Adding OpenMP to Your Code Using Cray Reveal

Helen He
NERSC User Services Group

October 10, 2013
Current Architecture Trend

- Multi-socket nodes with rapidly increasing core counts
- Memory per core decreases
- Memory bandwidth per core decreases
- Network bandwidth per core decreases
- Need a hybrid programming model with three levels of parallelism
  - MPI between nodes or sockets
  - Shared memory (such as OpenMP) on the nodes/sockets
  - Increase vectorization for lower level loop structures
Advantages of hybrid MPI/OpenMP

- Reduce number of MPI ranks per node
- Minimize network injection contention
- Avoids the extra communication overhead with MPI within node
- Reduce memory footprint
- Chance of overlapping MPI communication with OpenMP thread computation
What is Reveal

- A tool developed by Cray to help developing the hybrid programming model
- Part of the Cray Perftools software package
- Only works under PrgEnv-cray
- Utilizes the Cray CCE program library for loopmark and source code analysis, combined with performance data collected from CrayPat
- Helps to identify top time consuming loops, with compiler feedback on dependency and vectorization
- Loop scope analysis provides variable scope and compiler directive suggestions for inserting OpenMP parallelism to a serial or pure MPI code
Steps to Use Reveal on Edison (1)

- **Load the user environment**
  - % module swap PrgEnv-intel PrgEnv-cray
  - % module unload darshan
  - % module load perftools (current default is version 6.1.2)

- **Generate loop work estimates**
  - % ftn –c –h profile_generate myprogram.f90
  - % ftn –o poisson_serial –h profile_generate myprogram.o
    - Good to separate compile and link to keep object files
    - Optimization flags disabled with –h profile-generate
  - % pat_build –w myprogram (-w enables tracing)
    - It will generate executable “myprogram+pat”
  - Run the program “myprogram+pat”
    - It will generate one or more myprogram+pat+...xf files
  - % pat_report myprogram+pat...xf > myprogram.rpt
    - It will generate myprogram+pat....ap2 file
Steps to Use Reveal on Edison (2)

• Generate a program library
  – % ftn -O3 -hpl=myprogram.pl -c myprogram.f90
  – Optimization flags can be used
  – Build one source code at a time, with “-c” flag
  – Use absolute path for program library if sources are in multiple directories
  – User needs to clean up program library from time to time

• Launch Reveal
  – % reveal myprogram.pl myprogram+pat...ap2
Steps to Use Reveal on Hopper

• To use the newest version perftools/6.1.2, which is built upon cray-mpich/6.x.x
  – % module unload cray-libsci cray-mpich2
  – % module load cray-libsci/12.1.01
  – % module load cray-mpich/6.1.0
  – % module unload darshan
  – % module load perftools-lite/6.1.2

• To use perftools-lite/6.1.1 or older
  – % module unload darshan
  – % module load perftools-lite/6.1.2

• Follow the rest of steps for Edison
Cray Reveal GUI
Top loops with compiler loopmarks and feedback

Compiler loopmarks

Top loops

Compiler feedback
Compiler feedback explanation

Double click to explain
Compiler feedback explanation (2)

- A loop was unrolled.

The compiler unrolled the loop. Unrolling creates a number of copies of the loop body. When unrolling an outer loop, the compiler attempts to fuse replicated inner loops - a transformation known as unroll-and-jam. The compiler will always employ the unroll-and-jam mode when unrolling an outer loop; literal outer loop unrolling may occur when unrolling to satisfy a user directive (pragma).

This message indicates that unroll-and-jam was performed with respect to the identified loop. A different message is issued when literal outer loop unrolling is performed, as this transformation is far less likely to be beneficial.

For sake of illustration, the following contrasts unroll-and-jam with literal outer loop unrolling:

```c
for (i = 0; (i < 100); i++) {
    for (j = 0; (j < 10); j++) {
        a[i][j] = b[i][j] + 42.0;
    }
}
```

The literal outer unroll code performs the same sequence of memory operations as the original nest, while the unroll-and-jam transformation intensifies operations from outer loop iterations. The compiler employs literal outer loop unrolling only when the data dependencies in the loop, or a control flow involvement, prevent fusion of the replicated inner loops. Literal outer
Reveal scoping assistance

Right click to select loops

Start Scoping
Scoping Results
Suggested OpenMP directives
Save the directives to the original file.
Extensive “Help” topics in Reveal

- Program Library
  - Loop or loops to scope. To do this, right-click and select “Scope Loop” to scope.
  - Help the looping. A new window titled “Help the looping. A new window titled...”
  - When ready to start the looping, select “Start Looping”

- Reduction in an indexed function
  - There is a reduction to a variable which is in a called function.
  - The C programming model does not provide any method for automatically protecting the reduction variable.
  - For the loop to run correctly, the user needs to protect this reduction with a lock or change it to an atomic operation.

- Note:
  - Because of current limitations in the analysis, a global variable is initialized anywhere into the calling function, which may assume there is an initial reference in any loop rather than reference to the variable.
  - As a consequence of this, Reveal may display this scoping problem when it does not actually exist.

- Note:
  - Since the addition of a lock or changing to an atomic operation may significantly decrease performance, it may be necessary to do the function containing the reduction if it is called from other locations where the protection

- Create OpenMP clauses or directives
  - View scoping information by clicking on loops in the navigation panel that have been marked with a red or green icon. A new window with the title “Redeem OpenMP Scoping” will appear that contains the scoping information. Variables will be scoped shared, private, unscolded. Unresolved variables are identified so that you can focus on addressing conflicts or issues that prevent correct parallel execution.
  - Reduction variables are highlighted with a red “R” superscript and inline variables that caused an error are highlighted with a red “=” superscript.

- Change the scope of a specific variable by clicking on the scope presented for that variable.
  - Click on a variable in the “Scoping Results” and see highlighted occurrences of that variable within the “Source” panel.

- View and Save OpenMP directives

- Help the looping. A new window titled “Help the looping. A new window titled...”
Reveal helps to start adding OpenMP

- Only under PrgEnv-cray, with CCE compiler
- Start from most time consuming loops first
- Insert OpenMP directives
  - Make sure to save a copy of the original code first, since the saved new file will overwrite the original code
- There will be unresolved and incomplete variable scopes
- There maybe more incomplete and incorrect variables identified when compiling the resulted OpenMP codes
- User still needs to understand OpenMP, and resolves the issues.
- Verify correctness and performance
- Repeat as necessary
- No OpenMP tasks, barrier, critical, atomic regions, etc
More work after reveal (1)

Reveal suggests:

```c
#include <omp.h>

#pragma omp parallel for default(none) \
  unresolved (my_change,my_n) \
  shared (my_rank,N,i_max,u_new,u) \
  firstprivate (i)

for ( i = i_min[my_rank]; i <= i_max[my_rank]; i++ )
{
  for ( j = 1; j <= N; j++ )
  {
    if ( u_new[INDEX(i,j)] != 0.0 )
    {
      my_change = my_change
      + fabs ( 1.0 - u[INDEX(i,j)] / u_new[INDEX(i,j)] );
      my_n = my_n + 1;
    }
  }
}
```

Final code:
(Lots of changes from Reveal suggestions, but will still make the code slower than without OpenMP directives, so will not use any directives)

```c
#include <omp.h>

#pragma omp parallel for default(none) \
  private (my_change,my_n) \
  shared (my_rank,N,i_min,i_max,u_new,u) \
  private (j) \
  private (i)

for ( i = i_min[my_rank]; i <= i_max[my_rank]; i++ )
{
  for ( j = 1; j <= N; j++ )
  {
    if ( u_new[INDEX(i,j)] != 0.0 )
    {
      my_change = my_change
      + fabs ( 1.0 - u[INDEX(i,j)] / u_new[INDEX(i,j)] );
      my_n = my_n + 1;
    }
  }
}
```

```c
#pragma omp critical
{
  my_change = my_change
  + fabs ( 1.0 - u[INDEX(i,j)] / u_new[INDEX(i,j)] );
  my_n = my_n + 1;
}
```
Reveal suggests:

```c
#pragma omp parallel for default(none) \
  shared (f,N,u,u_new,i,h)

for ( i = i_min[my_rank] + 1; i <= i_max[my_rank] - 1; i++ )
{
  for ( j = 1; j <= N; j++ )
  {
    u_new[INDEX(i,j)] =
    0.25 * ( u[INDEX(i-1,j)] + u[INDEX(i+1,j)] +
             u[INDEX(i,j-1)] + u[INDEX(i,j+1)] +
             h * h * f[INDEX(i,j)] );
  }
}
```

Final code:

```c
#pragma omp parallel for default(none) \
  private (my_rank,j,i) \
  shared (f,N,u,u_new,h,i_min,i_max)

for ( i = i_min[my_rank] + 1; i <= i_max[my_rank] - 1; i++ )
{
  for ( j = 1; j <= N; j++ )
  {
    u_new[INDEX(i,j)] =
    0.25 * ( u[INDEX(i-1,j)] + u[INDEX(i+1,j)] +
             u[INDEX(i,j-1)] + u[INDEX(i,j+1)] +
             h * h * f[INDEX(i,j)] );
  }
}
```
Performance with OpenMP added

**Poisson_mpi_omp**, 4 MPI tasks, N=1200, on Edison

**poisson_omp**, nx=ny=1201, on Edison
More information

- % module load training
- See example codes, reports, detailed steps in README at:
  - $EXAMPLES/Edison2013/reveal
- Documentations:
  - % man reveal (when the “perftools” module is loaded)
  - Using Cray Performance Measurement and Analysis Tools
    http://docs.cray.com/books/S-2376-612/S-2376-612.pdf
National Energy Research Scientific Computing Center