Xinyu Chen, xchen786
GTID: 903612905

A brief explanation of how I developed my program

- ❑ Diffusion calculation:
  - Using a for loop to calculate temperature updates at each location
  - Using OpenMP to do the data parallelization that different threads update different heat location
  - There are no locks or synchronization barriers here. Therefore, it can be well parallelized.

- ❑ Modified binary search:
  - Using a while loop to find a suitable value of initial temperature by binary search
  - Place a function with a return value less than 1 before a condition with a return value greater than 1 to prevent code redundancy

- ❑ Using a Makefile to control the number of threads used in data parallelization

Why this program works correctly

- Test my program using reference value:
- an array size of 6000
- an initial location of 3000

```
ubuntu@ip-172-31-32-98 ~/CSE6010-22Fall/Assignments/A5 master ± make
Enter the number of threads for parallelization: 1
gcc -o diffusion diffusion.c -O0 -fopenmp -g -Wall -Werror -std=gnu99
export OMP_NUM_THREADS=1
ubuntu@ip-172-31-32-98 ~/CSE6010-22Fall/Assignments/A5 master ± time ./diffusion 6000 3000
the value of initial heat is 50
the maximum heat temperature is 0.997480
./diffusion 6000 3000  1.00s user 0.01s system 198% cpu 0.511 total
```

The maximum value is the same as mentioned in the assignment introduction.

- Also, by testing the program in a larger array size and the different numbers of threads, I can observe an improvement in performance based on runtime. (See the next slide)

Assessment of OMP performance

Runtimes using several numbers of threads for 1 array size

No. Threads	Array Size	Runtime / s
1	60000	8.20
2	60000	8.24
3	60000	8.17
4	60000	8.05
6	60000	7.94

- From this table, we can see that when the number of threads = 6, we have the shortest runtime.
- When the number of threads <= 3, there is no speedup (even has a performance degradation).

  This is common with small data sizes.