OS Lab 7

Sara Mohamed 900203032
Omar Harb 900201063
Mohamed Shaalan 900201539

Roles

Sara Mohamed: *parallel_compute + graphs*

Omar Harb: *parallel_compute + graphs*

Mohamed Shaalan: *sequential_compute + graphs*

Some Dependencies

Pseudocode and description

```
int getcountinfile(char*
filename)
{
     open file;
     int count = 0; int a;
     while (fscanf(a)!=EOF)
          count++;
     close file;
     return count;
}
```

// function to get number of ints in file

```
int* getArray(char* filename,  int N)
{
     open file;
     int arr[N];
     int i = 0; int a;
     while (fscanf(a)!=EOF)
     {
          arr[i] = a;
          i++;
     }
     close file;
     return arr;
}
```

// function to save ints from file into array

Sequential_compute

Pseudocode and description

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

    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);
}
```

childResult Dependency

Pseudocode and description

// returns result of computation from sindex to eindex in arr

```
int childResult(int* arr, int sindex, int eindex, int (*f)(int, int))
{
    if (sindex == eindex)
        return arr[sindex];   // if start and end index are equal, return the one element

    else if (sindex > eindex)
        return 0;             // if end index comes before start index, return 0

    else if (eindex == sindex+1)
        return f(arr[sindex], arr[eindex]);   // if two elements, return their computation

    else
        int result = f(arr[sindex], arr[sindex+1]);
        for (int i = sindex+2; i <= eindex; i++)
            result = f(result, arr[i]);    // compute result of all elements in interval
        return result;
}
```

Parallel_compute

Pseudocode and description

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

```
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

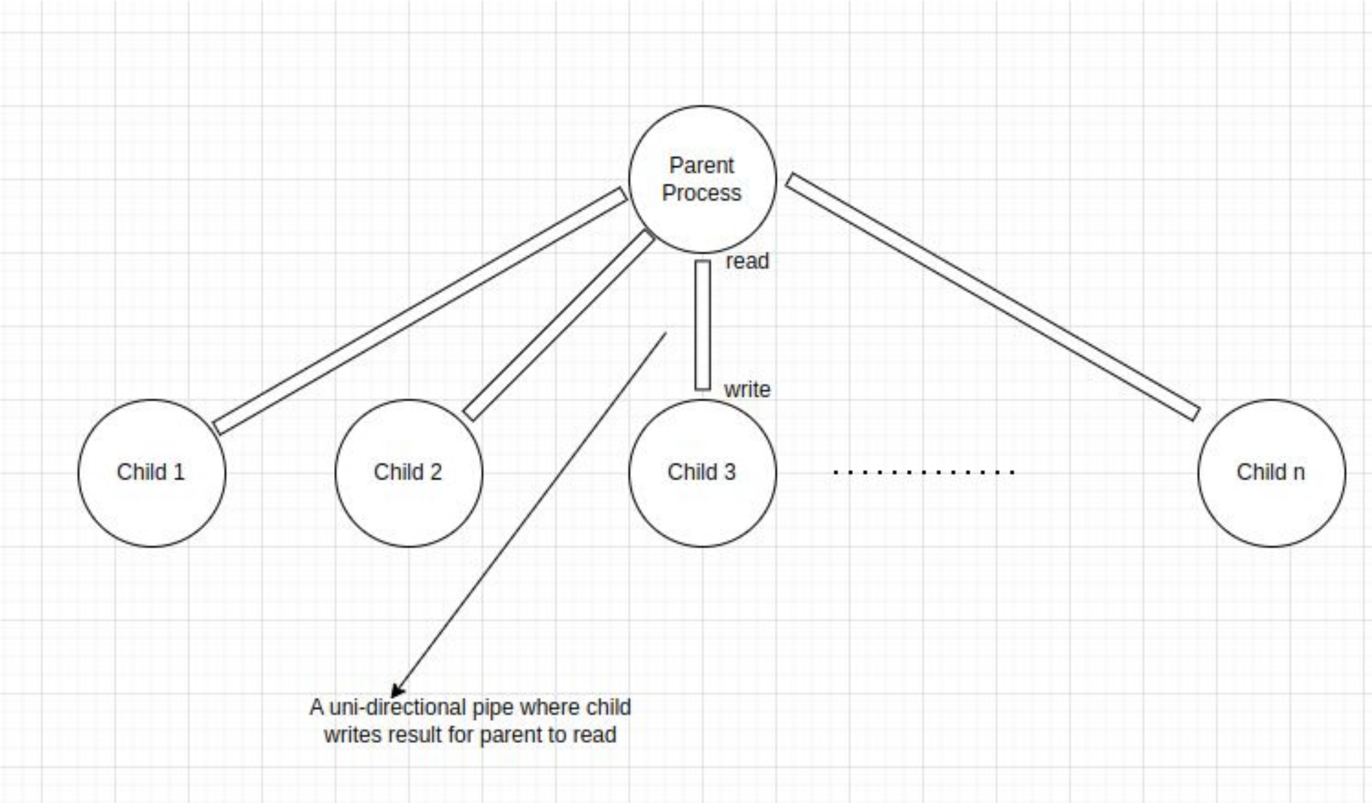
```
if (getpid() != parent_id)
     exit(0);  // children exit program after their computations

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);
}
```

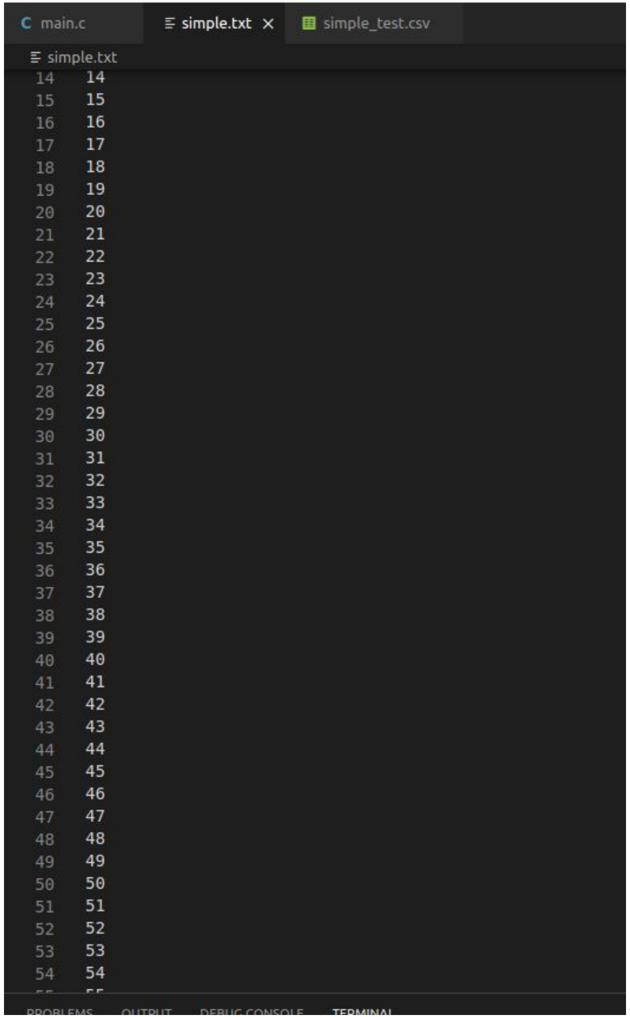
Parallel Compute Diagram



Each child shares a single unidirectional pipe with the parent. The read side is the parent's side, and the write side is the child's. Data (the child's computations) flows from the child to the parent to be combined in result and returned by the function at the end.

Example of Run

```
wrote from 1 to 500 to file 'simple.txt'.
[sequential compute] 125250, by process 2293023
        executed in 0.000003 seconds
[parallel compute with 16 processes]: 125250, by process 2293023
        executed in 0.000589 seconds
```



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

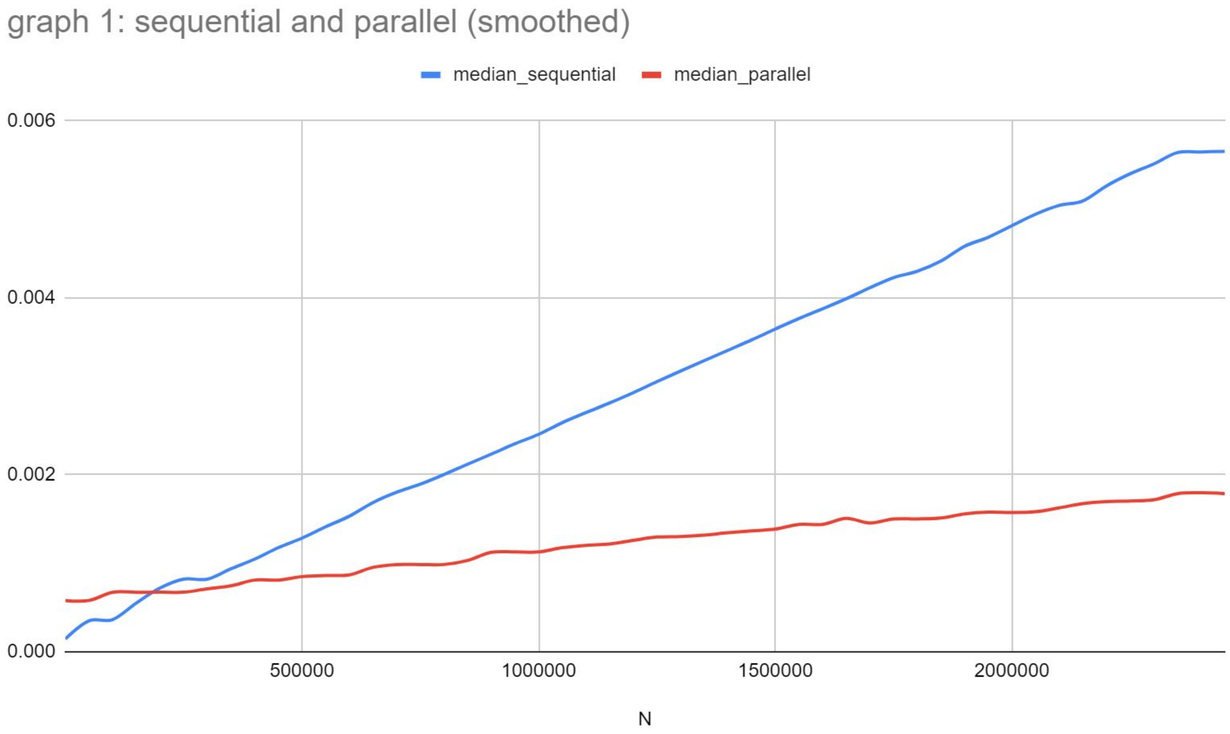
result = $\sum_{i=1}^{n} i = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$

$= 250(501) = 125250.$

Ranges and Results

This graph contains data ranging from N=1 to N=2.5M, with n_proc set at 16 processes.

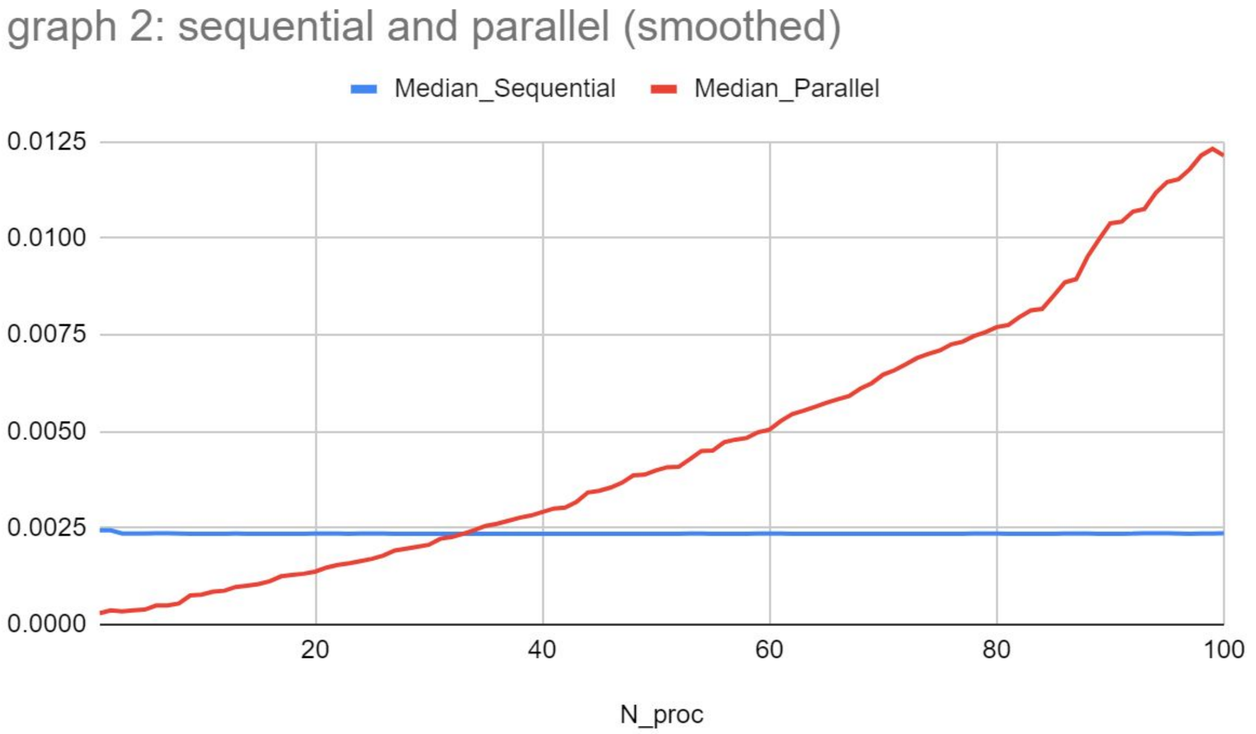
N	sequential	parallel	median_sequential	median_parallel
1	0.000001	0.000743	0.00014	0.000575
50001	0.00014	0.000575	0.000345	0.000575
100001	0.000345	0.000547	0.000359	0.000668
150001	0.000359	0.000668	0.000546	0.000668
200001	0.000546	0.000806	0.000715	0.000669
250001	0.000837	0.000646	0.000816	0.000669
300001	0.000715	0.000669	0.000816	0.000707
350001	0.000816	0.000707	0.000934	0.000741
400001	0.000934	0.000741	0.001043	0.000806
450001	0.001043	0.000806	0.001173	0.000806
500001	0.001173	0.000845	0.00128	0.000845
550001	0.00128	0.000795	0.001408	0.000858
600001	0.001408	0.000858	0.001528	0.000864
650001	0.001528	0.000864	0.001683	0.00095
700001	0.001683	0.00095	0.0018	0.000982
750001	0.0018	0.000991	0.001891	0.000982
800001	0.001891	0.000982	0.002	0.000982
850001	0.002	0.000977	0.002117	0.001028
900001	0.002117	0.001028	0.002231	0.00112
950001	0.002231	0.001189	0.002348	0.001124
1000001	0.002348	0.00112	0.002455	0.001124
1050001	0.002455	0.001124	0.002587	0.001172
1100001	0.002587	0.001172	0.0027	0.001198
1150001	0.0027	0.001198	0.00281	0.001213
1200001	0.00281	0.001213	0.002926	0.001255
1250001	0.002926	0.001255	0.003051	0.001293
1300001	0.003051	0.001299	0.003171	0.001299
1350001	0.003171	0.001293	0.003289	0.001315
1400001	0.003289	0.001341	0.003406	0.001341
1450001	0.003406	0.001315	0.003523	0.001362
1500001	0.003523	0.001383	0.003646	0.001383
1550001	0.003646	0.001362	0.003766	0.001436
1600001	0.003766	0.001436	0.003875	0.001436
1650001	0.003875	0.001555	0.003989	0.001504
1700001	0.003989	0.001408	0.004114	0.001452
1750001	0.004114	0.001504	0.004228	0.001498
1800001	0.004228	0.001452	0.0043	0.001498
1850001	0.0043	0.001498	0.004415	0.001508
1900001	0.004415	0.001508	0.004583	0.001555
1950001	0.004583	0.001574	0.004685	0.001574
2000001	0.004685	0.001555	0.004816	0.00157
2050001	0.004816	0.001579	0.004946	0.001579
2100001	0.004946	0.00157	0.005045	0.001623
2150001	0.005045	0.001672	0.005096	0.001672
2200001	0.005096	0.001623	0.005267	0.001695



Ranges and Results

This graph contains data ranging from n_proc=1 to n_proc=100, with N set at 1M data points

	A	B	C	D	E
1	N_proc	Sequential	Parallel	Median_Sequential	Median_Parallel
2	1	0.002349	0.000164	0.00244	0.000292
3	2	0.00244	0.000451	0.00244	0.000366
4	3	0.002442	0.000292	0.002358	0.000339
5	4	0.002358	0.000366	0.002355	0.000366
6	5	0.002351	0.000339	0.002355	0.000388
7	6	0.002355	0.000388	0.002361	0.000491
8	7	0.002361	0.000494	0.002361	0.000494
9	8	0.002361	0.000491	0.002356	0.000545
10	9	0.002356	0.000545	0.002352	0.000753
11	10	0.002348	0.000774	0.002352	0.000774
12	11	0.002352	0.000753	0.002352	0.000849
13	12	0.002352	0.000849	0.002352	0.000876
14	13	0.002355	0.000876	0.002355	0.000972
15	14	0.002348	0.000972	0.002352	0.001008
16	15	0.002357	0.001008	0.002352	0.001049
17	16	0.002352	0.001049	0.002351	0.00112
18	17	0.002348	0.00112	0.002351	0.001249
19	18	0.002351	0.001249	0.002351	0.001285
20	19	0.002351	0.001285	0.002352	0.001317
21	20	0.002352	0.001317	0.002355	0.00137
22	21	0.002361	0.00137	0.002355	0.001475
23	22	0.002355	0.001475	0.002355	0.001544
24	23	0.002352	0.001544	0.002352	0.001589
25	24	0.002355	0.001589	0.002355	0.001644
26	25	0.002351	0.001644	0.002356	0.001702
27	26	0.002356	0.001702	0.002356	0.001788
28	27	0.002356	0.001788	0.00235	0.001917
29	28	0.00235	0.001966	0.002349	0.001966
30	29	0.002349	0.001917	0.002349	0.002016
31	30	0.002349	0.002016	0.002349	0.002065
32	31	0.002351	0.002065	0.002351	0.00222
33	32	0.002348	0.00222	0.002348	0.002264
34	33	0.002351	0.002264	0.00235	0.002346
35	34	0.002346	0.002346	0.00235	0.002442
36	35	0.00235	0.002442	0.00235	0.002553
37	36	0.00235	0.002553	0.00235	0.00261
38	37	0.002351	0.00261	0.002351	0.002685
39	38	0.002347	0.002685	0.002351	0.002766
40	39	0.002362	0.002766	0.002353	0.002824
41	40	0.002351	0.002824	0.002351	0.002913
42	41	0.002353	0.002913	0.002353	0.003003
43	42	0.00235	0.003003	0.00235	0.003029
44	43	0.002353	0.003029	0.002351	0.003175
45	44	0.00235	0.003175	0.002351	0.003419
46	45	0.002351	0.003419	0.002352	0.003463



Ranges and Results

- Testing and runs were conducted on a PC in the Systems Engineering Lab with 16 CPU cores (hence setting n_proc at 16)

- Smoothing of graphs was conducted by getting the median of elements i.e. element 1 was the ***median*** of reading 1, 2, and 3; and element 2 was the ***median*** of reading 2, 3, and 4; etc.

- In the first graph, intersection occurs at approximately N = 200,000. Parallel_compute outperforms sequential_compute at that approximate value.
  In the second graph, intersection occurs at approximately n_proc = 33~34. Sequential compute outperform parallel_compute at that value approximately.