System Practicum

CS-307

(Semester - 6)

Assignment - 2
Report

Team Members:

Shubham Saurav (B19222)

Prashant Kumar (B19101)

Ravi Kumar (B19191)

Saloni Patidar (B19111)

Sagar Tarafdar (B19110)

Gaurav Sahitya (B19083)

Part - 1: Crashing the system

OS: Ubuntu 20.04 on Virtual Machine

We installed the kernel 5.17 version and built it on the system.

Initial Kernel Info:
Version: 5.11.0-27-generic
Size: 74MB

Kernel version (after customizing)
Version: 5.17
Size: 203MB

Removing Modules:

1. Android Drivers
2. USB support
3. Bluetooth Module
4. Touch Screen, etc

Writing Modules for crashing the kernel:
Libraries used:

1. #include <linux/module.h>
2. #include <linux/init.h>
3. #include <linux/kernel.h>

Crashing kernel:

1. Crashing by including infinite loop

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init load(void){
  while(1){
     printk(KERN_ALERT "Crashing using module\n");
  }
  return 0;
}
```

```
static void unload(void){
  printk(KERN_ALERT "Unload Crash module\n");
}

module_init(load);
module_exit(unload);
```

2. Crashing by including a line which will dereference a NULL pointer;

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init load(void){
 printk(KERN_INFO "Crashing using module\n");
 *(int *)0=0;
  return 0;
}


static void unload(void){
  printk(KERN_INFO "Unload Crash module\n");
}

module_init(load);
module_exit(unload);
```

Part - 2: Linux Process

Created four different Child process:
[i] C1 reads from keyboard and encrypt it using a mapping table and send it to Parent (eg A converted to F, G to I etc). It reads the mapping table from a predefined file.
[ii] C2 reads data from a text file and speaks it using “espeak” linux utility.
[iii] C3 copies a given input file. Here do not use the linux “cp” utility. Code the copy function.
[iv] C4 does CPU monitoring (total %CPU used etc) and periodically sends stats to the parent.

Created each child process using **fork()** system call. For sending data to the parent process we have used **pipe()** system calls. We use 2 file descriptors for each file where for reading we use "fd[0]" and for writing we use "fd[1]".

On running the exe file all four child processes and parent processes will run.
In the command line you can pass any uppercase letter and Child 1 will send the encrypted data to the parent through pipe. Child 2 will use espeak to speak data in the text file. Child 3 and child also do their work.

On running the exe file
./q2 mapping_table.txt espeak.txt mapping_table.txt copy.txt
Example of the output is:
15.2192%
6.20301%
R
R W
6.70846%
6.49189%
2.5%
1.25392%

Part 3 : Multithreaded Merge Sort

1. To run the program use run 'make' command on terminal
2. Then run q3.out <thread_size> <input file> <output file>

We run the program by varying the array size and the thread size.

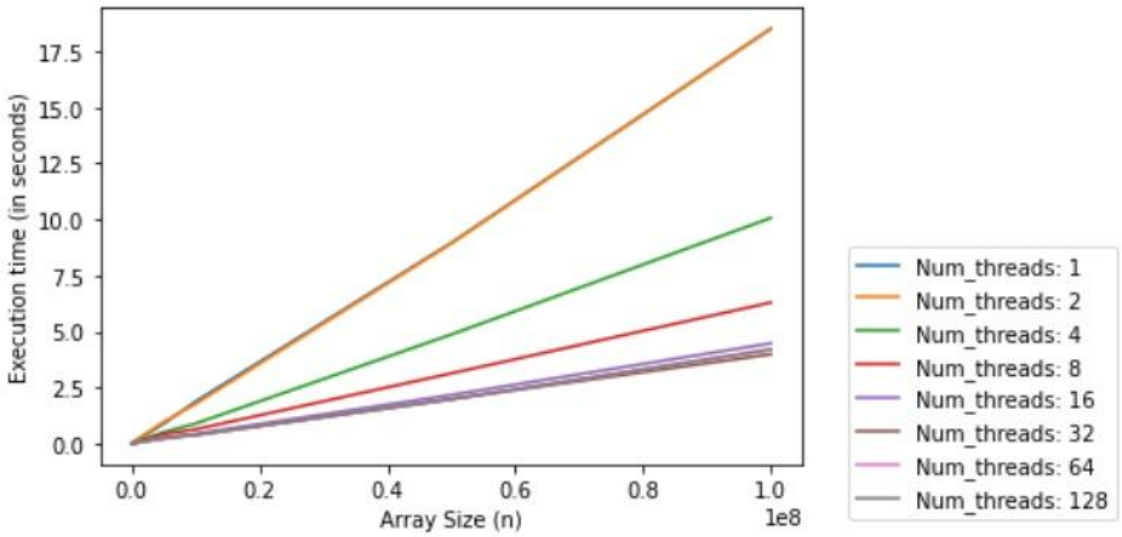
We varied the thread size from 1 to 128 and then we varied the size of the array from 100 to 200000000.

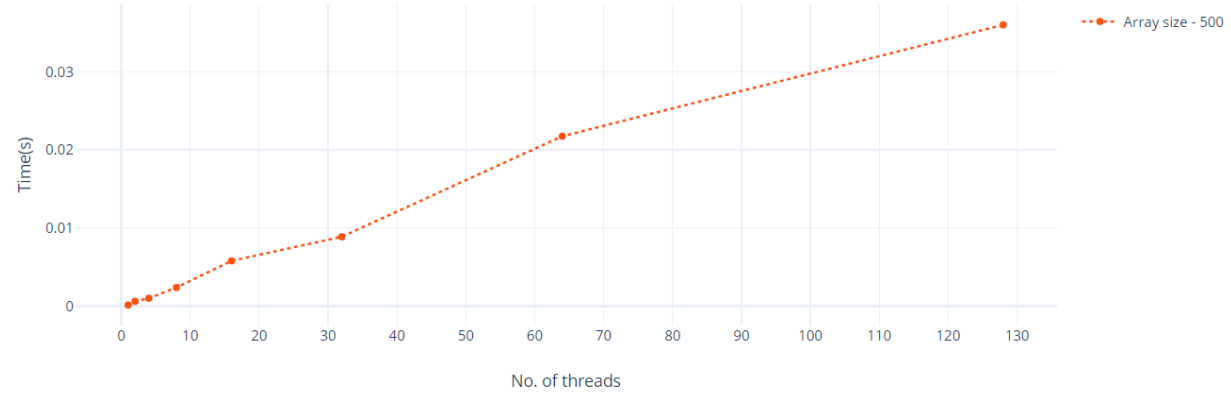
Below is the execution time of the Multithreaded merge sort algorithm.

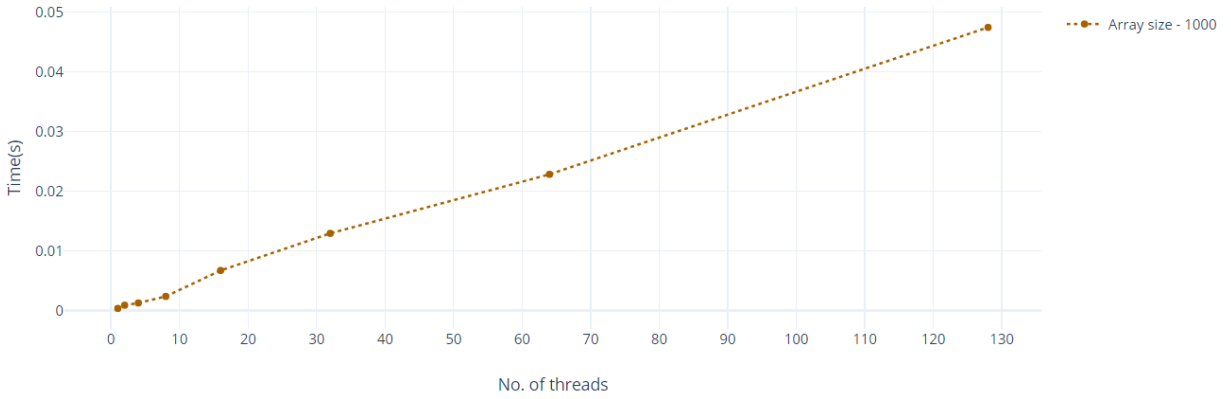
Array Size (n)	Execution time (in seconds) for number of threads= 1	Execution time (in seconds) for number of threads= 2	Execution time (in seconds) for number of threads= 4	Execution time (in seconds) for number of threads= 8	Execution time (in seconds) for number of threads= 16	Execution time (in seconds) for number of threads= 32	Execution time (in seconds) for number of threads= 64	Execution time (in seconds) for number of threads= 128
100	0.000015	0.000523	0.000755	0.001860	0.005966	0.006268	0.015261	0.030865
500	0.000092	0.000421	0.000716	0.001676	0.002691	0.007194	0.011158	0.025498
1000	0.000191	0.000534	0.000769	0.001551	0.003531	0.008031	0.010236	0.021560
5000	0.001156	0.001458	0.001286	0.002494	0.003277	0.006723	0.012686	0.021077
10000	0.002294	0.002650	0.001890	0.002064	0.003176	0.007342	0.010595	0.020678
50000	0.010553	0.015172	0.006613	0.004952	0.005418	0.006306	0.011482	0.021446
100000	0.022886	0.021907	0.012210	0.007711	0.006939	0.008104	0.014867	0.021335
500000	0.099426	0.094688	0.055053	0.037528	0.022328	0.024604	0.028246	0.031094
1000000	0.188824	0.221960	0.117694	0.075988	0.053004	0.055152	0.049400	0.053165
5000000	0.885503	0.974007	0.508922	0.423643	0.226541	0.225696	0.250091	0.276141
10000000	1.875324	1.768807	0.891474	0.635255	0.435368	0.386324	0.378622	0.385604
50000000	8.950847	8.925085	4.863241	3.129386	2.167057	1.985514	1.969322	1.978099
100000000	18.505887	18.498835	10.061554	6.289899	4.469993	3.980061	4.167245	4.205616

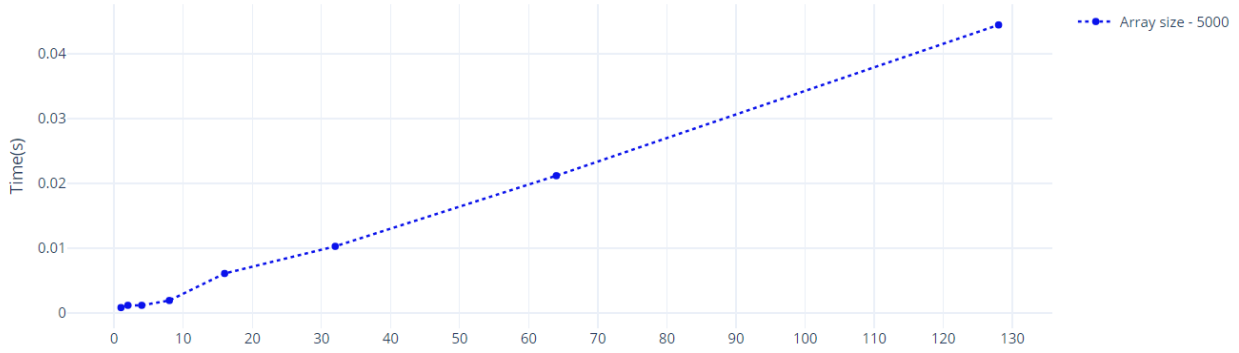
Table 1. Showing Execution time of the algorithm

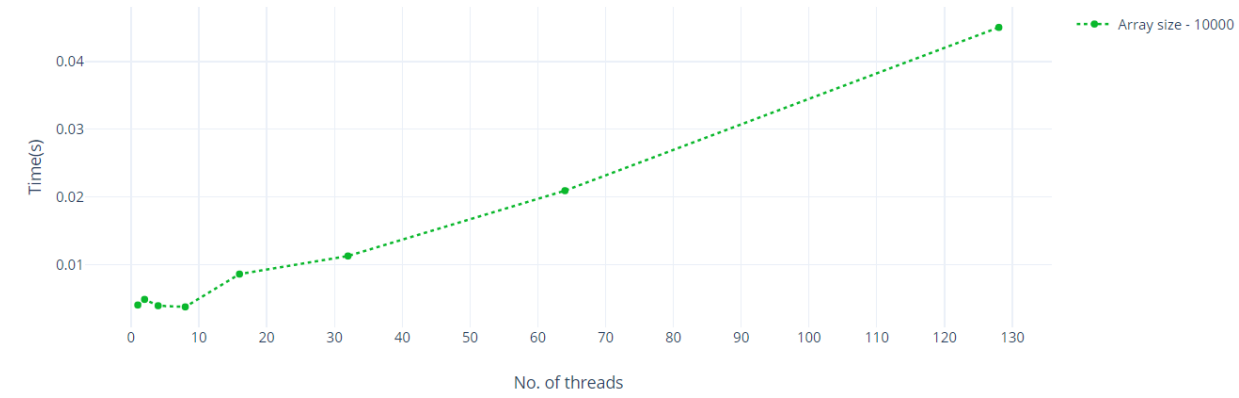
When the size of the array is higher then on increasing the size of the threads decreases the execution time but upto a certain extent. After increasing the thread size more and more, the execution time is increased or remains almost the same. This is because we only have 4 core processors upon which 4 threads will work fine, but after increasing the thread value, we are unnecessarily creating the thread and allocating memory which is taking time.
For a 2GB input file, for 1 thread it is taking 181.32674 seconds while for 32 threads it is taking 27.2345 seconds.
The graphs are as follows:

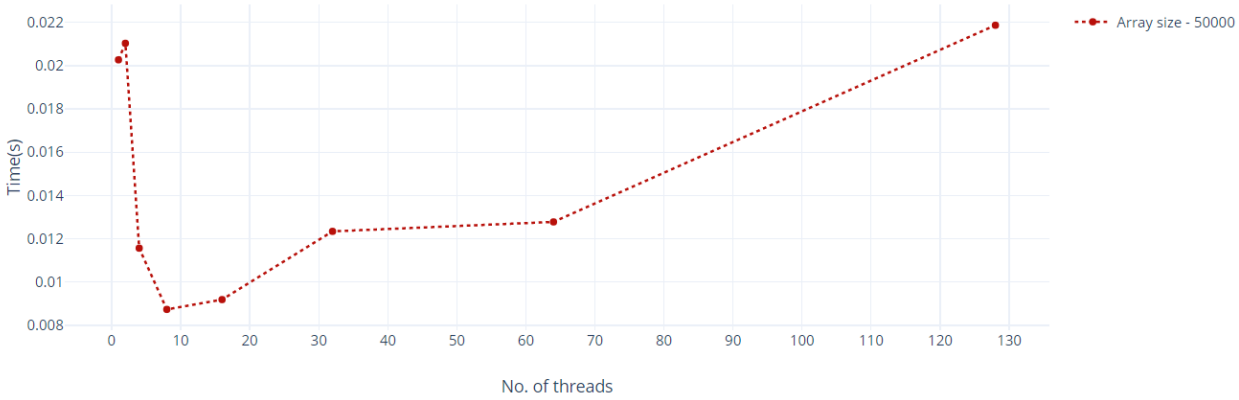


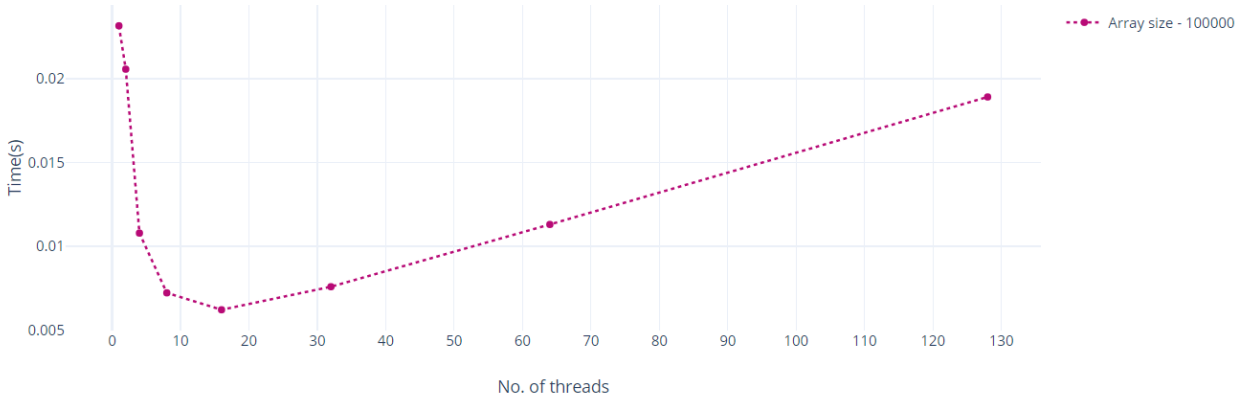

Array size - 500
Time(s)
0.03
0.02
0.01
0
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads

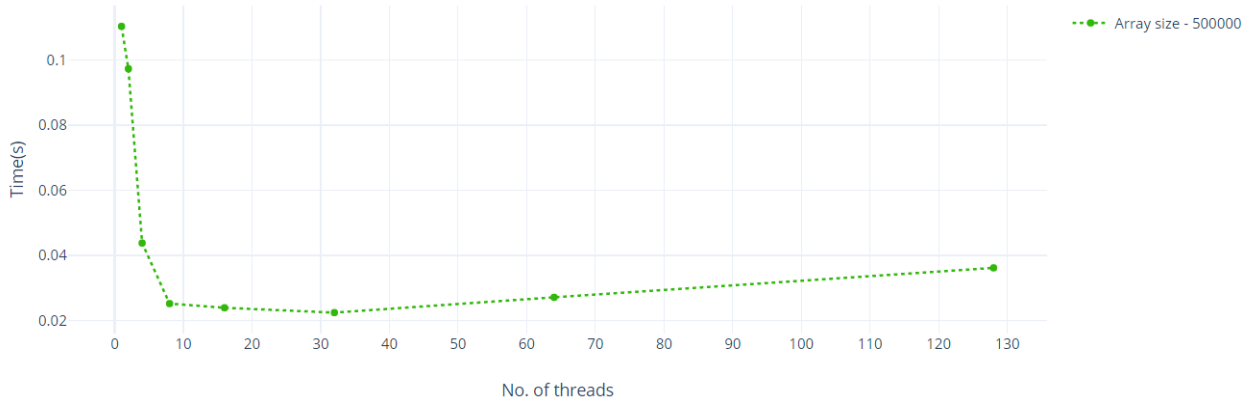

Array size - 1000
Time(s)
0.05
0.04
0.03
0.02
0.01
0
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads

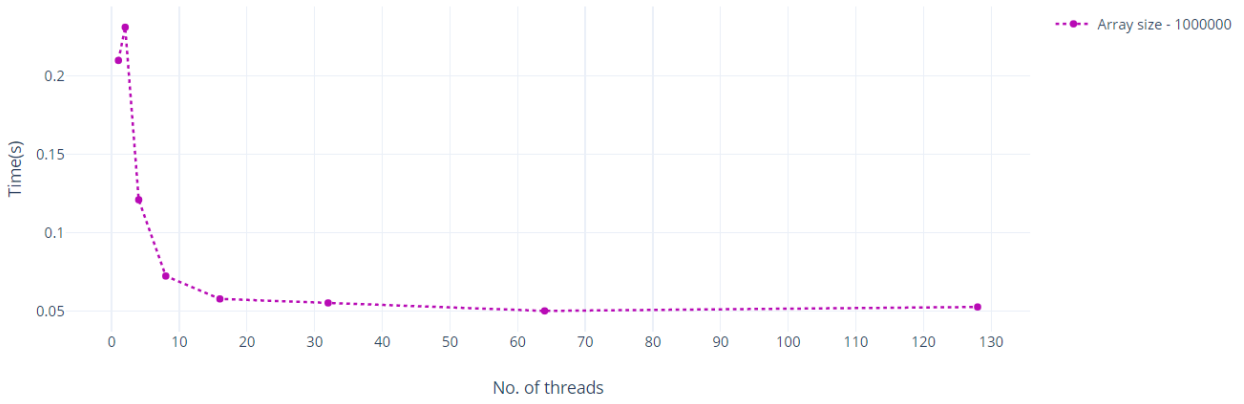

Array size - 5000
Time(s)
0.04
0.03
0.02
0.01
0
0 10 20 30 40 50 60 70 80 90 100 110 120 130

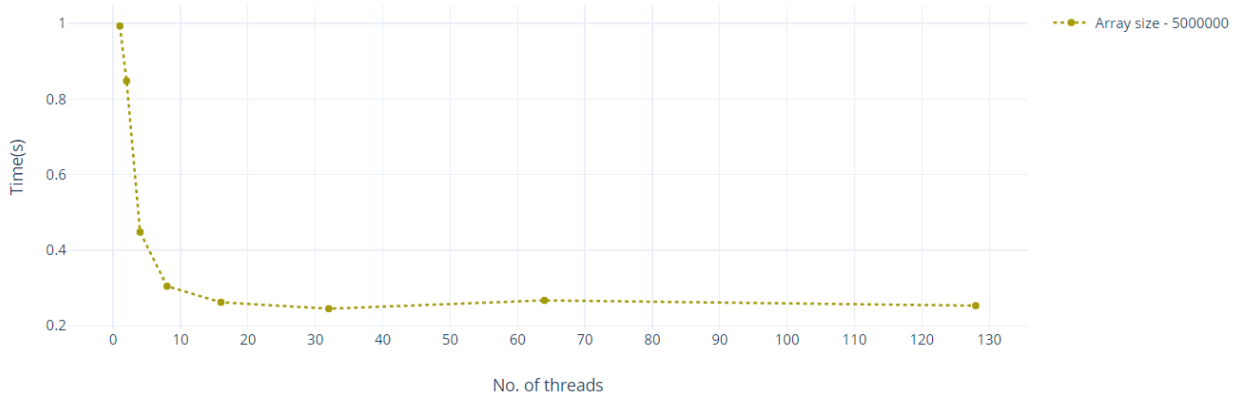

Array size - 10000
Time(s)
0.04
0.03
0.02
0.01
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads

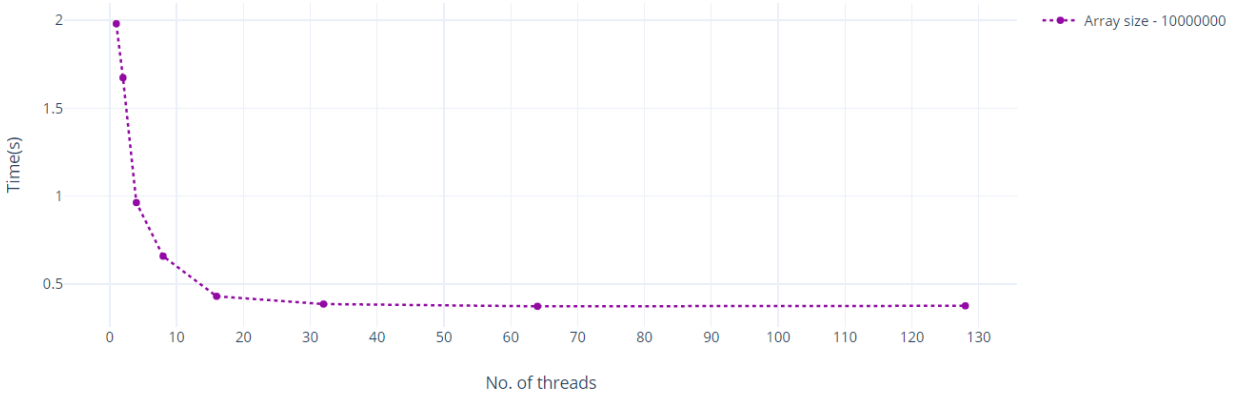

Array size - 50000
Time(s)
0.022
0.02
0.018
0.016
0.014
0.012
0.01
0.008
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads

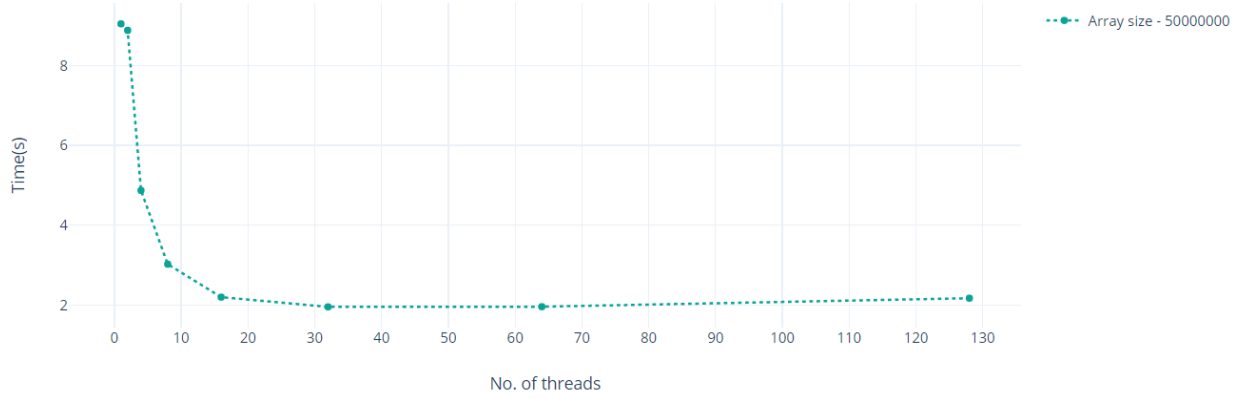

Array size - 100000
Time(s)
0.02
0.015
0.01
0.005
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads

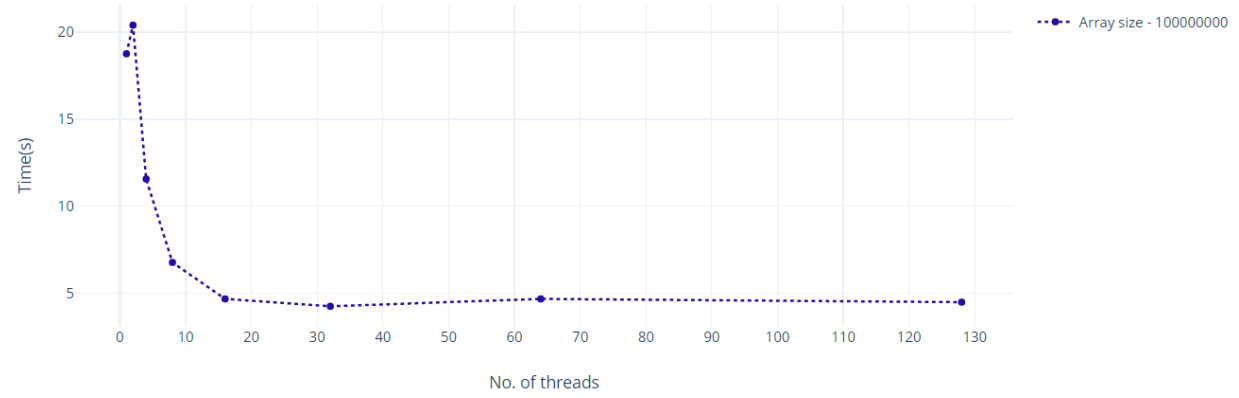

Array size - 500000
Time(s)
0.1
0.08
0.06
0.04
0.02
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads

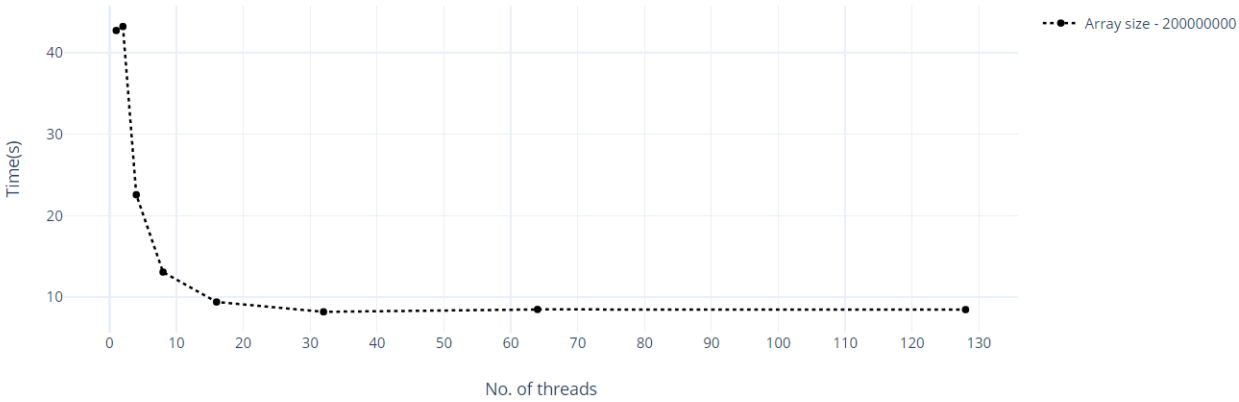

Array size - 1000000
Time(s)
0.2
0.15
0.1
0.05
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads


Array size - 5000000
Time(s)
1
0.8
0.6
0.4
0.2
0 10 20 30 40 50 60 70 80 90 100 110 120 130
No. of threads


Array size - 10000000
Time(s)
2
1.5
1
0.5
0
10
20
30
40
50
60
70
80
90
100
110
120
130
No. of threads


Array size - 50000000
Time(s)
8
6
4
2
0
10
20
30
40
50
60
70
80
90
100
110
120
130
No. of threads


Array size - 100000000
Time(s)
20
15
10
5
0
10
20
30
40
50
60
70
80
90
100
110
120
130
No. of threads


Array size - 200000000
Time(s)
40
30
20
10
0
10
20
30
40
50
60
70
80
90
100
110
120
130
No. of threads