Assignment 3
OpenMP Programming Assignment

B. Wilkinson and C Ferner: Modification date Feb 27, 2014 (minor correction in red since Feb 20).

Overview

In this assignment, you will write and execute run OpenMP programs. ***For Spring 2014, we are going to try a new approach so watch for announcements on the home page for any corrections/clarifications.*** First you will test your programs on your own computer. This will require you to have a C compiler that will compile OpenMP programs. The C compiler you used in Assignment 2 may already be suitable (e.g. gcc version 4.7 onwards). Later you will test the programs on the UNC-C 16-core **cci-grid05.uncc.edu** system. This approach reduces issues of faulty user programs running on the UNC-C cluster affecting response time. Users can also do local editing and testing before running on the cluster.

In the *Preliminaries*, OpenMP compiler is installed if necessary.

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. You are also asked to parallelize matrix multiplication using the work sharing **for** directive and draw conclusions. Code for sequential matrix multiplication is given.

Part 2 asks you run the hello world and matrix multiplication programs on **cci-grid05.uncc.edu**.

Part 3 asks you to write your own OpenMP program to model the static heat distribution of a room with a fireplace (two dimensions only) using the stencil pattern. This part also asks you to generate graphical output. Refer to a separate document on how to produce this. The last task in this part is for extra credit and asks you to modify the heat distribution program for dynamic heat distribution where the heat sources vary with time.

Preliminaries – Installing OpenMP compiler

Install a C compiler that will compile OpenMP programs on your own computer if such a compiler does not already exist - see next for possibilities. Although the most recent version of the OpenMP standard is version 4 (July 2013), a compiler that supports just the basic directives (e.g. OpenMP version 2) is sufficient.

Linux, Macs, and Linux Environment on Windows (Cygwin): The GNU gcc compiler (gcc version 4.7 onwards) already supports OpenMP and will compile OpenMP programs by using –fopenmp option, i.e. the command to compile the OpenMP program omp_hello.c is:

```
cc –fopenmp –o omp_hello omp_hello.c
```

Execute the program with the command:

```
./omp_hello
```

Native Windows: As usual, a little more challenging. Possibilities:

- Microsoft Visual Studio 2005 onwards -- already comes with a C/C++ compiler that supports OpenMP version 2.0: http://msdn.microsoft.com/en-us/library/tt15eb9t%28v=vs.120%29.aspx

- TDM-gcc – Port of gcc version 4.8.1 for Windows: http://tdm-gcc.tdragon.net/

I will post any other approaches brought to my attention on the course home page.

Note the Windows port of gcc in minGW (http://www.mingw.org/) is for gcc version 3.4.5. Version 4.7 at least is necessary.

What to submit for Preliminaries

Your submission document should clearly describe how you installed the OpenMP compiler if you needed to.

Part 1 – OpenMP Tutorial (25%)

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 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.*

Create this program and call it **omp_hello.c**. Compile the program on your own computer. Execute the program. 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 number of threads will depend upon the particular computer system. 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 class notes. Here just try adding the following function:

```
    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 **omp_worksharing_for.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];

   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 */
   return(0);
}
```

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.

Time of execution

Measure the execution time by instrumenting the MPI code with the OpenMP routine **omp_get_wtime()** at the beginning and end of the program and finding the elapsed in time. The function **omp_get_wtime()** returns a **double**.

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 **omp_worksharing_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 4

Your submission document should include the following:

1) Screenshot from compiling and running the program
2) Your conclusions.

Task 4 – Matrix Multiplication

A sequential program for matrix multiplication given here:

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

int main(int argc, char *argv) {
   omp_set_num_threads(8);//set number of threads here
   int i, j, k;
   double sum;
   double start, end;               // used for timing
   double A[N][N], B[N][N], C[N][N];

   for (i = 0; i < N; 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();   //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 computation: %f seconds\n", end-start);
   return(0);
}
```

The size of the *N* x *N* matrices in the program is set to 100 x 100. Change this to the maximum size you can use on your system without getting a segmentation fault.

You are to parallelize this matrix multiplication program in two different ways:[1]

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 Task 4

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

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

        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 2 Executing on cluster (25%)

For this assignment, we will test the hello world and matrix multiplication programs on the UNCC **cci-grid0x.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.

Log onto the UNCC cluster gateway **cci-gridgw.uncc.edu**. In your home directory, create a directory called **Assign3** 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 **Assign4** directory.

What to submit for Part 2

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

Part 3 Heat Distribution (50%)

In this part of the assignment, you are to first fully test your programs on your own computer and then upload and test on ***cci-grid05***

The objective is to write an OpenMP program that will model the static heat distribution of a room with a fireplace using a stencil pattern. Although a room is 3 dimensional, we will be simulating the room with 2 dimensions. The room is 10 feet wide and 10 feet long with a fireplace along one wall as depicted in Figure 1. The fireplace is 4 feet wide and is centered along one wall (it takes up 40% of the wall, with 30% of the walls on either side). The fireplace emits 100º C of heat (although in reality a fire is much hotter). The walls are considered to be 20º C. The boundary values (the fireplace and the walls) are considered to be fixed temperatures.

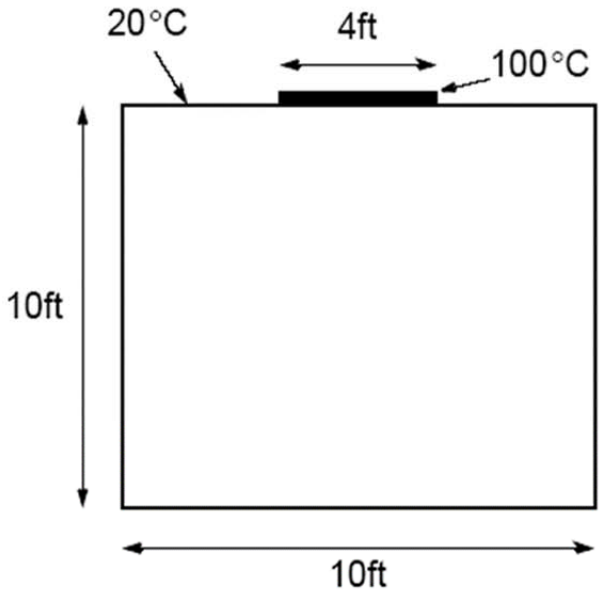

Figure 1: 10x10 Room with a Fireplace

We can find the temperature distribution by dividing the area into a fine mesh of points, $h_{i,j}$. The temperature at an inside point can be taken to be the average of the temperatures of the four neighboring points, as illustrated in Figure 2:

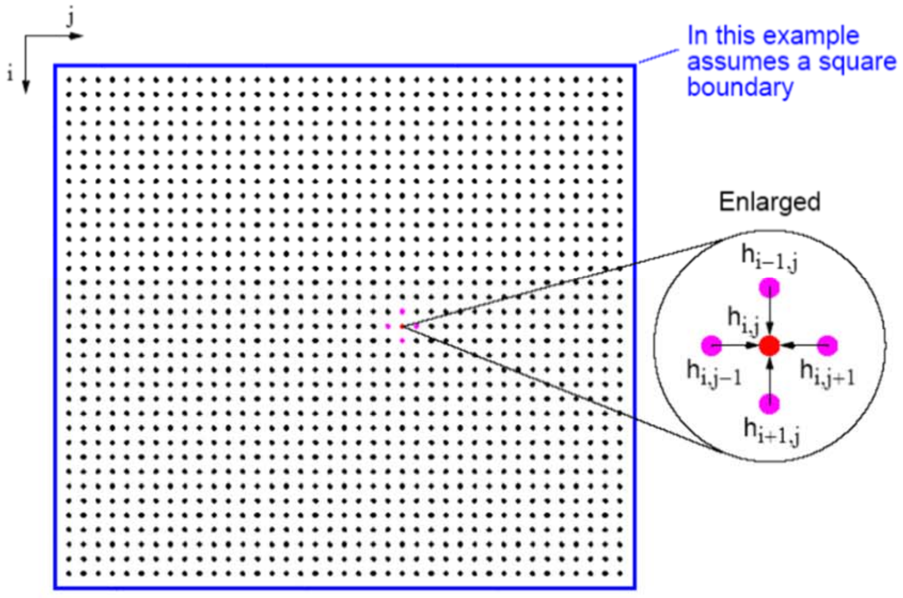

Figure 2: Determining Heat Distribution by a Finite Difference Method

For this calculation, it is convenient to describe the edges by points adjacent to the interior points. The interior points of $h_{i,j}$ are where $0 < i < n$, $0 < j < n$ $[(n - 1) \times (n - 1)$

interior points]. The edge points are when $i = 0$, $i = n$, $j = 0$, or $j = n$, and have fixed values corresponding to the fixed temperatures of the edges. Hence, the full range of $h_{i,j}$ is $0 \leq i \leq n$, $0 \leq j \leq n$, and there are $(n + 1) \times (n + 1)$ points. We can compute the temperature of each point by iterating the equation:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

$(0 < i < n, 0 < j < n)$ for a fixed number of iterations or until the difference between iterations of a point is less than some very small prescribed amount. This iteration equation occurs in several other similar problems; for example, with pressure and voltage. More complex versions appear for solving important problems in science and engineering. In fact, we are solving a system of linear equations. The method is known as the *finite difference* method. It can be extended into three dimensions by taking the average of six neighboring points, two in each dimension. We are also solving Laplace's equation.

Sequential Code. Suppose the temperature of each point is held in an array h[i][j] and the boundary points h[0][x], h[x][0], h[n][x], and h[x][n] $(0 \leq x \leq n)$ have been initialized to the edge temperatures. The calculation as sequential code could be

```
for (iteration = 0; iteration < limit; iteration++) {
  for (i = 1; i < n; i++)
    for (j = 1; j < n; j++)
      g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);
  for (i = 1; i < n; i++)
                                       /* update points */
    for (j = 1; j < n; j++)
      h[i][j] = g[i][j];
}
```

using a fixed number of iterations. Notice that a second array g[][] is used to hold the newly computed values of the points from the old values. The array h[][] is updated with the new values held in g[][]. This is known as Jacobi iteration. Multiplying by 0.25 is done for computing the new value of the point rather than dividing by 4 because multiplication is usually more efficient than division. Normal methods to improve efficiency in sequential code carry over to parallel code and should be done where possible in all instances. (Of course, a good optimizing compiler would make such changes.)

Task 1 Sequential program

The above code can be improved to avoid actually copying the array into another array by using a 3-dimensional array of size N x N x 2, say A[i][j][x] and every iteration, alternate between x = 0 to x =1. Re-write the code given to use a 3-dimensional array avoiding the update loop and complete as a C program to model the heat distribution of room given in Figure 1. Instrument the code so that the elapsed time is displayed. Have *N* x *N* points and *T* iterations where *N* and *T* are defined constants in the program. Output on the console the values of every *N*/8 x *N*/8 points i.e. 8 x 8 points irrespective of the value of *N*.

Test your C program on your own computer and then on cci-grid05.uncc.edu, with *N* = 1000 and *T* = 5000.

What to submit for Task 1

Your submission document should include *(but is not limited to)* the following:

- Your sequential C program listing to solve the heat distribution problem
- On your own computer
    - Screenshot of compiling the program
    - Screenshot of command to execute the program and program output
- On cci-grid05.uncc.edu
    - Screenshot of compiling the program
    - Screenshot of command to execute the program and program output

Task 2 - OpenMP Program

Modify the sequential program in Task 1 to be an OpenMP program. Include:

- Code to compute the heat distribution on in parallel (OpenMP)
- Code to compute the heat distribution on sequentially
- Code to ensure both sequential and parallel versions of heat distribution calculation produce the same correct results**.**
- Statements to time the execution of both sequential and parallel versions.
- Compute the speed-up factor and display.

Test the program on your computer and finally execute on cci-grid05.uncc.edu.

What to submit for Task 2

Your submission document should include *(but is not limited to)* the following:

- Your OpenMP program listing to solve the heat distribution problem
- On your own computer
    - Screenshot of compiling the program
    - Screenshot of command to execute the program and program output
- On cci-grid05.uncc.edu
    - Screenshot of compiling the program
    - Screenshot of command to execute the program and program output

Task 3 Graphical output

Make your sequential heat distribution programs in Task 2 produce X11 graphics displaying the temperature contours at 10°C intervals in color. Execute your program on cci-grid05.uncc.edu and forward the X11 output to your computer. Repeat for the OpenMP program. (Details of X11 code and forwarding are given separately.)

What to submit for Task 3:

Your submission document should include *(but is not limited to)* the following:

- Your OpenMP program listing with X11 code to solve the heat distribution problem
- Screenshot of compiling the program on cci-grid05.uncc.edu
- Screenshot of command to execute the program on cci-grid05.uncc.edu
- Screenshot of the graphical output of the program forwarded to on your computer
- Conclusions

Task 4 Dynamic Heat Distribution (Extra credit +15%)

Previously the heat distribution assumed that the heat sources did not vary with time and we solved Laplace's equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

With heat sources that vary, this equation becomes the general Heat equation (Poisson's equation):

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = \frac{1}{\alpha}\frac{\partial f}{\partial t}$$

Where α is a constant and *f*, the temperature, varies with time.

Modify the heat distribution program for dynamic heat distribution where the heat sources vary with time. You will need to do some research on how to reformulate the difference equations. First write a sequential program and then add OpenMP directives to parallelize.

What to submit for Task 4:

Your submission document should include *(but is not limited to)* the following:

- Your OpenMP program listing with full comments
- Screenshot of compiling the program on cci-grid05.uncc.edu

- Screenshot of command to execute the program on cci-grid05.uncc.edu and results
- 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 "Include in your submission document" section at the end of each 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.

Derivation of Jacobi Iteration Equation

(from [3] page 357)

The steady-state heat distribution is governed by Laplace's equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

(in two dimensions). The two-dimensional solution space is "discretized" into a large number of solution points, as shown in Figure 3. If the distance between the points in the x and y directions, Δ, is made small enough, the

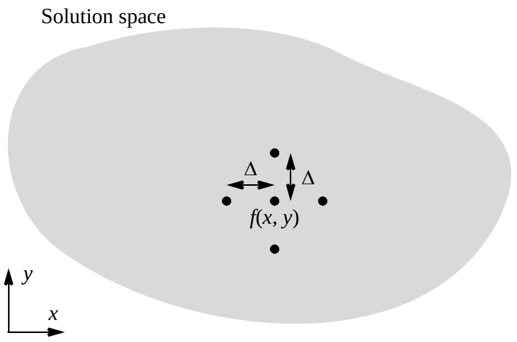

Figure 3 Finite difference method.

central difference approximation of the second derivative can be used:

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta^2}[f(x+\Delta, y) - 2f(x, y) + f(x-\Delta, y)]$$

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{\Delta^2}[f(x, y+\Delta) - 2f(x, y) + f(x, y-\Delta)]$$

[See Bertsekas and Tsitsiklis (1989) for proof.] Substituting into Laplace's equation, we get

$$\frac{1}{\Delta^2}[f(x+\Delta, y) + f(x-\Delta, y) + f(x, y+\Delta) + f(x, y-\Delta) - 4f(x, y)] = 0$$

Rearranging, we get

$$f(x, y) = \frac{[f(x-\Delta, y) + f(x, y-\Delta) + f(x+\Delta, y) + f(x, y+\Delta)]}{4}$$

The formula can be rewritten as an iterative formula:

$$f^k(x, y) = \frac{[f^{k-1}(x-\Delta, y) + f^{k-1}(x, y-\Delta) + f^{k-1}(x+\Delta, y) + f^{k-1}(x, y+\Delta)]}{4}$$

where $f^k(x, y)$ is the value obtained from kth iteration, and $f^{k-1}(x, y)$ is the value obtained from the $(k-1)$th iteration. By repeated application of the formula, we can converge on the solution.

Further Information

[1] Wikipedia "Laplace's equation" http://en.wikipedia.org/wiki/Laplace%27s_equation
[2] Wikipedia "Heat equation" http://en.wikipedia.org/wiki/Heat_equation
[3] Barry Wilkinson and Michael Allen, *Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers 2nd edition*, Prentice Hall Inc., 2005.