

Mulard: A Multigroup Radiation Diffusion Compact-App

Using Application Proxies for Co-Design of Future HPC Computer Systems and Applications, Supercomputing 2012

November, 2012

Thomas A. Brunner



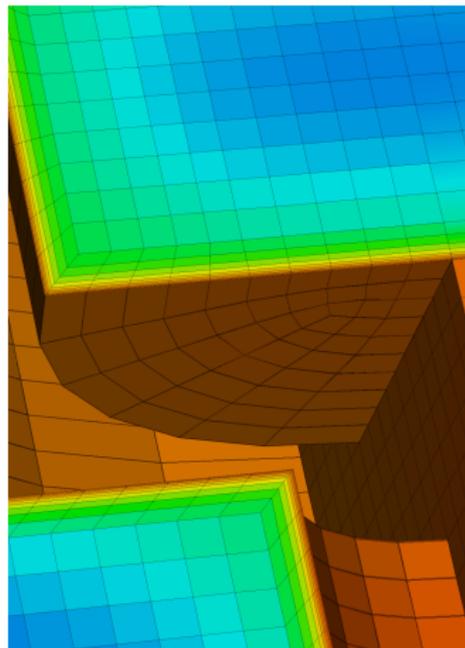
LLNL-PRES-581073

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC



Production codes have lots of features that make them difficult to optimize

- We use unstructured meshes as a form of adaptivity, greatly reducing our zone counts
- Multiple physics packages run (usually) sequentially
- Problem specification is completely abstract, often behind opaque function calls
- The code often has multiple algorithm choices for different problems
- Code is maintained by physics experts, not performance experts



Material interfaces and gradients are resolved with unstructured mesh

Mulard is written to emulate a production code

- Collection of loosely coupled classes manage complexity of algorithms
- Supports running several different problems in 2D and 3D
- Adds extra opportunity for parallelism by solving multiple equations together
- Meant to be easily understandable and flexible more than high performance

Goals for Mulard

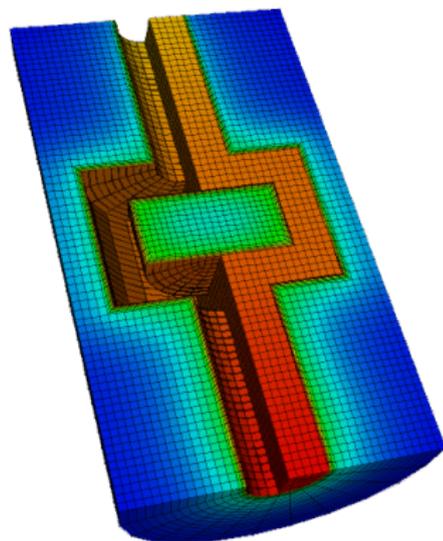
- Explore on-node parallelism (threads, GPU, etc.)
 - Explore data storage layouts for optimal performance
 - More flexible than the production codes
 - Have some of the complications of the production code
-
- Mulard letters come from 'MULTigroup RAdiation Diffusion'
 - A mulard is a sterile, hybrid duck raised for food

High energy-density physics experiments are dominated by energy transfer between radiation and material

- Photons of different frequencies, E_g diffuse through material at different rates, $\nabla \cdot D_g \nabla E_g$
- Each group of photons deposits energy in the material (u) at the rate $\sigma_g E_g$
- Material radiates photons into each group, $\sigma_g B_g(T(u))$

$$\frac{\partial E_g}{\partial t} - \nabla \cdot D_g \nabla E_g = \sigma_g [B_g(T) - E_g] + S_g$$

$$\frac{\partial u}{\partial t} = \sum_g \sigma_g [E_g - B_g(T)] + Q,$$



Radiation flows through a pipe,
heating material

The discretization is highly parallelizable

for all zones do

for all integration points do

compute basis functions and gradients
transform to real space

for all groups do

compute material properties

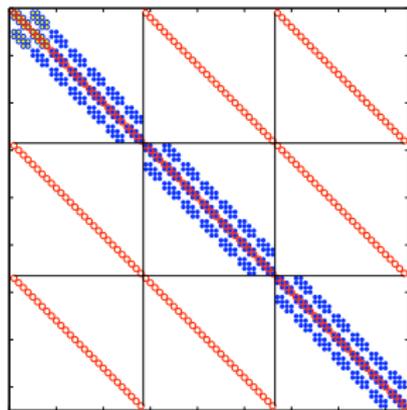
accumulate to local **submatrix**

accumulate **submatrix** into **global matrix**

for all groups do

solve **linear system** for E_g

add contribution of E_g to **material**



Global sparse matrix structure for 3 groups. Each group (box) forms a **diffusion matrix**, **coupled** to other groups locally.

- First three nested loops can be computed in any order
- At inner most level, vectorized code should be possible

Mulard comes with three levels of applications

All use MFEM, an open source finite element library

- Self-contained and easy to use
- <http://code.google.com/p/mfem/>
- Mulard is the full featured, multigroup code
 - Solves different problems via an abstract material interface to test difference performance issues and be slightly more production-like
 - Abstracts details of solvers and matrix storage, making it easy to switch them out
 - Runtime selection of finite element order and other algorithmic options
- Duckling solves only one problem
 - Has many algorithmic run-time options
 - Has lots of code for calculating the quality of the solution.
- Hatchling is similar to Mantevo's miniFE.
 - Integrates the finite elements over the mesh
 - Solves the matrix

The different applications vary considerably in code size

Code	Files	Lines	Comments
MFEM	107	39,468	4,399
Mulard	26	4383	1,299
Duckling	3	969	367
Hatchling	1	430	170

- We use a small portion of MFEM
 - Unstructured mesh management (reading, storing, accessing)
 - Finite element operations
 - Sparse linear algebra storage and solvers.
- Many components are independent; for example, it is easy to switch all the global sparse matrix stuff out for your own.

Code Tour

- Where to find it:

<http://portal.nersc.gov/project/training/files/SC12/Mulard/>

- Doxygen:

<http://portal.nersc.gov/project/training/files/SC12/Mulard/html>

- How to build it: On NERSC:

- `module load cmake/2.8.9`

- `cp -R /global/project/projectdirs/training/2012/SC12/Mulard .`

- How to run it: `./ReadAndRunMe.sh`

- A quick tour of Hatchling

- A quick tour of Mulard

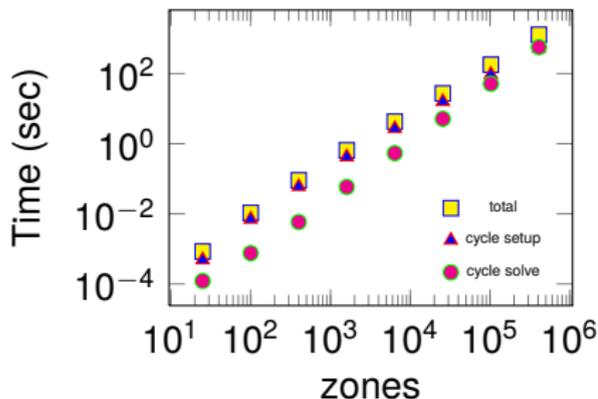
- You can use Visit to look at the simulation output:

<http://visit.llnl.gov>

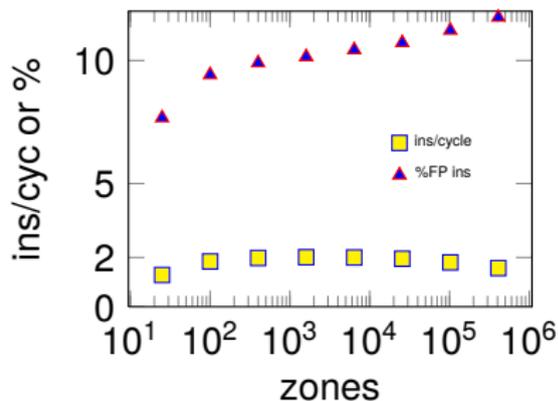
Real work increases with mesh size

(These are really old numbers)

Run times

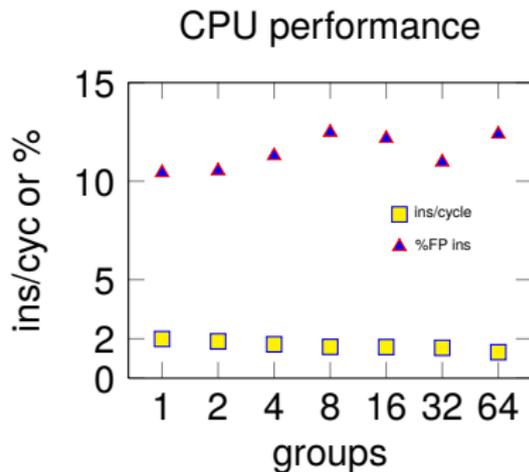
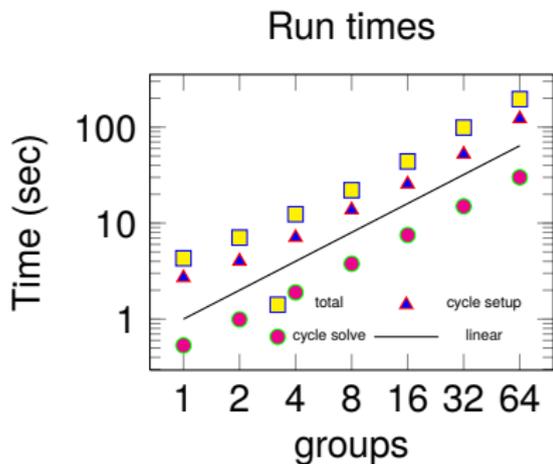


CPU performance



- As mesh size increases
 - Number of time steps also increases
 - Percentage of floating point instructions goes up
 - Saturate memory bandwidth (IPC goes down)
- In each cycle, setup time shouldn't be higher than solve time
 - But there are some known optimizations in production code

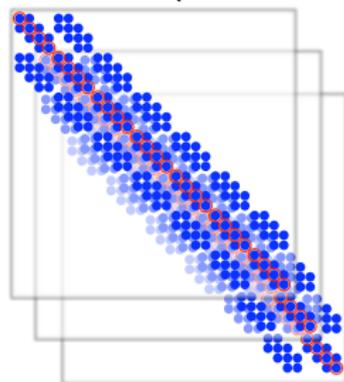
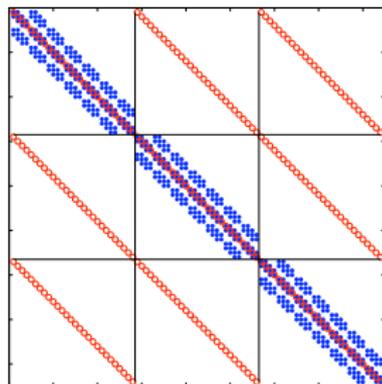
As groups increase, everything grows roughly linearly



- Instructions per cycle goes down as bandwidth saturates
- Floating point instruction fraction is roughly constant.

Opportunities for code exploration

- Explore on-node parallelism: OpenMP, CUDA, TBB, etc.
- Assemble matrices in parallel
- Solve matrices in parallel
- Reorder data to store block diagonal matrices
- Reorder loops
- Rewrite loops to use raw data instead of nice interface
 - How much overhead are we paying for “maintainable” code?
- Explore more advanced algorithms
 - Nonlinear coupling instead of linearized.
 - Which finite element order?



Conclusions

- Mulard scales well with zones or groups
- Designed to explore trade-offs of on-node parallelism (OpenMP, CUDA) in a more production-like code
- There are many levels of parallelism to explore with multiple equations to solve

See Also

- The library we're based on:
<http://code.google.com/p/mfem/>
- Links to all LLNL mini-apps: <http://codesign.llnl.gov>
- <https://computation.llnl.gov/casc/ShockHydro/>