Parallel and Distributed Processing
Class 18
OpenMP I

Attique Dawood

07 March, 2025

Last Modified: Friday 14th March, 2025, 00:58

1 Announcements

- None

2 Revision

- The Monte Carlo method
- C++11 threads

3 Overview

- OpenMP

4 OpenMP

- MP is for multi-programming.
- Provided with most compilers.
- Need to tell compiler to enable openmp.
- Uses `#pragma` directives.

4.1 Parallel regions or blocks

Can be specified with:

```
#pragma omp parallel
... this line of code will be parallelised ...

OR

#pragma omp parallel
{
    ... parallel block of code ...
}

```

```
Can also specify num of threads to create with
#pragma omp parallel num_threads(5)
```

4.2 Helper functions

```
omp_get_thread_num();   // get thread identifier also known as
   thread ID or rank
omp_set_num_threads();  // number of threads to create with next
   parallel block
omp_get_max_threads();  // defaults to available logical cores, can
   be set with omp_set_num_threads(), this is used if
   set_num_threads is not specified
```

4.3 Parallel loop

```
#pragma omp parallel for
for (int i=0; i<100; i++)
{
    ... loop iterations are shared between threads ...
}
```

Can also do

```
#pragma omp parallel
{
    ... threads running in parallel ...
    #pragma omp for
    for (int i=0; i<10; i++)
        ... parallel loop shared by threads...

    for (int i=0; i<150; i++)
        ... not parallelised so ALL threads will run this loop for
           all iterations...
}
```

4.4 Critical section and atomic statements

```
int Counter = 0;
#pragma omp parallel
{
    int tid = omp_get_thread_num();

    #pragma omp atomic
    Counter++;

    #pragma omp critical
    cout << "Hello " << "World! this is thread: " << tid << endl;
}
```

Atomic statements only work for read/write operations. Critical sections can be used to protect entire blocks of code.

5 Private and shared variables

Variables declared globally but specified as private get a copy for each thread (like a local variable) and can be accessed only by those threads. Shared variables act like global.

6 Reduction operation

Reduction operation takes an operator (+,-,/,* etc.) and a list of variable names. The specified operation is performed on all the variables after a loop.

For example, array sum can be done using a sum variable and specifying this in the reduction clause. The compiler takes care of partial sums and then adds them up. Also, no need for atomic statement as that is also taken care of by the compiler.

```
int Sum = 0;
#pragma parallel for reduction(+:Sum)
for (int i=1; i<=1000; i++)
    Sum = Sum+i;    // No need for atomic
cout << "Sum is: " << Sum << endl;
```

7 Complete example

```
// OMP thread creation, summation, reduction and Monte Carlo
   implementation to calculate pi
#include <omp.h>
#include <iostream>
#include <chrono>
#include <random>
std::default_random_engine re;
double randDouble(double min, double max)
{
    std::uniform_real_distribution<double> unif(min,max);
    return unif(re);
}
using std::cout;
using std::endl;
using std::chrono::high_resolution_clock;
using std::chrono::duration_cast;
using std::chrono::duration;
using std::chrono::milliseconds;

int main()
{
    //omp_set_num_threads(5);   // This is used if num_threads not
       specified with #pragma omp parallel
    int counter = 0;
    cout << "Max threads: " << omp_get_max_threads() << endl;
    #pragma omp parallel num_threads(8)
    {
        //const unsigned int tID = omp_get_thread_num();
        #pragma omp for
        for (int i=0; i<5; i++)
        {
            cout << "tID for: " << omp_get_thread_num() << endl;
            #pragma omp atomic
```

```
32              counter++;
33          }
34          #pragma omp critical    // Only 1 thread can execute
               critical block of code OR next statement
35          {
36              cout << "Hello " << omp_get_thread_num() << "world " <<
                   omp_get_thread_num() << endl;
37          }
38          #pragma omp critical
39          cout << "HelloC " << omp_get_thread_num() << " non-critical
               " << omp_get_thread_num() << endl;
40      }
41      //int partialSum;
42      int totalSum = 0;
43      #pragma omp parallel //shared(totalSum) private(partialSum)
44      {
45          cout << "This is thread no: " << omp_get_thread_num() <<
               endl;
46          //totalSum = 0;
47          //partialSum =0;
48          #pragma omp for reduction(+:totalSum)
49          for (int i=1; i<=1000; i++)
50              totalSum=totalSum+i;
51          //partialSum+=i;
52
53          //#pragma omp critical
54          //totalSum+=partialSum;
55      }
56      cout << "Total sum: " << totalSum << endl;
57      cout << "Counter is: " << counter << endl;
58
59      // Monte Carlo to approximate pi
60      // pi = 4*M/N
61      auto t1 = high_resolution_clock::now();
62      int N = 1024*1024*32;
63      int M = 0;
64      for (int i=0; i<N; i++)
65      {
66          double x = randDouble(0,1.0);
67          double y = randDouble(0,1.0);
68          //cout << "x: " << x << ", y: " << y << endl;
69          if((x * x + y * y) < 1.0)
70              M++;
71      }
72      double pi = 4.*(double)M/N;
73      cout << "Pi: " << pi << endl;
74
75      auto t2 = high_resolution_clock::now();
76      duration<double, std::milli> ms_doubleC = t2 - t1;
77      cout << "CPU time Monte Carlo single: " << ms_doubleC.count() <<
            "ms" << endl;
78
79      t1 = high_resolution_clock::now();
80      M = 0;
81      int partialM;
82      #pragma omp parallel private(partialM)
83      {
```

```
84          //  partialM = 0;   // initialise private variable for each
                thread
85          #pragma omp for reduction(+:M)
86          for (int i=0; i<N; i++)
87          {
88              double x = randDouble(0,1.0);
89              double y = randDouble(0,1.0);
90              //cout << "x: " << x << ", y: " << y << endl;
91              if((x * x + y * y) < 1.0)
92                  M++;
93                  //partialM++;   // if not using reduction
94          }
95
96          //pragma atomic
97          //M+=partialM;
98      }
99      pi = 4.*(double)M/N;
100     cout << "Pi: " << pi << endl;
101     t2 = high_resolution_clock::now();
102     ms_doubleC = t2 - t1;
103     cout << "CPU time Monte Carlo multi: " << ms_doubleC.count() <<
            "ms" << endl;
104
105     return 0;
106 }
```

References

8 Additional References

1. https://curc.readthedocs.io/en/latest/programming/OpenMP-C.html
2. https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf
3. https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf