ECE 420 Parallel and Distributed Programming Assignment 2 Solutions

Instructor: Di Niu
Email: dniu@ualberta.ca
Department of Electrical and Computer Engineering
University of Alberta

1. In the serial Trapezoidal Rule calculation, we have a for loop containing 1-9999 iterations, each of which calculates the area of a stripe. Then the areas are summed up to give the total approximate area under the function of interest. Suppose now we want to parallelize this for loop with OpenMP using 2 threads. What are the iterations assigned to Thread 0 and Thread 1, respectively, when the clause `schedule(static,2)` is used? What are the iterations assigned to Thread 0 and Thread 1, respectively, when the clause `schedule(guided)` is used?
Note: we assume each iteration takes a constant amount of time and ignore all the scheduling/ synchronization/work assignment overhead associated with the dynamic assignment of work.

Answer: refer to the slides for description of different scheduling policies.

`schedule(static,2):`
Thread 0: 1 2 5 6 … 9997 9998
Thread 1: 3 4 7 8 … 9999

`schedule(guided):`
If we assume each iteration takes a constant amount of time, and ignore all the scheduling/ synchronization/work assignment overhead associated with the dynamic assignment of work, then we have the following allocation:

Thread 0: 1-5000
Thread 1: 5001-9999

This is because after the first chunk of iterations 1-5000 is assigned to thread 0, thread 0 will be always busy until 5000 iterations later. During this time, thread 1 can take all of the rest of the dynamic assignments in chunks of sizes 2500, 1250, 625, 312, 156, 78, 39, 20, 10, 5, 2, 1, 1, respectively. There are 4999 iterations in total in all of these dynamic assignments. And thread 1 still finishes earlier than thread 0.

If we assume each iteration takes an **unpredictable** amount of time, and consider the scheduling/ synchronization/work assignment overhead associated with the dynamic assignment of work, then the assignment of iterations are actually unpredictable.

2. An OpenMP solution to the Odd-Even Transposition Sort is given in the following program. Explain why this program might be inefficient. Modify the program to reduce the runtime.

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Answer: one potential problem is the overhead associated with forking and joining the threads. The OpenMP implementation may fork and join thread_count threads on each pass through the body of the outer loop. We can improve the performance by using the following code:

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

3. The following program finds the maximum value from an array `data`. How can we modify it to improve the performance?

```
int largest = 0;
#pragma omp parallel for
for ( int i = 0; i < 1000; i++ ) {
  #pragma omp critical
  if (data[i] > largest)
     largest = data[i];
}
```

Answer: one improved version (see slides) is

```
int largest = 0;
int lp

#pragma omp parallel private(lp)
{
  lp = 0;
  #pragma omp for
  for ( int i = 0; i < 1000; i++) {
    if ( data[i] > lp )
            lp = data[i];
  }
  if ( lp > largest ) {
    #pragma critical
    {
      if ( lp > largest )
            largest = lp;
    }
  }
}
```

4. The following two program segments implement matrix-vector multiplication, parallelizing it by rows and by columns, respectively.

```
// Parallelize by rows
# pragma omp parallel default(none) shared(v2,v1,matrix,tam) private(i,j)
{
#   pragma omp for
    for (i = 0; i < tam; i++)
        for (j = 0; j < tam; j++)
            v2[i] += matrix[i][j] * v1[j];
}

// Parallelize by columns
# pragma omp parallel default(none) shared(v2,v1,matrix,tam) private(i,j)
{
```

```
      for (i = 0; i < tam; i++)
#           pragma omp for
            for (j = 0; j < tam; j++)
                  v2[i] += matrix[i][j] * v1[j];
}
```

Are these programs correct? If not, please fix the problems.

Answer: the row version is correct. The column version is wrong because of a race condition when writing to `v2[i]`. We can fix it by making private versions of `v2[i]`, filling them in parallel, and then merging them with a critical section. One possible correction is the following code:

```
#pragma omp parallel
{
    float v2_private[tam] = {};
    int i,j;
    for (i = 0; i < tam; i++) {
        #pragma omp for
        for (j = 0; j < tam; j++) {
            v2_private[i] += matrix[i][j] * v1[j];
        }
    }
    #pragma omp critical
    {
        for(i=0; i<tam; i++) v2[i] += v2_private[i];
    }
}
```

Note that we did not explicitly define anything shared or private here. One way is to explicitly define everything. Alternatively, by defining `i` and `j` (and `v2_private`) inside the parallel section, they are made private.

5. What's a possible output result of the following program?

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

int main()
{
    omp_set_nested(1);
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
```

```
        if (omp_get_thread_num() == 0)
            omp_set_num_threads(4);       /* line A */
        else
            omp_set_num_threads(6);       /* line B */

        /* The following statement will print out
         *
         * 0: 2 4
         * 1: 2 6
         *
         * omp_get_num_threads() returns the number
         * of the threads in the team, so it is
         * the same for the two threads in the team.
         */
        printf("%d: %d %d\n", omp_get_thread_num(),
               omp_get_num_threads(),
               omp_get_max_threads());

        /* Two inner parallel regions will be created
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         */
        #pragma omp parallel
        {
            #pragma omp master
            {
                /* The following statement will print out
                 *
                 * Inner: 4
                 * Inner: 6
                 */
                printf("Inner: %d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* line C */
        }

        /* Again two inner parallel regions will be created,
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         *
         * The omp_set_num_threads(7) call at line C
         * has no effect here, since it affects only
         * parallel regions at the same or inner nesting
         * level as line C.
         */

        #pragma omp parallel
        {
            printf("count me.\n");
        }
    }
    return(0);
}
```

Answer:

```
0: 2 4
Inner: 4
1: 2 6
Inner: 6
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
```

6. The following is a piece of C code to calculate a Fibonacci number for $n = 5$:

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

int fib(int n) {
      if (n<2)
            return n;
      else {
            return fib(n-1) + fib(n-2);
      }
}

int main (int argc, char *argv[])
{
      int result;
      printf("Doing sequential fibonacci:\n");
      result = fib(5);
      printf("Result is %d\n", result);
      return 0;
}
```

1) Parallelize the program above using OpenMP Sections directives.

Answer:

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

int fib(int n) {
      if (n<2)
            return n;
      else {
```

```
        int i,j;
#       pragma omp parallel sections firstprivate(n) shared(i,j)
        {
#           pragma omp section
            i = fib(n-1);

#           pragma omp section
            j = fib(n-2);
        }
            return i + j;
      }
}

int main (int argc, char *argv[])
{
      int result;
      printf("Doing Fibonacci (with OpenMP sections):\n");
      result = fib(5);
      printf("Result is %d\n", result);
      return 0;
}
```

2) Parallelize the program above using OpenMP Tasks directives.

Answer:

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


int fib (int n) {
      int i, j;
      if (n<2)
            return n;
      else {
#           pragma omp task shared(i)
                  i=fib(n-1);
#           pragma omp task shared(j)
                  j=fib(n-2);
#           pragma omp taskwait
                  return i + j;
      }
}

int task_fib(int n)
{
    int result;
#   pragma omp parallel
    {
#       pragma omp single nowait
            result = fib(n);
    }
```

```
      return result;
}


int main (int argc, char *argv[])
{
      int result;
      printf("Doing parallel Fibonacci (using OpenMP tasks):\n");
      result = task_fib(5);
      printf("Result is %d\n", result);
      return 0;
}
```

3) If the following environment variables are set, what are the number of threads that are ever launched in the parallel programs 1) and 2), respectively? Assume no `num_threads` clause or OpenMP functions were used in your program.

```
export OMP_NUM_THREADS = 3
export OMP_NESTED = TRUE
export OMP_DYNAMIC = FALSE
```

Answer: in 1) using Sections, since `OMP_NESTED = TRUE,` which means the nested parallelism is enabled, the number of threads ever launched in the program is 15:

fib(5) -> fib(4) + fib(3) 1 original thread +2 more (to make it a team of 3)
fib(4) -> fib(3) + fib(2) +2 more (to make it a team of 3)
fib(3) -> fib(2) + fib(1) +2 more (to make it a team of 3)
fib(3) -> fib(2) + fib(1) +2 more (to make it a team of 3)
fib(2) -> fib(1), fib(0) +2 more (to make it a team of 3)
fib(2) -> fib(1), fib(0) +2 more (to make it a team of 3)
fib(2) -> fib(1), fib(0) +2 more (to make it a team of 3)

In 2) using Tasks, there are always 3 threads in total.