Parallel Computing 2012

OpenMP Programming Assignment

Due Date: Dec 17, 2012 (strict deadline!)

Part 1 Tutorial

Part 1 provides some basic skills in running OpenMP programs.

Acknowledgement: This part is derived from the OpenMP tutorial at
https://computing.llnl.gov/tutorials/openMP/exercise.html

Task 1 Preliminaries

For this assignment, Lonestar system will be used – four quad-core processor (16 core) shared memory system. You can ssh directly into this computer.

export lonestar=129.114.53.21
ssh *-o PreferredAuthentications=password* username@lonestar *Or* ssh username@lonestar

It will prompt for your password. Create a directory called **OpenMP** and cd into this directory.
Use the command "idev" to get access to a compute node in interactive job mode.

Task 2: Compile a "hello work" program

This assignment requires a compiler that compiles OpenMP programs. The newer versions of the gcc compiler will compile OpenMP programs but we can also use the Intel C++ compiler called icpc (icc) as this has good support for multithreaded OpenMP programs.

Obtain the hello world program from here. The listing is 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 */
}
```

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.

Compile the program using the command:

icc –openmp –o omp_hello omp_hello.c

Execute the program with the command:

./omp_hello

You should get a listing showing 16 threads such as:

```
Hello World from thread = 0
Number of threads = 16
Hello World from thread = 4
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 8
Hello World from thread = 15
Hello World from thread = 9
Hello World from thread = 14
Hello World from thread = 10
Hello World from thread = 13
Hello World from thread = 12
Hello World from thread = 11
```

Alter the number of threads to 4 by altering the environment variable with the command:

export OMP_NUM_THREADS=4

(bash shell). Check value is correct with:

echo $OMP_NUM_THREADS

Re-execute the program.

Task 3: Work sharing

This task explores the use of the **for** work-sharing construct. The program provided here 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];

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

chunk = CHUNKSIZE;

#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 */

}
```

This program has an overall **parallel** region within which there is a work-sharing **for** construct. 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.

Alter the code from **dynamic** scheduling to **static** scheduling and repeat. What are your conclusions?

Time of execution

Measure the execution time by instrumenting the OpenMP code with the OpenMP routine **omp_get_wtime()** at the beginning and end of the program and finding the difference in time.

Task 4: Work-sharing with the sections construct

This task explores the use of the **sections** construction. The program provided here 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];
```

```
/* Some initializations */
for (i=0; i<N; i++) {
  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 */

}
```

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 worksharing construct. Within the sections construct, there are individual section blocks that are to be executed once by one member of the team of threads.

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

Part 2 Matrix Multiplication

Acknowledgement: This part has been provided by Professor B. Kurtz, Appalachian State University.

In this part, you are to write you own OpenMP program. You are given a skeleton sequential program for matrix multiplication here and below

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

```
#define M 500
#define N 500

int main(int argc, char *argv) {
    //set number of threads here
    omp_set_num_threads(8);

    int i, j, k;
    double sum;
    double **A, **B, **C;

    A = malloc(M*sizeof(double *));
    B = malloc(M*sizeof(double *));
    C = malloc(M*sizeof(double *));

    for (i = 0; i < M; i++)   {
        A[i] = malloc(N*sizeof(double));
        B[i] = malloc(N*sizeof(double));
        C[i] = malloc(N*sizeof(double));
    }

    double start, end;

    for (i = 0; i < M; i++) {
        for (j = 0; j < N; j++)     {
            A[i][j] = j*1;
            B[i][j] = i*j+2;
            C[i][j] = j-i*2;
        }
    }

    start = omp_get_wtime();

    for (i = 0; i < M; i++) {

           for (j = 0; j < N; j++)   {
            sum = 0;
                     for (k=0; k < M; k++) {
                         sum += A[i][k]*B[k][j];
                     }
                     C[i][j] = sum;
            }
    }

    end = omp_get_wtime();

    printf("Time of computation: %f\n", end-start);

}
```

You are to parallelize this algorithm in two different ways:

1. Add the necessary pragma to parallelize the outer **for** loop

2. Remove the pragma for the outer **for** loop and create a pragma for the middle **for** loop

and collect timing data given one thread, four threads, eight threads, and 16 threads and two matrix sizes.

You will find that when you run the same program several times, the timing values can vary significantly. Therefore for each set of conditions, collect ten data values and average them. You are encouraged to use a spreadsheet program either from MS Office or OpenOffice to store your data and perform the necessary calculations.

Here are the conditions you should collect data for:

1. No parallelization at all (that is, the given program)
2. Parallelizing the outer loop with 1, 4, 8, and 16 threads using matrix sizes 50x50 and 500x500
3. Parallelizing the middle loop with 1, 4, 8, and 16 threads using matrix sizes 50x50 and 500x500

Collect timing data for each case; average the result based on ten test runs each.

Make tables and graphs of your average timing data and put them in the submitted report. After you have reported your results, try to explain them as best as possible. Include two source code files (outer loop - 8 threads-500x500 and middle loop - 8 threads-500x500).

Assignment Submission

Submit at github site (dmacsassignments.github.com) by the due date using your own github used id. Combine all your insights, observation and sample outputs into one PDF file. The code and pdf together should be tarred and the submission should be named "name_openmp.tar". (Use just simple tar and not tar.bz2, tar.bz etc). *All students must work individually.*

Note: The files required for this assignments are provided at the "here" links. A file at this link can be downloaded on Lonestar using the command "wget link".

References:

1. Lecture slides used in the class (TACC Training material):
http://www.tacc.utexas.edu/c/document_library/get_file?uuid=ab5adf90-5581-4644-a640-3391ef5b9b9a&groupId=13601
2.Another set of slides used in the class: http://parlab.eecs.berkeley.edu/sites/all/parlab/files/openmp_basics.pdf
3. OpenMP Lab (TACC training material):
http://www.tacc.utexas.edu/c/document_library/get_file?uuid=79f050cd-70d7-45fc-b5ad-c33cab9c608e&groupId=13601
4. TACC Lab user environment (lists the basic commands you need to know):
http://www.tacc.utexas.edu/c/document_library/get_file?uuid=e7812ce9-0190-4d9b-b3a5-c9729cc13710&groupId=13601