**ASSIGNMENT 2**

**GROUP: 08**

**ADEBOWALE ABIODUN SAHEED (e12045326)**

**JIMOH HAMMED (e12045329)**

1. (1.) **Question:** What do $a$ and $t$ count?

   **Answer:** $a$ is the thread that is currently in use and $t$ is the number of times thread $a$ is used.

   (2.) Values for all elements in $a$ and $t$.

Table 1: Values for all elements in a

| Case/$a$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Static | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 |
| static,1 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 |
| static,3 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 0 | 0 | 0 |
| dynamic,1 | 1 | 2 | 2 | 1 | 2 | 3 | 1 | 2 | 1 | 2 | 3 | 1 | 2 | 3 | 1 |
| dynamic,5 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 0 |
| guided,2 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 2 | 1 | 1 |

Table 2: Values for all elements in t

| Case/$t$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Static | 4 | 4 | 4 | 3 |
| static,1 | 4 | 4 | 4 | 3 |
| static,3 | 6 | 3 | 3 | 3 |
| dynamic,1 | 0 | 6 | 6 | 3 |
| dynamic,5 | 10 | 0 | 5 | 0 |
| guided,2 | 3 | 4 | 8 | 0 |

   (3.) **Question:** What is a possible performance problem with the assignment to the $t$ array?

   **Response:** The possible performance problem is the dynamic change in array $t$ because the worker thread get a share of work and when work is done, it get a new share **and the size of share can have huge impact on performance.**

2. **Fixed problem:** The fix is done by making `count_odd` shared instead of private. Reason for this is because, for private variable (i.e `count_odd`), each thread has unique access unlike the shared where all threads have access to the same shared memory, hence each thread can read and write the shared variable `count_odd`.

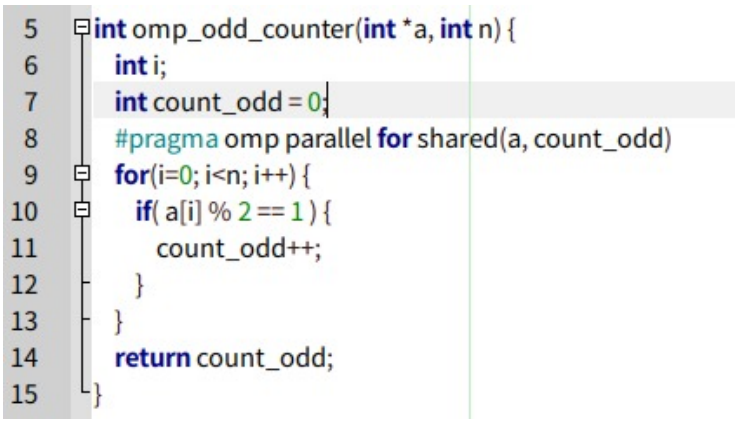


Figure 1

3. (1.) **Question:** What is the output of the three different versions of the program when the programs are executed with `OMP_NUM_THREADS=4`?

   **Response:**

   Version A: Output is 5

   Version B: Output is 5

   Version C: Output is 5.

   (2.) **Question:** How often is the function `omp_tasks`(int v) called in each version of the program when being executed with `OMP_NUM_THREADS=4`?

   **Response:** We added a counter to `omp_tasks` function and printed it out in the main function after the program process is done.

   Version A: called 9 times

   Version B: called 35 times

   Version C: called 9 times.

```
5   static int num_called = 0;
6
7   int omp_tasks(int v) {
8     ++num_called ;
9     int a, b;
10    if( v <= 2 ) {
11      return 1;
12    } else {
13      #pragma omp task shared(a,b)
14      a = omp_tasks(v-1);
15      #pragma omp task shared(a,b)
16      b = omp_tasks(v-2);
17      #pragma omp taskwait
18      return a + b;
19    }
20  }
21
22  int main() {
23    int res;
24    #pragma omp parallel
25    {
26      #pragma omp master
27      res = omp_tasks(5);
28    }
29    printf("res=%d\n", res);
30    printf("num_called=%d\n", num_called);
31    return 0;
32  }
```

Figure 2: Snippet of `omp_tasks` for listing 2 : Version A

The idea is the same for listing 3 : Version B and Listing 4 : Version C.

4. **Question:** Check whether you can apply the collapse clause to increase the potential parallelism.

   **Response:** Yes, we can merge the two loops by the nested loops because they are perfectly nested, that is, the second loop is perfectly nested in the first. Hence we can merge the potential parallelism with collapse.

   (1.) Minimum running time of the three experiments.

Table 3

| Dimension(N) | Number of threads(P) | Minimum running time |
|---|---|---|
| 100 | 1 | 0.160948 |
| 100 | 2 | 0.080868 |
| 100 | 4 | 0.040691 |
| 100 | 8 | 0.02084 |
| 100 | 16 | 0.010885 |
| 100 | 24 | 0.007785 |
| 100 | 32 | 0.012127 |
| 1000 | 1 | 16.131054 |
| 1000 | 2 | 8.186223 |
| 1000 | 4 | 4.111476 |
| 1000 | 8 | 2.073852 |
| 1000 | 16 | 1.054052 |
| 1000 | 24 | 0.710967 |
| 1000 | 32 | 0.553332 |

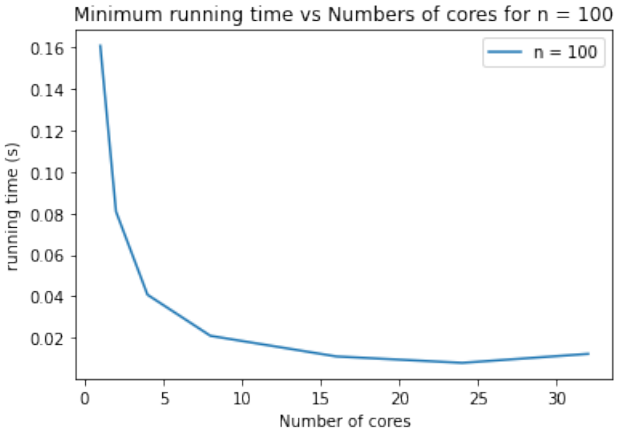


Figure 3

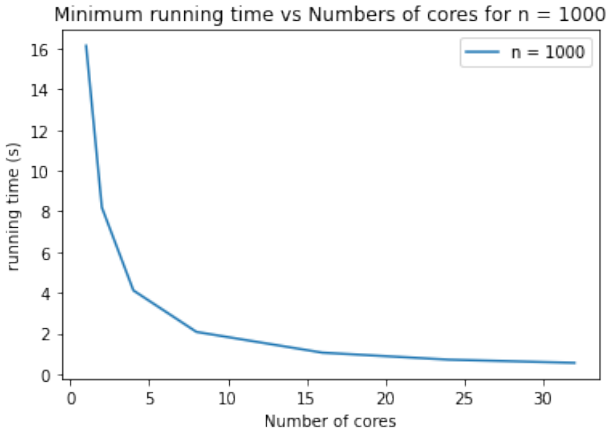


Figure 4

**Observations:** As the number of cores increases, the running time decreases and we also observed that as the number of dimension (n) increases from 100 to 1000, the minimum

time increases 100 times.

(2.) Schedule parameter influence on running time.

Table 4

| Dimension(N) | Number of threads(P) | OMP schedule | Minimum running time |
|---|---|---|---|
| 1000 | 16 | static | 2.428371 |
| 1000 | 16 | static, 1 | 1.030411 |
| 1000 | 16 | dynamic, 1 | 1.054307 |
| 1000 | 16 | guided, 10 | 1.399861 |

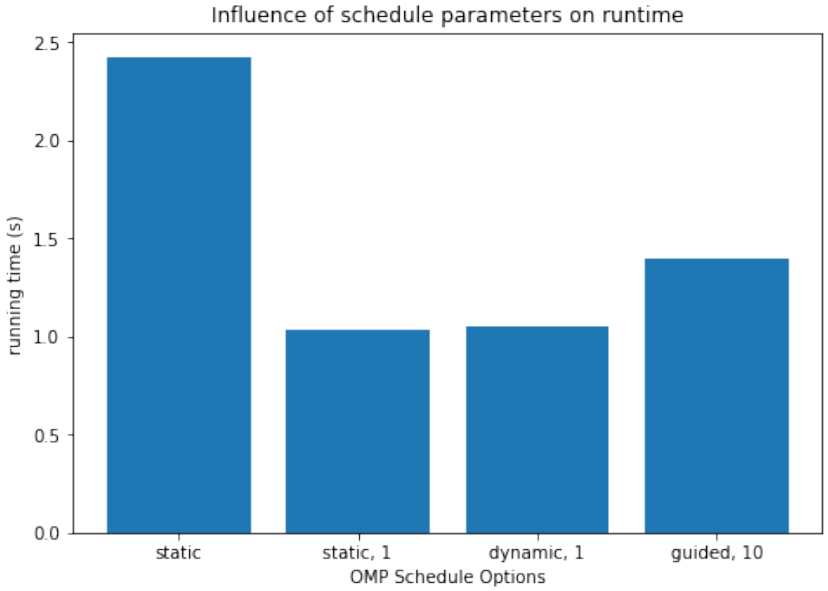


Figure 5

**Observations:** We noticed that the more the chunk size, the higger the run-time. For static without specifying the chunk size, the iteration space 1000 was divided by the number of threads 16, hence the chunk size in this case is approximately 62 which is larger than the chunk size for the rest of the omp schedule. Although, both static,1 and dynamic,1 have the same chunk size. The static,1 assigned the chunk to the thread in order of the thread number but dynamic,1 executes as chunk size many iterations and then request for another chunk till chunk is done. The guided,10 is similar to the dynamic in terms of implementation but the chunk size changes over time and 10 defines the minimum chunk size, so what we can't see is the actual chunk size that is used but we know that it's greater or equals 10.

5. (1.) Minimum running time of the three experiments (Strong scaling). Input is input1.png.

Table 5

| No. of filter application(r) | Number of threads(P) | Minimum running time |
|---|---|---|
| 1 | 1 | 0.0375154 |
| 1 | 2 | 0.0462122 |
| 1 | 4 | 0.0239887 |
| 1 | 8 | 0.0288063 |
| 1 | 16 | 0.026024 |
| 1 | 24 | 0.0268065 |
| 1 | 32 | 0.0419064 |

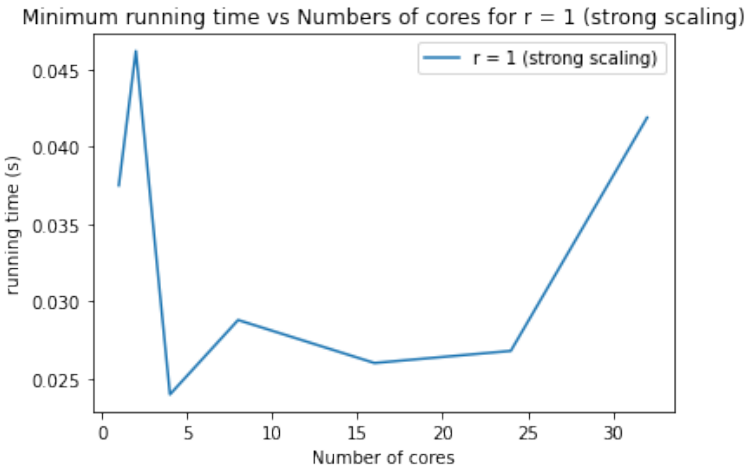


Figure 6

(2.) Minimum running time of the three experiments (weak scaling). Input is input1.png.

Table 6

| No. of filter application(r) | Number of threads(P) | Minimum running time |
|---|---|---|
| 1 | 1 | 0.0375121 |
| 2 | 2 | 0.131669 |
| 4 | 4 | 0.0775685 |
| 8 | 8 | 0.0427748 |
| 16 | 16 | 0.0242519 |
| 24 | 24 | 0.062618 |
| 32 | 32 | 0.0947295 |

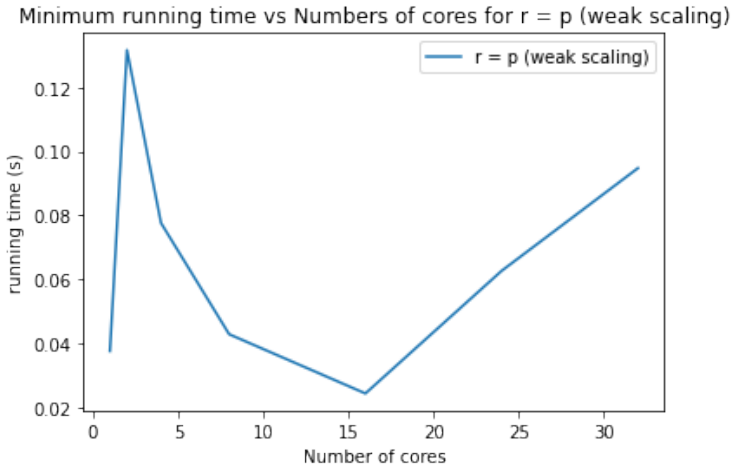


Figure 7

**Observations:** For strong scaling, we can see that the time varies with the number of processors at a fixed r (number of filter application). On the other hand, where we vary $r = p$, there is a significant change in running time(s) as the number of cores changes (i.e increase in time, except for $p = 1$ and $p = 16$).

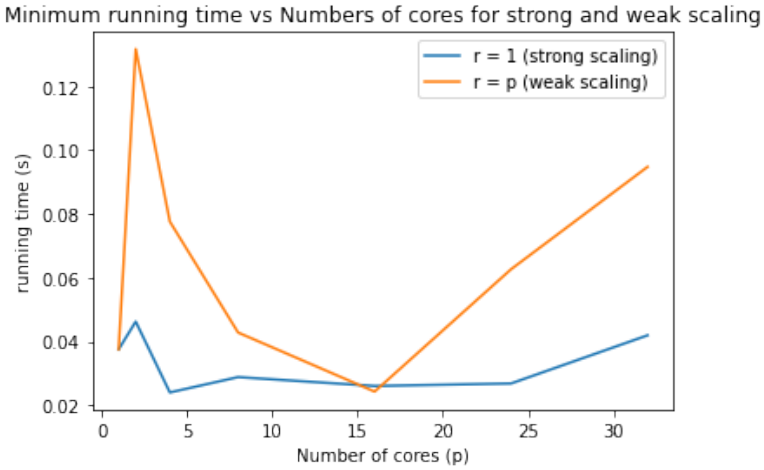


Figure 8: Strong vs Weak scaling