INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Spring Semester, 2019
Introduction to Parallel Scientific Computing
(CSE504)

Tutorial II
February 21, 2019

Due: **08.03.19** **Instructor: Dr. Pawan Kumar** Maximum Marks: 20

Instructions:

Assignments submitted after the deadline won't be considered for grading. It is a good idea to type the code rather than copy pasting it. Write answers to your question in some text editor and attach your answer along with codes in a zip file with your roll number as filename.

1. (**Dynamic Schedule Overhead** Dynamic scheduling of for loop iterations by OpenMP is better when the iterations may take very different amounts of time. However, there is some overhead to dynamic scheduling. Code the following program.

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

#define THREADS 10
#define N 100000000

int main ( ) {
  int i;

  printf("Running %d iterations on %d threads dynamically.\n", N, THREADS);
  #pragma omp parallel for schedule(dynamic) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */

  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

Compare the above code with the code below:

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

Student's name: Question 1 continues on the next page...

```
#include <stdio.h>

#define THREADS 10
#define N 100000000

int main ( ) {
  int i;

  printf("Running %d iterations on %d threads statically.\n", N, THREADS);
  #pragma omp parallel for schedule(static) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */

  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

Now consider the code with guided schedule: This scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced:

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

#define THREADS 10
#define N 100000000

int main ( ) {
  int i;

  printf("Running %d iterations on %d threads guided.\n", N, THREADS);
  #pragma omp parallel for schedule(guided) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* a loop that doesn't take very long */

  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

1. Which of the three codes runs faster? Report the times of each of them.

2. What is the run time if the chunk size in dynamic scheduling is set to 100 by replacing `schedule(dynamic)` by `schedule(dynamic, chunk)` in the first code?

Now consider the case when the loop iterations take different amounts of time for different iterations:

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

#define THREADS 4
#define N 16

int main ( ) {
  int i;

  #pragma omp parallel for schedule(guided) num_threads(THREADS)
  for (i = 0; i < N; i++) {
    /* wait for i seconds */
    sleep(i);

    printf("Thread %d has completed iteration %d.\n", omp_get_thread_num( ), i);
  }

  /* all threads done */
  printf("All done!\n");
  return 0;
}
```

Do the following:

1. Compare the above code with `schedule(static)` and `schedule(dynamic)`. Which of these three is fastest? Why?

Things we learn:

- ✓ OpenMP automatically splits for loop iterations for us. Depending on our program, the default behavior may not be ideal.
- ✓ For loops where each iteration takes roughly equal time, static schedules work best, as they have little overhead.
- ✓ For loops where each iteration can take very different amounts of time, dynamic schedules, work best as the work will be split more evenly across threads.
- ✓ Specifying chunks, or using a guided schedule provide a trade-off between the two.
- ✓ Choosing the best schedule depends on understanding your loop.

2. Consider the following code that computes the value of pi using monte carlo method.

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


static long num_trials = 100000000;

int main ()
{
   long i;  long Ncirc = 0;
   double pi, x, y, test, time;
   double r = 1.0;   // radius of circle. Side of squrare is 2*r

   time = omp_get_wtime();
   #pragma omp parallel
   {

      #pragma omp ...
          printf(" %d threads ",omp_get_num_threads());

      seed(-r, r);
#pragma omp ...
      for(i=0;i<num_trials; i++)
      {
         x = drandom();
         y = drandom();

         test = x*x + y*y;

         if (test <= r*r) Ncirc++;
       }
    }

    pi = 4.0 * ((double)Ncirc/(double)num_trials);

    printf("\n %ld trials, pi is %lf ",num_trials, pi);
    printf(" in %lf seconds\n",omp_get_wtime()-time);

    return 0;
}
```

Do the following:

1. Save this file as `pi_mc.c`
2. Compile this file first on your laptop as follows:
   `gcc -o pi_mc pi_mc.c random_par.c random.h -fopenmp`

If it compiles without error, then run your code as follows

```
time ./pi_mc
```

If it runs OK, then proceed to next step.

3. Send only the source files (`pi_mc.c, random_par.c, random.h`) to your home directory on ABACUS server using `scp` command as follows:

   ```
   scp pi_mc pi_mc.c random_par.c random.h user-xx@abacus.iiit.ac.in:~/.}
   ```

   where you replace `user-xx` is your user id alloted by TA. The files `random_par.c` and `random.h` shown above are attached as additional files.

4. Re-compile your code exactly as in item 1 above on the ABACUS machine so that compiler can optimize your code for the ABACUS machine which may have different Caches, registers, etc.

Now, answer the following:

1. Export the maximum number of threads to be used via the environment variable, for example

   ```
   export OMP_NUM_THREADS=2
   ```

   to use two cores, and then use

   ```
   omp_wtime()
   ```

   to time execution of parallel region. Repeat this (no need to recompile) by changing the environment variable `OMP_NUM_THREADS` as above with thread counts: 1,2,4,6, and plot the time obtained using the `time` command as above. Plot the times using `gnuplot`.

   (a) What is the speedup on 6 cores relative to 1 core?
   (b) What is the speedup of 4 cores relative to 1 core?
   (c) Why do you think the time taken on 4 cores is not 1/4th of the time taken on 1 core?

3. Parallelize the Jacobi method using openmp. [3]

In this exercise we consider the Jacobi method for the iterative solution of a boundary value problem (heat equation). In Jacobi iteration, a new matrix of values is computed from a given one by assigning to each inner element of the new matrix the mean value of the four neighboring elements of the old matrix. The iteration is repeated until the maximum change of an element is below a predefined limit. Parallelize the sequential code for the Jacobi iteration (by filling the lines `#pragma omp ...`):

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;
extern double eps;
extern double **New_Matrix(int m, int n);
extern void Delete_Matrix(double **matrix);
int solver(double **a, int n)
{
```

```
int i,j;
double h;
double diff;
int k = 0;
double **b = New_Matrix(n,n);

if (b == NULL) {
cerr << "Jacobi: Can't allocate matrix\n";
exit(1);
}

do {
diff = 0;
for (i=1; i<n-1; i++) {
for (j=1; j<n-1; j++) {
b[i][j] = 0.25 * (a[i][j-1] + a[i-1][j]
  + a[i+1][j] + a[i][j+1]);

/* Determine the maximum change of the matrix elements */
h = fabs(a[i][j] - b[i][j]);
if (h > diff)
diff = h;
}
}

/*
** Copy intermediate result into matrix 'a'
*/
for (i=1; i<n-1; i++) {
for (j=1; j<n-1; j++) {
a[i][j] = b[i][j];
}
}

k++;
} while (diff > eps);

Delete_Matrix(b);

return k;
}
```

Do the following:

1. Save this in a file solver-jacobi.cpp
2. Parallelize by filling the line `#pragma omp parallel for ...` at appropriate place(s). Make sure to choose private and shared variables correctly. Check if it runs correctly by

comparing with sequential program.

3. First compile `solver-jacobi.cpp` on your laptop, if it works correctly, then send your files to CSTAR machine, and recompile on CSTAR machine as follows

   ```
   g++ -fopenmp -o heat heat.cpp solver-jacobi.cpp
   ```

   where `heat.cpp` is attached in assignment zip file. Answer the following:

   (a) Run your binary `heat` with various threads, and plot the run times against the number of threads. What speedup you get on 6 cores compared to using 1 core.

   (b) Compare static, dynamic, and guided scheduling with chunk sizes 100 and 200. Which is better?

4. Type the following program that illustrates race conditions in a multithreaded code.

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

int main() {
    const int REPS = 1000000;
    int i;
    double balance = 0.0;

    printf("\nYour starting bank account balance is %0.2f\n", balance);

    // simulate many deposits
    // #pragma omp parallel for                      // A1
    // #pragma omp parallel for private(balance)     // B1
    for (i = 0; i < REPS; i++) {
    // #pragma omp atomic                            // C1
        balance += 10.0;
    }

    printf("\nAfter %d $10 deposits, your balance is %0.2f\n",
                REPS, balance);

    // simulate the same number of withdrawals
    // #pragma omp parallel for                      // A2
    // #pragma omp parallel for private(balance)     // B2
    for (i = 0; i < REPS; i++) {
    // #pragma omp atomic                            // C2
        balance -= 10.0;
    }

    // balance should be zero
    printf("\nAfter %d $10 withdrawals, your balance is %0.2f\n\n",
            REPS, balance);

    return 0;
```

```
}
```

Do and answer the following:

1. Compile and run 10 times; note that it always produces the final balance 0
2. To parallelize, uncomment `A1+A2,` recompile and rerun, compare results
3. Try 1: recomment `A1+A2`, uncomment `B1+B2`, recompile/run, compare on 1 cores and 4 cores
4. To fix: recomment `B1+B2`, uncomment `A1+A2, C1+C2,` recompile and rerun, compare on 1 core and 4 cores.
5. Uncomment A1 and A2 and append reduction `reduction(+:balance)` and `reduction(-:balance)` clauses in A1 and A2 resp. Compare step 4 with this step.
