OS Lab 9

Sara Mohamed 900203032
Omar Harb 900201063
Mohamed Shaalan 900201539

Roles

Omar Harb: threads_compute, bonus functions, results and graphs

Mohamed Shaalan: mmap_compute, bonus functions, results and graphs

Sara Mohamed: threads_compute, bonus functions, results and graphs

Inputs and Outputs of mmap

Inside out mmap_compute() function, we initialize the output "result" using the mmap() function

```
int* result = mmap(
        NULL,
        sizeof(int),
        PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS,
        0,
        0
);
```

- `*addr = NULL`, so kernel can place mapping wherever it sees fit
- `length = sizeof(int)` , as result is a shared int
- `prot = PROT_READ | PROT_WRITE`, so all processes sharing this space can read/write
- `flags = MAP_SHARED | MAP_ANONYMOUS`, so all processes mapped to this object share the space; and so that the mapping is anonymous, i.e. not connected to any files
- `fd = 0`
- `offset = 0`, to start from the beginning of the file open on the fd descriptor

Pseudocode of functions

pipes_compute():

```
double pipes_compute(int (*f)(int, int),
char* filename, int n_proc)
{
    if (n_proc == 0)
        exit;

    int N = getcountinfile(filename);
    int* arr = getArray(filename, N);
    start clock;
    pid_t parentid = getpid();

    int count = 0, index = 0;
    int tempread, tempwrite;

    int fdesc_arr[n_proc][2];
    int nperprocess = N/n_proc;
    int extraN = nperprocess + (N%n_proc);


    for (i=0 to n_proc-1)
        if (pipe(fdesc_arr[i]) == -1)
            print error message;
            exit(1);  // initializing pipes, one per
                      child process
```

pipes_compute():


```
while (fork()!=-1 && count < n_proc)
{
    if (getpid() != parent_id)
        close(fdesc_arr[count][0]);    // closes read side of pipe

        if (count == 0)   // if first element, get result using index and extraN
            tempwrite = childResult(index, index + extraN - 1, f);
            write(fdesc_arr[count][1], &tempwrite, sizeof(tempwrite));
            close(fdesc_arr[count][1]);   // close write side after done
        else              // else, get result using index and nperprocess
            tempwrite = childResult(index, index + nperprocess - 1, f);
            write(fdesc_arr[count][1], &tempwrite, sizeof(tempwrite));
            close(fdesc_arr[count][1]);   // close write side after done
        break;
    else
        if (count == 0)
            index += extraN;
        else
            index += nperprocess;
        count++;
}
```

This branch eliminates the need for another pipe- the parent increments the index properly before forking the child that will use that index

child branch

parent branch

pipes_compute():


```
    if (getpid() != parent_id)
        exit;  // children exit program after their computations

    count = 0;
    if (getpid() == parent_id)
        close(fdesc_arr[count][1]);   // close write side of first pipe
        read(fdesc_arr[count][0], &tempread, sizeof(tempread));
        result = tempread;
        count++;        // read first result, store in tempread, increment count

        while (count < n_proc)
            close(fdesc_arr[count][1]);
            read(fdesc_arr[count][1], &tempread, sizeof(tempread));
            result = f(result, tempread);
            count++;  // read remaining results, combine together

    end clock;
    print result;
    free arr;
    return (end clock - start clock);
}
```

Pseudocode of functions

sequential_compute():

```
double sequential_compute(int
(*f)(int, int), char* filename)
{
     int N =
getcountinfile(filename);
     int* arr = getArray(filename,
N);
     start clock;
     int result = 0;

     if (N==0)
          result = 0;
          stop clock;

     else if (N==1)
          result = arr[0];
          stop clock;
     else
          result = arr[0];
          for (i = 1 to N-1)
               result = f(result, arr[i]);
          stop clock;

     free arr;
     print result to screen;

     return (end clock - start clock);
}
```

Pseudocode of functions

mmap_compute():

```
const int N = getcountinfile(filename);
    int* arr = getArray(filename, N);

    initialize result mmap;

    if(result == MAP_FAILED) {
        printf("map failed!\n");
        return 0.0;
    }

    *result = 0;

    Start clock;
    pid_t parent_id = getpid();
```

```
int count = 0, index = 0;

int nperprocess = N / n_proc;

int extraN = nperprocess + (N %
n_proc);
```

//identical to pipes_compute(), we find the number of elements each child process will compute the result of

mmap_compute():

```
while (fork() != -1 && count < n_proc)
{
        if (getpid() != parent_id)
            if(count == 0)  // if first element, get result using index and extraN
                 *result = f(*result, childResult(arr, index,
                 index+extraN-1, f));  //add to result
            else  //else, get result using index and nperprocess
                 *result = f(*result, childResult(arr, index,
                 index+nperprocess-1, f));
             break;
        else
            if(count == 0)
                index += extraN;
            else
                index += nperprocess;
            count++;
}
```

2

child branch

parent branch

mmap_compute():

```
if(getpid() != parent_id)
        exit;  //children exit
    else
        wait for all children to exit;


    end clock;
    print result;

    int err = munmap(result, sizeof(int));   //unmap shared memory
    if(err != 0) {
        printf("unmapping failed\n");
        return -1.0;
    }

    return start clock - end clock;
}
```

Pseudocode of functions

Dependencies: thread_result(), thread_data struct, global variables

```
void* thread_result(void* arg) {
    thread_data* td = (thread_data*) arg;
    if(td->sindex > td->eindex)
        return NULL;

    else
        pthread_mutex_lock(&mutex1);

        // critical section
        for(int i = td->sindex to  td->eindex) {
            tresult = f(tresult, td->arr[i]);

        pthread_mutex_unlock(&mutex1);
}
```

```
// global variables
int tresult = 0;
pthread_mutex_t mutex1;

typedef struct {
    int* arr;
    int sindex;
    int eindex;
} thread_data;
```

Pseudocode of functions

thread_compute():

```
double single_process_thread_compute(int (*f)(int, int), char* filename,
int n_thread) {

    int N = getcountinfile(filename);
    int* arr = getArray(filename, N);

    start clock;
    int nperprocess = N / n_thread;
    int extraN = nperprocess + (N % n_thread);

    pthread_t tid[n_thread];

    thread_data tdata[n_thread];   //array of data associated to each thread
```

thread_compute():

2

```
int error, index = 0;
    for(int i = 0 to n_thread - 1) {
        tdata[i].arr = arr;  // "assign" each thread the array
        tdata[i].sindex = index;  // "assign" start index = index
        if(i == 0)
            tdata[i].eindex = index+extraN-1;
            index += extraN;
        else
            tdata[i].eindex = index+nperprocess-1;
            index += nperprocess;
        error = pthread_create(&(tid[i]), NULL, thread_result, (void*)
&tdata[i]);
        if(error != 0)
            printf("Thread can't be created: [%s]\n", error);
            i--; revert index value;  // try again since we have to create n_thread
threads
    }

    for(int i = 0; i < n_thread; i++)
        pthread_join(tid[i], NULL);
    // join all threads
```

```
end clock, free arr, print result;
return (end clock - start clock);
```

Run Example

```
Wrote from 1 to 500 to file 'test.txt'.
[sequential compute] 125250, by process 106826
        executed in 0.000002 seconds
[multi pipes compute]: 125250, by process 106826
        executed in 0.000565 seconds
[mmap_compute with 16 child processes]: 125250, by process 106826
        executed in 0.000507 seconds
[single process thread compute with 16 threads]: 125250, by process 106826
        executed in 0.000308 seconds
```

```
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
```

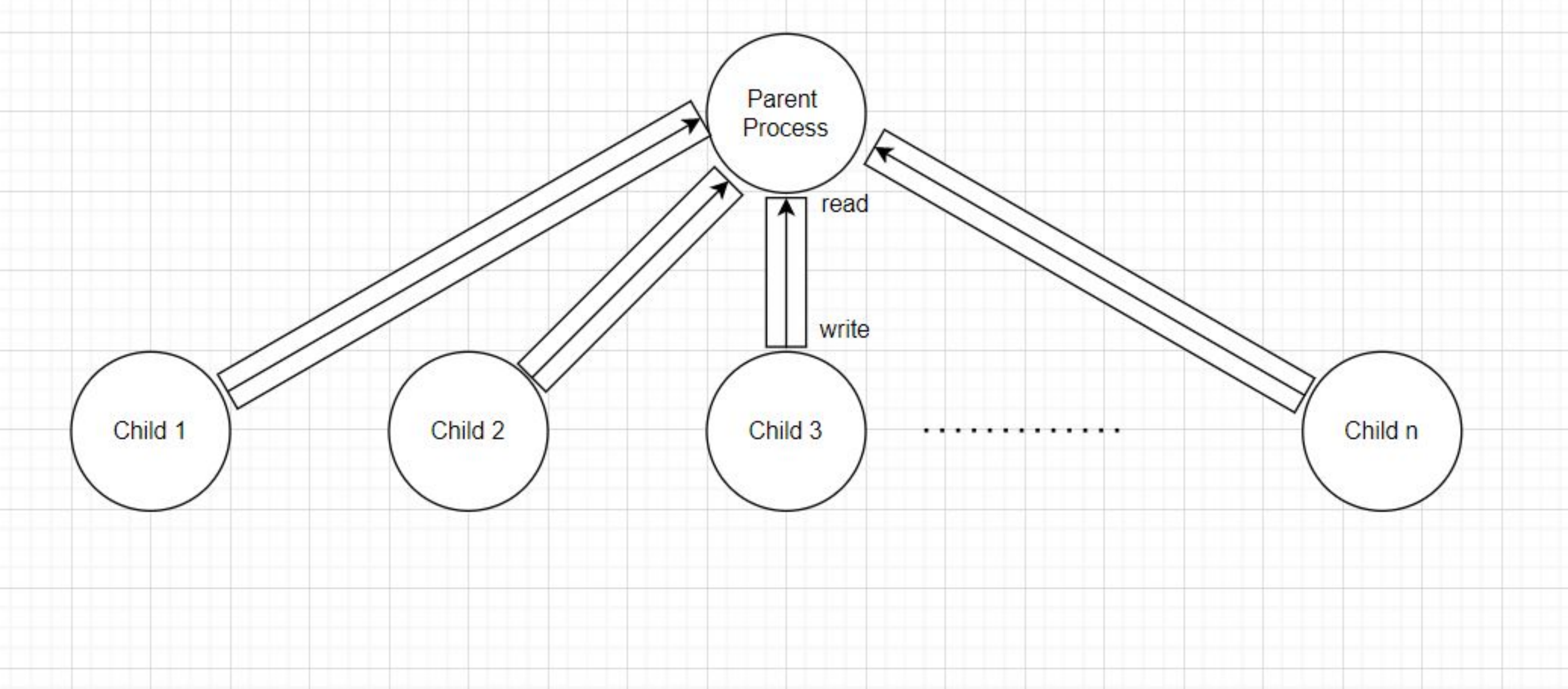
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

This run enters 500 elements (1, 2, 3, ..., 500) to file and then computes them using the four functions

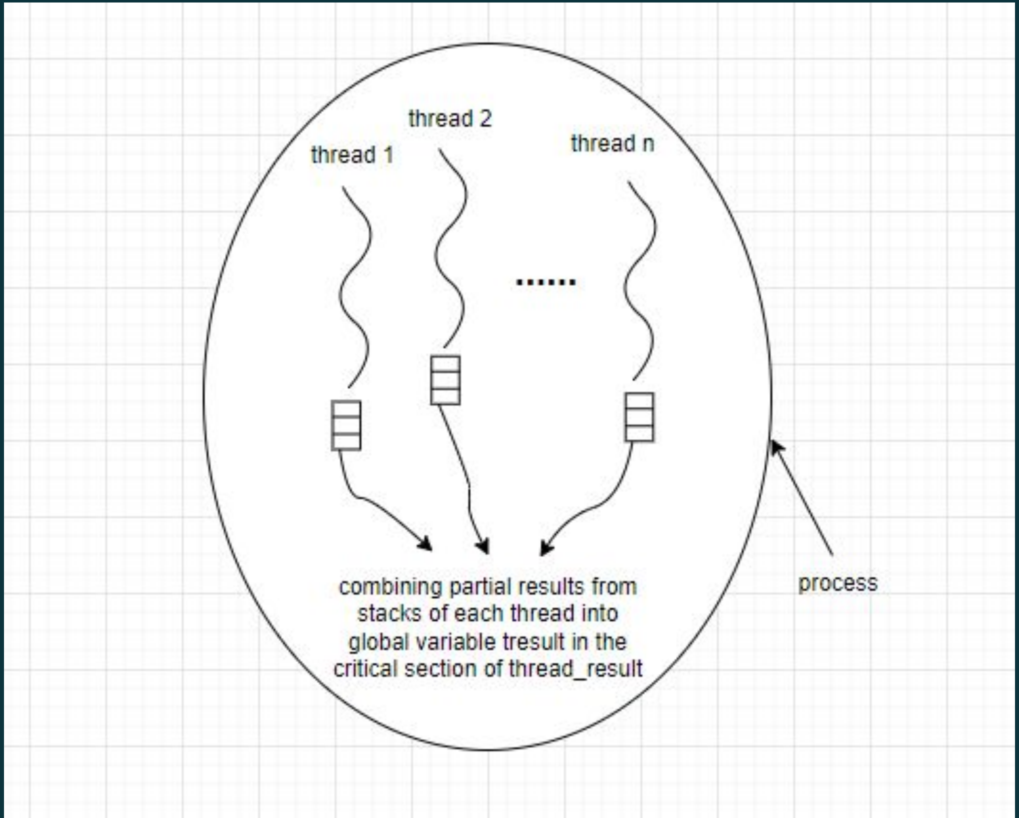
$$\text{result} = \sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

$$= 250(501) = 125250.$$

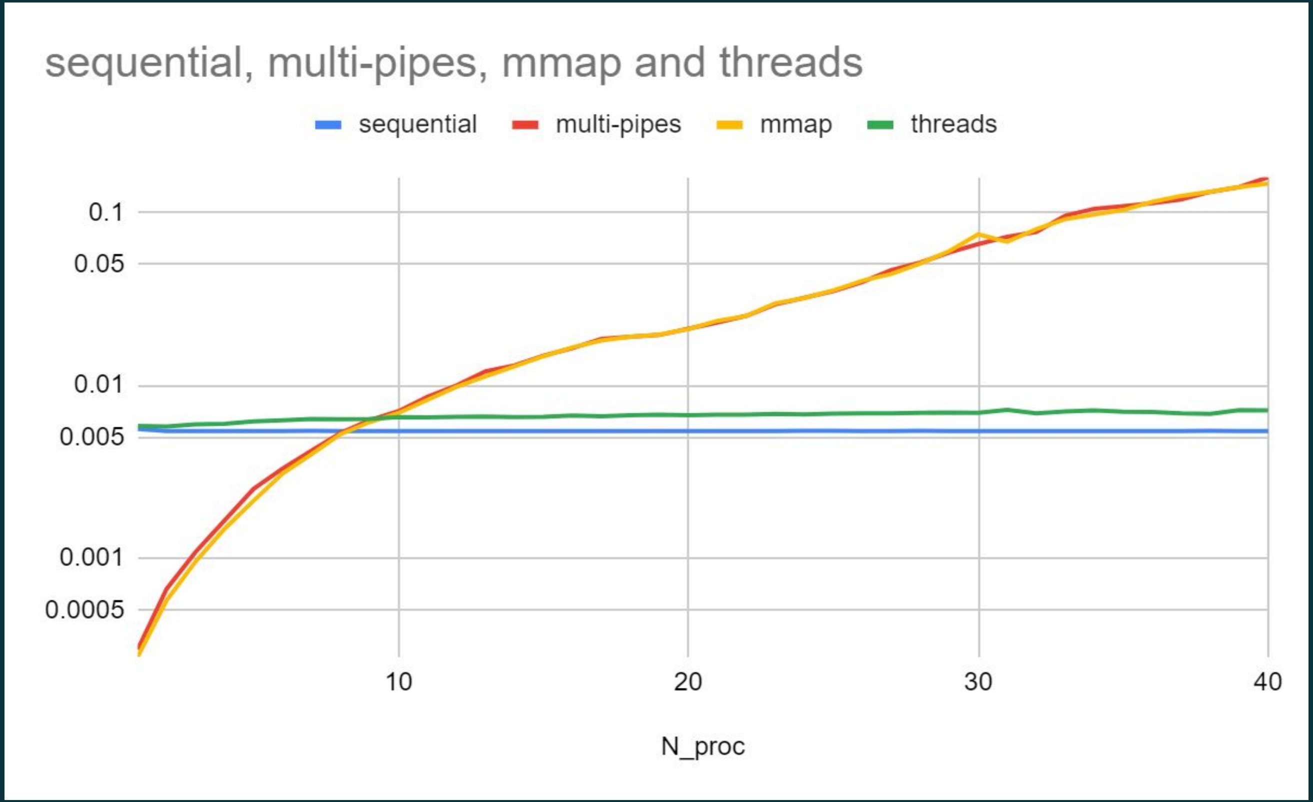
Diagrams: Pipes



Diagrams: Threads

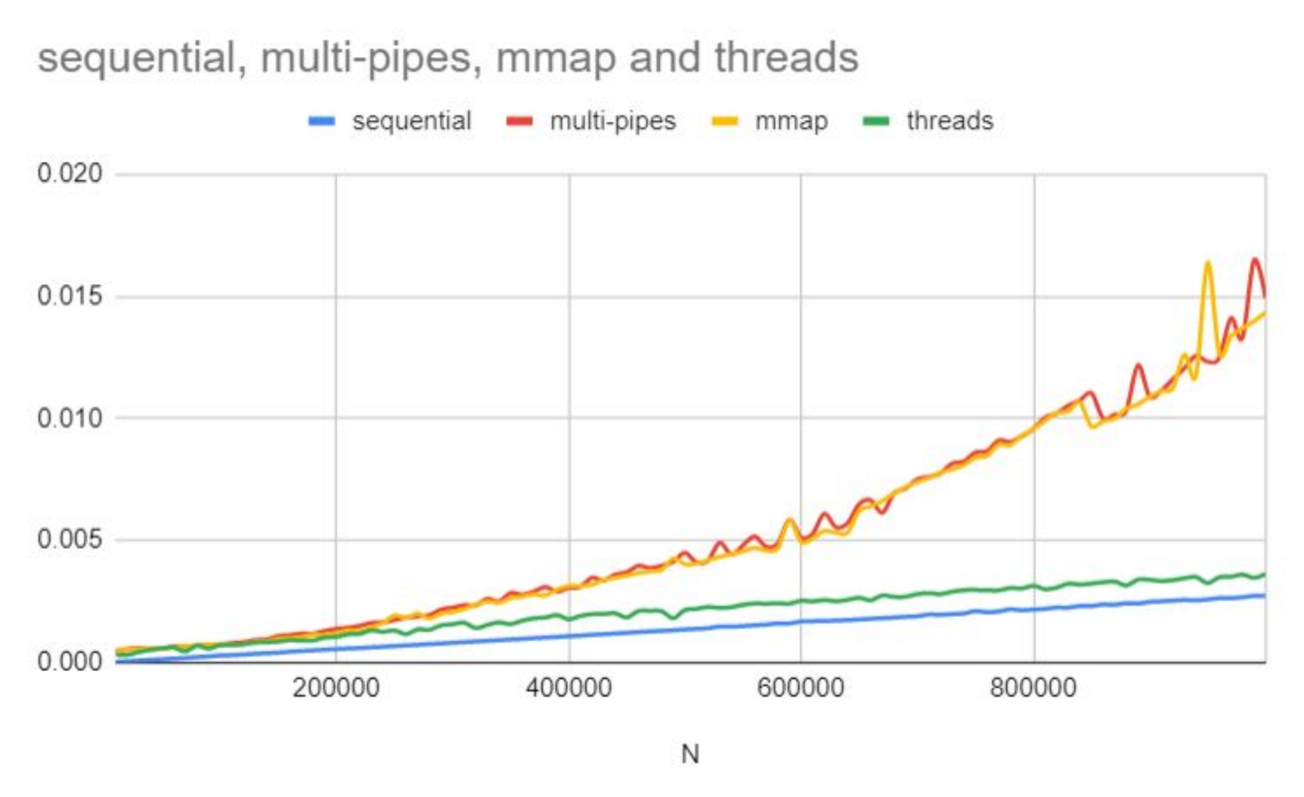


Graphs



The graph contains data with N fixed at 2M, and n_proc ranging from 1 to 40. Tests were conducted on a PC with 8 CPU cores. Intersection occurs at n_proc = 8. Graph presented in log scale for clearer differentiation of results.

Graphs

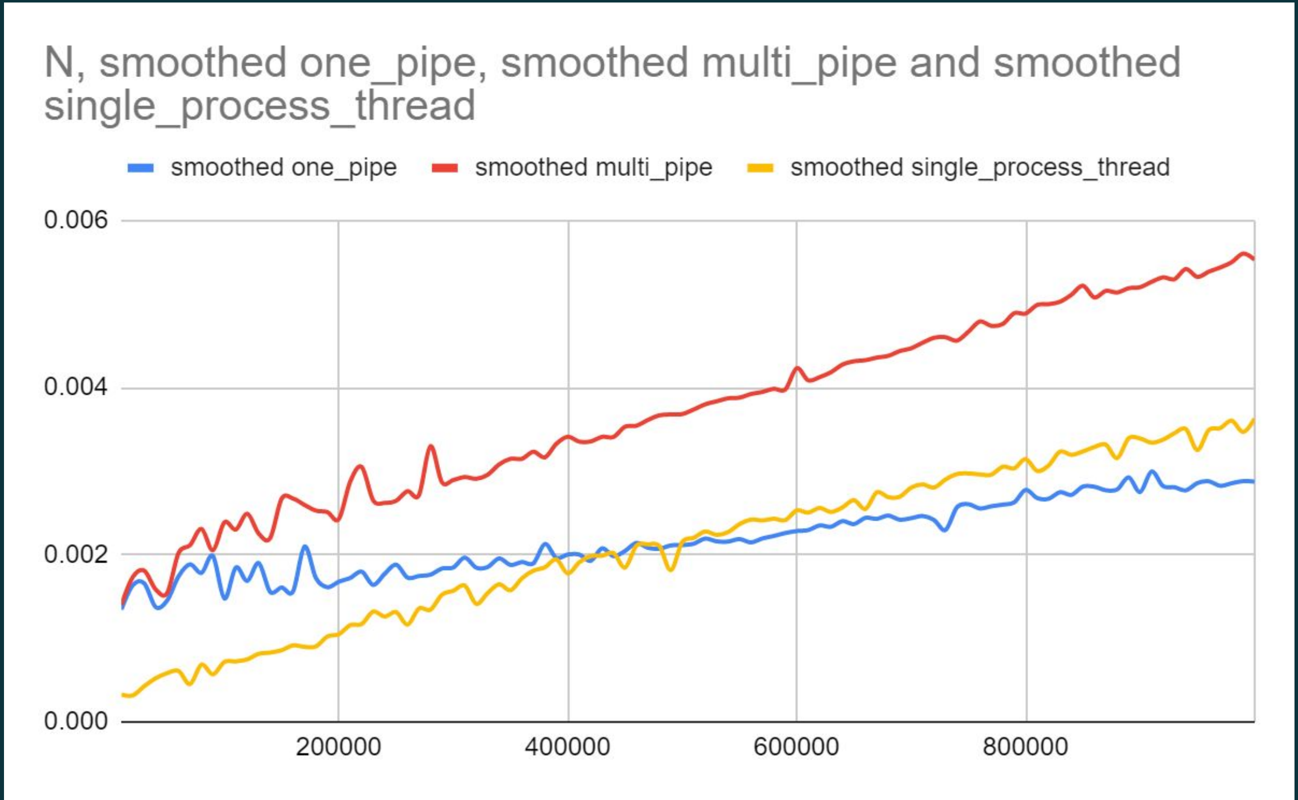


The graph contains data with n_proc and n_thread fixed at 8, and N ranging from 10K to 1M. Tests were conducted on a PC with 8 CPU cores.

Bonus Functions

- created `multi_process_thread_compute`, `single_process_thread_compute`, `multi_pipe_compute`, and `one_pipe_compute` functions (attached in zipped file)
- For comparison graph, fixed n_thread and n_proc at 8, as testing was conducted on a PC with 8 CPU cores
- `Multi_process_thread_compute` generated error, so did not include it in our generated graph (but still attached in zipped file)

Bonus Graph



The graph contains data with n_proc and fixed at 8, and N ranging from 10K to 1M. Tests were conducted on a PC with 8 CPU cores.

Ranges and Results

- Testing and runs were conducted on a PC in the Systems Engineering Lab with 8 CPU cores
- Smoothing of graphs was conducted by getting the median of three readings.
- Fixing n_proc and n_thread to 8, we see that sequential compute always takes the least time, as it has no overhead tied to it, and thread_compute() nears it with increasing values of N.
- Fixing N to 2M, we see that with n_proc and n_thread less than 8, sequential and thread_compute take longer time, but the behaviour reverses for values on n_proc and n_thread greater than 8.
- Ranges of tests chosen to both reflect full graph view while maintaining details and information for the smaller values.