Lightning tutorial
Loop level parallelization of C/C++ using OpenMP

Robert Sinkovits
Director Scientific Computing Applications

SDSC SAN DIEGO SUPERCOMPUTER CENTER
UC San Diego

OpenMP - API for parallel programming

- OpenMP is an application programming interface for thread-level, shared memory parallelization in C/C++ and Fortran
- Parallelization enabled by directives that are not part of the language
  - Fortran: special comments recognized by OpenMP capable compilers
  - C/C++: macros recognized by the preprocessor
- OpenMP has many advanced features and a run-time library for accessing/modifying parameters that affect the parallelization

- In this brief tutorial, we'll focus on the most commonly used feature of OpenMP – loop level parallelization

OpenMP – Loop level parallelization in C/C++

Two basic scenarios

- All iterations take the same amount of time
- Iterations take varying amounts of time

The first case is more common, but with a very minor change to the pragma, we can easily adopt to the second case

Static decomposition: Use when iterations take same amount of time

```
#pragma omp parallel for
for (int i = 0; i < niter; i++) {
  -- loop body –
}
```

Each thread is statically assigned niter/nthread iterations

- For example, if niter = 100 and nthread = 5
- Thread 0 assigned iterations 0-19
- Thread 1 assigned iterations 20-39
- …
- Thread 5 assigned iterations 80-99

Dynamic decomposition: Use when iterations take different amounts of time

```
#pragma omp parallel for schedule(dynamic, chunk)
for (int i = 0; i < niter; i++) {
  -- loop body –
}
```

Threads dynamically assigned “chunk” iterations as they become idle

- For example, if niter = 100, nthread = 3 and chunk = 10
- Thread 0 assigned iterations 0-9
- Thread 1 assigned iterations 10-19
- Thread 2 assigned iterations 20-29
- First idle thread assigned iterations 30-39
- Next idle thread assigned iterations 40-49
- Continue assigning chunks until loop complete

Build and run parallel executable

- All major C/C++ and Fortran compilers are OpenMP capable (Intel, PGI, GNU, PathScale, IBM). Just need to compile with appropriate option.
- One caveat is that option is compiler dependent (C++ examples below)
  - Intel: icpc -openmp
  - PGI: pgCC -mp
  - GNU: g++ -fopenmp

- Set the environment variable before executing
  - export OMP_NUM_THREADS=24
  - ./a.out

Why not always use dynamic scheduling?

- Dynamic scheduling can introduce overhead
- Depending on the thread count, number of iterations and the chunk size, can introduce load imbalance even if all iterations take the same amount of time

100 iterations, 4 threads, chunk size = 20, all iterations take same time

- Thread 0 assigned 0-19
- Thread 1 assigned 20-39
- Thread 2 assigned 40-59
- Thread 3 assigned 60-79
- One thread gets 80-99, other threads are idle

If we had used static scheduling, load balancing would not be an issue!

Do I need to worry about threads overwriting each others results?

Yes! We can avoid this by using the OpenMP PRIVATE clause or by declaring variables inside the loop.

```
#pragma omp parallel for private(t)
for (int i = 0; i < niter; i++) {
  t = sqrt(x[i]*x[i] + y[i]*y[i]);
  z[i] = exp(t) + log(t);
}
```

```
#pragma omp parallel for
for (int i = 0; i < niter; i++) {
  double t = sqrt(x[i]*x[i] + y[i]*y[i]);
  z[i] = exp(t) + log(t);
}
```

I personally prefer this approach – easier to keep track of private variables

So I now know everything about OpenMP?

No! You now know the basics of parallelizing a loop, but there's much more to learn

- Reduction variables (calculate sum inside a loop)
- CRITICAL and ATOMIC directives (restrict execution to single thread)
- The runtime library (find the number of active threads, set limits on thread count, etc.)
- Parallel regions, nested parallelism and other advanced topics

For more information, go to

- https://computing.llnl.gov/tutorials/openMP/
- http://www.openmp.org/

Exercise

We'll now execute a code containing a parallelized loop where we solve a different eigenvalue problem at each iteration. We'll look at two cases

- The same sized matrix (N x N) is used at each iteration
- The size of the matrix grows for later iterations
  N = M + (iteration number) / 5

export OMP_NUM_THREADS=X; ./openmp_eigen_static 500 500 E

export OMP_NUM_THREADS=X; ./openmp_eigen_static 500 700 U

export OMP_NUM_THREADS=X; ./openmp_eigen_dynamic 500 700 U

(where X = 1, 2, 4, 8, 16; run times will vary from 2 to 75 seconds)

* Eigenvalue problems are common in many areas of scientific computing. Don't worry if you don't know what it is. For the purpose of this exercise just think of it as "something that takes a lot of CPU time"