Assignment 1
OpenMP Tutorial Assignment

B. Wilkinson and C Ferner: Modification date Aug 5, 2014

Overview

In this assignment, you will write and execute run some simple OpenMP programs as a tutorial. First you will test OpenMP programs on your own computer. *This will require you to have completed the "pre-assignment" installing the provided virtual machine (or a native Linux installation with the course software stack).*

Later you will test the programs on the UNC-Charlotte's 4-processor (16-core) **cci-grid05.uncc.edu** system. This approach reduces issues of faulty user programs running on the UNC-C cluster that can affect the system in a deleterious manner. Users can also do local editing and testing before running on the cluster.

Part 1 provides basic practice in coding, compiling and running OpenMP programs, covering hello world program, timing, using work sharing for, and sections directives. The OpenMP code is given.

Part 2 asks you to parallelize matrix multiplication using the work sharing **for** directive and draw conclusions. Code for sequential matrix multiplication is given.

Part 3 asks you run the hello world and matrix multiplication programs on **cci-grid05.uncc.edu**.

Preliminaries

The OpenMP programs for this assignment are to be held in the directory **~/ParallelProg/OpenMP**, which is already created in the provided virtual machine, with the sample programs. Cd to this directory.

Part 1 – OpenMP Tutorial (30%)

The purpose of this part is to become familiar with OpenMP constructs and programs, using your own computer.

Task 1 – "hello world" program

An OpenMP hello world program called **hello.c** is given below:

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

```
#include <stdlib.h>

int main (int argc, char *argv[])  {
   int nthreads, tid;

// Fork a team of threads giving them their own copies of variables

   #pragma omp parallel private(nthreads, tid)
   {
   tid = omp_get_thread_num();                  // Obtain thread number
   printf("Hello World from thread = %d\n", tid);

   if (tid == 0) {  // Only master thread does this
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
   }
   }              // All threads join master thread and disband
   return(0);
}
```

This program has the basic **parallel** construct for defining a single parallel region for multiple threads. It also has a **private** clause for defining a variable local to each thread. *Remember that OpenMP constructs such as parallel have their opening braces on the next line and not on the same line.*

Compile the program on your own computer using the command:

```
cc –fopenmp hello.c –o hello
```

Execute the program with the command:

```
./hello
```

You should get a listing showing a number of threads such as:

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 1
```

The default number of threads will depend upon the particular computer system[1]. Execute the program at least four times. Explain your output. Why does the thread order change?

Alter the number of threads to 32. There are actually three ways to do this, see the class notes. Here just try adding the following function:

[1] Usually the number of cores available. Hyperthreading might double this number.

```
omp_set_num_threads(32);
```

before the parallel region **pragma**. Re-execute the program.

What to submit for Task 1

Your submission document should include the following:

1) Screenshot from compiling and running the hello world program on your computer and explanation of output and thread order.
2) Program listing of the hello world program with the number of threads altered
3) Screenshots of the output from running the program with 32 threads.

Task 2 – Work sharing with the for construct

This task explores the use of the **for** work-sharing construct. The following program **worksharing.c** adds two vectors together using a work-sharing approach to assign work to threads:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

int main (int argc, char *argv[]) {
   int nthreads, tid, i, chunk;
   float a[N], b[N], c[N];
   double start, end;               // used for timing

   for (i=0; i < N; i++)
      a[i] = b[i] = i * 1.0;         // initialize arrays

   chunk = CHUNKSIZE;

   start = omp_get_wtime();   //start time measurement

   #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
   {
      tid = omp_get_thread_num();
      if (tid == 0){
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
      printf("Thread %d starting...\n",tid);

      #pragma omp for schedule(dynamic,chunk)
      for (i=0; i<N; i++){
         c[i] = a[i] + b[i];
         printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
      }
```

```
   } /* end of parallel section */

   end = omp_get_wtime();      //end time measurement
   printf("Time of computation: %f seconds\n", end-start);
   return(0);
}
```

This program has an overall **parallel** region within which there is a work-sharing **for** construct. Also the time of execution is recorded by instrumenting the code with **omp-get-wtime()** in two places.

Compile and execute the program. Depending upon the scheduling of work different threads might add elements of the vector. It may be that one thread does all the work. Execute the program several times to see any different thread scheduling. In the case that multiple threads are being used, observe how they may interleave.

Experimenting with Scheduling

Alter the code from **dynamic** scheduling to **static** scheduling and repeat. What are your conclusions? Alter the code from **static** scheduling to **guided** scheduling (chunk size is irrelevant) and repeat. What are your conclusions?

What to submit for Task 2

Your submission document should include the following:

1) A copy of the source program with timing added;
2) Screenshot from compiling and running the program with the original dynamic scheduling;
3) Screenshots from running the program with static and with guided scheduling;
4) Your conclusions about the different scheduling approaches.

Task 3 – Work-sharing with the sections construct

This task explores the use of the **sections** construction. The program **sections.c** below adds elements of two vectors to form a third and also multiplies the elements of the arrays to produce a fourth vector.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 50

int main (int argc, char *argv[]) {
   int i, nthreads, tid;
   float a[N], b[N], c[N], d[N];

   for (i=0; i<N; i++) {      // Some initializations, arbitrary values
      a[i] = i * 1.5;
```

```
      b[i] = i + 22.35;
      c[i] = d[i] = 0.0;
   }

   #pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
   {
      tid = omp_get_thread_num();
      if (tid == 0)  {
         nthreads = omp_get_num_threads();
         printf("Number of threads = %d\n", nthreads);
      }
      printf("Thread %d starting...\n",tid);

      #pragma omp sections nowait
      {
         #pragma omp section
         {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++)   {
               c[i] = a[i] + b[i];
               printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
         }

         #pragma omp section
         {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++)  {
               d[i] = a[i] * b[i];
               printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
         }
      }                        // end of sections

      printf("Thread %d done.\n",tid);

   }                           // end of parallel section
   return(0);
}
```

This program has a parallel region but now with variables declared as shared among the threads as well as private variables. Also there is a sections work sharing construct. Within the sections construct, there are individual section blocks that are to be executed once by one member of the team of threads. *Remember that OpenMP constructs such as sections and section have their opening braces on the next line and not on the same line.*

Compile and execute the program and make conclusions on its execution.

What to submit for Task 3

Your submission document should include the following:

1) Screenshot from compiling and running the program

2) Your conclusions.

Task 4 – For construct with private variables

In this section we will explore private variables. Compile and execute the following code, called **privatetest.c**.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 100000

int main(int argc, char *argv) {
      omp_set_num_threads(2);             //set number of threads here
      int i, j, x, tid;
      double start, end;                  // used for timing

      start = omp_get_wtime();            //start time measurement
      #pragma omp parallel for private(x,tid)
      for (i = 0; i < N; i++)
      for (j = 0; j < N; j++) {
            tid = omp_get_thread_num();
            x=tid;
      }
      end = omp_get_wtime();              //end time measurement
      printf("Time of parallel computation: %f seconds\n", end-start);

      return(0);
}
```

(i) Repeat with the number of threads being 1, 2, 4, and 8. Plot the execution time against number of threads. Monitor the CPU usage in the Task Manager (for a Windows system) or Activity Monitor in **Applications** > **Utilities** for a Mac. Discuss the results

(ii) Remove **x** from the private clause i.e. **private(tid)**. Compile and execute for two threads. Explain the difference in the execution time.

What to submit for Task 4

Your submission document should include the following:

1) Foer (i) above, a graph of the execution time against number of threads, a screenshot of the CPU usage, and a discussion of the results.
2) For (ii) above, the execution time with and without **x** as private variable and an explanation.

Part 2 – Matrix Multiplication (35%)

A skeleton of a matrix multiplication program, **matrixmult.c**, given here:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 256

int main(int argc, char *argv) {
      omp_set_num_threads(2);//set number of threads here
      int i, j, k, x;
      double sum;
      double start, end;                  // used for timing
      double A[N][N], B[N][N], C[N][N], D[N][N];

      // set some initial values for A and B
      for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++)     {
                  A[i][j] = j*1;
                  B[i][j] = i*j+2;
            }
      }


      // sequential matrix multiplication
      start = omp_get_wtime();      //start time measurement
      for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++)   {
                  sum = 0;
                  for (k=0; k < N; k++) {
                        sum += A[i][k]*B[k][j];
                  }
                  C[i][j] = sum;
            }
      }
      end = omp_get_wtime();  //end time measurement
      printf("Time  of  sequential  computation:  %f  seconds\n",  end-
start);

      // Add OpenMP matrix multiplication here, result in array D[N][N]
      //    ...



      // check sequential and parallel versions give same answers
      int error = 0;
      for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++)   {
                  if (C[i][j] != D[i][j]) error = -1;
            }
      }
      if (error == -1) printf("ERROR, sequential and parallel versions
give different answers\n");

      return(0);
}
```

This program includes the sequential matrix multiplication code with the result in **C[N][N]**. Parallel version that you are to add, is to produce its result in **D[N][N]**. The final part of the program checks that the parallel results match the sequential results to make sure the results of your parallel version are correct.

The size of the ***N*** x ***N*** matrices in the program is set to 512 x 512. What happens if you increase the size to 1000 x 1000. Why?

You are to parallelize this matrix multiplication program in two different ways:[2]

1. Add the necessary **pragma** to parallelize the outer **for** loop in the matrix multiplication;
2. Remove the **pragma** for the outer **for** loop and add the necessary **pragma** to parallelize the middle **for** loop in the matrix multiplication;

In both cases, collect timing data with 1 thread, 4 threads, 8 threads, and 16 threads. You will find that when you run the same program several times, the timing values can vary significantly. Therefore add a loop in the code to execute the program 10 times and display the average time.

What to submit from for Part 2

Your submission document should include the following:

1) For the outer matrix multiplication loop parallelized
    a. Source program listing
    b. One screenshot from compiling and running the program
    c. Graphical results of the average timings
2) For the middle matrix multiplication loop parallelized
    a. Source program listing
    b. One screenshot from compiling and running the program
    c. Graphical results of the average timings
3) Your conclusions and explanations of the results.

Part 3 Executing on cluster (30%)

For this assignment, we will test the hello world and matrix multiplication programs on the UNCC **cci-gridgw.uncc.edu** cluster. Specifically, we will use **cci-grid05** – four quad-core processor (16 core) shared memory system. *First carefully read the separate instructions on using this cluster.* You cannot ssh directly into **cci-grid05**. You must log in through the gateway node **cci-gridgw.uncc.edu** first.

[2] Nested for loops was introduced in OpenMP version 2.5 that could parallelize both loops but not tried here.

Log onto the UNCC cluster gateway **cci-gridgw.uncc.edu**. In your home directory, create a directory called **OpenMP** to hold all the files for this part and **cd** into this directory. Transfer the OpenMP hello world and matrix multiplication programs to this directory.

We now want to execute these programs on **cci-grid05.** From **cci-gridgw.uncc.edu**, ssh into **cci-grid05** with the command:

[<username@cci-gridgw ~]$ ssh cci-grid05

Compile and execute the programs from the **OpenMP** directory.

What to submit for Part 3

Your submission document should include the following:

- Sample output for the hello world program on cci-grid05
- Sample output for the matrix multiplication program on cci-grid05
- Conclusions

Grading

Every task and subtask specified will be allocated a score so make sure you clearly identify each part/task you did. Make sure you include everything that is specified in the "What to include ..." section at the end of each task/part. **Include all code, not as screen shots of the listings but complete properly documented code listing.**

Assignment Submission

Produce a ***single pdf*** document that show that you successfully followed the instructions and performs all tasks by taking screenshots and include these screenshots in the document. Submit by the due date as described on the course home page.