ACM40640/PH504 Practical 4

ICHEC

2022/23 Spring

1 Vector addition

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

1. 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.

2. Use the runtime schedule and environment variables to modify the schedule of the loop.

```
export OMP_SCHEDULE="guided, 4"

other valid values:
  dynamic[, n]
  guided[, n]
  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.
```

2 Dot Product

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

1. Parallelise the code using OpenMP.

2. Split threads over number of iterations in the dot product function; use reduction clause.

3. Generate a new version of the code using the critical directive instead of the reduction clause

3 Race Condition

In the following code (see Fig. 1 or Fig. 2), the coder wants to generate an array with the same elements. The array 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 1. Sample C code showing a race condition.

```
 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 2. Sample Fortran code showing a race condition.