Lab: Adding Parallelism to Code

Todd Evans

Feb 3rd, 2017

Login

Setup

1. Login to Stampede KNL

```
$ ssh <username>@login-knl1.stampede.tacc.utexas.edu
```

2. Enter your password and token
3. Extract the lab if you don't have it

```
$ tar -xf ~train00/knl_opt_training.tgz
```

4. start an idev session

```
$ idev -m 60
```

5. Move into the directory

```
$ cd knl_opt
```

Ex 1: Opt Reports, Cache Reuse, & OpenMP

OpenMP can Interfere with Compiler Optimization

- Compiler will often reorder loops for efficient cache reuse
  - Remember, $\times 100$ faster to get data from cache than memory
- `#pragma omp parallel for` can confuse it
- Let's look at `vector.c` $\rightarrow$ `omp_vector.c` with OMP
- This looks pretty good right?

```
  for (n=0;n < 10000;n++){
#pragma omp parallel for
    for(i=0;i<N;i++)
      for(j=0;j<N;j++)
        a[i]+=B[i][j]*c[j];
}
```

Ex 1: Opt Reports, Cache Reuse, & OpenMP

1. Compile w/o OpenMP
   ```
   $ icc omp_vector.c -o vector -xhost
   ```
2. Run it: Results should be just like this morning (if you weren't there try running
3. Compile w/ OpenMP
   ```
   $ icc omp_vector.c -o omp_vector -xhost -qopenmp
   ```
4. set number of threads to 1 (default)
   ```
   $ export OMP_NUM_THREADS=1
   ```
5. Run it
   ```
   ./omp_vector
   ```
6. Oh noooo!
7. What happened?
8. try increasing thread number and running

Ex 1: Opt Reports, Cache Reuse, & OpenMP

At least throwing a bunch of threads at the problem helps

1. But why is 1 thread of OpenMP sooo much slower than no OpenMP
2. Compile w/o OpenMP and qopt-report=5
   `$ icc omp_vector.c -o vector -xhost -qopt-report`
3. open `omp_vector.optreport` and inspect
4. Notice line
   `remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 2 1 3 )`
5. Compile w/ OpenMP and qopt-report
   `$ icc omp_vector.c -o omp_vector -xhost -qopenmp -qopt-report`
6. Is that line still there?

Ex 1: Opt Reports, Cache Reuse, & OpenMP

The compiler did not optimize nested loops because of OpenMP pragma

1. Oops!
2. Let's reorder loops by hand

```
#pragma omp parallel for
  for(i=0;i<N;i++){
    for (n=0;n < 10000;n++)
      for(j=0;j<N;j++)
        a[i]+=B[i][j]*c[j];
}
```

3. Compile and rerun
4. Vary number of threads
5. on line 45 change `c[j]` $\rightarrow$ 2
6. compile and rerun at multiple thread count

Ex 1: Opt Reports, Cache Reuse, & OpenMP

One last comment

1. The vector product operation benefitted from using more than 68 cores
   1.1 But we only have 68 *physical* cores so why
   1.2 Hardware threads help to hide *latency* (memory access is slow)
2. Code with good memory access patterns does not benefit from additional cores
   2.1 `c[j]` $\rightarrow$ 2 fills VPUs better
   2.2 It doesn't need to hide latency because cache reuse is so good
   2.3 additional cores just add overhead

Ex 2: Load Balancing w/ OpenMP

We'll look at a code with unequal work per iteration

1. `omp_prime.c` will compute the number of primes between 0 and 10 million
2. As iteration count increasing more work required to factorize

```
#pragma omp parallel for schedule(static) reduction(+:j)
  for (n=0;n <N;n++) {
    if (is_prime(n))
      j++;
}
```

3. Compile `omp_prime.c` with openmp
4. Run with 1,2,4,16, 34, 68 threads. Record run times
5. Modify to use `dynamic` scheduling
6. Run with 1,2,4,16, 34, 68 threads: Record run times
7. Which one scales better and why?
8. Vary chunk sizes . Can you improve?

Ex 3: Parallelize Heat Equation

We will parallelize heat.c

- By adding OpenMP and Vectorization
- Solves steady state of 2D heat equation
  - Left wall is held constant at 1, other walls are held at 0
- stencil update scheme is defined in calc_up.c

```
void calc_up(int x,int y,int Nx,int Ny,double u[Nx][Ny],
                                        double up[Nx][Ny]) {
  up[x][y]=u[x][y]+0.01*(u[x-1][y]+u[x+1][y]-2*u[x][y])+
                           0.01*(u[x][y-1]+u[x][y+1]-2*u[x][y]);
}
```

Ex 3: Parallelize Heat Equation

Start with OpenMP Parallelization

1. Compile `calc_up.c`
   `$ icc calc_up.c -c`
2. Compile and run `heat.c`
   `$ icc calc_up.o heat.c -o heat -xhost`
3. Run, then record time and sum (sum is to check correctness)
4. Open `heat.c` and take a look
   4.1 I'm only timing `t` loop (time to solution)
   4.2 Where might make sense to add OpenMP parallelization?
5. Add OpenMP pragmas and compile with `-qopenmp -qopt-report=5`
   5.1 hint: 2 places work really well
   5.2 you might need to make your iteration variables private (check sum!)
6. Record time at different thread counts and ensure sum is unchanged

Ex 3: Parallelize Heat Equation

Start with OpenMP Parallelization

1. You should see a huge difference with more threads
2. What about vectorization though?
   2.1 take a look at `heat.optrpt`
   2.2 Was every loop vectorized?
3. Let's get vectorization out of `calc_up`
4. Cross-file inline it:
```
$ icc calc_up.c -ipo
$ icc calc_up.o heat.c -o heat -xhost -ipo
-qopt-report
```
5. open ipo_out.optrpt and inspect inline and vectorization sections
6. now run heat at different thread count and compare

Ex 3: Parallelize Heat Equation

- We saw some nice speedup: $\sim$17s $\rightarrow$ $\sim$0.1s
- You would see even more difference for larger problem
- Not perfect but much better than waiting around for unoptimized code
- If you tried to generate an opt report with ipo in the last step you're probably a little dissapointed...
- Apparently it does not handle `-ipo` very well