Homework 3 Report

This is the same as Readme.md

Hardware specs:

Experiments are run on CIMS crunchy1, which according to `lscpu` is x86_64 architecture with 64 cores and each core has 2 threads.

Question 1

(a)

Static scheduling assigns equal size of chunk to the two threads. Therefore, for the first for-loop, the time will be:

```
thread 1: 1+2+...+(n-1)/2,

thread 2: (n-1)/2+1 +  (n-1)/2+2 + ... + n-1

(let's just assume n is an odd number for simplicity, and this does not affect the
generality of our discussion.)
```

Similarly, for the second for-loop, we have:

```
thread 1: n-1 + n-2 + ... + (n-1)/2+1

thread 2: (n-1)/2 + (n-1)/2-1 + ... + 1
```

In total, we can see that both of the thread will spend 1 + 2 + ... + n-1 = n(n-1) /2 miliseconds.

After each for-loop, the threads need to synchronize. For the first for-loop, thread 1 spends less time than thread 2, so it will spend:

[(n-1)/2+1 + (n-1)/2+2 + ... + n-1] - [1+2+...+(n-1)/2] = (n-1)^2 / 4

waiting for thread 2.

Similarly, for the second for-loop, the thread 2 will spend the same amount of time (n-1)^2 / 4 waiting for thread 1.

In total, (n-1)^2 / 2 will be spend in waiting (synchronization).

(b): For each for-loop, the execution time will be less imbalanced if we use schedule(static, 1).

For the first for-loop:

```
Thread 1: 1 + 3 + 5 + ... + (n-2) = (n-1)^2/4

Thread 2: 2 + 4 + 6 + ... + (n-1) = (n-1)(n+1) / 4
```

For the second for-loop, similarly:

```
Thread 1: 2 + 4 + 6 + ... + (n-1) = (n-1)(n+1) / 4

Thread 2: 1 + 3 + 5 + ... + (n-2) = (n-1)^2/4
```

Combined, the total execution time for both of the thread is the not changed: 1 + 2 + ... + n-1 = n(n-1) /2 miliseconds. But there will be less time spent for waiting to synchronize since it is less imbalanced, so the total running time will be shorter.

(c):

In general, using schedule(dynamic, 1) will improve since it will dynamically assign work to threads and results in better load balancing. But in this specific case, schedule(static, 1) and schedule(dynamic, 1) will be the same. This is because the assignment is not changed given the specific execution time of f(i)s.

(d):

Since f(x) is an independent function, we can use the directive `nowait` for each of the for-loops. This eliminates the implicit synchronization at the end of each for-loop. So at the end each thread will spend exactly n(n-1) /2 miliseconds and no more waiting time.

Question 2

The max number of threads is set to 64 for this question, and the experiments of thread 1, 2, 4, 8, 16, 32, 64 are shown below:

```
sequential-scan = 0.638685s
run with 1 threads      parallel-scan   = 0.534414s     error = 0
run with 2 threads      parallel-scan   = 0.427734s     error = 0
run with 4 threads      parallel-scan   = 0.381150s     error = 0
run with 8 threads      parallel-scan   = 0.297201s     error = 0
run with 16 threads     parallel-scan   = 0.348355s     error = 0
run with 32 threads     parallel-scan   = 0.352518s     error = 0
run with 64 threads     parallel-scan   = 0.352525s     error = 0
```

Question 3

Since the convergence is slow, I set the number of iterations to 100 for different Ns, which is the same as assignment 1. Note that, to comply with the handing rules, the number of threads, the N and the max iterations have to be modify in the source code. Also, the right hand side f(x,y) is set to 1 for simplicity. One should change it to a vector for general case.

For Jacobian 2D:

Threads	**seq**	**1**	**2**	**4**	**8**	**16**	**32**	**64**
N=100	0.013248	0.031984	0.017096	0.009588	0.006920	0.007546	0.010835	0.524076
N=1000	1.177591	2.899415	1.478437	0.761617	0.396977	0.204938	0.140522	0.703573
N=10000	130.323083	281.000741	141.376350	70.619156	36.532573	19.125628	12.516912	12.839569

For Gauss-Seidel 2D:

Threads	**seq**	**1**	**2**	**4**	**8**	**16**	**32**	**64**

Threads	seq	1	2	4	8	16	32	64
N=100	0.015471	0.028812	0.015523	0.008998	0.006812	0.007818	0.013221	0.746888
N=1000	1.402649	2.566101	1.316912	0.686153	0.355611	0.190818	0.144216	1.060069
N=10000	137.948482	240.959659	122.255339	62.039789	32.786162	17.990499	13.334537	13.695364