Report of Home Work 7:

1.

a.
```
Hello World from thread = 0
Number of threads = 16
Hello World from thread = 15
Hello World from thread = 4
Hello World from thread = 11
Hello World from thread = 12
Hello World from thread = 2
Hello World from thread = 14
Hello World from thread = 9
Hello World from thread = 13
Hello World from thread = 8
Hello World from thread = 3
Hello World from thread = 7
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 10
Hello World from thread = 1
```

16 threads are used. Because there are 16 cores available in the node.

b.

(First time):

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 1
```

(Second time):

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 1
```

(Third time was like First time), (4th time):

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 3
Hello World from thread = 2
```

The order of thread's output changes each time. This is because there is no predefined priority for selecting threads to execute their job. So the output would be in random order of existing threads.

2. **a.**

Description of the output:

For Static Scheduling: N is 100 and therefore there is 100 iterations available in the loop. The number of threads has been set to 4. Then, the loop has been divided to sections of 10 (Chunk size) and is distributed in a round robin manner then after. That means the threads have these elements of the resulted vector C:

```
thread0: C[0]C[1]..C[9];
thread1:C[10]C[11]..C[19];
thread2:C[20]C[21]..C[29];
thread3:C[30]C[31]..C[39];

thread0:C[40]C[41]..C[49];
thread1:C[50]C[51]..C[59];
thread2:C[60]C[61]..C[69];
thread3:C[70]C[71]..C[79];

thread0:C[80]C[81]..C[89];
thread1:C[90]C[91]..C[99];
```

For Dynamic Scheduling: Again, N is 100 and therefore there is 100 iterations available in the loop. However in this case, there is no static dictation to the threads which forces them to do a specific portion of the loop. Once a thread is ready to do a job, the job is dedicated to it. (Of course, as the chunk size is again 10, then ten consecutive iterations is given to each thread at each time) So by running the code for the next time compared to previous time, it is not expected to see the same portion of the loop executed by a specific thread:

<u>First Try:</u>

```
thread0:    C[0]C[1]..C[9];
            C[50]C[51]..C[59];
            C[80]C[81]..C[89];

thread1:    C[10]C[11]..C[19];
            C[60]C[61]..C[69];
            C[90]C[91]..C[99];

thread2:    C[30]C[31]..C[39];
            C[70]C[71]..C[79];

thread3:    C[20]C[21]..C[29];
            C[40]C[41]..C[49];
```

<u>Second Try:</u>

```
thread0:    C[0]C[1]..C[9];
            C[40]C[41]..C[49];
            C[60]C[61]..C[69];

thread1:    C[30]C[31]..C[39];
            C[90]C[91]..C[99];

thread2:    C[20]C[21]..C[29];
```

```
             C[50]C[51]..C[59];

thread3:     C[10]C[11]..C[19];
             C[70]C[71]..C[79];
             C[80]C[81]..C[89];
```

And surely, the magnitude of the element is unchanged in all cases. That means in all cases `C[i]=2*i`.

b.

I think dynamic scheduling is very good option for those cases in which each iteration has variable amounts of work or where some processors are faster than others. With static scheduling there is no way to load-balance the iterations. Dynamic scheduling load-balance the iterations automatically. For instance in static scheduling, it is very probable that when some of the processors are idle (after doing their portion of job) the overall work is not finished yet because one of the processors is still executing its iteration.

On the other hand, static is also a good option for those loops with good balance among iterations. Moreover, the use of dynamic schedules usually incurs high scheduling overheads and its non-predictive behavior tends to degrade data locality (non–reuse of data across loops or multiples instances of the same loop).

3. As there are two sections in the code, while two of all threads are doing the job, some threads do not execute any iteration. I saw that the threads which calculate c[:] and d[:] vary at each running time. At the first try, thread 0 and thread 1 worked, the next time the thread 0 and thread 10, and third only 0 did the whole job. Therefore, Which thread does work is non-deterministic in this case. It depends that how fast they become ready to be used.

4. The scheduling is set STATIC with chunk size of 10. Matrix A is 62 x 15 and B is 15 x 7. So Matrix C is 62 x 7. There are three "for loops" in matrix multiplication. The outer loop is over rows of A and second and third loops are over columns of B and columns of A, respectively. It is seen in my results that the assignment of threads for doing the iterations in loop is based on outer loop (rows of A). Remembering that we have set the number of threads to 4, therefore:

```
Thread 0 starting matrix multiply...
Thread=0 did row=0
Thread=0 did row=1
        .
        .
Thread=0 did row=9

Thread=0 did row=40
Thread=0 did row=41
        .
        .
```

```
Thread=0 did row=49

Thread 1 starting matrix multiply...
Thread=1 did row=10
Thread=1 did row=11
         .
         .

Thread=1 did row=19

Thread=1 did row=50
Thread=1 did row=51
         .
         .

Thread=1 did row=59

Thread 2 starting matrix multiply...
Thread=2 did row=20
Thread=2 did row=21
         .
         .

Thread=2 did row=29

Thread=2 did row=60
Thread=2 did row=61

Thread 3 starting matrix multiply...
Thread=3 did row=30
Thread=3 did row=31
         .
         .

Thread=3 did row=39
```

(I have rearranged the place of the output a little bit)

5.

Here is the section of the code that prints the requested information:

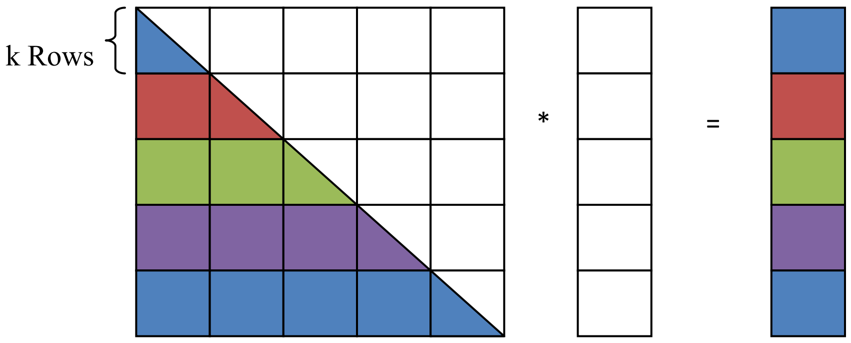
```c
/* Printing Info from Master thread */
if (tid == 0)
     {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);

        printf("Printing Info From Master Thread (Thread %d)\n", tid);

        /* Get environment information */
        printf("Number of processors available = %d\n", omp_get_num_procs());
        printf("Number of threads being used = %d\n", omp_get_num_threads());
        printf("Max number of threads available = %d\n", omp_get_max_threads());
        if (omp_in_parallel()!=0){
            printf("You are in parallel region\n");
        }
        else{
            printf("You are not in parallel region\n");
        }

        if (omp_get_dynamic()!=0){
            printf("Dynamic threads are enabled\n");
        }
        else{
            printf("Dynamic threads are not enabled\n");
        }

        if (omp_get_nested()!=0){
            printf("Nested parallelism is supported\n");
        }
        else{
            printf("Nested parallelism is not supported\n");
        }


   }
```

<u>the output:</u>

```
Number of threads = 16
Printing Info From Master Thread (Thread 0)
Number of processors available = 16
Number of threads being used = 16
Max number of threads available = 16
You are in parallel region
Dynamic threads are not enabled
Nested parallelism is not supported
```

6.

For parallelizing the code, I used the following distribution:



Each color is done by one thread. All of the matrices is in the shared memory. (Please note that this distribution is cyclic, for the example above Nthread=4 and n=k*5)

For timing analysis, I used *omp_get_wtime()* before and after parallel region.

The Results:

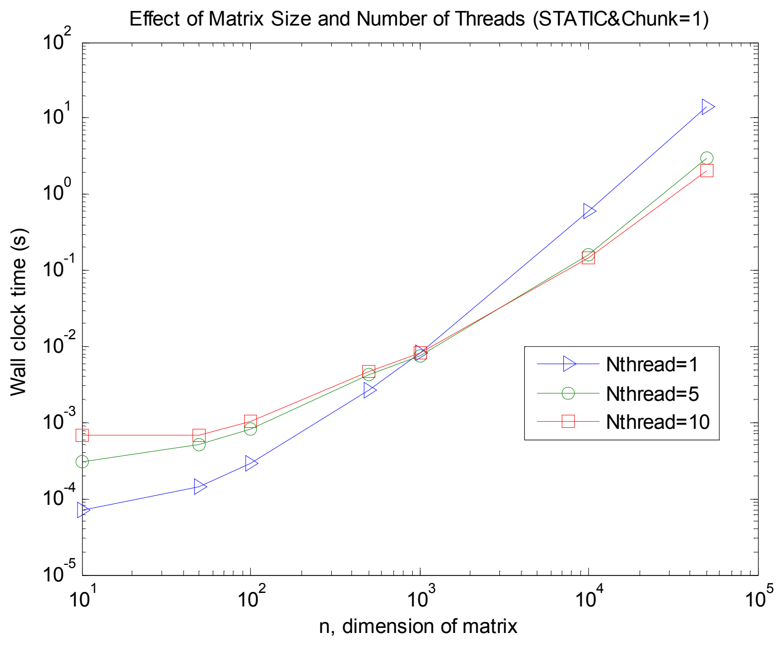


Figure 1. time vs. dimension for 3 different Nthreads; STATIC and chunk=1

The above figure, shows the dependency of "wall clock time" on matrix dimension for three different thread numbers; for STATIC scheduling and chunk size of one. It is obvious and expected that with increasing matrix dimension, time goes up. However, the behavior of the curve for different thread numbers (Total number of threads) is interesting. Although before matrix dimension of 1000 increasing the number of threads increases the total time, after that value the reverse trend is observed. So practically, it could be concluded that in this specific situation (static and ...) increasing the number of threads is worthy only for large matrices. It may be attributed to increased overhead time as we increase the number of threads in small dimensions. However, for large size matrices, the imposed balance in distributing the tasks among the threads surpasses the increment of overhead and we finally see lower wall clock time.

The next figure (figure. 2) represents the effect of chunk size on wall clock time for STATIC scheduling and matrix dimension of 5000. It is seen that the difference is not very deterministic and is in the order of 0.02 s. Generally speaking, the time decreases when we change the chunk size from a very low value to larger values. However, increasing the chunk further results in increasing the clock time. It was somehow expected, as the granularity is often related to how balanced the workload is between threads. It is easier to balance the workload of a large number of small tasks but too many small tasks can lead to excessive parallel overhead. Increasing granularity too much can create load imbalance.

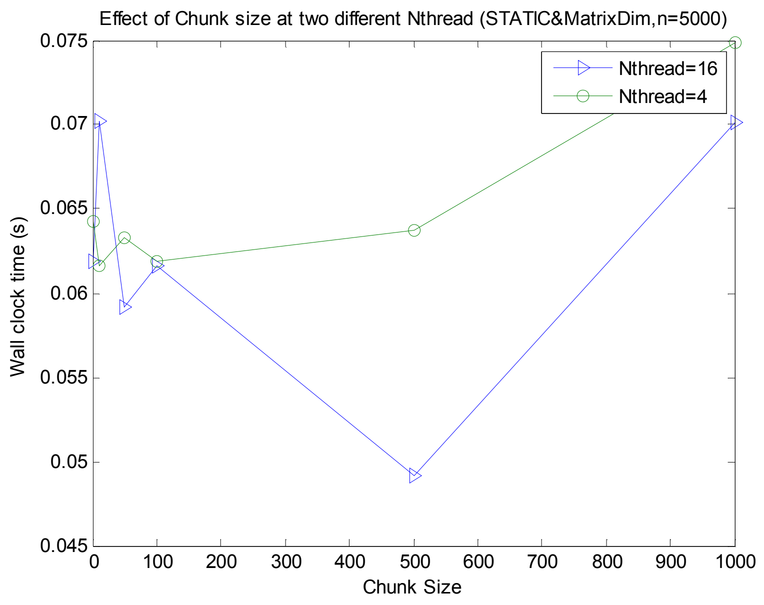


Figure 2. time vs. chunk size; STATIC and n=5000.

Figure. 3 shows the time vs. matrix dimension for DYNAMIC scheduling.

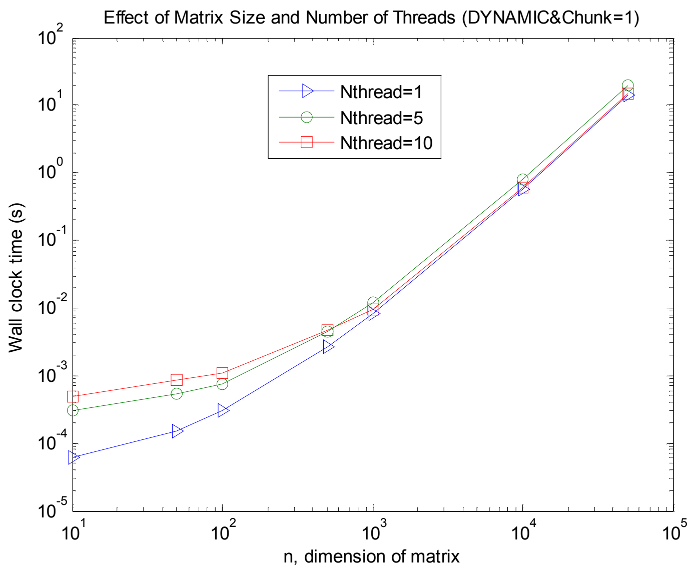


Figure 3. time vs. n for different Nthreads; DYNAMIC, chunk=1.

It is demonstrated in this figure that compared to STATIC scheduling, the first part behavior (before 1000) is the same as STATIC. However, as opposed to STATIC, the time is nearly the same for different number of threads. So, from this point of view, increasing the number of threads with this situation will not give us any advantage over serial code. This could be attributed to the chunk size of 1 that we have used here. Because it would raise the overhead time, as there are so many tasks to be scheduled dynamically. With this interpretation, increasing the chunk size should improve the results of higher number of threads in large matrix sizes:

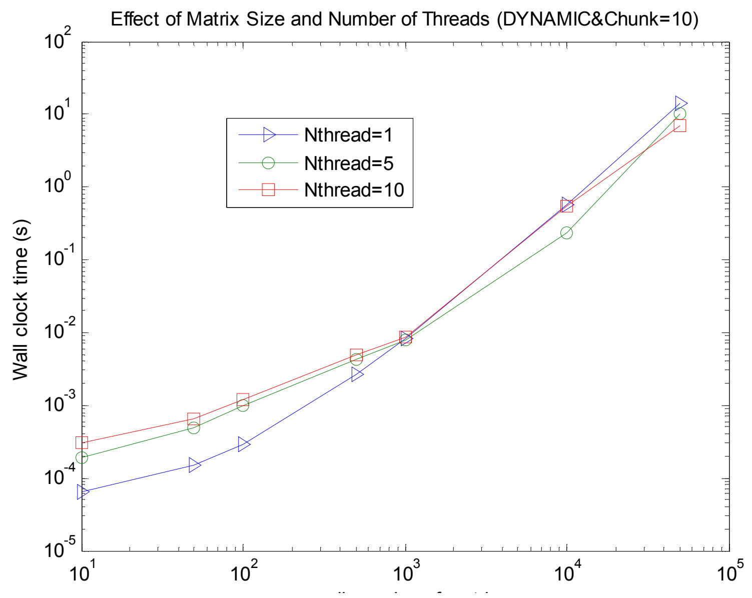


Figure 4. time vs. n;DYNAMIC, chunk=10.

So it seems that our expectation is somehow true.

The next figure represents the chunk size dependency for DYNAMIC scheduling. Compared to STATIC, the behavior here is more deterministic. The same interpretation here holds but with more significance. The time rises as we incorporate the imbalance of tasks over the threads; e.g. with noticeably increasing the chunk size.

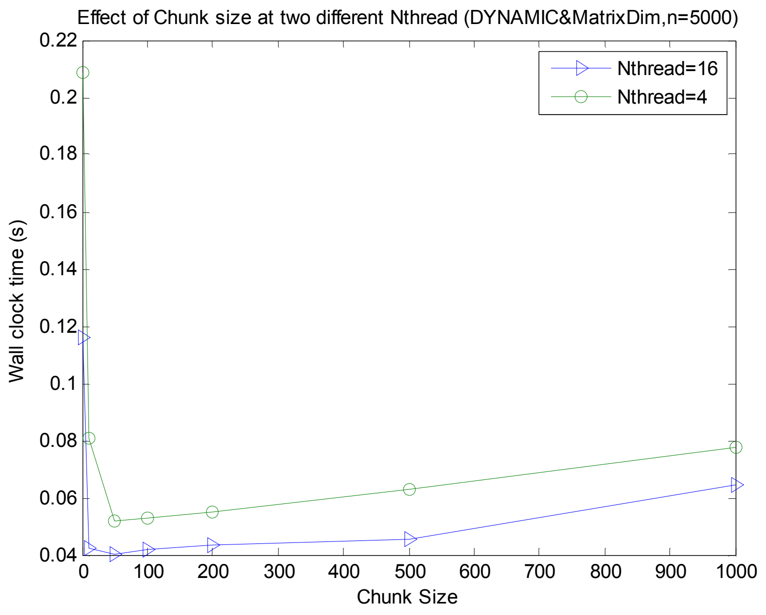


Figure 5. time vs. chunk size for DYNAMIC scheduling and n=5000.

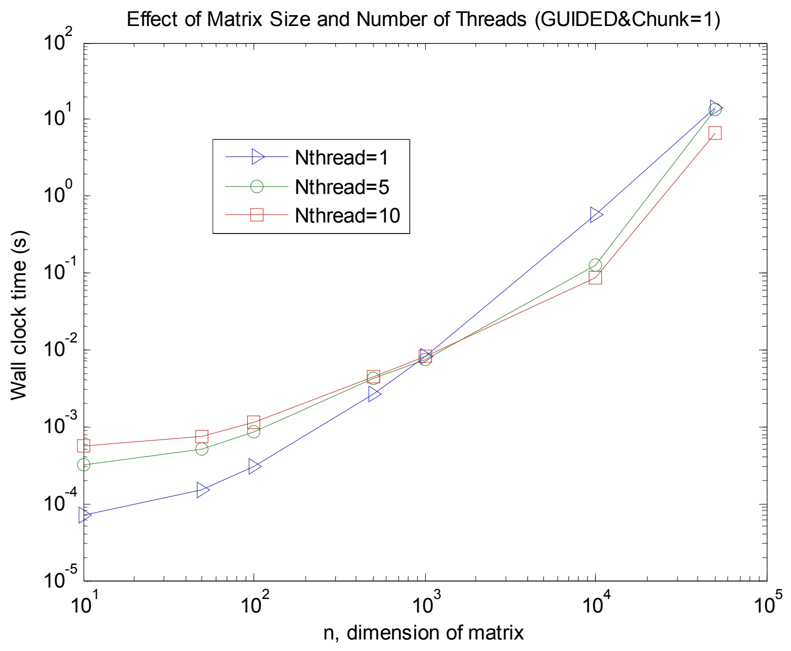


Figure 6. time vs. n for GUIDED scheduling.

The figure above shows time vs. dimension for guided scheduling. The same behavior and interpretation holds here as STATIC. The time dependency on chunk size is also similar to STATIC.

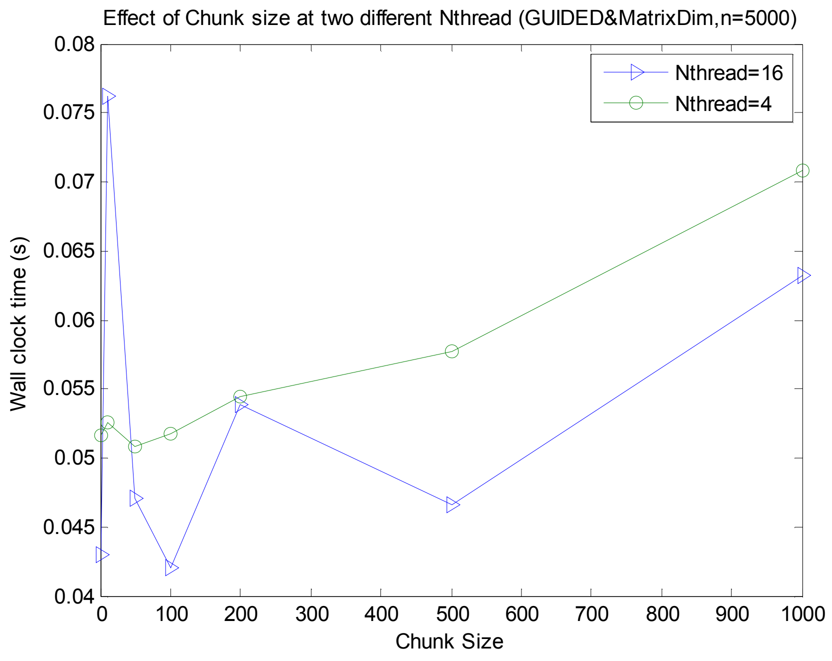


Figure 7. time vs. chunk for GUIDED scheduling.

Figure 8 shows the wall clock time for different scheduling policies.

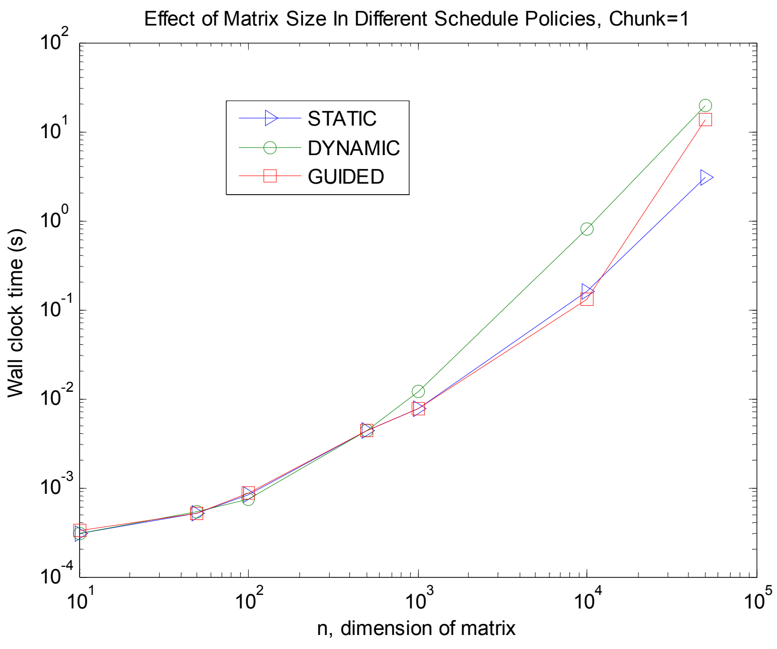


Figure 8. time vs n for different policis.

It is seen that interestingly, dynamic and static policies does not have better timing performance over static in this example. Guided and STATIC is close to each other at most of the time and they are better than dynamic. The better performance of GUIDED over DYNAMIC may be attributed to more balanced distribution of tasks especially in this example of triangular matrix. Because the distribution of row to threads is not very well balanced. For chunk size of 10, the behavior is very similar:

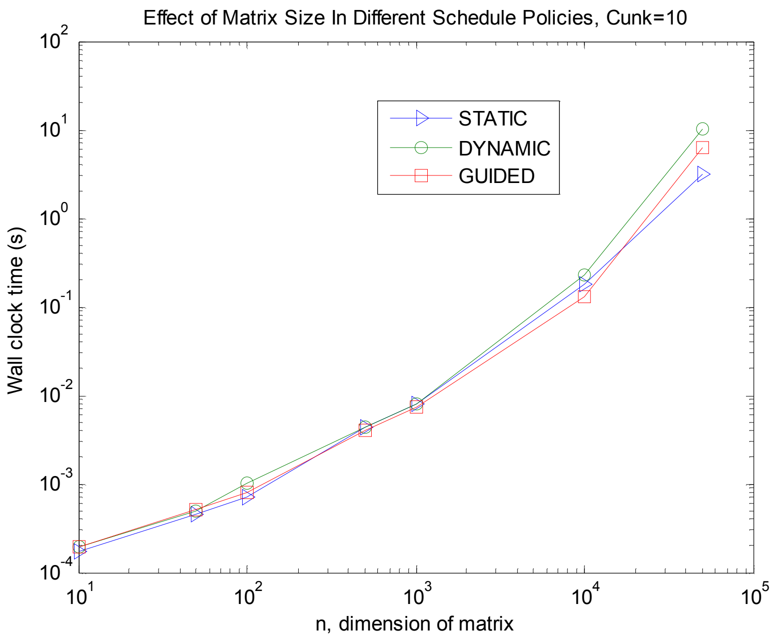


Figure 9. time vs. n for different policies.