Reaction Networks with OpenACC

Michael Zingale, Adam Jacobs, & Max Katz
(Stony Brook)

(+ enormous help from Oscar Hernandez @ OLCF)

Reacting Flow & Splitting

- We want to solve an advective-diffusion-reaction system that looks like

$$\phi_t = -A(\phi) + D(\phi) + R(\phi)$$

- Standard approach: operator splitting
  - Treat each process independent of the others
  - Strang: switch the order of operations each step to get 2nd order accuracy
- Ex: advection-reaction: $\phi^{n+1} = R_{\Delta t/2} A_{\Delta t} R_{\Delta t/2} \phi^n$

$$\phi_i^\star \;:\; \frac{d\phi}{dt} = R(\phi)\,;\; \phi(0) = \phi_i^n\,;\; t \in [t^n, t^{n+1/2}]$$

$$\phi_i^{\star\star} = \phi_i^\star - \Delta t[A(\phi^\star)]_i^{n+1/2}$$

$$\phi_i^{n+1} \;:\; \frac{d\phi}{dt} = R(\phi)\,;\; \phi(0) = \phi_i^{\star\star}\,;\; t \in [t^{n+1/2}, t^{n+1}]$$

Splitting

- Reaction rates are very temperature sensitive—over a timestep, the rate evaluation should know how the temperature is changing.
  - Split system of ODEs appears as:

$$\frac{dX_k}{dt} = \dot{\omega}(X_k, \rho_0, T_0)$$

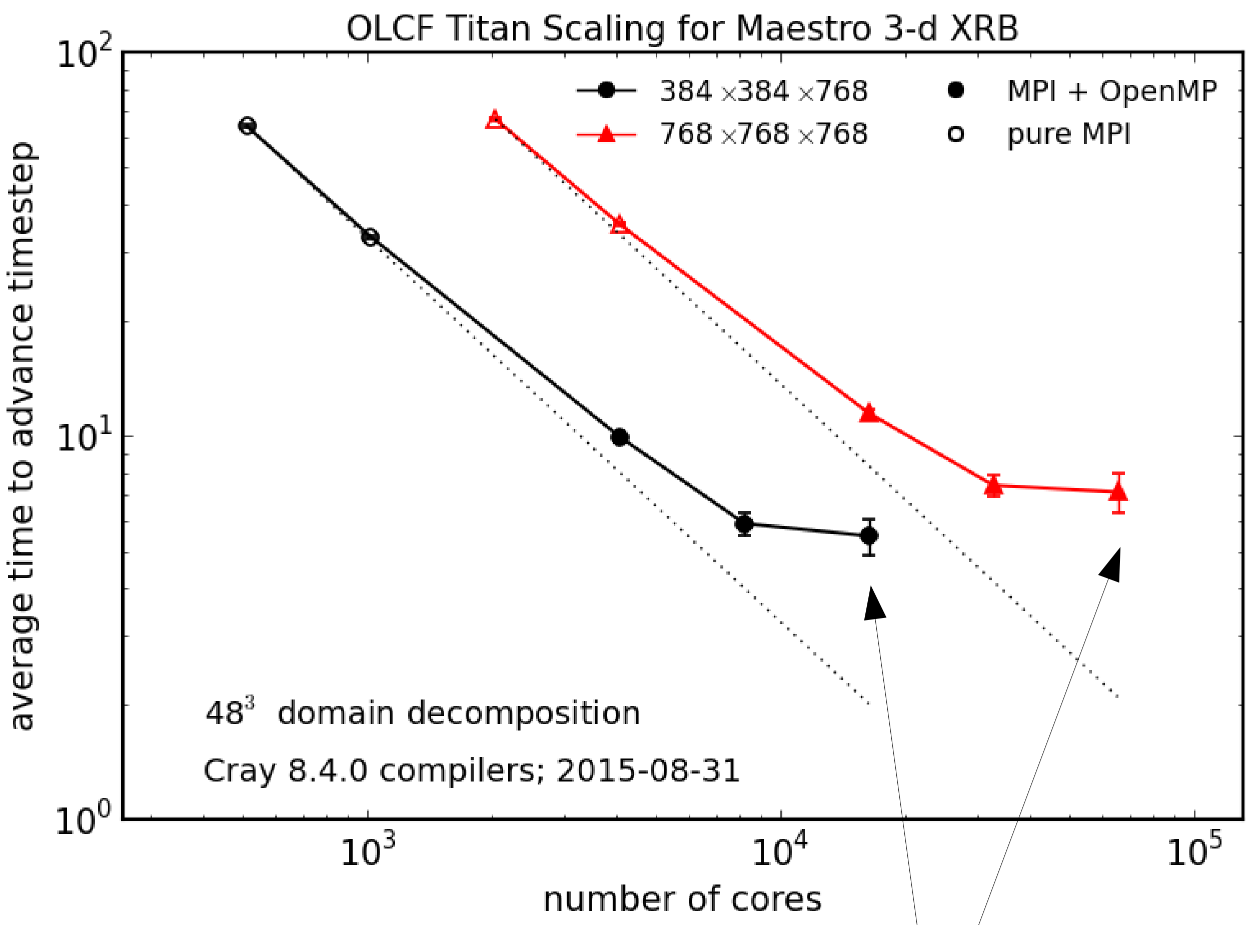
$$\frac{dT}{dt} = \frac{1}{c_p}\left(-\sum_k \xi_k \dot{\omega}_k + H_{\mathrm{nuc}}\right)$$

This assumes constant pressure in the T evolution, for a compressible code / explosive nucleosynthesis, you might do constant rho

    - Note: even with this energy equation, you are missing the actions of advection and diffusion, and not allowing density to self-consistently evolve

Current Code Performance

- Maestro is our low Mach number hydrodynamics code (https://github.com/BoxLib-Codes/MAESTRO)
  - uses 2nd order Godunov method for advection, approximate projection (enforced via multigrid) for the velocity constraint, and reactions via splitting
  - Written in Fortran 2003+
  - MPI/OpenMP parallelism via BoxLib library
- Reactions can take ~20% of our runtime
  - network cost scales as N^2



these points are where we do 1 MPI task with 16 OpenMP threads per node

Network Integration

- Reaction networks are stiff (loosely speaking—there is a wide range in timescales of interest that are changing in the system)
  - Implicit integration methods are needed
    - E.g.: backward Euler
    $$\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t \mathbf{f}(t^{n+1}, \mathbf{y}^{n+1})$$
    - Make a guess at the new-time solution, expanding the righthand side about this guess gives a linear system. Solve for the correction and iterate
    $$\left(\mathbf{I} - \Delta t\ \mathbf{J}|_{k-1}\right) \Delta\ \mathbf{y}_{k-1} = \mathbf{y}^n - \mathbf{y}_{k-1}^{n+1} + \Delta t \mathbf{f}(t^{n+1}, \mathbf{y}_{k-1}^{n+1})$$
    $$\mathbf{y}_k^{n+1} = \mathbf{y}_{k-1}^{n+1} + \Delta \mathbf{y}_{k-1}$$
    - Higher-order methods are constructed similarly
  - Expense is in the RHS and Jacobian evaluation

Putting it onto GPUs

- Just evaluate the Jacobian and RHS on GPUs
  - This will require a lot of data transfer
  - Ideally, we need a vectorized integrator that hits the RHS evaluation for all zones in the vector at the same time
    - This still can be expensive
- Put the entire ODE integration on the GPUs
  - Each zone can be treated independently
  - Data transfer only at the start and end of the integration
- Which integrator?
  - We use VODE as our workhorse—it is old (circa 1970) and common-block heavy
    - Not easy to OpenACC (no `threadprivate` directive as in OpenMP)
  - Our plan is to switch to a modern integrator written by a colleague
    - However, we've found that modern Fortran design patterns can cause issues with OpenACC compilers (more on that later)

Test Driver

- We built a simple first-order integration test driver
  - uses the same design patterns as our production networks
    - lots of Fortran modules
    - derived types
  - freely available: https://github.com/BoxLib-Codes/ode-openacc
- We require OpenACC 2.0
  - extensive use of functions, etc., is not supported in OpenACC 1.0
- We had lots of compiler crashes along the way—worked closely with the OLCF compiler team, who reported bugs to PGI for us.
- Overriding goal: we want to make minimal changes to the code, as we run on a variety of architectures and don't have the people to maintain two branches indefinitely.

Test Driver

- Code outline:
  - main loop over zones in the input vector—this is the parallelism
    - `!$acc loop gang vector`
  - numerical Jacobian constructed via simple differencing
  - linear system solved with LAPACK `dgesv`
    - we have our own copy of the LAPACK and BLAS source
    - customizations:
      - remove multiple exit points, add `!$acc routine seq`
      - get rid of character arrays (OpenACC doesn't seem to like these)

Lessons Learned

- PGI compilers offer better support then Cray (at the moment)
  - they became our target, especially since Cray will not be on summit
  - latest PGI (version 15.7) was needed to get things compiling and giving the correct answer
- Fortran things to avoid:
  - *character arrays*: not supported, seem to be a low priority for OpenACC developers. Replacing these with integer arrays is usually possible
  - *arrays of parameters (constants)*: rewrite these as allocatable arrays and initialize them
  - print statements, multiple exit points, stop statements, …
- Main summary:
  - the latest compilers are much better at implementing OpenACC now
  - more complex codes that didn't work a while back may be feasible now
  - take a look at our example for something that uses a range of modern Fortran for inspiration