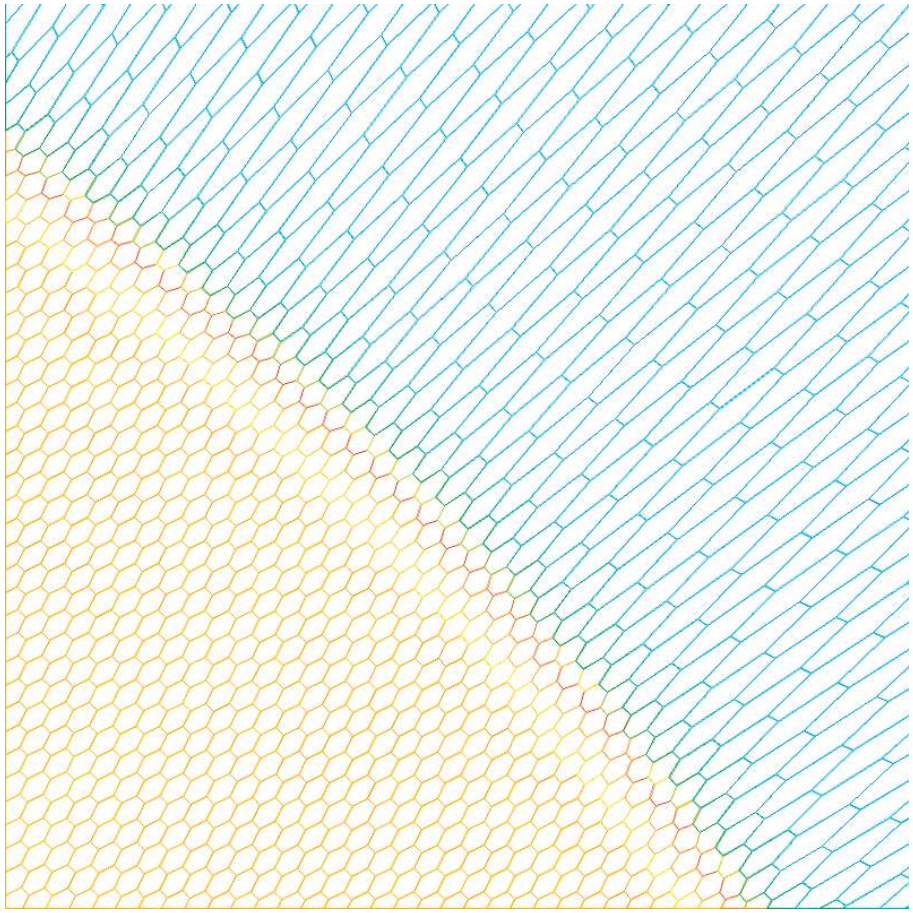
PENNANT test problems

test name	# zones	# cycles	mesh type
nohsquare	32400	4055	structured, all square zones
nohpoly	22801	6817	unstructured, hexagon zones
sedov	72900	1909	structured, all square zones
leblanc	57600	1937	structured, all square zones

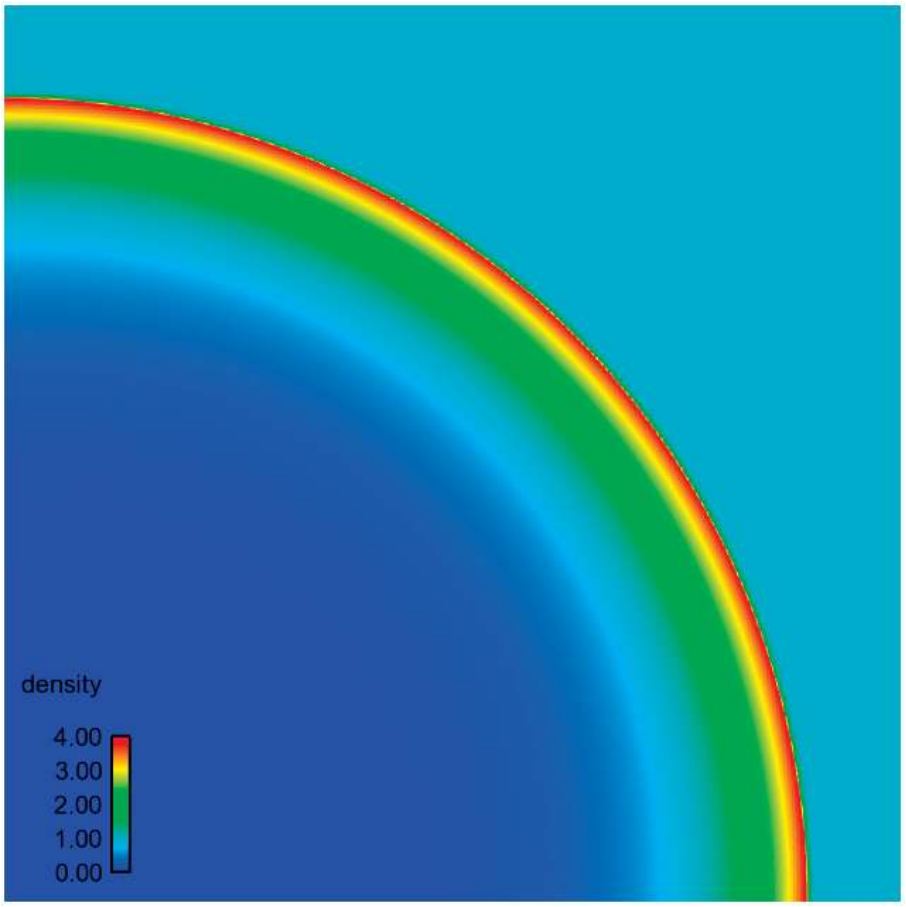
- Problem sizes are chosen such that (# zones $\times$ # cycles) is similar across all problems
- PENNANT distribution includes all of these tests, plus some smaller versions for debugging

PENNANT test problems: Noh





PENNANT test problems: Sedov, Leblanc



Los Alamos NATIONAL LABORATORY EST. 1943

UNCLASSIFIED

NNSA

Test platforms

Moonlight:

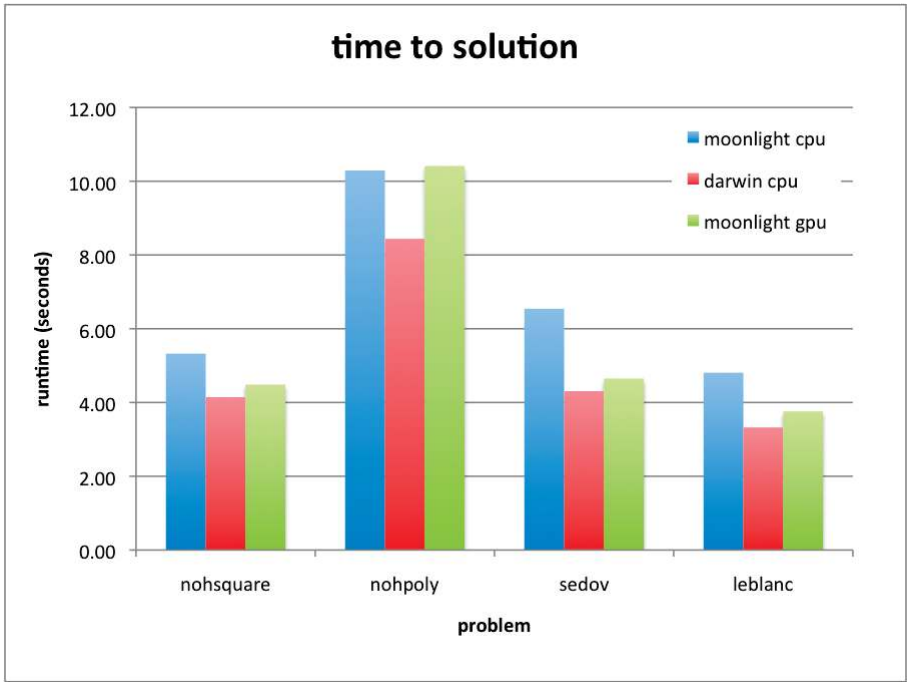
- Two 8-core Intel Xeon E5-2670s (Sandy Bridge) per node
- 16 cores/threads per node total
- Each core runs at 2.60 GHz
- Two NVidia M2090 GPUs per node (for these tests only one GPU was used)

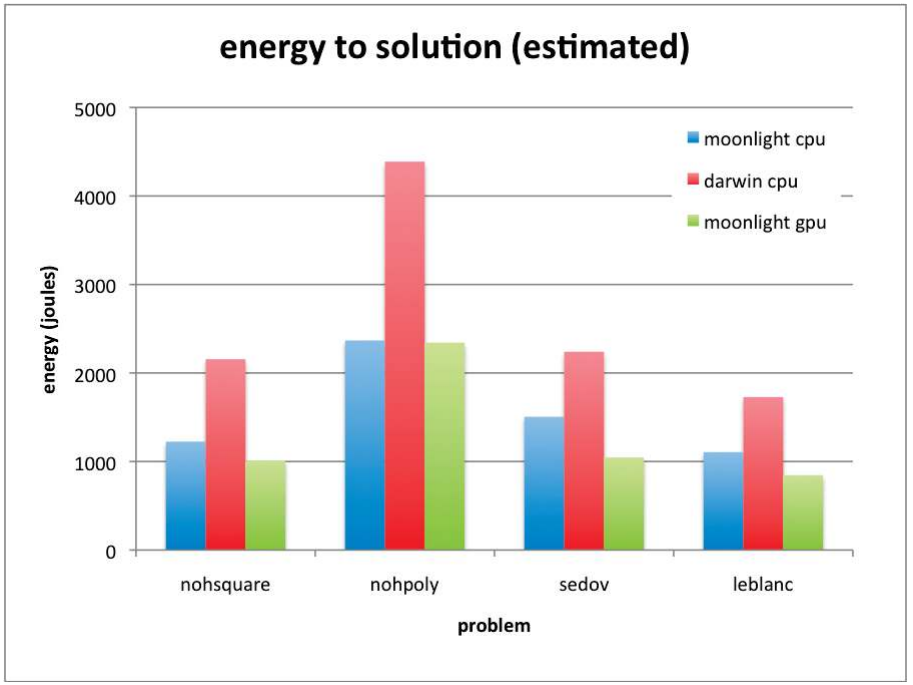
Darwin:

- Four 8-core Intel X6550s (Nehalem) per node
- 32 cores per node total
- Each core runs at 2.00 GHz
- Cores have hyperthreading enabled, allowing for a total of 64 OpenMP threads

All CPU timings use all available threads, except as noted

Basic results - timing and energy



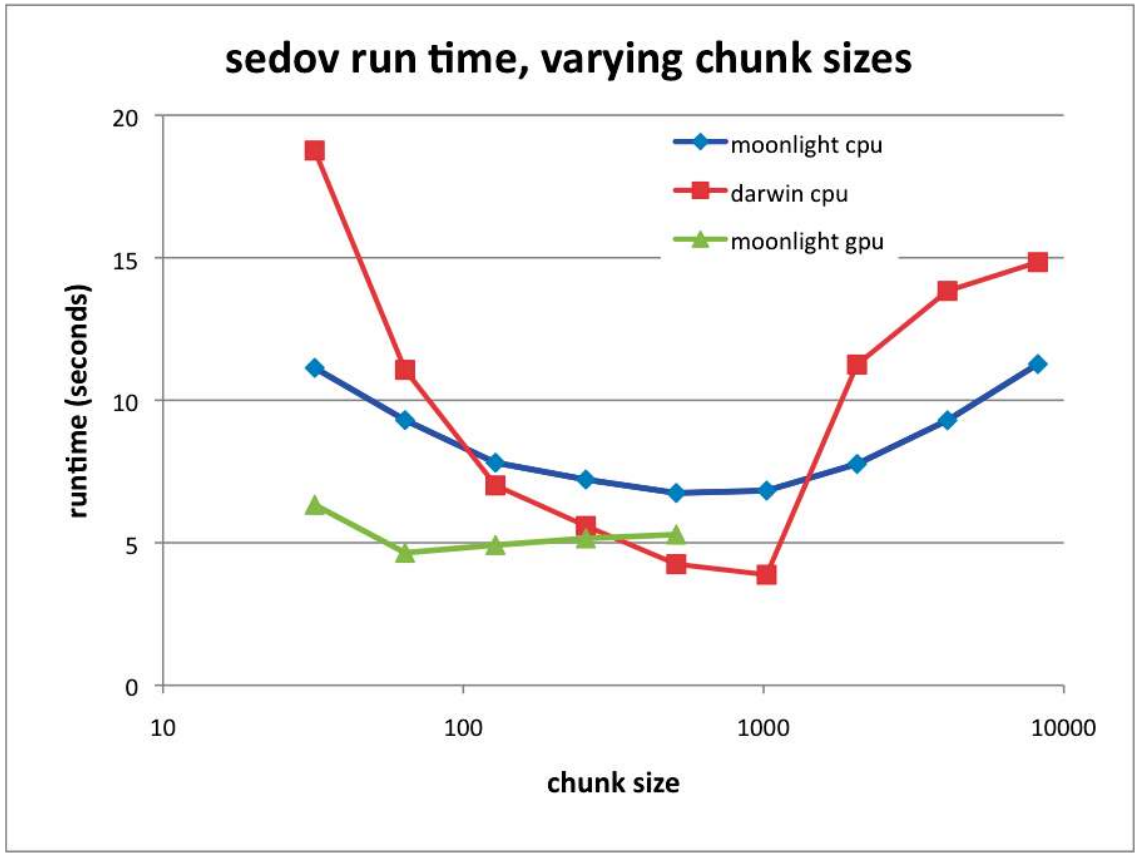


- Darwin CPU gives shortest time to solution (though not by too much)
- But Darwin uses the most energy by far; Moonlight GPU or CPU is more energy-efficient

Optimal chunk size

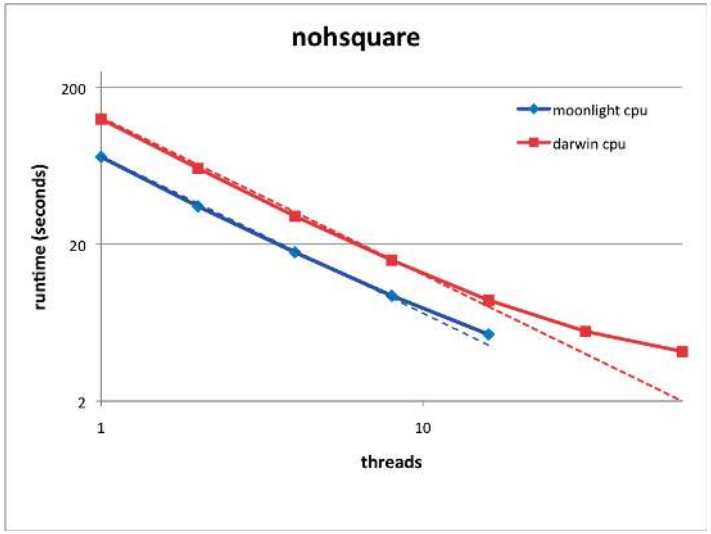
- If chunks are too large, performance will be bad...
  - won't fit in cache
  - load imbalance
- But if chunks are too small, performance will be bad...
  - more dependencies, synchronization between chunks
  - on the CPU, more time spent in loop startup, shutdown
- There should be a "sweet spot" between the minimum and maximum possible chunk size – can we find it empirically?

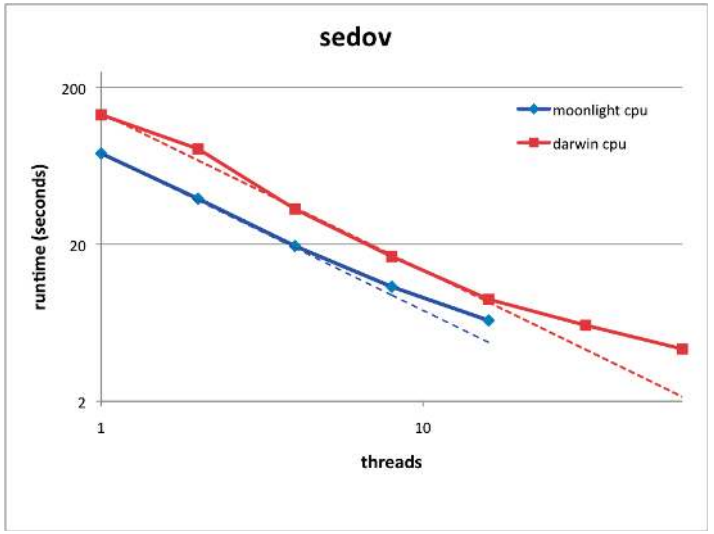
Optimal chunk size (2)

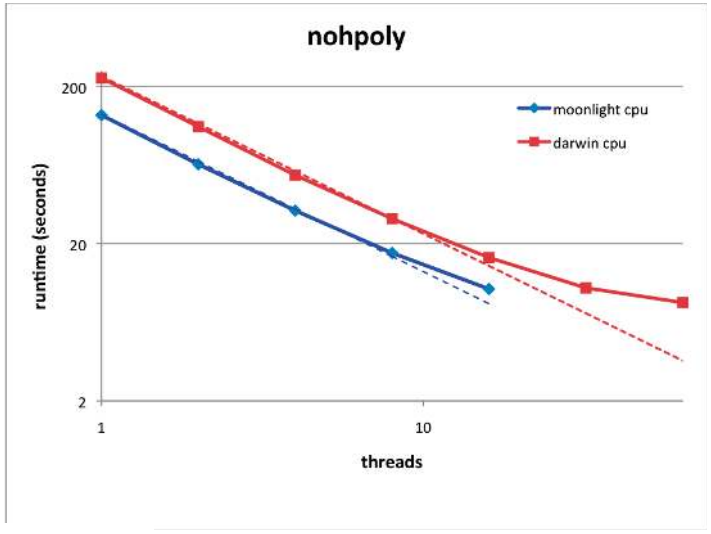


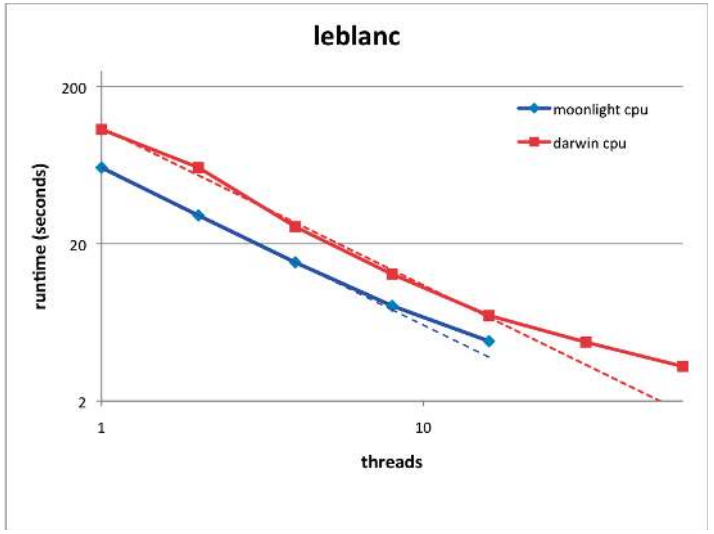
Based on these results, all other tests use 512 for CPU, 64 for GPU

Scaling studies - OpenMP









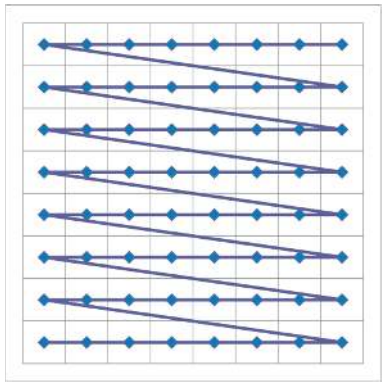
Scaling studies - OpenMP (2)

- Good scaling on 16 threads of Moonlight (12-14x speedup)
- Good scaling on 32 threads of Darwin (22-23x)
- Starts to level off at 64 threads of Darwin (27-33x)

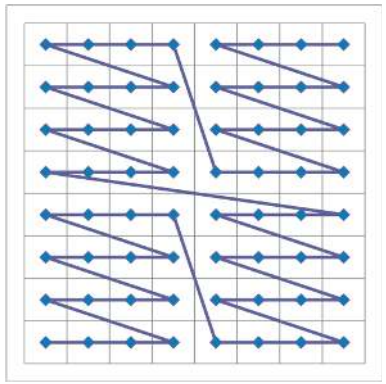
This suggests that we've (mostly) succeeded in making the chunks independent

Mesh renumbering for memory locality

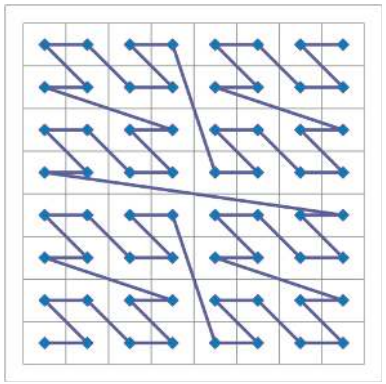
- On GPUs, memory access is optimized for the case when threads access memory locations that are contiguous (or nearly so)
- CPUs are similar (cache line access)
- So can we improve memory performance by renumbering?



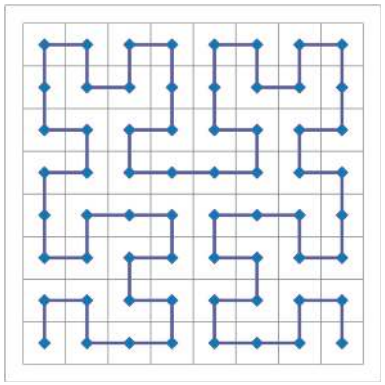
row-column



block

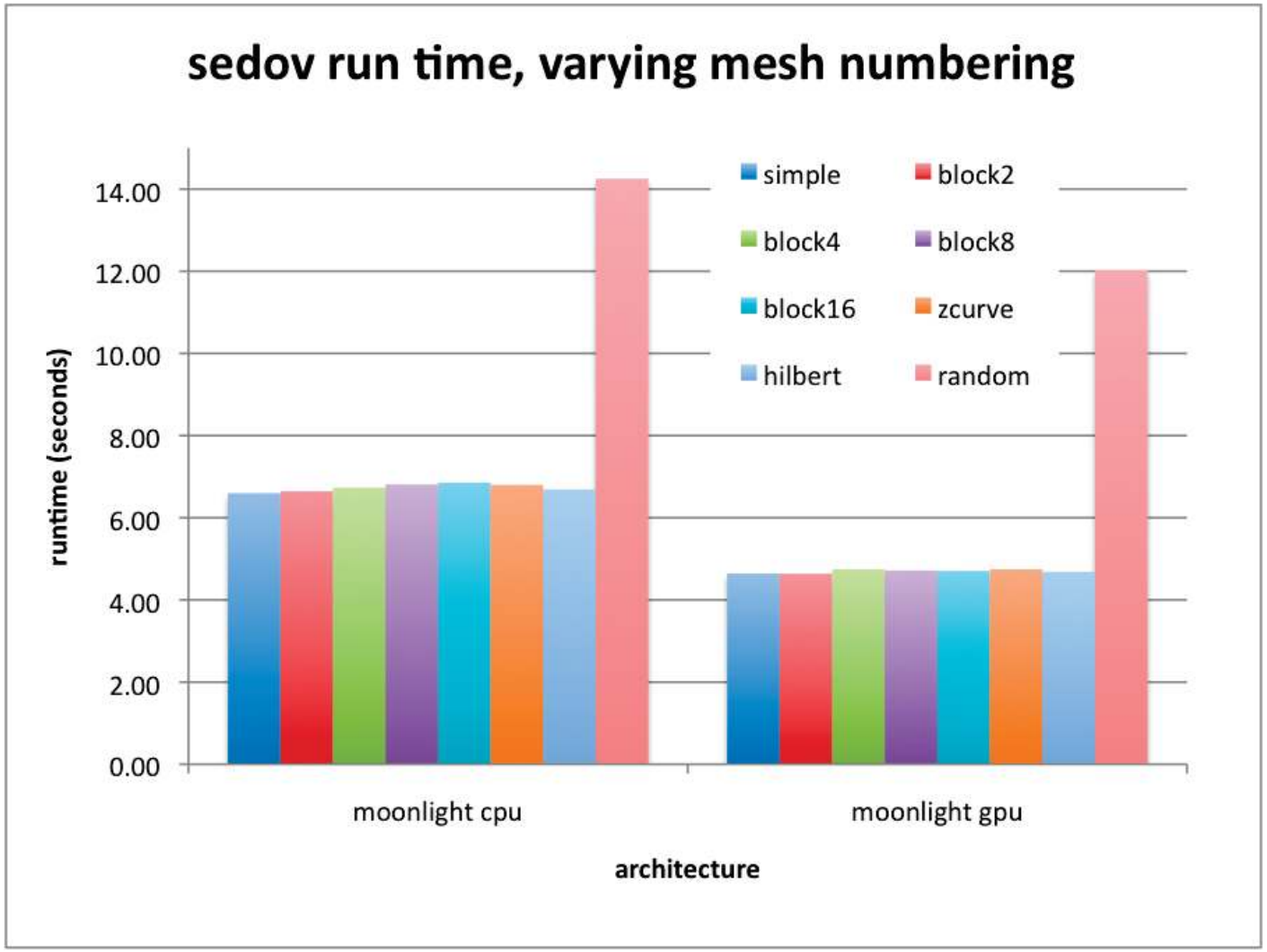


z-curve



Hilbert curve

Mesh renumbering for memory locality (2)



Mesh renumbering for memory locality (3)

- Random numbering gives awful performance – no surprise
- Other schemes are very close together (3-4%)
- Apparently any numbering scheme with some locality gives about the same performance

Conclusions

PENNANT is:

- able to reproduce the basic functionality of the much larger FLAG code
- usable as a testbed for new architectures, programming models, optimization ideas
- able to do high-level parallelization of unstructured mesh problems effectively

Future work

- Implement MPI version of PENNANT
- Test PENNANT in a FLAG-like mode (MPI-only, no chunking) and compare against other PENNANT versions
- Test more rigorously how well PENNANT can predict FLAG performance
- Implement MPI+OpenMP, MPI+CUDA versions
- Implement other GPU versions (OpenCL and/or OpenACC)
- Test on Intel MIC (using OpenMP version)
- Test alternate strategies for corner-to-point gathers (edge-coloring)
- Make PENNANT more widely available for other co-design efforts

Acknowledgements

Thanks to:

- Mikhail Shashkov and the ASCR “Mimetic Methods for PDEs” project, and the ASC Hydrodynamics project, for providing support for this work
- the many Lagrangian Applications Project members who have contributed to the FLAG code; parts of the PENNANT code and documentation are adapted from their work
- the Intel EPOCH workshop, June 2012, for optimization ideas