Improving the Thread Scalability of the Community Atmosphere Model

Kyle Marcus - University at Buffalo NY
SIParCS Summer 2015

CISL Group (John Dennis & Ben Jamroz)
Outline

● Motivation
● Programming models
● HOMME (dynamic core)
● Summer Work
● Results
● Future work
Motivation - Current Gen

Yellowstone Supercomputer

- 2.6-GHz Sandy Bridge AVX
- 72,576 processor cores
- 2 GB/core, 32 GB/node
- 1.504 petaflops peak
- production summer 2012
Motivation - Next Generation

Argonne Aurora SC
- 180 petaflops
- Production in 2018

Xeon Phi Coprocessor
- Knights Hill - 3rd Gen
- Large vector units

72 Cores x 4 threads/core!
72*4 = 288 cores / Phi
Motivation - Scalability

- Need to scale to new systems
- More compute power
- More memory
- Larger vectors
- Take advantage of new hardware!

Intel Haswell
18 cores * 2 threads/core = 36 threads
Community Atmosphere Model (CAM)
Summer work with OpenMP

- **HOMME** community model
  - High-Order Method Modeling Environment
  - Default *dynamical core* of the Community Atmosphere Model (CAM) and the Community Earth System Model (CESM).
  - Uses fully unstructured quadrilateral based finite element meshes on the sphere, such as the *cubed-sphere mesh* for quasi-uniform resolution.
Cubed-Sphere Mesh
Parallelize vertical code

```c
ifdef (COLUMN_OPENMP) {
    #omp parallel ...
}
```
What is OpenMP?

**Open Multi-Processing** is an **API** that supports multi-platform shared memory multiprocessing programming in **C**, **C++**, and **Fortran**.
First steps in project

1. Determine if COLUMN_OPENMP sections scale correctly
   - Add in timing code to all COLUMN_OPENMP sections and run the code multiple times with different number of cores
   - There are 132 different COLUMN_OPENMP parallel sections that need to be tracked and timing code added to
#if (defined COLUMN_OPENMP)
!$omp parallel do private(k, q, lap_p)
#endif

do q=1,qsize
    do k=1,nlev
        lap_p(:,:,)=qtens(:,:,k,q,ie)
        qtens(:,:,k,q,ie)=laplace_sphere_wk(lap_p,deriv,elem(ie),var_coef=var_coef1)
    enddo
enddo
Adding timing code

- Used Python to text process Fortran files and look for the string "COLUMN_OPENMP"
- Once found the timing function `t_startf()` was hooked into the code before the beginning of the parallel section and at the end of the section `t_stopf()` was added.
- Automated the process for fast changes
call t_startf('col_omp_testing_viscosity4')
#if (defined COLUMN_OPENMP)
!$omp parallel do private(k, q, lap_p)
#endif
do q=1,qsize
do k=1,nlev
   lap_p(:,:,)=qtens(:,:,k,q,ie)
   qtens(:,:,k,q,ie)=laplace_sphere_wk(lap_p,deriv,elem(ie),var_coef=var_coef1)
endo
dendo
call t_stopf('col_omp_testing_viscosity4')

Start of timing code
naming convention is 'col_omp_testing' + filename + count

End of timing code
Test runs with timing code added

- Ran scalability tests on COLUMN_OPENMP HOMME code in the HPC Lab cluster.
- Used the Haswell 4 socket system where each socket has 18 cores for a total of 72
- Haswell system is equipped with hyper-threading which we did not use while testing.
4 Socket Node
Running tests

- Must interface with the SLURM scheduler
- We used mpirun instead of srun
- Configured it to run without hyper-threading so we were only using the hardware cores
- This allowed for running scaling tests on a single node because it had so many cores
<table>
<thead>
<tr>
<th>Rank</th>
<th>Pid</th>
<th>Node name</th>
<th>Pin cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>158716</td>
<td>hl01.ssg.ucar.edu</td>
<td>{18,19,54,55}</td>
</tr>
<tr>
<td>1</td>
<td>158717</td>
<td>hl01.ssg.ucar.edu</td>
<td>{20,21,56,57}</td>
</tr>
<tr>
<td>2</td>
<td>158718</td>
<td>hl01.ssg.ucar.edu</td>
<td>{22,23,58,59}</td>
</tr>
<tr>
<td>3</td>
<td>158719</td>
<td>hl01.ssg.ucar.edu</td>
<td>{24,25,60,61}</td>
</tr>
<tr>
<td>4</td>
<td>158720</td>
<td>hl01.ssg.ucar.edu</td>
<td>{26,27,62,63}</td>
</tr>
<tr>
<td>5</td>
<td>158721</td>
<td>hl01.ssg.ucar.edu</td>
<td>{28,29,64,65}</td>
</tr>
<tr>
<td>6</td>
<td>158722</td>
<td>hl01.ssg.ucar.edu</td>
<td>{30,31,66,67}</td>
</tr>
<tr>
<td>7</td>
<td>158723</td>
<td>hl01.ssg.ucar.edu</td>
<td>{32,33,68,69}</td>
</tr>
<tr>
<td>8</td>
<td>158724</td>
<td>hl01.ssg.ucar.edu</td>
<td>{34,35,70,71}</td>
</tr>
<tr>
<td>9</td>
<td>158725</td>
<td>hl01.ssg.ucar.edu</td>
<td>{0,1,36,37}</td>
</tr>
<tr>
<td>10</td>
<td>158726</td>
<td>hl01.ssg.ucar.edu</td>
<td>{2,3,38,39}</td>
</tr>
<tr>
<td>11</td>
<td>158727</td>
<td>hl01.ssg.ucar.edu</td>
<td>{4,5,40,41}</td>
</tr>
<tr>
<td>12</td>
<td>158728</td>
<td>hl01.ssg.ucar.edu</td>
<td>{6,7,42,43}</td>
</tr>
<tr>
<td>13</td>
<td>158729</td>
<td>hl01.ssg.ucar.edu</td>
<td>{8,9,44,45}</td>
</tr>
<tr>
<td>14</td>
<td>158730</td>
<td>hl01.ssg.ucar.edu</td>
<td>{10,11,46,47}</td>
</tr>
<tr>
<td>15</td>
<td>158731</td>
<td>hl01.ssg.ucar.edu</td>
<td>{12,13,48,49}</td>
</tr>
<tr>
<td>16</td>
<td>158732</td>
<td>hl01.ssg.ucar.edu</td>
<td>{14,15,50,51}</td>
</tr>
<tr>
<td>17</td>
<td>158733</td>
<td>hl01.ssg.ucar.edu</td>
<td>{16,17,52,53}</td>
</tr>
</tbody>
</table>

number of MPI processes: 18

This job has 18 MPI processes and 2 vertical threads associated with each MPI process.

We wanted hyper-threading turned off so in order to do that, twice the number of threads were assigned to each MPI process

```bash
export I_MPI_DOMAIN=4,compact
mpiexec -n 18 ...
```
Results

- Timings showed some portions of the COLUMN_OPENMP code that has scalability issues
- Removed some COLUMN_OPENMP sections of the code (edges)
- 2 sections in viscosity, 3 sections in advection
call t_startf('col_omp_testing_viscosity4')
#if (defined COLUMN_OPENMP)
!$omp parallel do private(k, q, lap_p)
#endif
do q=1,qsize
  do k=1,nlev
    lap_p(:,:,)=qtens(:,:,k,q,ie)
    qtens(:,:,k,q,ie)=laplace_sphere_wk(lap_p,deriv,elem(ie),var_coef=var_coef1)
  enddo
enddo
#endif

collapse these do loops, split work up into teams?

Create a **single parallel region** (master thread) and turn threading on when needed

create start and end bounds for each thread, this may enforce **cache reuse** on the node

call t_stopf('col_omp_testing_viscosity4')
call t_startf('col_omp_testing_viscosity4')
#if (defined COLUMN_OPENMP)
!$omp parallel do collapse(2) private(k, q, lap_p)
#endif
do q=1,qsize
  do k=1,nlev
    lap_p(:,:,)=qtens(:,k,q,ie)
    qtens(:,:,k,q,ie)=laplace_sphere_wk(lap_p,deriv,elem(ie),var_coef=var_coef1)
  enddo
enddo
call t_stopf('col_omp_testing_viscosity4')

Collapse the 2 loops into 1 single loop. This can be done because both k and q are set up as private variables between the loops.

Increased performance because of less overhead of scheduling jobs for different threads.
### Collapse Results - 3x1x12

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Count</th>
<th>Walltotal</th>
<th>Wallmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORIGINAL</td>
<td>viscosity4</td>
<td>38880</td>
<td>9.849440e-01</td>
<td>0.351</td>
</tr>
<tr>
<td>COLLAPSE(2)</td>
<td>viscosity4</td>
<td>38880</td>
<td>7.451851e-01</td>
<td>0.269</td>
</tr>
<tr>
<td>ORIGINAL</td>
<td>viscosity5</td>
<td>38880</td>
<td>9.688257e-01</td>
<td>0.342</td>
</tr>
<tr>
<td>COLLAPSE(2)</td>
<td>viscosity5</td>
<td>38880</td>
<td>7.987231e-01</td>
<td>0.278</td>
</tr>
</tbody>
</table>

~23% decrease in execution time
One large parallel section?

Creating one large parallel and turning on/off parallelism when needed

#pragma omp parallel
{
    #pragma omp for[...]
    for(...)
    #pragma omp single
    { ... }
    #pragma omp for[...]
    for(...)
}

Intel defines what happens and how the program is compiled.

The threads are created at the beginning of the first OpenMP section and destroyed when the program terminates.

Penalties are created for the fork/join model

Add a nowait clause to the for loops - issues with loops inside single loops
NOWAIT combined Results - 6x1x6

<table>
<thead>
<tr>
<th>Name</th>
<th>Wallmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>advection6</td>
<td>0.057</td>
</tr>
<tr>
<td>advection7</td>
<td>1.825</td>
</tr>
<tr>
<td></td>
<td>1.882</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Wallmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>advection6</td>
<td>n/a</td>
</tr>
<tr>
<td>advection7</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>1.584</td>
</tr>
</tbody>
</table>

#pragma omp parallel
{
  #pragma omp single
  {
    for ( ... ) {
      #pragma omp for[...]
      for ( ... ) {
      }
    // Issues with inside parallel loop
  }
}
Future Work

- Study the 5 sections of code
- Look at new OpenMP 4 features - tasks
- More testing to get better performance
- Single loops inside parallel loops
  - Rewrite these loops to not use single outside loops
  - Schedule different sections for outside single loops in threads
Thank You

Kyle Marcus
University at Buffalo, NY
kmarcus2@buffalo.edu
What does code scalability mean?

**Scalability** is the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

**Weak** VS **Strong** scaling
How do you scale your code?

Many different threading architectures

- MPI
- **OpenMP**
- PThread
OpenMP language extensions

- **parallel control structures**
  - governs flow of control in the program
  - parallel directive

- **work sharing**
  - distributes work among threads
  - do/parallel do and section directives

- **data environment**
  - scopes variables
    - shared and private clauses

- **synchronization**
  - coordinates thread execution
    - critical and atomic directives
    - barrier directive

- **runtime functions, env. variables**
  - runtime environment
    - omp_set_num_threads()
    - omp_get_thread_num()
    - OMP_NUM_THREADS
    - OMP_SCHEDULE
OpenMP Example

```c
for (i = 0; i < N; i++) {
    pixel[i] = red_filter(pixel[i]);
}
```

```c
#pragma omp parallel for
for (i = 0; i < N; i++) {
    pixel[i] = red_filter(pixel[i]);
}
```
ne = 6

MPI x Horz x Vert