OpenMP Programming

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Chemical Engineering & Materials Science
Department of Biological Sciences
University of Southern California

Email: anakano@usc.edu
OpenMP

- Portable application program interface (API) for shared-memory parallel programming based on multi-threading by compiler directives
- OpenMP = Open specifications for Multi Processing
- OpenMP homepage
  www.openmp.org
- OpenMP tutorial
  computing.llnl.gov/tutorials/openMP
- Process: an instance of program running
- Thread: a sequence of instructions being executed, possibly sharing resources with other threads within a process

MPI (distributed memory)  OpenMP (shared memory)
Fork-join parallelism

- **Fork:** master thread spawns a team of threads as needed
- **Join:** when the team of threads complete the statements in the parallel section, they terminate synchronously, leaving only the master thread

- **OpenMP threads communicate by sharing variables**
OpenMP Example: `omp_example.c`

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

void main() {
    int nthreads, tid;
    nthreads = omp_get_num_threads();
    printf("Sequential section: # of threads = %d\n", nthreads);
    /* Fork multi-threads with own copies of variable */
    #pragma omp parallel private(tid)
    {
        /* Obtain & print thread id */
        tid = omp_get_thread_num();
        printf("Parallel section: Hello world from thread %d\n", tid);
        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Parallel section: # of threads = %d\n", nthreads);
        }
    } /* All created threads terminate */
}
```

- **Obtain the number of threads & my thread ID** (cf. MPI_Comm_size & MPI_Comm_rank)

- **By default, all variables are shared unless selectively changing storage attributes using private clauses**
OpenMP Example: `omp_example.c`

- **Compilation on** `hpc-login3.usc.edu`
  
  ```
  source /usr/usc/openmpi/default/setup.sh (if bash)
  gcc -o omp_example omp_example.c -fopenmp
  ```

- **Slurm script**
  ```
  #!/bin/bash
  #SBATCH --nodes=1
  #SBATCH --ntasks-per-node=1
  #SBATCH --cpus-per-task=2
  #SBATCH --time=00:00:59
  #SBATCH --output=omp_example.out
  #SBATCH -A lc_an2
  WORK_HOME=/home/rcf-proj/an2/anakano
  cd $WORK_HOME
  export OMP_NUM_THREADS=2
  ./omp_example
  ```

- **Output**
  ```
  Sequential section: # of threads = 1
  Parallel section: Hello world from thread 1
  Parallel section: Hello world from thread 0
  Parallel section: # of threads = 2
  ```

Set the # of threads using environment parameter.
Setting the Number of Threads

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

void main () {
    int nthreads,tid;
    #pragma omp_set_num_threads(2);
    nthreads = omp_get_num_threads();
    printf("Sequential section: # of threads = %d\n",nthreads);
    /* Fork multi-threads with own copies of variable */
    #pragma omp parallel private(tid)
    {
        /* Obtain & print thread id */
        tid = omp_get_thread_num();
        printf("Parallel section: Hello world from thread %d\n",tid);
        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Parallel section: # of threads = %d\n",nthreads);
        }
    } /* All created threads terminate */
}
```

- Setting the number of threads to be used in parallel sections within the program (no need to set OMP_NUM_THREADS); see `omp_example_set.c`
OpenMP Programming Model

- OpenMP is typically used to parallelize (big) loops
- Use synchronization mechanisms to avoid race conditions (i.e., the result changes for different thread schedules)
- Critical section: only one thread at a time can enter

```c
#pragma omp parallel
{
    ...
    #pragma omp critical
    {
        ...
    }
    ...
}
```

Threads wait their turn—only one at a time executes the critical section
Example: Calculating $\pi$

- **Numerical integration**

\[ \int_0^1 \frac{4}{1 + x^2} \, dx = \pi \]

- **Discretization:**

\[ \Delta = \frac{1}{N} \Rightarrow \text{step} = \frac{1}{\text{NBIN}} \]

\[ x_i = (i+0.5)\Delta \quad (i = 0, \ldots, N-1) \]

\[ \sum_{i=0}^{N-1} \frac{4}{1 + x_i^2} \Delta \approx \pi \]

```c
#include <stdio.h>
#define NBIN 100000
void main() {
    int i; double step,x,sum=0.0,pi;
    step = 1.0/NBIN;
    for (i=0; i<NBIN; i++) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = sum*step;
    printf("PI = %f\n",pi);
}
```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
void main() {
    double step, sum=0.0, pi;
    step = 1.0/NBIN;
    #pragma omp parallel
    {
        int nthreads, tid, i;
        double x;
        nthreads = omp_get_num_threads();
        tid = omp_get_thread_num();
        for (i=tid; i<NBIN; i+=nthreads) {
            x = (i+0.5)*step;
            #pragma omp critical
            sum += 4.0/(1.0+x*x);
        }
    }
    pi = sum*step;
    printf("PI = %f\n", pi);
}

Thread-private variables: Either declare private or define within a parallel section

Shared variables

Private (local) variables

This has to be atomic
Race Condition

- **Race condition**: Output is dependent on the sequence or timing of how multiple threads are executed.
- Race condition arises if the read & write operations below are not atomic (a set of operations is atomic if they are executed without being interrupted by other operations).

```plaintext
sum = sum + 1;
```

**Possible scenarios**

- t0 r 0
- t0 r 0
- t1 r 0
- t0 w 1
- t0 w 1
- t1 r 1
- t1 w 2
- t1 w 3

```
0 or 2
1 or 3
2 or 3
0 or 1
```

```
t0 r 0
0 or 2
1 or 3
2 or 3
```
Critical Section

- Critical section degrades scalability, *cf.* Amdahl’s law

\[
T_P = fT_1 + (1 - f) \frac{T_1}{P} \\
S_P = \frac{T_1}{T_P} = \frac{1}{f + \frac{1 - f}{P}} \rightarrow \frac{1}{f} \quad (P \rightarrow \infty)
\]

for \(i=\text{tid}; \ i<\text{NBIN}; \ i+=\text{nthreads}) \) {
  \[
  x = (i+0.5) \times \text{step};
  \]
  \[
  \#\text{pragma omp critical} \quad f \sim 0.5
  \]
  \[
  \text{sum} += \frac{4.0}{1.0+x\times x};
  \]
}

- How to get rid of the critical section?
Avoid Critical Section: `omp_pi.c`

Data privatization: Give each thread a dedicated accumulator

```c
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
int nthreads,tid;
double step,sum[MAX_THREADS]={0.0},pi=0.0;
step = 1.0/NBIN;
#pragma omp parallel private(tid)
{
    int i;
    double x;
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    for (i=tid; i<NBIN; i+=nthreads) {
        x = (i+0.5)*step;
        sum[tid] += 4.0/(1.0+x*x);
    }
}
for(tid=0; tid<nthreads; tid++) pi += sum[tid]*step;
printf("PI = %f\n",pi);
```
Avoid Critical Section: “Wrong” Way

```c
#include <stdio.h>
#include <omp.h>
#define NBIN 100000

void main() {
    double step,sum=0.0,pi;
    step = 1.0/NBIN;
    # pragma omp parallel
    {
        int nthreads,tid,i;
        double x;
        nthreads = omp_get_num_threads();
        tid = omp_get_thread_num();
        for (i=tid; i<NBIN; i+=nthreads) {
            x = (i+0.5)*step;
            // #pragma omp critical
            sum += 4.0/(1.0+x*x);
        }
    }
    pi = sum*step;
    printf("PI = %f\n",pi);
}
```

[anakano@hpc-login3 src]$ ./omp_pi_critical
PI = 3.141593
[anakano@hpc-login3 src]$ ./omp_pi_noncritical
PI = 0.558481 ← 16-thread run
Load Balancing

- Interleaved assignment of loop-index values to threads balances the loads among the threads

```c
for (i=tid; i<NBIN; i+=nthreads) {
    ...
}
```

A bad example