Coarse vs. fine-level threading in the PENNANT mini-app

Charles R. Ferenbaugh
Applied Computer Science, CCS-7
Los Alamos National Laboratory

DOE Centers of Excellence Performance Portability Meeting
April 19, 2016
A brief overview of PENNANT

- Implements a small subset of basic physics from the LANL rad-hydro code FLAG
- 2-D staggered-grid Lagrangian hydrodynamics on general unstructured meshes (arbitrary polygons)
- Contains about 3300 lines of C++ source code
  - 2300 lines used in the main hydro cycle
  - 1000 lines of support code: mesh generators, graphics output, ...
  - Compare to > 600K lines for FLAG
- Has complete implementations for multicore CPUs (MPI + OpenMP) and GPUs (CUDA)
Three basic threading models

- Threading models fall into three basic categories:
  - Data-parallel, loop level
  - Data-parallel, higher level
  - Task-parallel
- These are not mutually exclusive; can be combined
- For this talk, I’m mainly interested in the first two (data-parallel)
Threading model 1: Data-parallel, loop level

- Example in OpenMP:

  ```c
  #pragma omp parallel for
  for (int n = 0; n < num_points; ++n) {
    z[n] = a * x[n] + y[n];
    w[n] = z[n] * x[n];
    ...
  }
  ```

- Relatively easy to implement in legacy code
- Other systems that support this: RAJA, Kokkos; CUDA, OpenCL; Thrust
Threading model 2: Data-parallel, high level (used in the baseline version of PENNANT)

- Example in OpenMP:
  ```c
  #pragma omp parallel for
  for (int c = 0; c < num_chunks; ++c) {
    run_step1(nbegin[c], nend[c]);
    run_step2(nbegin[c], nend[c]);
    run_step3(pbegin[c], pend[c]);
    ...
  }
  ```

- This has more work in the parallel region than the loop-level version does
- Requires some refactoring
- Other systems that support this: CUDA, OpenCL; RAJA, Kokkos (later versions)
PENNANT versions tested

- Four variants of MPI+OpenMP PENNANT
  - **coarse**: baseline version, has 5 fairly large chunk-level parallel for loops per hydro cycle
  - **medium**: splits up some loops, has 13 chunk loops per cycle
  - **fine**: puts every function call in its own loop, has 30 chunk loops per cycle
  - **loop**: instead of chunk-level threading, adds loop-level pragmas to all loops

- Three variants of CUDA PENNANT
  - **coarse, medium, fine**: as above, with each OpenMP parallel for loop translated into a CUDA kernel in CUDA
  - CUDA has no explicit loops, so there’s no **loop** variant
Threading model comparison: Sandy Bridge, OpenMP

- Runtime increases as size of parallel sections decreases
- Relative difference between versions increases slightly with problem size
Threading model comparison: Fermi GPU, CUDA

- As on the CPU, runtime increases as size of parallel sections decreases
Threading model comparison: various architectures

- Only the Sedov test problem is shown here; other tests give similar results
Threading model comments

- Why are the **medium** and **fine** versions slower?
  - Probably due to higher memory turnover, more context switching

- Why is the **loop** version even slower, especially on BG/Q?
  - Memory turnover, context switching apply even more here
  - **loop** version contains many atomic operations for side-to-zone reductions
    - These are particularly slow on BG/Q
    - Could remove these with a more flexible iteration schedule, as in RAJA or Kokkos
Pros and cons of high-level data parallelism

Pros:
- Performs better than loop-level on a range of architectures
- Can use many existing kernels with minimal change

Cons:
- For legacy codes, requires some refactoring work at driver level
- Requires reasoning about thread safety, synchronization between kernels
What about task-level parallelism?

- This is a longer-term question – most task-parallel runtimes are still maturing, not in production use.
- These will likely have similar coarse vs. fine issues.
  - In codes with smaller kernels/tasks, more time is spent in overhead and scheduling.
  - The Legion developers at Stanford have observed this.
Conclusions

- Coarse-grained threading over chunks performs better than fine-grained chunk threading or loop-level threading
  - This will probably be the method of choice for data parallelism in new codes
  - For legacy codes with limited resources, loop-level threading may be a good compromise

- Task-level parallelism will probably have similar issues; this will need further study
Thanks for your attention!

Charles Ferenbaugh
cferenba@lanl.gov
github.com/losalamos/PENNANT
Backup slides...
Test problems

<table>
<thead>
<tr>
<th>test name</th>
<th># zones</th>
<th># cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>nohsquare</td>
<td>129600</td>
<td>7677</td>
</tr>
<tr>
<td>nohpoly</td>
<td>63001</td>
<td>9876</td>
</tr>
<tr>
<td>sedov</td>
<td>291600</td>
<td>3882</td>
</tr>
<tr>
<td>leblanc</td>
<td>230400</td>
<td>3775</td>
</tr>
</tbody>
</table>

Problem sizes are chosen so that all problems do roughly the same amount of work (zones × cycles)
## Platforms used

Runs were done using a single node/card of each of the following:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Cores</th>
<th>Threads / Core</th>
<th>Frequency (MHz)</th>
<th>Peak Gflops</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon E5-2670 (Sandy Bridge)</td>
<td>16</td>
<td></td>
<td>2600</td>
<td>332.8</td>
<td>230</td>
</tr>
<tr>
<td>Intel MIC (KNC)</td>
<td>60</td>
<td>4</td>
<td>1050</td>
<td>1065</td>
<td>225</td>
</tr>
<tr>
<td>Blue Gene/Q</td>
<td>16</td>
<td>4</td>
<td>1600</td>
<td>204.8</td>
<td>63</td>
</tr>
<tr>
<td>Nvidia M2090 (Fermi)</td>
<td>512</td>
<td></td>
<td>1300</td>
<td>665</td>
<td>225</td>
</tr>
<tr>
<td>Nvidia K20X (Kepler)</td>
<td>2688</td>
<td></td>
<td>732</td>
<td>1310</td>
<td>235</td>
</tr>
</tbody>
</table>