Practical 2

ICHEC

April 22, 2020

1 Hello world!

1. **Build and run** the code in Fig. 1.
2. Write a simple OpenMP code to fork a team of threads, obtain and print the thread IDs, have the master thread print the total number of threads.
3. Write a simple OpenMP code with three parallel regions each with a different number of threads, have the master thread print the total number of threads. Define the number of threads as an environment variable. Then, use **omp_set_num_threads()** function and/or **num_threads()** clause to change them for each parallel region.

```
gfortran source.F90 -o source.X -fopenmp
```

```
gcc source.c -o source.X -fopenmp
```

```
#include<stdio.h>
#include<omp.h>
int main(void){
  int tid, nthreads;
#pragma omp parallel private(tid),shared(nthreads)
  {
    tid=omp_get_thread_num();
    nthreads=omp_get_num_threads();
    printf("Hello from thread %d out of %d\n", tid,nthreads);
  }
 }
```

```
 program hello
  use omp_lib
  implicit none
  integer :: tid, nthreads
!$omp parallel private(tid), shared(nthreads)
    tid=omp_get_thread_num()
    nthreads=omp_get_num_threads()
    write(*,'(a,1x,i0,1x,a,i0)'), 'Hello from thread',tid,&
      'out of',nthreads
!$omp end parallel
 end program hello
```

Figure 1. OpenMP Hello World! samples. Top C and bottom Fortran.

2 Vector addition

Write a simple program in your favourite language adding two vectors of double precision numbers, $c = a + b$ with each of length n.

1. Read in n and generate the vectors. Compute the addition multiple times to get a descent run time.
2. Parallelise the code using OpenMP
3. The iterations of the loop will be distributed dynamically in chunk sized pieces. No synchronization is required. Experiment with modifying the chunk size, array size and using a static and guided distribution.

3 Dot Product

Reuse the code dotNaive.c or dotNaive.f90 from the previous week.

1. Parallelise the code using OpenMP. Split threads over number of iterations and the dot product plus use a reduction clause.
2. Have the summary stats over thread not iteration.
3. Use environment variables to modify the schedule of the loop.
4. Generate a new version of the code using OpenMP but this time use the critical directive instead of the reduction clause
5. For each of the versions above generate a scalability curve using $n = 10^4$ and $n = 10^6$ for 1, 3, 6 and 12 threads.
6. Do the curve above for at least two different schedules of your choice.

```
export OMP_SCHEDULE="guided, 4"

other valid values:
  dynamic[, n]
  guided[, n]
  runtime
  static[, n]

If specifying a chunk size with n,
the value of n must be an integer value of 1 or greater.

The default scheduling algorithm is static.
```

4 Race Condition

In the following code (see Fig. 2 or Fig. 3), the coder wants to generate an array with the same elements. The array is is initialised randomly but the value is passed in sequence from thread

0 to $nthreads - 1$. The program does not work as there is a race condition and the threads do not execute in sequence.

1. Try with different values of n and see how the program changes. You only need a few threads and the program should not take long to run.

2. How could you remove the race condition?

3. How can you ensure that the random initial value is passed from a thread to the next inside the parallel construct.

```c
#include<stdio.h>
#include<stdlib.h>
#include<omp.h>
#include<time.h>

int main(void){
    int i, j, n, tid, astart, nthreads, *a;

/* Enter the array size */
    printf("Please enter the size of the array\n");
    scanf("%d",&n);
    if (n<2 || n>1000) {
      printf(" Enter a positive number in range 2<n<10001\n");
      exit(1);
    }

    a = (int *) malloc(n*sizeof(int));
    if (a == NULL) {
      printf(" Cannot allocate array for id %d, stopping\n",tid);
      exit(2);
    }
    a[0] = 0;


/* Start of parallel region */
#pragma omp parallel private(i,j,tid), shared(nthreads,n,a)
{
      tid=omp_get_thread_num();
      nthreads=omp_get_num_threads();

/* Generate different random numbers */
      srand(time(NULL)*tid);
      a[0] = a[0] + rand()%11;


/* Set all elements per thread equal */
      for (j=1; j<n; j++) a[j] = a[0];

      printf("Hello from thread %d out of %d my a is: %d\n",
            tid,nthreads,a[0]);

} /* end parallel region */

/* Check that value from last thread saved */
    printf("Hello from the master thread my a is: %d\n",a[0]);

    free(a);
    return 0;
}
```

Figure 2. Sample C code showing a race condition.

```fortran
program race
    use omp_lib
    implicit none
    integer (kind=4) :: i, n, tid, nthreads, astart, ierr
    real (kind=4) :: ainit(1)
    logical (kind=1) :: test
    integer (kind=4), allocatable :: a(:)

! Read size of array
    write(6,*) ' Please enter the size of the array 2<n<10001 '
    read(5,*) n
    if (n.LE.2 .OR. n.GT.1000) then
      write(6,*) ' Array size must be in the range 2<n<10001, stopping '
      stop
    endif

    allocate(a(n),stat=ierr)
    if (ierr .NE. 0) then
      write(6,*) ' Cannot allocate arrays '
      stop
    endif

    a(1)= 0


! Start of parallel region
!$omp parallel private(tid,ainit), shared(nthreads,a)
      tid=omp_get_thread_num()
      nthreads=omp_get_num_threads()


      call random_seed()
      call random_number(ainit)
      a = a(1) + nint(10.0*ainit(1))


      write(*,'(a,1x,i0,1x,a,i0,a,1x,i0)') 'Hello from thread', &
          tid,'out of ',nthreads, ",my a is:",a(1)
!$omp end parallel
! End parallel region


        write(*,'(a,1x,i0)') 'Hello from master thread, my a is:',a(1)

  deallocate(a)

end program race
```

Figure 3. Sample Fortran code showing a race condition.