High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 2

OpenMP

- Shared address space programming
- “High-level” approach compared to POSIX threads that is useful in scientific and other loop-based applications
- Comprised primarily of compiler directives (pragmas) as well as a library and environment variables
  Example directive: `#pragma omp parallel num_threads(4)`
- Note: *directives* can have *clauses*
- Online reference:
  `https://computing.llnl.gov/tutorials/openMP`

Simple OpenMP example

```c
#include <stdio.h>

int main(void)
{
  // create a team of threads for the following structured block
  #pragma omp parallel
  {
    printf("Hello, world!\n");
  }
  // team of threads join master thread after the structured block

  return 0;
}
```

Compiling OpenMP programs

- Compile with: `gcc -fopenmp file.c`
- Without `-fopenmp`, the directives will be ignored, and but your program will still run, using a single thread
  - This is a *feature*, not a bug, since good OpenMP programs should run correctly with a single thread.

`#pragma omp parallel`

- `omp parallel` creates a team of threads for the following structured block
- the threads *join* the master thread at the end of the structured block
- note: master thread and worker threads look the same in the parallel region
- this is the “top-level” OpenMP directive. Other directives are combined with this one.

Setting the number of threads to be created

In increasing priority:

- a default number of threads is used, normally equal to the number of cores (or more if hyperthreading is turned on)
- shell variable `OMP_NUM_THREADS`
- function `omp_set_num_threads(num)`
- `num_threads` clause in `omp parallel` directive: `#pragma omp parallel num_threads(num)`

Each thread has its own id

```c
#include <stdio.h>
#include <omp.h>

int main(void)
{
  #pragma omp parallel
  {
    int thread_id = omp_get_thread_num();
    printf("Thread %d says: Hello World\n", thread_id);

    if (thread_id == 0) {
        int nthreads = omp_get_num_threads();
        printf("Thread %d: number of threads is %d\n",
          thread_id, nthreads);
    }
  }

  return 0;
}
```

Shared vs. private variables

Consider this example:

```c
#include <stdio.h>

int main(void)
{
  int counter = 0;

  #pragma omp parallel
  counter++;

  printf("counter: %d\n", counter);

  return 0;
}
```

By default, all variables are shared.

Private variables

It is useful for threads to have their own copy of variables. Variables are private if:

- they are declared within the parallel region (not possible in Fortran)
- they are made private using the `private` clause
- indices of worksharing loops are automatically private (will see this later)

All other variables are shared by default.

`private` clause

```c
#include <stdio.h>

int main(void)
{
  int counter = 0;

  #pragma omp parallel private(counter)
  counter++;

  printf("counter: %d\n", counter);

  return 0;
}
```

What is the printed value of counter?

`private` clause

```c
#include <stdio.h>

int main(void)
{
  int counter = 100;

  #pragma omp parallel private(counter)
  {
    printf("counter: %d\n", counter);
  }

  printf("master thread counter: %d\n", counter);

  return 0;
}
```

`private` clause

- The initial value of a privatized variable is undefined (in C++ the initial value may be set by the default constructor).
- The value of the variable outside of the parallel region is not changed by the threads in parallel region.

`firstprivate` clause

- The value outside the parallel region is used to initialize each private copy

omp for directive

- Easy way to parallelize a for loop.
- This is the second most important OpenMP directive.

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<N; i++)
    x[i] = x[i] + alpha * y[i];
}
```

The loop index *i* is automatically privatized.

Shorthand

The above example is often abbreviated to

```
#pragma omp parallel for
for (i=0; i<N; i++)
  x[i] = x[i] + alpha * y[i];
```

This is a special case of combining this pattern of two directives into one.

Common patterns

```
#pragma omp parallel for
for (i=0; i<N; i++)
  blah1;

#pragma omp parallel for
for (i=0; i<N; i++)
  blah2;
```

In modern OpenMP implementations, threads are not actually destroyed at the end of the first parallel region. This avoids overhead of thread creation every time threads are spawned.

Loop scheduling

- How are the loop iterations assigned to threads?

Loop scheduling

- How are the loop iterations assigned to threads?
- By default, the loop is split as evenly as possible into chunks, one chunk for each thread.
- This default can be changed with the `schedule` clause

`schedule` clause

Example: `#pragma for schedule(static)`

- `static` using one chunk for each thread (default)
- `static,10` using chunk size 10
- `dynamic` chunks of size 1, assigned whenever threads are “free”
- `dynamic,10` dynamic chunks of size 10
- `guided` chunk size adaptively reduces, with smallest size 1
- `guided,2` guided with smallest chunk size 2
- `runtime` read schedule from `OMP_SCHEDULE` shell variable

`schedule` clause

- dynamic and guided scheduling may have high overhead; often want to use large chunks to reduce this overhead (unless work per iteration is large)
- for guided, the chunk size is proportional to the number of iterations remaining divided by the number of threads (but bounded by the smallest chunk size, which may help reduce overhead)
- static is often good, because memory access for contiguous iterations may be similar or nearby

Example demonstrating loop scheduling

Program outputs the thread id that executes each loop iteration.

```c
#include <stdio.h>
#include <omp.h>

int main()
{
  int i;

  #pragma omp parallel schedule(runtime)
  {
    int thread_id = omp_get_thread_num();
    #pragma omp for
    for (i=0; i<16; i++)
      printf("iteration %2d done by thread %d\n", i, thread_id);
  }

  return 0;
}
```

Reductions using a for loop

```
double sum = 0.;

#pragma omp parallel for
for (i=0; i<N; i++)
  sum = sum + a[i];
```

Above will not work correctly because `sum` is shared.

Reductions using a for loop

Solution: use `reduction` directive

```
double sum;

#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++)
  sum = sum + a[i];

printf("%f\n", sum);
```

`sum` will be privatived, initialized, and properly summed into the shared variable outside the parallel region.

Loop iteration dependencies

- What if there are dependencies between the loop iterations?

Loop iteration dependencies

- What if there are dependencies between the loop iterations?
- Then it is incorrect to use `omp for` to parallelize the loop.

Exercise 2 - due at the beginning of next class

Perform a Brownian dynamics simulation for $n = 10000$ particles in 3-D space initially randomly distributed in a unit box $(0,1) \times (0,1) \times (0,1)$. The propagation formula for particle i is

$$x(i) = x(i) + \sqrt{2\Delta t} \cdot y(i)$$

where $x(i)$ is the position vector of particle i, $\Delta t = 10^{-4}$ is the time step size, and $y(i)$ is a Brownian displacement, which has random components uniformly distributed between -1 and 1.

Write a program that performs 5000 time steps and outputs the average distance moved by the particles every 500 time steps. Note that the average distance is expected to grow proportionally with the square root of time. **Use OpenMP to parallelize your program.**

- Submit a short report with the following sections:
  - Listing of your program.
  - Graph of average distance moved from 0 to 5000 time steps.
  - Table of the execution time for a few different loop scheduling options, and using an appropriate number of threads.

Download these slides at
`http://edmondchow.com/hpc/openmp.pdf`