AM250 HW5: OpenMP

Owen Morehead

May 17 2022

1 Running the Programs

A `README.md` file is included which contains the commands necessary to compile and run each of the programs below. Brief instructions are also included here. To compile each of the programs, the command used is:

```
gfortran -fopenmp program.f90 -o program
```

where the name of the program, `program.f90`, is the name provided in each section title below, and the name of the executable, `program`, was chosen to be the exact same name as the program without the `.f90` extension. After the executables are made (they are also attached in the homework submission), they can be run interactively on `Grape` using the commands:

```
export OMP_NUM_THREADS=N
./program
```

where the first line is used to decide how many, N threads to run the program with. I am using the `export` command instead of, `setenv OMP_NUM_THREADS N`, since I am working on bash and not csh.
The number of threads used to run the program varies as I tested the program for multiple threads. Details are stated in the `README.md` file and also under each section below for the specific output contained in this deliverables report.

2 Hello World – hello_world_omp.f90

This program writes out "Hello World" from each thread, stating the thread rank. The total number of threads used is also printed out by the 0 thread. Indeed we see that the output is non-deterministic.

```
-bash-4.1$ gfortran -fopenmp hello_world_omp.f90 -o hello_world_omp
-bash-4.1$ ls
 hello_world_omp  hello_world_omp.f90  hello_world_omp.f90~
-bash-4.1$ export OMP_NUM_THREADS=8
-bash-4.1$ ./hello_world_omp
  Hello World from thread =            4
  Hello World from thread =            5
  Hello World from thread =            0
  Number of threads =            8
  Hello World from thread =            6
  Hello World from thread =            2
  Hello World from thread =            3
  Hello World from thread =            7
  Hello World from thread =            1
```

Figure 1: Screenshot of the terminal output which shows the result from running the hello_world_omp.f90 program in parallel on OpenMP using 8 threads.

3 Matrix Product – matprod_omp.f90

This program runs using single precision floating point arithmetic and initializes two symmetric matrices A and B with random values between 0 and 5, and the size declared by the user at runtime. It then calcu-

lates the matrix product, $C = AB$ and finds the minimum value in the C matrix and its location in the matrix. There are two version which are both written in the `matprod_omp.f90` program. The first version uses `OpenMP PARALLEL` DO to loop over the matrix elements in order to compute the matrix multiplication. The second version uses `OpenMP WORKSHARE` with `Fortran's MATMUL` to calculate the matrix multiplication, `MINVAL` to find the minimum value, and `MINLOC` to calculate the minimum values location in the C matrix. All floating point arithmetic is done in single precision. Screenshots of the terminal output showing the total compute time of both methods side by side are shown below. Only the parallel computation is included inside the start and end `OMP_GET_WTIME` calls. The variables for this timer are in double precision. The results were computed for a variety of total threads, and a variety of matrix sizes.

The results below start with smaller matrices and end with the larger matrices. The image screenshots are in the following order, with total times also included in the table below.

Matrix Size	Total Threads	DO LOOP TIME (s)	WORKSHARE TIME (s)
20	4	2.795×10^{-4}	5.829×10^{-5}
100	10	7.189×10^{-3}	1.158×10^{-3}
500	10	0.374	0.168.
1000	5	2.806	0.523
1000	10	1.671	0.525
1000	16	1.139	0.531
2000	10	9.890	4.118
2000	16	6.515	4.122.

We see that as expected, the parallel `WORKSHARE` method using `Fortran's` built-in functions computes the results faster than the parallel `DO LOOP` method. As more threads are used on the same sized large matrix (i.e. 1000×1000 and 2000×2000), the `DO LOOP` method is computed in a shorter time where the `WORKSHARE` method stays around the same, thus the time difference between the two methods becomes less, however the `WORKSHARE` time is still faster. For example, the `WORKSHARE` method is around 3 times faster than the `DO LOOP` method when using 10 threads to compute a 1000×1000 matrix, and the `WORKSHARE` method is only around 2 times faster when 16 threads are used to compute the 1000×1000 matrix, due to the fact that the `DO LOOP` method becomes faster, while the `WORKSHARE` method stays around the same total computation time. A possible reason the `WORKSHARE` method obtains around the same 'quick' total time for different amounts of threads is because `WORKSHARE` has overhead to setup the problem with however many threads are used. Therefore, for more threads, there is more overhead time, which can mask the actual computation time. In other words, the cost of setting up the problem could be more or around equal to the the cost of computation. This can explain why the total time stays around the same for different threads being used to compute on the same sized matrix. A more detailed performance evaluation on these two methods would better reveal the results we see here. Nevertheless, the respective terminal output screenshots are shown below.

```
-bash-4.1$ export OMP_NUM_THREADS=4
-bash-4.1$ ./matprod_omp
 Enter the dimension of the square arrays:
20
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    73.583611     and its location in C: (   6.0000000       19.000000     )
 Work took :  2.79508065432310104E-004 seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    73.583611     and its location in C: (   6.0000000       19.000000     )
 Work took :  5.82949724048376083E-005 seconds
```

Figure 2: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 4 threads on 20×20 matrices in the matrix multiplication.

```
[-bash-4.1$ export OMP_NUM_THREADS=10
[-bash-4.1$ ./matprod_omp
 Enter the dimension of the square arrays:
[100
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    447.52853     and its location in C: (   82.000000       45.000000     )
 Work took :  7.18864798545837402E-003 seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    447.52853     and its location in C: (   82.000000       45.000000     )
 Work took :  1.15800509229302406E-003 seconds
```

Figure 3: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 10 threads on 100×100 matrices in the matrix multiplication.

```
[-bash-4.1$ export OMP_NUM_THREADS=10
[-bash-4.1$ ./matprod_omp
 Enter the dimension of the square arrays:
[500
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    2622.4695     and its location in C: (   126.00000       271.00000     )
 Work took :  0.37403974286280572      seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    2622.4695     and its location in C: (   126.00000       271.00000     )
 Work took :  0.16817525308579206      seconds
```

Figure 4: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 10 threads on 500×500 matrices in the matrix multiplication.

```
[-bash-4.1$ export OMP_NUM_THREADS=5
[-bash-4.1$ ./matprod_omp
 Enter the dimension of the square arrays:
[1000
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    5414.7705     and its location in C: (   975.00000       355.00000     )
 Work took :   2.8063287851400673      seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    5414.7705     and its location in C: (   975.00000       355.00000     )
 Work took :  0.52327633788809180      seconds
```

Figure 5: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 5 threads on 1000×1000 matrices in the matrix multiplication.

```
[-bash-4.1$ export OMP_NUM_THREADS=10
[-bash-4.1$ ./matprod_omp
 Enter the dimension of the square arrays:
[1000
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    5414.7705     and its location in C: (   975.00000       355.00000     )
 Work took :   1.6712223519571126      seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    5414.7705     and its location in C: (   975.00000       355.00000     )
 Work took :  0.52496930910274386      seconds
```

Figure 6: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 10 threads on 1000×1000 matrices in the matrix multiplication.

```
-bash-4.1$ export OMP_NUM_THREADS=16
-bash-4.1$ ./matprod_omp
 Enter the dimension of the square arrays:
1000
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    5414.7705     and its location in C: (   975.00000       355.00000     )
 Work took :   1.1387517328839749      seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    5414.7705     and its location in C: (   975.00000       355.00000     )
 Work took :  0.53090052003972232      seconds
```

Figure 7: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 16 threads on 1000×1000 matrices in the matrix multiplication.

```
-bash-4.1$ export OMP_NUM_THREADS=10
-bash-4.1$ ./matprod_omp
 Enter the dimension of the square arrays:
2000
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    11245.772     and its location in C: (   914.00000       1035.0000     )
 Work took :   9.8901696549728513      seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    11245.772     and its location in C: (   914.00000       1035.0000     )
 Work took :   4.1178901239763945      seconds
```

Figure 8: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 10 threads on 2000×2000 matrices in the matrix multiplication.

```
 Enter the dimension of the square arrays:
2000
 ----- PARALLEL DO LOOP VERSION -----

 min value in C:    11245.772     and its location in C: (   914.00000       1035.0000     )
 Work took :   6.5154801551252604      seconds

 ----- PARALLEL WORKSHARE VERSION -----

 min value in C:    11245.772     and its location in C: (   914.00000       1035.0000     )
 Work took :   4.1219169530086219      seconds
```

Figure 9: Screenshot of the terminal output which shows the result from running the matprod_omp.f90 program in parallel on OpenMP using 16 threads on 2000×2000 matrices in the matrix multiplication.