1

a)

SISD	MISD
SIMD	MIMD

Flynn’s taxonomy is a system to classify architectures. The differences are whether the architecture uses single or multiple instruction streams and similarly for data streams.

i)

Using MPI you write one program, which means you only write one set of instructions, but when executed, there are multiple processes with its own instruction stream, so in that sense MPI seems to fall under multiple instruction. As for the data streams, you can write to different places individually, but you might want to collect the data before writing it once, but given that you have the choice I would then say that it falls under MIMD.

b)

i)

MPI uses message passing to coordinate between ranks. Otherwise there is no shared memory, so MPI does not fit under Shared Memory.

c)

i)

Given that memory isn’t shared, it would then naturally be distributed.

2

a)

i)

My implementation uses the stupid approach where all the ranks compute their sums then send them off to rank 0. This means that rank 0 has to receive p times, where p is the number of other ranks. The other ranks might also have to wait for the preceding ranks to do their send operations before they can do their own send and then carry on.

ii)

It is possible to reduce this bottleneck by applying a binary tree. For example half of the ranks can have a buddy rank, who sends them their sum, then in the next iteration half of those ranks send their sums to another one. This way the amount of ranks left is halved each time. Eventually one rank is the only one left, and that rank has only received O(lg p) messages, which is certainly better than O(p)

b)

i)

As of now, the load is split between each rank, where the job is to sum up a number of unit calculations, where each calculation is the reciprocate of the log of an integer

ii)

Computationally my program doesn’t have a simple fix that will reduce the bottleneck without changing the structure of the problem. One thing that could be done would be to simply use more ranks, so that each process gets less work to do, but that's not really the best way to optimize it since we have a limited amount of processors. A better way might be to use calculus to try to find a way to calculate the sum directly with some integral or something.

2

i)

The amount of operations as a function of the interval n and number of processes p has two parts. The calculations per process are O(n/p), so the program as a whole has to do O(n) calculations.

ii)

Due to my implementation, the total amount of MPI operations are O(n).

iii)

The average amount of MPI operations per process is about 2.

iv)

The maximum number of MPI operations is O(n), which is horrible.

v)

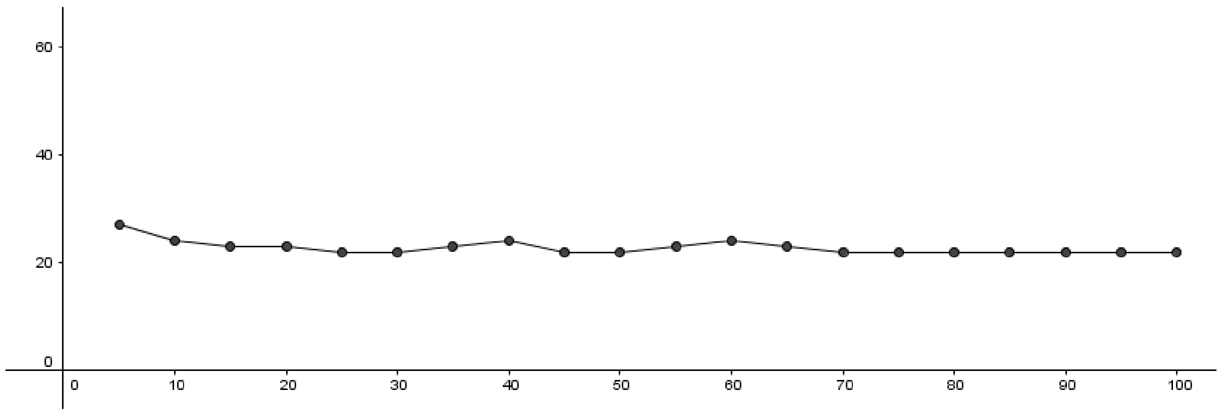
The full table of run times:

	1	2	4	8
5	27	1051	1044	1057
10	24	1039	1043	1060
15	23	1030	1041	1055
20	23	1031	1041	1054
25	22	1034	1033	1055
30	22	1041	1043	1054
35	23	1042	1051	1061
40	24	1045	1042	1065
45	22	1036	1040	1055
50	22	1034	1043	1051
55	23	1040	1039	1058
60	24	1046	1043	1062
65	23	1040	1052	1054
70	22	1039	1038	1053
75	22	1039	1038	1053
80	22	1042	1032	1056
85	22	1037	1047	1053
90	22	1036	1036	1056
95	22	1042	1039	1054
100	22	1033	1041	1056
50000	23	1030	1045	1054
100000	24	1042	1040	1054
150000	25	1032	1042	1054
200000	26	1028	1041	1055

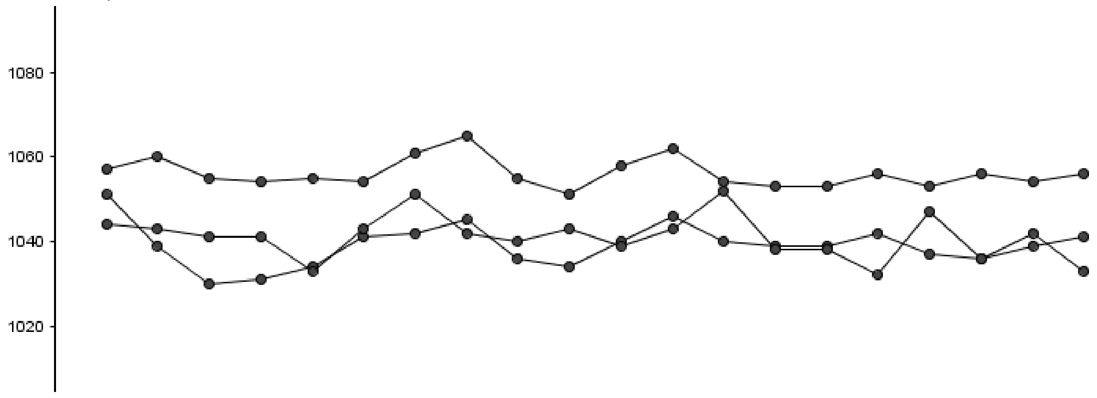
250000	28	1040	1043	1047
300000	28	1043	1045	1061
350000	29	1032	1042	1056
400000	30	1038	1037	1054
450000	31	1043	1046	1059
500000	32	1036	1041	1051
550000	34	1048	1045	1042
600000	34	1039	1041	1052
650000	34	1033	1042	1054
700000	36	1038	1045	1054
750000	37	1043	1053	1054
800000	38	1050	1039	1049
850000	38	1055	1038	1048
900000	40	1034	1047	1055
950000	48	1051	1041	1059
1000000	53	1040	1056	1054

50000000	1013	1544	1451	1275
100000000	2001	2618	1560	1476
150000000	2988	2567	1828	1687
200000000	3993	3032	2076	1907
250000000	5011	3520	2366	2112
300000000	5977	4010	3586	2321
350000000	6923	4485	3864	2522
400000000	10659	5014	4308	2777
450000000	8915	5501	3442	2953
500000000	9976	8979	3647	3216
550000000	10922	6498	5545	3369
600000000	11884	7113	4234	3581
650000000	12819	7526	6337	3826
700000000	13820	8046	4684	4095
750000000	14750	8458	4968	4508
800000000	15830	9048	7488	4497
850000000	16820	9504	7957	4668
900000000	17777	10035	8385	4852
950000000	18824	16192	8834	5032
1000000000	19854	11000	9217	5355

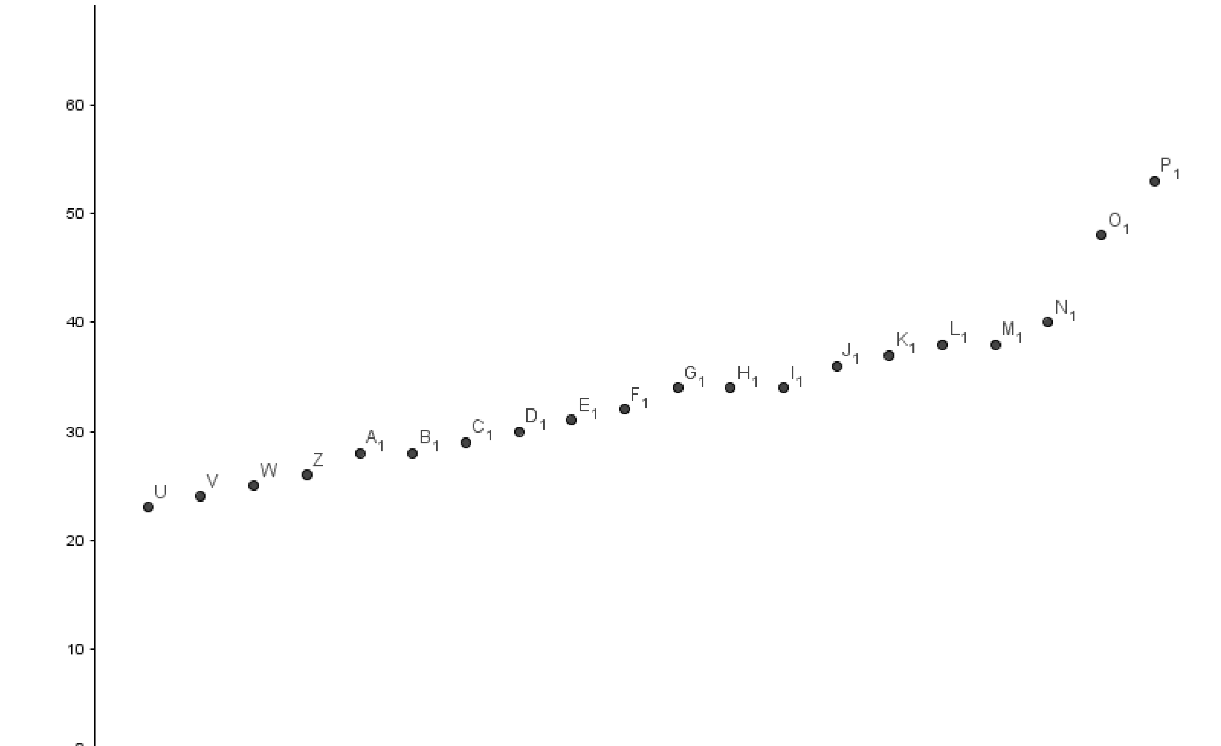
Which can be represented visually in a couple of graphs



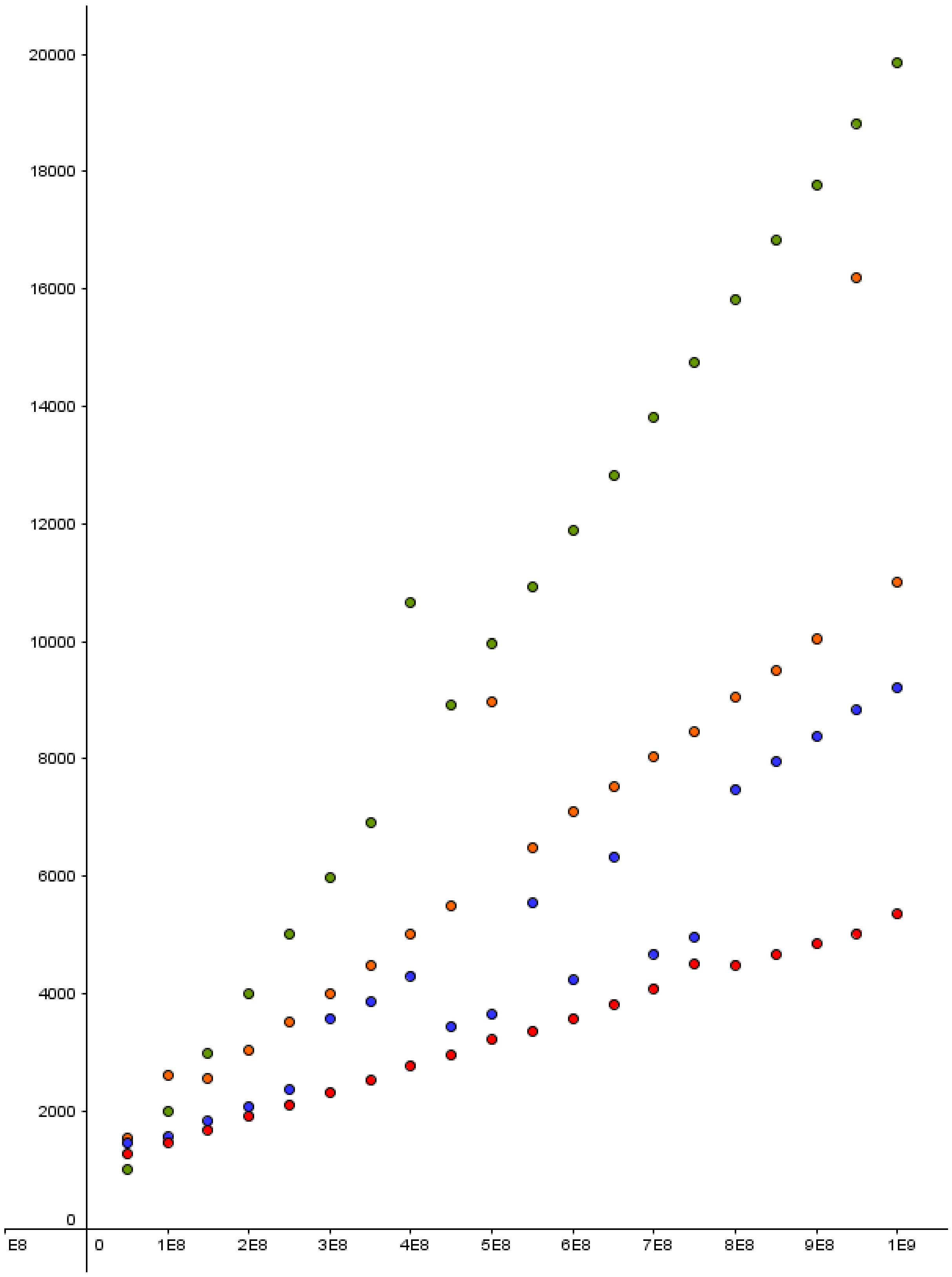
First off, the smallest intervals running with 1 processor. This is a very boring result, because the run time doesn't vary at all.



Second graph shows the same phenomenon when multiple processors are in use. The run time is pretty much the same across all of them as well, which indicates that there is an iherent cost in running in parallel, but it doesn't seem to matter much how many programs are run in parallell.



Thist graph shows the run time for 1 process in the medium range of intervals. At least here we see that the program starts to spend more time to compute the larger intervals. For the multi proccess results, the values were pretty much the same as in the smallest range, and not changing, so I deceided to not graph those values. This seems to indicate cost of doing the computations must still have been small enough for the MPI interface to be the most important factor.



Lastly, for the largest intervals, I placed all of the results in one graph, and they seem to make very straight lines. The data points are colour coded where the results from running one, two, four and eight processes are codes as green, orange, blue and red respectively. There seems to be some outliers from the lines, which might happen because the computer might not have run all of our processes at the same time. Interestingly the blue dots seem to fall on two different lines, which might be because the amount of processes running at the same time just happened to be about the same several times