HW02 (HPC . ICS632 Oct. 2015) ---------- Ehsan Kourkchi

Exercise 1:

Q1:

The main program: `exercise1.c`

To compile on `uhhpc` cluster:

```
$ icc   exercise1.c -o exercise1_0.x   -Ofast -D N=2500 -D MODE=0 -fopenmp -mcmodel medium -shared-intel
```

Mode 0: For running the program in the serial mode. The order of nested loops is i-k-j, where the i-loop is the most outer loop.

```
for(i=0; i<N; i++)
  for(k=0; k<N; k++)
    for(j=0; j<N; j++)
       C[i][j] += A[i][k]*B[k][j];
```

All the other modes run the program in parallel mode.

Mode 1: j-loop runs in parallel mode – no race condition
Mode 2: k-loop runs in parallel mode – race condition
Mode 3: i-loop runs in parallel mode – no race condition

In mode `1 (3)`, each thread has it's unique number for `j (i)`. Since the summation is being done over `C[i][j]`, having a unique index in a thread, does not produce any race condition, because the other threads do not allow to play with the same memory.

In mode 2, the `k`-loop is run in parallel mode, and therefore there is a chance that the same values of I and j be produced in each thread. So multiple threads would try to change `C[i][j]` and the race condition happens. To avoid that, we use `"pragma omp atomic"`. However since different threads have to wait for each other to do the summation in a right order.

Q2:

For N=2500, the running time in serial mode is ~5 second.
T_0 = 5.49 sec → Average running time in serial mode

HW02 (HPC . ICS632 Oct. 2015) ---------- Ehsan Kourkchi

All the reported results are for 10 iterations.
To do that automatically, I used the bash script "`task.bash`", that compiles the program for each iteration and then stores the outputs in multiple files, i.e. `output.xx.yy.txt`, where `xx` is the version number (mode 1, mode2 or mode 3) and `yy` is the number of used threads to run the program. Figure 1, also is generated by the python code: `exercise1_times.py`, where it plots the results of all three version together.

These are the results of running the program in parallel mode. 'T' is the runnig time and 'c' is the number of cores.

Mode 1: `j`-loop in parallel (**`exercise1_1.x`**)
T_1 = 7.35 sec
c_1 = 2

```
for(i=0; i<N; i++)
  for(k=0; k<N; k++)
    for(j=0; j<N; j++)
       C[i][j] += A[i][k]*B[k][j];
```

In this case the inter-most loop is in parallel and for each `i` and `k`, the number of produced parallel forks is of the order of N^2. So it sounds to be inefficient to make this many threads. In fact, the running time is more dominated by the time needed to produce threads. Since creating threads and killing them takes some time, in this mode, as seen in the Figure 1 (blue dots), the running time increases with the number of threads. When the number of threads increases, more time is needed to produce them. Since this action takes way more time than the actual calculation of the matrix calculations, the total running time reflects more of thread making operation.

Also for each thread, `j` is constant. So each thread loads a big chunk of `B[k][j]` just to use one element. So there seem to be tons of load misses in this version.

Mode 2: `k`-loop in parallel (**`exercise1_2.x`**)
T_2 = 24.00 sec
c_2 = 19

```
for(i=0; i<N; i++)
  for(k=0; k<N; k++)
    for(j=0; j<N; j++)
       C[i][j] += A[i][k]*B[k][j];
```

In this case, due to the race condition, different threads have to wait for each other. This takes some time. Compared to the above version, the number of thread is N times smaller and therefore, thread generating does not dominant the matrix calculations. As seen in Figure 1 (red dots), increasing the number of threads actually help to solve the problem faster. For each thread, `k` is constant. So when calling `B[k][j]`, in the inner loop, we can make use of the memory locality, however `k` is the second index of `A[i][k]` and still there are some memory load miss. Compared to the version 1, this version is still slow, because of the race condition.

Mode 3: `i`-loop in parallel

T_3 = 1.24 sec

c_3 = 18

```
for(i=0; i<N; i++)
  for(k=0; k<N; k++)
    for(j=0; j<N; j++)
       C[i][j] += A[i][k]*B[k][j];
```

There is no race condition in this version. Therefore each thread can work safely.
Also there is an optimum use of memory locality, by using the best order of the indices. Also the number of threads is N times less than version 2, that discussed above. So, the thread creation time should be negligible (or much less) compared to the actual calculations. As seen in Figure 1 (green dots), the running time decreases with increasing the number of threads.

Q3:

The most efficient version in this case, is the version 3, when the `i`-loop is parallelized. The bes running time is `1.24 sec`, with `18` threads.

Compared to the serial version, the speedup is `5.49/1.24 = 4.4`
Using 18 cores, one expects to see a large speedup. In ideal case the speedup would be close to 18, however, the performance of the program becomes dominated by the network communication and the time that is needed to generate threads. As seen in Figure 2, there is no much change in the performance, when we go from 10 cores to 20 cores. This means that, the calculations of the matrix multiplications almost take no time, and all the observed time reflects the needed time for the system setup.

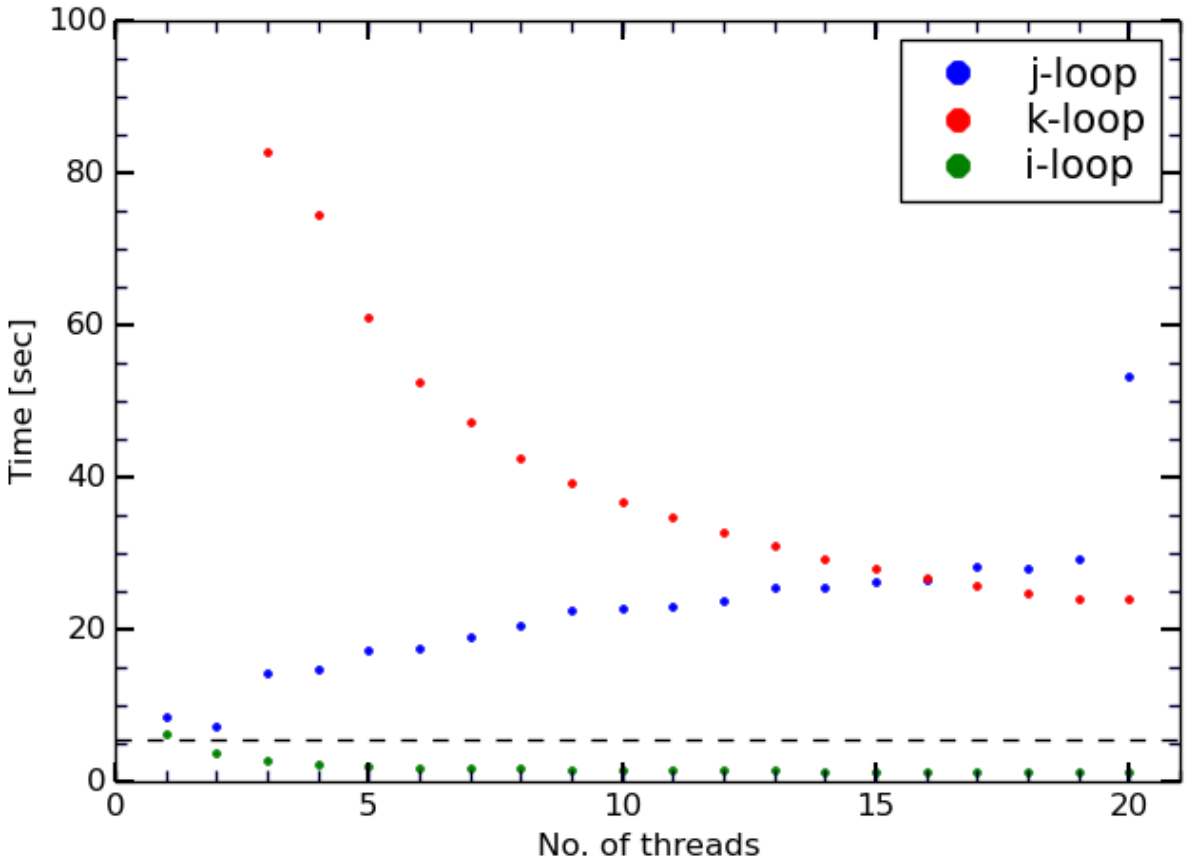

Figure 1: The running time of different versions of the matrix multiplication program. The horizontal axis shows the number of threads. Different colors are different modes, where different loops run in parallel. The run-time of the serial version of the program, is represented by the horizontal dashed line.

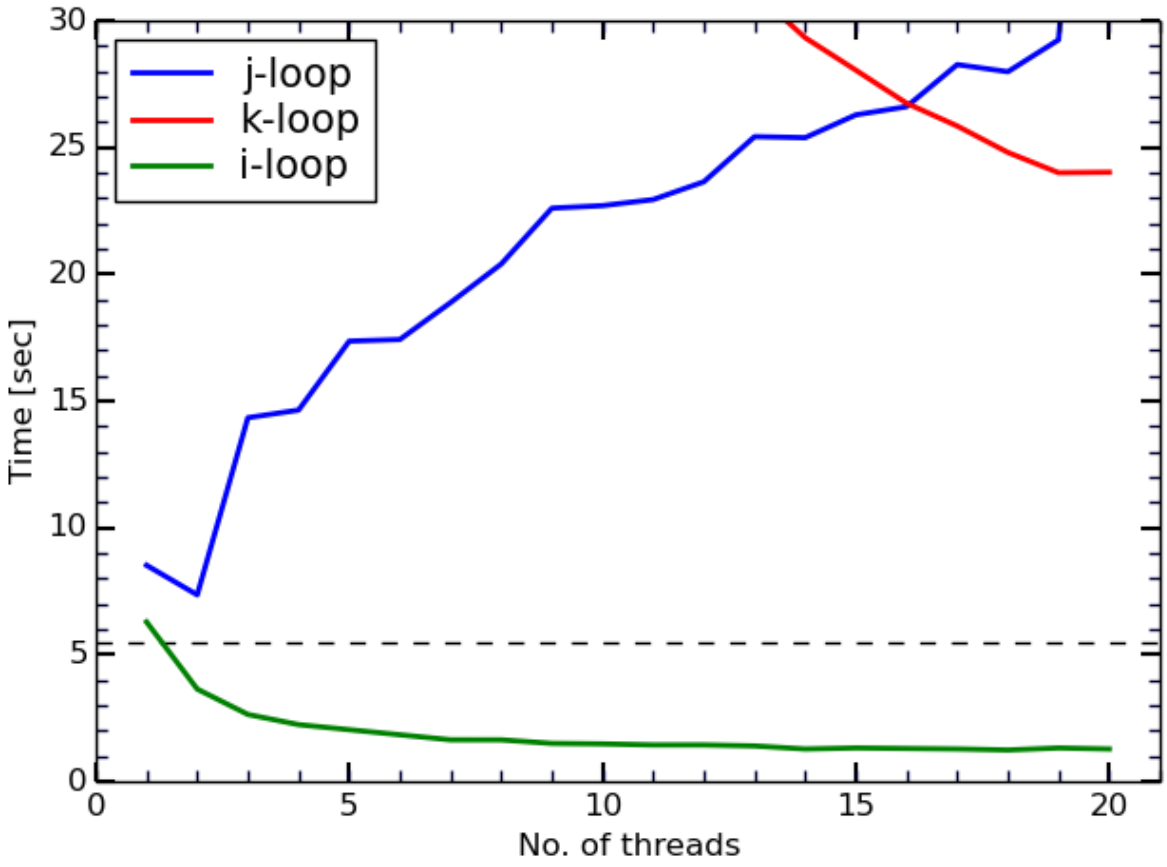

Figure 2:The same as figure 1. In order to see more details of the behavior of the i-loop when it runs in parallel mode (i.e. green curve). The dashed line shows the running-time of the serial mode.

Exercise 2:

Q1:

```
A[i][j] = (3*A[i-1][j] + A[i+1][j] + 3*A[i][j-1] + A[i][j+1])/4;
```

For any pair of `i` and `j`, `A[i][j]` depends on its neighbor elements.

	`A[i-1][j]`	
`A[i][j-1]`	`A[i][j]`	`A[i][j+1]`
	`A[i+1][j]`	

If the loop of `i` or `j` are parallelized, then it means that each thread does the

calculations separately, while it needs to have access to the results of the other threads. Since each thread does the calculations independently, each `A[i][j]` values might be updated before its neighbors get updated. This way, the order of calculations would be wrong and wrong numbers get mixed up.

Looking at the order of loop over `i` and `j`, the elements of A get updated from the left to right and from the top to bottom. As seen in the above table, in order to calculate `A[i][j]`, first the orange values should be updated, while the blue values are still old.

This helps to understand the way the values of the matrix A are getting updated:

Q2:

	1	2	3	4	5	6	7	8	9	10	
	2	3	4	5	6	7	8	9	10	11	
	3	4	5	6	7	8	9	10	11	12	
	4	5	6	7	8	9	10	11	12	13	
	5	6	7	8	9	10	11	12	13	14	
	6	7	8	9	10	11	12	13	14	15	
	7	8	9	10	11	12	13	14	15	16	
	8	9	10	11	12	13	14	15	16	17	
	9	10	11	12	13	14	15	16	17	18	
	10	11	12	13	14	15	16	17	18	19	

For example, the above table shows the matrix A, when N=10. The gray area shows the region where the matrix is updated in the nested `i-j` loop. Based on the order of the operations, those elements which are represented with the same numbers (in the above table), can be updated independently. The numbers also show th order of the dependency of elements. It means that to calculate the cells with the number P, all cells with numbers in the range (1, P-1) should be calculated first.

This table shows that we are able to calculate all the diagonal elements wit the same numbers in parallel loops.

This is the piece of the code which runs in parallel:

```
  // Loop for num_iterations iterations
  for (iter = 0; iter < num_iterations; iter++) {

    // upper left half of the matrix A
    for (n=1; n < N+1; n++) {
      #pragma omp parallel for shared(A) \
    private(j)
      for (i=1; i < n+1; i++) {
    j = n - (i-1);
    A[i][j] = (3*A[i-1][j] + A[i+1][j] + 3*A[i][j-1] + A[i]
[j+1])/4; }}

    // lower right half of the matrix A
    for (n=N+1; n < 2*N; n++) {
      #pragma omp parallel for shared(A) \
    private(j)
      for (i=n-N+1; i < N+1; i++) {
    j = n - (i-1);
    A[i][j] = (3*A[i-1][j] + A[i+1][j] + 3*A[i][j-1] + A[i]
[j+1])/4; }}

    }
```

For the entire code, please refer to `exercise2.c` .

Q3:

N = 12000
# of iterations = 17

The average running time when using the original serial code:
T_0 = 10.10 sec

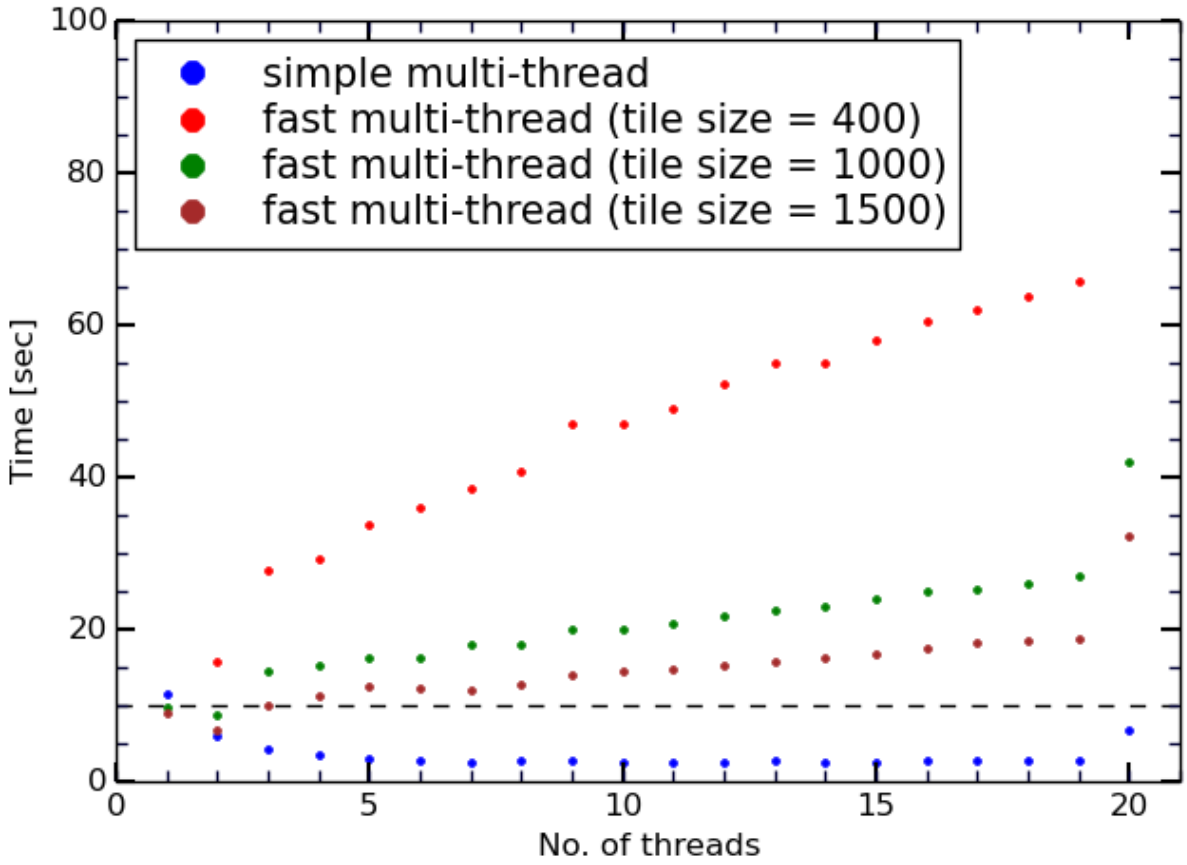

Figure 3:The running time of different version of the exercise2 code. The dashed horizontal line shows the running-time of the serial code without any parallelization. The blue dots show the results for the simple parallelization. Red, Green, and Brown dots show the results of the tiled-version of the code, for different tile sizes.

As seen in the above diagram, the parallel version of the program is faster than the serial mode. The minimum run-time is achieved when 11 threads are in use:

T_{min} = 2.47 sec
# of threads = 11

and compared to the serial version, the speedup is `10.10/2.47 = 4.09`
As observed, for number of threads greater than 5, there is no significant change in the performance of the code, and the running time is relatively constant. It means that, the performance is probably governed by another factor, like memory locality. Because in each thread, `i` is constant, and on the right side of the following relation, `A[i][j-1] and A[j][j+1]` are next to each other in a memory sequence. However for the other tow terms (denoted by blue

color), the cahe load miss kills the performance.

```
A[i][j] = (3*A[i-1][j] + A[i+1][j] + 3*A[i][j-1] + A[i][j+1])/4;
```

Q4: The idea is to increase the performance of the above code with tiling the matrix. This code is called `exercise2_fast.c` and the results of the running time is shown in Figure 3 for a few different tile size.

N = 120,000
# of iterations: 17

Tile size = 400
T_{min} = 15.86 sec
# of threads = 2

Tile size = 1000
T_{min} = 8.69 sec
# of threads = 2
speed-up: `10.10/8.69 = 1.16`

Tile Size = 1500
T_{min} = 6.89 sec
# of threads = 2
speed-up: `10.10/6.89 = 1.47`

Q5:
As seen in Figure 3, tiling does not improve the performance. Even increasing the number of threads, increases the running-time.

As seen above, using 2 threads and tile sizes of 1000, and 1500, slightly improves the performance however in general, the total performance for larger number of threads is worse.

- If N is the size of the matrix A, for the simple parallel mode (no tiling) we have:

```
the number of parallel loops: 2(N-1) + 1 = 2N-1 =~ 2N
```

- If N is the size of the matrix A, and P is the number of tiles in each side

then:

```
the tile size: M = N/P
number of tiles: P^2
The number of parallel loops:  (2M-1)*P^2 = 2NP-P^2 =~ 2NP
```

My guess is that since the number of the parallel loops is much larger in the tiled-version, the performance might be worse. Another guess is that for small tile size and large number of threads, more cores might be idle in the entire process, because at each levels threads must complete their jobs in order to jump to another level. The number of idle threads decreases if the size of the tile (sub matrices) increases and/or the number of cores decreases.