CS701 - PPA - 1

Open Multiprocessing (OpenMP)

The specification of the OpenMP Application Program Interface (OpenMP API) provides a model for parallel programming that is portable across shared memory architectures from different vendors. Compilers from numerous vendors support the OpenMP API. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, work-sharing constructs, and synchronization constructs, and they provide support for sharing and privatizing data.

Note: (a) Include running times of the serial and parallel versions of the loop for all questions (`double omp_get_wtime()` is a useful function). Deadline: 8pm, September 30. Pack your report, code, screenshots and other files in an archive and mail to `cs701.nitk@gmail.com`. (b) This is a team assignment. At most two students per team. One submission per team.

Hello World Example

```
#include "omp.h" /* OpenMP compiler directives, APIs are declared here */
void main()
{

  /* Parallel region begins here */
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf(`` hello(%d) '', ID);
    printf(`` world(%d) \n'', ID);
  }
}
```

Compilation

```
$ gcc -o hello helloworld.c -fopenmp
$ ./hello
```

Programs to Implement

1. Hello World Program

Create a default number of OpenMP threads. For each thread OpenMP can create, print a Hello World message.

2. Hello World Program - Version 2

Create a default number of OpenMP threads. For each thread OpenMP can create, pass the thread id to a function. The function prints out a Hello World message and the thread id. The prototype of the function could be `void printHello(threadID)`.

3. DAXPY Loop

D stands for Double precision, A is a scalar value, X and Y are one-dimensional vectors of size 2^{16} each, P stands for Plus. The operation to be completed in one iteration is X[i] = a*X[i] + Y[i]. Your task is to compare the speedup (in execution time) gained by increasing the number of threads. Start from a 2 thread implementation. How many threads give the max speedup? What happens if no. of threads are increased beyond this point? Why?

Speedup = $\frac{T_N}{T_1}$. T_N: Execution time of a n-threaded OpenMP program; T_1: Execution time of the uniprocessor implementation.

4. Matrix Multiply

Build a parallel implementation of multiplication of large matrices (Eg. size could be 1000x1000). Repeat the experiment from the previous question for this implementation. Think about how to partition the work amongst the threads - which elements of the product array will be calculated by each thread?

5. Calculation of π

This is the first example where many threads will cooperate to calculate a computational value. The task of this program will be arrive at an approximate value of π. The value of π is given by the following integral:

$$\pi = \int_0^1 \frac{4.0}{1+x^2} dx \tag{1}$$

This can be approximated as the sum of the areas of rectangles under the curve shown in Figure 1. In the figure, δx is the width of the rectangle with its height equal to $F(x_i)$ at the middle of interval i.

$$\pi \approx \sum_{i=0}^{N} F(x_i)\Delta x$$

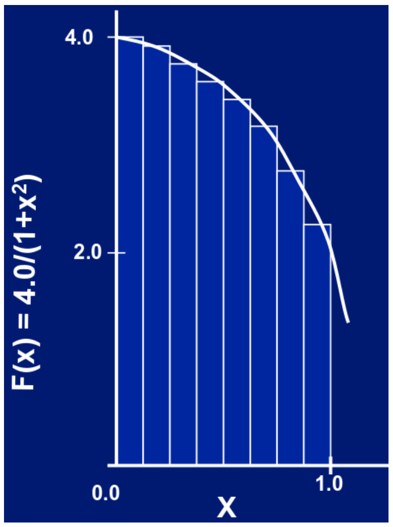

Figure 1: The sum of the areas of the rectangles is an approximation for the value of π.

The serial version of the code follows.

```
static long num_steps = 100000;
double step;
void main ()
{
  int i;
  double x, pi, sum = 0.0;
  step = 1.0/(double)num_steps;
  for (i=0; i<num_steps; i++) {
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Your task is to create a parallel version of the π program using a parallel construct. Which of the variables are shared and which ones are private? In addition to a parallel construct, you may need one or more of the following library routines:

- `int omp_get_num_threads();`: No. of threads in the team.
- `int omp_get_thread_num();`: returns the ThreadID of the calling thread.
- `double omp_get_wtime();`: returns time in seconds since a fixed point in the past.

Points to consider:

- How many threads will your program spawn?

- Assuming each thread has calculated its partial sum ($\frac{4.0}{1.0+x^2}$). The `sum` variable should not be modified by multiple threads simultaneously - `sum` should be modified inside the critical region. Consider the following example:

  ```
  float Sum;
  #pragma omp parallel
  {
    float partial_sum;
    int i, id, nthrds;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();

    for(i=id;i<niters;i+nthrds){
      partial_sum=partial_sum_calculate(i);

      /* Critical section - Only one thread modifies Sum at a time.*/
      #pragma omp critical
        Sum = consumePartialSum(partial_sum);
    }
  }
  ```

  For the `Sum = consumePartialSum(partial_sum);` statement, each thread waits for its turn to execute instructions in the `consumePartialSum` function. The `critical` clause of OpenMP marks the critical region (implements *Mutual Exclusion*).

  For the π calculation, the same effect can be obtained using the `atomic` clause. The `atomic` clause instructs the compiler that the modification of the memory location must complete atomically. A thread will complete the *read from memory-modify in register-write back to memory location* sequence of operations as if it was a single instruction. The thread will not be context switched out during this operation. Only one thread can executing an atomic operation. Example:

  ```
  float Sum;
  #pragma omp parallel
  {
    float partial_sum;
    int i, id, nthrds;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();

    for(i=id;i<niters;i+nthrds){
      partial_sum=partial_sum_calculate(i);

      /* Critical section - Only one thread modifies Sum at a time.*/
      #pragma omp atomic
        Sum = Sum + partial_sum;
    }
  }
  ```

6. Calculation of π - Worksharing and Reduction

Improve the previous implementation of the π calculation using worksharing and the reduction clause.

The OpenMP `parallel` construct creates a "Single Program Multiple Data" (SPMD) program. Each thread redundantly executes the same code. Worksharing splits up pathways through the code between threads.

The loop worksharing construct (`for`) splits up loop iterations among the threads in a team. The following code shows the usage of the worksharing `for` construct.

```
#pragma omp parallel
{
  #pragma omp for
  for (i=0;i<N;i++){
```

```
    work_for_thread(i);
  }
}
```

Note that the worksharing `for` construct should be used after eliminating any dependences between loop iterations.

In the π calculation loop, the partial sums calculated by each thread ($\frac{4.0}{1.0+x^2}$). The partial sums have to be accumulated into the `sum` variable. This step is called reduction. Use the OpenMP `reduction` clause to calculate the sum. (To think: given 16 numbers to add up with 16 processors in hand, what is the fastest way to sum up the numbers?)

```
double  ave=0.0, A[MAX];
int i;
#pragma omp parallel for reduction (+:ave)
  for (i=0;i< MAX; i++) {
    ave + = A[i];
  }
ave = ave/MAX;
```

- The worksharing `for` construct has been appended to the `parallel` clause. This is short hand (instead of using two compiler directives as in the previous example).
- The reduction clause is `reduction (op : list)`. `op` indicates the operation to be used for reduction (+ in the case of the π program). `list` are the variables in reduction (`sum` in the π serial code).

7. Calculation of π - Monte Carlo Simulation

The objective of this program will be to estimate the value of π using the probability of landing a dart inside a circular boundary on a dart board. The probability of landing a dart inside the circle is proportional to ratio of areas:

$$P = \frac{A_c}{A_r} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

For this question:

- Serial version of the code: Compute π by randomly choosing points, count the fraction that falls in the circle. Compute pi. Write your own pseudo random number generator.
- Create a parallel version of this program. Use the `private` clause to specify the variables local to a thread.

Bonus:

- Make the random number generator threadsafe.
- Make your random number generator numerically correct (non-overlapping sequences of pseudo-random numbers).

8. Producer-Consumer Program

The producer-consumer sequence involves a producer process filling in data into a memory location and a consumer process reading from the memory location and performing an action on it. An example producer-consumer operation is for the consumer to calculate the sum of an array after it has been populated by the producer. The example serial code is shown below. Parallelize the following Producer-Consumer program.

```
int main()
{
  double *A, sum, runtime;
  int flag = 0;
  A = (double *)malloc(N*sizeof(double));
  runtime = omp_get_wtime();
  fill_rand(N, A);        // Producer: fill an array of data
  sum = Sum_array(N, A);  // Consumer: sum the array
  runtime = omp_get_wtime() - runtime;
  printf(" In %lf seconds, The sum is %lf \n",runtime,sum);
}
```

For the parallel implementation, take care of these:

- There are two threads - a producer and a consumer thread. Use the `sections` and `section` clauses to assign code blocks to each thread. Example:

```
#pragma omp parallel
{
  #pragma omp sections // Assign code in each  section to individual threads
  {
    #pragma omp section
      X_calculation();  // Thread 1
    #pragma omp section
      y_calculation();  // Thread 2
    #pragma omp section
      z_calculation();  // Thread 3
  } // default barrier here.
}
```

- How does the consumer know that the producer has completed populating the array? The consumer can be notified by setting a global flag. *Hint:* Use the `flush` clause. The `flush` clause forces an update to the memory location as soon as a thread has modified a variable. The producer writes a flag that can be flushed. The following code sequence flushes the updated value of the flag variable to memory.

```
  flag = 1;
  #pragma omp flush (flag)
```

- The consumer thread waits (in an loop) for the producer to complete populating the array - it waits for the flag to be updated.

On successful completion, the code implements pairwise synchronization between threads.

9. Producer-Consumer Program - Version 2

Implement another version of the Producer-Consumer program. In this version, the producer thread updates the queue (Enqueue operation) and the consumer thread reads from the tail (Dequeue operation). The situation is illustrated in Figure 2).

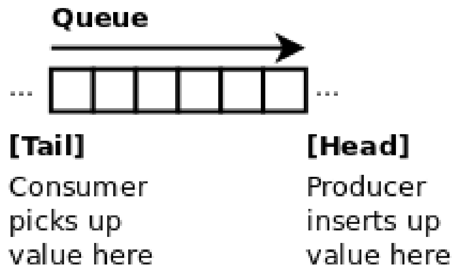

Figure 2: Illustration of the producer-consumer queue situation.

The pseudocodes for the enqueue and dequeue operations follow.

```
if (tail == NULL)         if (head->next == NULL)
update head and tail      update head and tail
else                      else
update tail               update head
```

Points to note:

- The producer thread updates the `head` and the `tail` when the queue is empty. On a non-empty queue only the `head` is modified by the producer.

- In a non-empty queue where total elements is greater than 1, the consumer updates the `tail` value only. When there is only one element in the queue, the consumer has to update both the `head` and the `tail` variable.

- One way of preventing two threads from reading and writing a memory location simultaneously is by marking the code with the `critical` clause. Another synchronization primitive is the *lock*. OpenMP's lock functions are: `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`, `omp_destroy_lock()`. Example usage of the code follows:

```
omp_lock_t lck;
omp_init_lock(&lck); // create lock.

#pragma omp parallel private (tmp, id) // each thread has its own copy of tmp, id
{
  id = omp_get_thread_num();
  tmp = do_lots_of_work(id);

  /* The first thread to acquire the lock gets to execute printf.
     Other threads wait for their turn */
  omp_set_lock(&lck);

  printf(``%d %d'', id, tmp);

  /* The lock is released. */
  omp_unset_lock(&lck);

}
omp_destroy_lock(&lck);   // free memory
```

- The implementation of the queue is your choice. The code may update random values in the queue. You may terminate the program after 100 write-read operations.

10. Producer-Consumer Program - Version 3

Repeat the Producer-Consumer implementation from Q9. In this version, create a multiple number (your choice) of producer and consumer threads. The number of producer threads may be different from the number of consumer threads. One added complication in this situation is that there are multiple producers(consumers) vying to update the `head`(`tail`) value.

11. Molecular Dynamics Simulation

In this experiment, your task is to parallelize a simple molecular dynamics simulation. The code for the calculation of particle forces of the melting of solid argon is presented in the subroutine `forces` (in `forces.c`).

- Use the worksharing `parallel for` construct and atomics in your implementation.
- Which variables should be SHARED, PRIVATE or REDUCTION variables.
- Experiment with different schedules kinds.

1. Once you have a working version, move the parallel region out to encompass the iteration loop in `main.c`.
   - Code other than the forces loop must be executed by a single thread (or workshared).
   - How does the data sharing change?
2. The atomics are a bottleneck on most systems.
   - This can be avoided by introducing a temporary array for the force accumulation, with an extra dimension indexed by thread number.
   - Which thread(s) should do the final accumulation into `f`?

Epilogue

This document borrows heavily from the excellent *A "Hands-on" Introduction to OpenMP* tutorial by Tim Mattson and Larry Meadows[1] (Videos on Youtube are here Tim Mattson's OpenMP Video Tutorial). For the OpenMP definitions of all the terms used in this document refer to the OpenMP 4.0 specification document[2]. The API examples document from the OpenMP website is here[3]. OpenMP website [4] has the OpenMP specification, resources and other useful links. Some excellent references to understand concepts with hands-on exercises are listed in the References section ([5][6])[7][8][9].

References

[1] T. Mattson and L. Meadows, "A "Hands-on" Introduction to OpenMP," http://openmp.org/mp-documents/omp-hands-on-SC08.pdf, OpenMP tutorial at SC08.

[2] "OpenMP Application Program Interface Specification," http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[3] "OpenMP API Examples," http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf.

[4] "OpenMP Website," http://openmp.org/wp.

[5] B. Barney, "OpenMP," https://computing.llnl.gov/tutorials/openMP/, Lawrence Livermore National Laboratory.

[6] Wikipedia, "OpenMP — Wikipedia, the free encyclopedia," 2015. [Online]. Available: https://en.wikipedia.org/w/index.php?title=OpenMP&oldid=685091436

[7] J. Landman, "OpenMP in 30 Minutes," http://www.linux-mag.com/id/4609/.

[8] J. Yliluoma, "Guide into OpenMP: Easy multithreading programming for C++," http://bisqwit.iki.fi/story/howto/openmp/.

[9] M. Süss and C. Leopold, "Common mistakes in OpenMP and how to avoid them: a collection of best practices," in *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming*, ser. IWOMP'05/IWOMP'06. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 312–323. [Online]. Available: http://portal.acm.org/citation.cfm?id=1892830.1892863