

## Self-Evaluation Exercise on OpenMP

1. Write three ways in which the total number of threads can be specified in an OpenMP program.

In the source code: `omp_set_num_threads(NUM)`

On the command prompt: `export OMP_NUM_THREADS=2`

In the pragma: `#pragma omp .... num_threads(THREADS)`

2. Consider these two codes. Are they equivalent?

### Code A:

```
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

### Code B:

```
double res[MAX]; int i;
#pragma omp parallel for
for (i=0; i< MAX; i++) {
    res[i] = huge();
}
```

Answer: yes, they are equivalent.

3. Consider this piece of code.

```

const int MAX=6;
int big(int k) {return k;}
int main() {
    int i, j, A[MAX];
    j = 5;
    for (i=0; i< MAX; i++) {
        j +=2;
        A[i] = big(j);
    }
}

```

Can this be parallelized simply by writing "#pragma omp parallel for" just before the for loop? If yes, show the final value of array A. If no, change the code suitably to allow for-loop parallelization and then, show the final value of array A.

Answer: No, this for loop has loop-carried dependency. So, we need to remove this dependency, so that all loop iterations become independent, as follows:

```

const int MAX=6;
int big(int k) {return k;}
int main() {
    int i, A[MAX];
    for (i=0; i< MAX; i++) {
        int j = 5 + 2*i;
        A[i] = big(j);
    }
}

```

4. Replace WRITE\_PRAGMA\_HERE by suitable pragma to correctly compute average value in parallel.

```
double avg=0.0, A[MAX]; int i;
WRITE_PRAGMA_HERE
for (i=0;i< MAX; i++) {
    avg += A[i];
}
avg = avg/MAX;
```

Answer: the following pragma needs to be added.

**#pragma omp parallel for reduction (+:avg)**

5.

Consider “Serial\_ComputingPi\_IntegrationApproximation.cpp” program which shows serial code. The code computes pi by approximating integration operation.

Parallelize the code by adding suitable OpenMP directive.

Side Note: The pi value can also be computed Monte Carlo approach. See the following link: <http://jakascorner.com/blog/2016/05/omp-monte-carlo-pi.html>

See the attached file.

6. In CNNs, the computation pattern of convolution layers and fully-connected (FC) layers can be modeled as matrix-matrix multiplication (MMM) and matrix-vector multiplication (MVM), respectively. Since MVM is just a simple case of MMM, we will now focus on MMM. As we will discuss later during this course, in AlexNet, convolution layers and FC layers account for 93\% and ~7% of total computations. Thus, any improvement in performance of convolution layer can directly improve the performance of AlexNet (and other CNNs in general).

See "Serial\_MatMul.cpp". Parallelize it using OpenMP. Ensure that you clearly specify private/shared scope for all the variables. Observe speedup using different number of threads.

See attached file.

7. Consider the following code to compute pi. If required, replace WRITE\_PRAGMA\_HERE with suitable pragma to compute pi correctly. Otherwise, just say "no code required".

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

static long num_steps = 10000000; double step;
#define NUM_THREADS 2

int main ()
{ double pi; step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
  int nthreads;
  #pragma omp parallel
  {
    int i, id,nthrds; double x, sum;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    if (id == 0) nthreads = nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
    }
    //WRITE_PRAGMA_HERE

    pi += sum * step;
```

```
}
```

```
printf("pi = %f\n", pi);
```

```
}
```

Answer: #pragma omp critical