Parallel and Distributed Computing - Final Paper

Author: **Abdul Hamid Mangorangca**
Final Revision: **June 8, 2025**

Abstract

I ran a performance test of three `Merge Sort` approaches: (**1) sequential**, **(2) multiprocessing Pool on an Apple M1**, and **(3) MPI (on an Apple M1 and a small server cluster)**, where I used random arrays generated using `numpy`, with sizes from from 10^3 to 10^7. I then measured runtimes and calculated speedups. MPI on the M1 gave the best peak speedup (~6.9x), while the Pool approach topped at ~3.5x.

Introduction

Sorting big lists is common in data work, according to the various readings I did on the Internet. `Merge Sort` is easy to parallelize. We test two ways to run it in parallel:

- ***Pool (shared-memory)*** on an Apple M1
- **MPI *(distributed-memory)*** on both the Apple M1 and a 2-node cluster
  The goal is to see how fast each method runs and how well it scales.

Methods

The code stuff was written in Python. The standard python merge sort provided by the PDF (mentioned below in this section) with NumPy for merging, both of witch has my own modifications to it.

The pool file splits array into P parts, sort in parallel with Python's `multiprocessing.Pool`, then merge.

The mpi file uses `mpi4py`, **where on M1**, all ranks on the same machine, and **where on server** cluster, ranks split across two servers.

I tested P = 2, 4, 8, 16, 32 (Pool); 2,4,8 (MPI on Apple M1); 6, 12 (MPI server Cluster). I then measured sequential time T_1 and parallel time T_P, then speedup $S = T_1/T_P$.

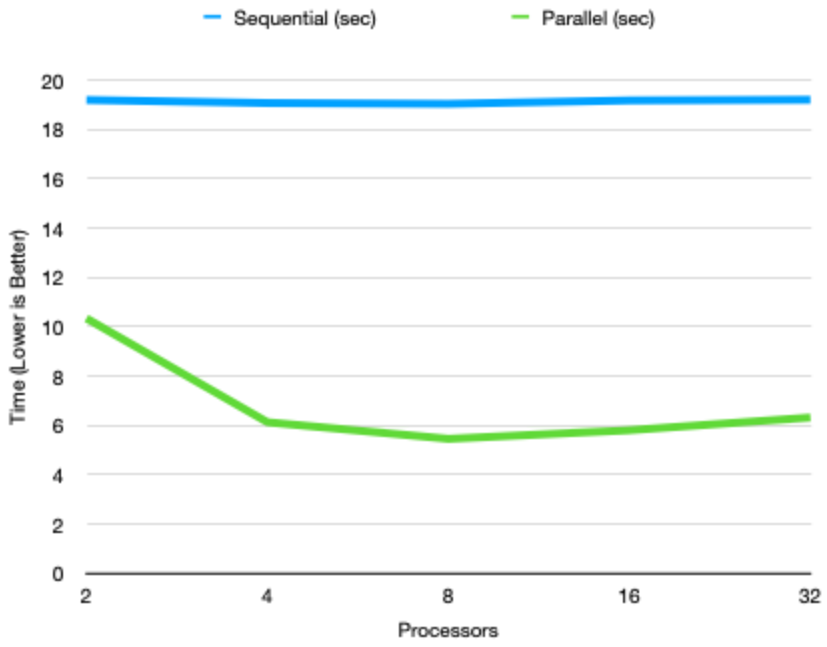
Note that beyond 8 processors is not possible for MPI on Apple M1 as it only has 8 processors, otherwise it will result in errors.

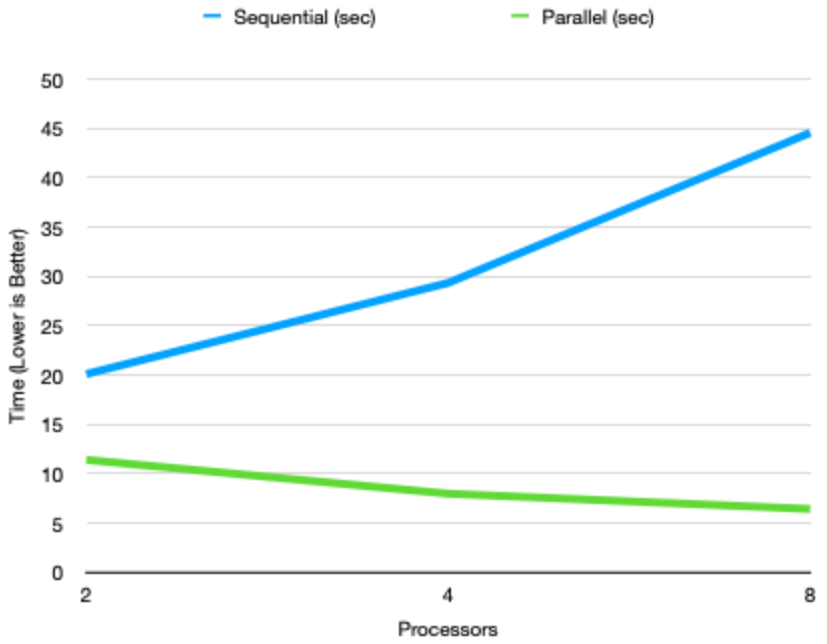
PDF: `Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses (Edited by Prasad, Gupta, Rosenberg, Sussman, Weems)`

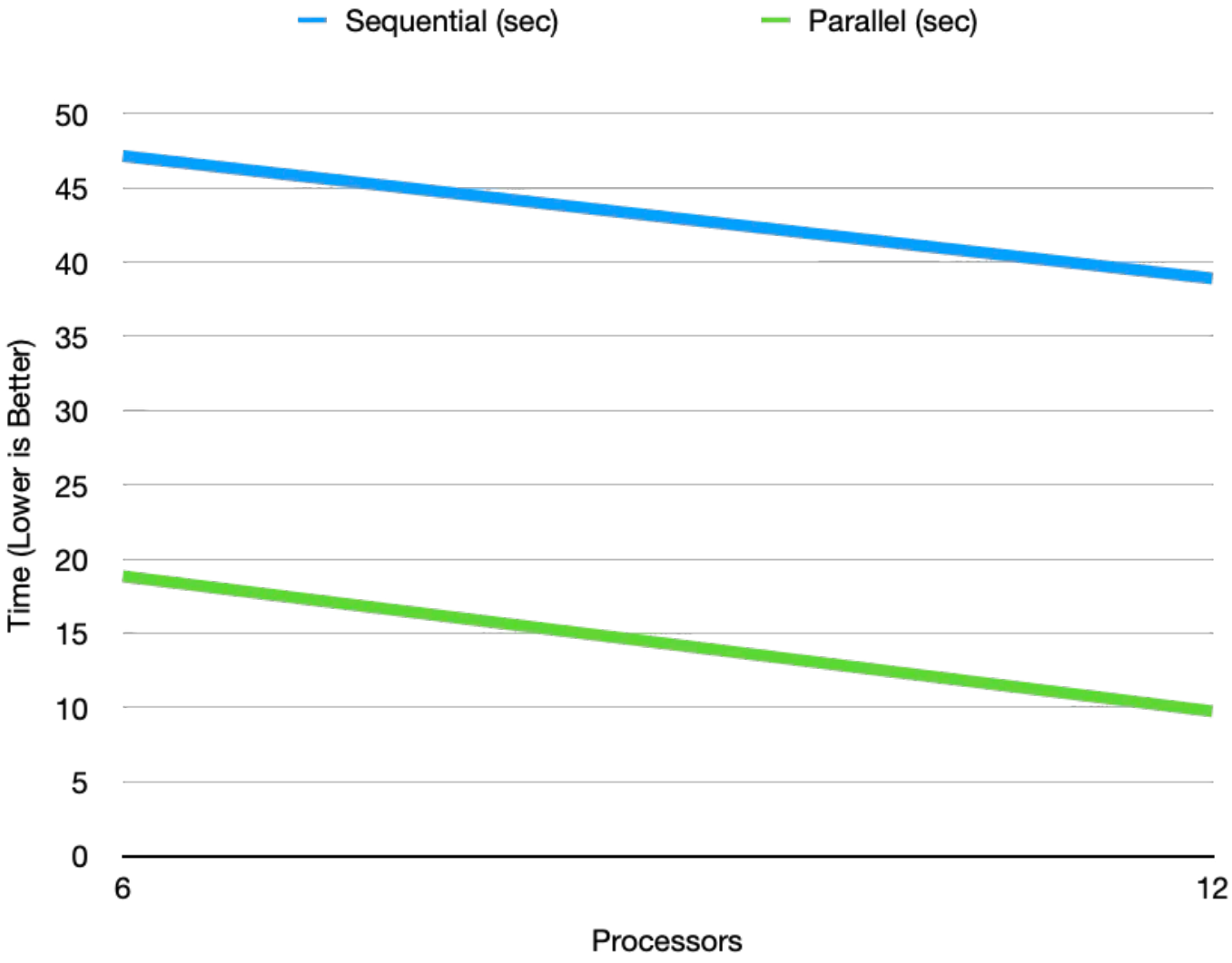
Results

Runtimes for $N = 10^7$

Impl.	Procs	T_1 (s)	T_P (s)
Pool on M1	2	19.18	10.33
	4	19.05	6.12
	8	19.02	5.45
	16	19.16	5.80
	32	19.19	6.31
MPI on M1	2	20.08	11.39
	4	29.32	7.96
	8	44.52	6.41
MPI on Cluster	6	47.15	18.84
	12	38.89	9.73


Sequential (sec)
Parallel (sec)
Time (Lower is Better)
20
18
16
14
12
10
8
6
4
2
0
2
4
8
16
32
Processors

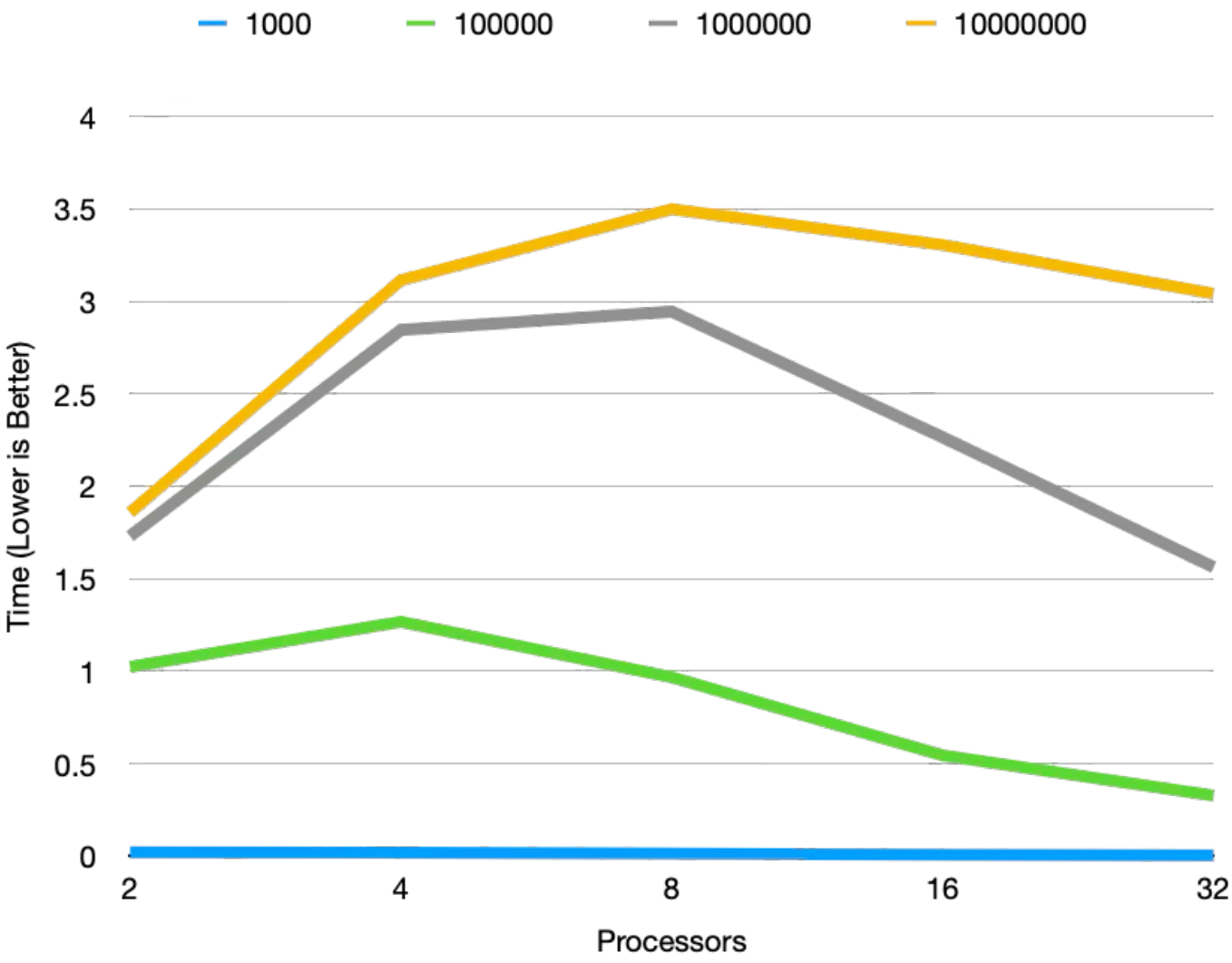

Sequential (sec)
Parallel (sec)
Time (Lower is Better)
50
45
40
35
30
25
20
15
10
5
0
2
4
8
Processors



Speedups

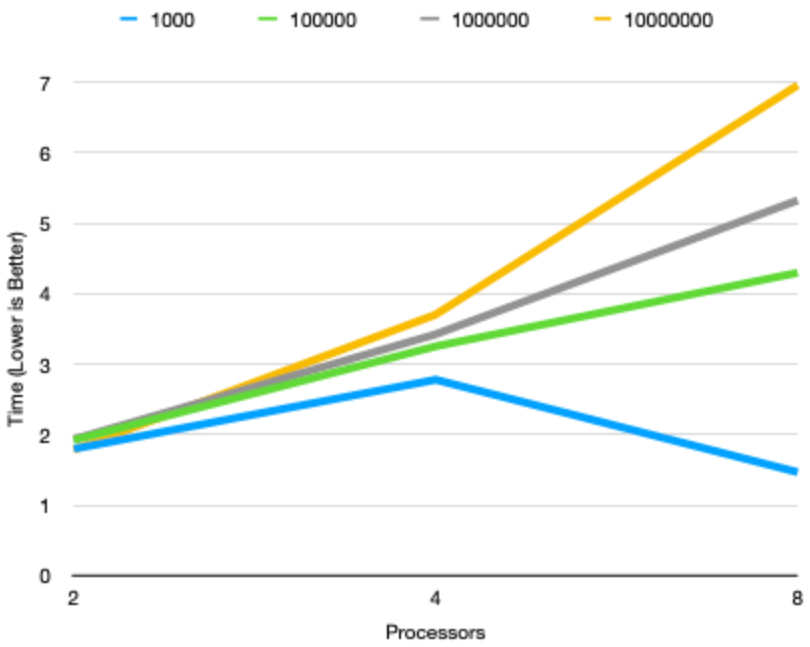
Pool on M1

P	10^3	10^5	10^6	10^7
2	0.020	1.023	1.730	1.857
4	0.018	1.267	2.845	3.113
8	0.013	0.967	2.944	3.498
16	0.007	0.544	2.264	3.304
32	0.004	0.327	1.562	3.040



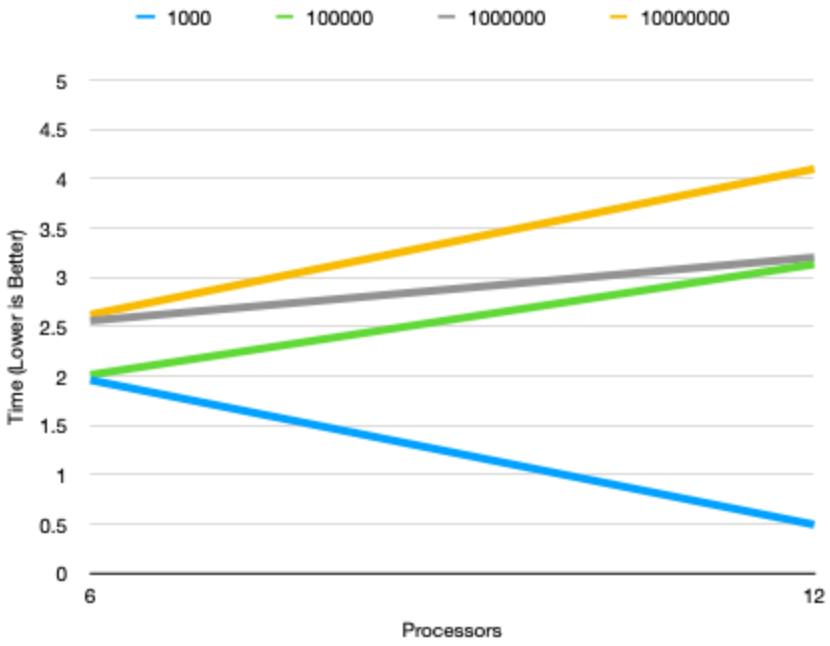
MPI on M1

P	10^3	10^5	10^6	10^7
2	1.792	1.918	1.928	1.780
4	2.774	3.247	3.418	3.697
8	1.463	4.290	5.317	**6.949**



MPI on Cluster

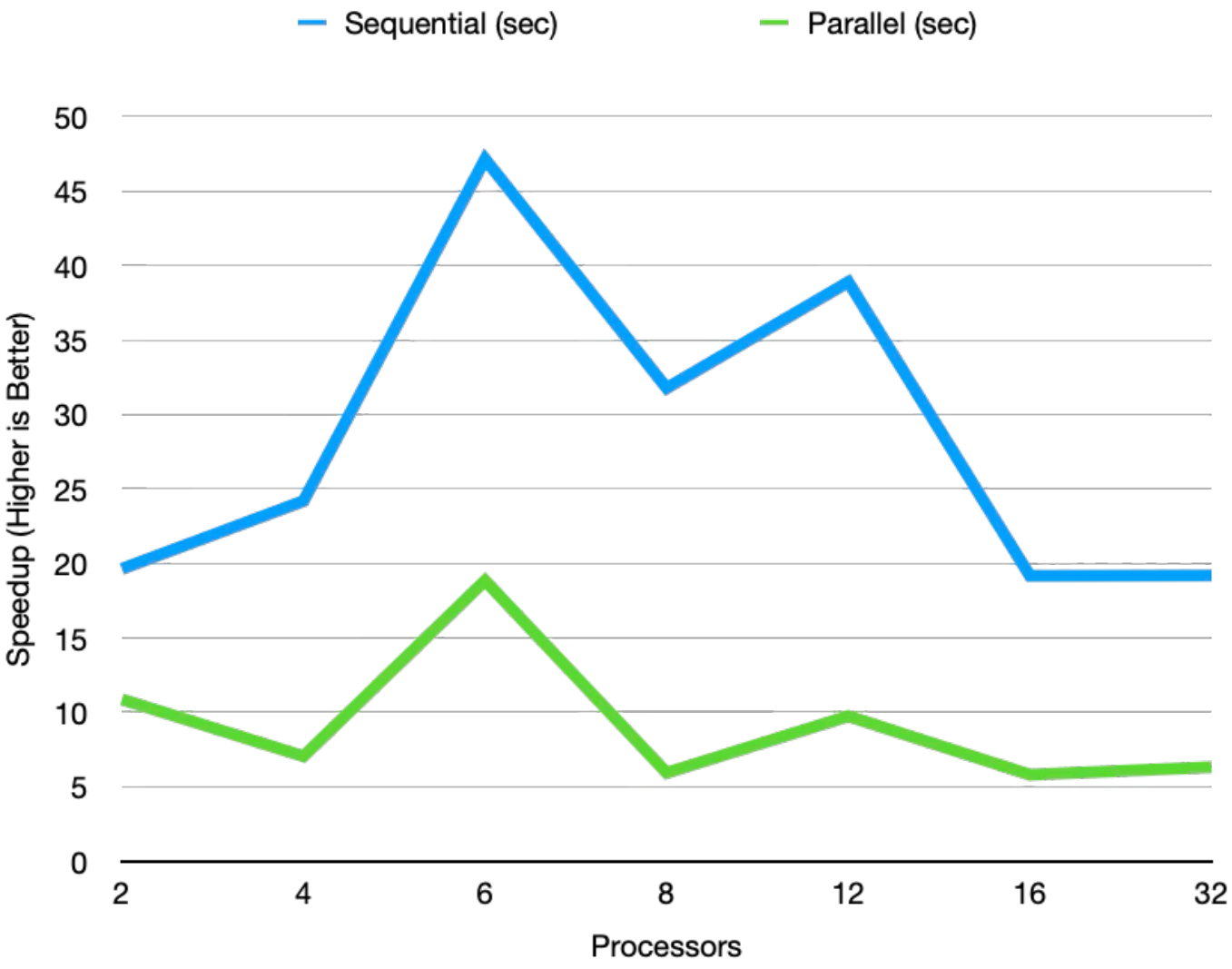
P	10^3	10^5	10^6	10^7
6	1.959	2.011	2.558	2.620
12	0.493	3.130	3.200	4.097



For fun, I tried to combine everything.

Runtime

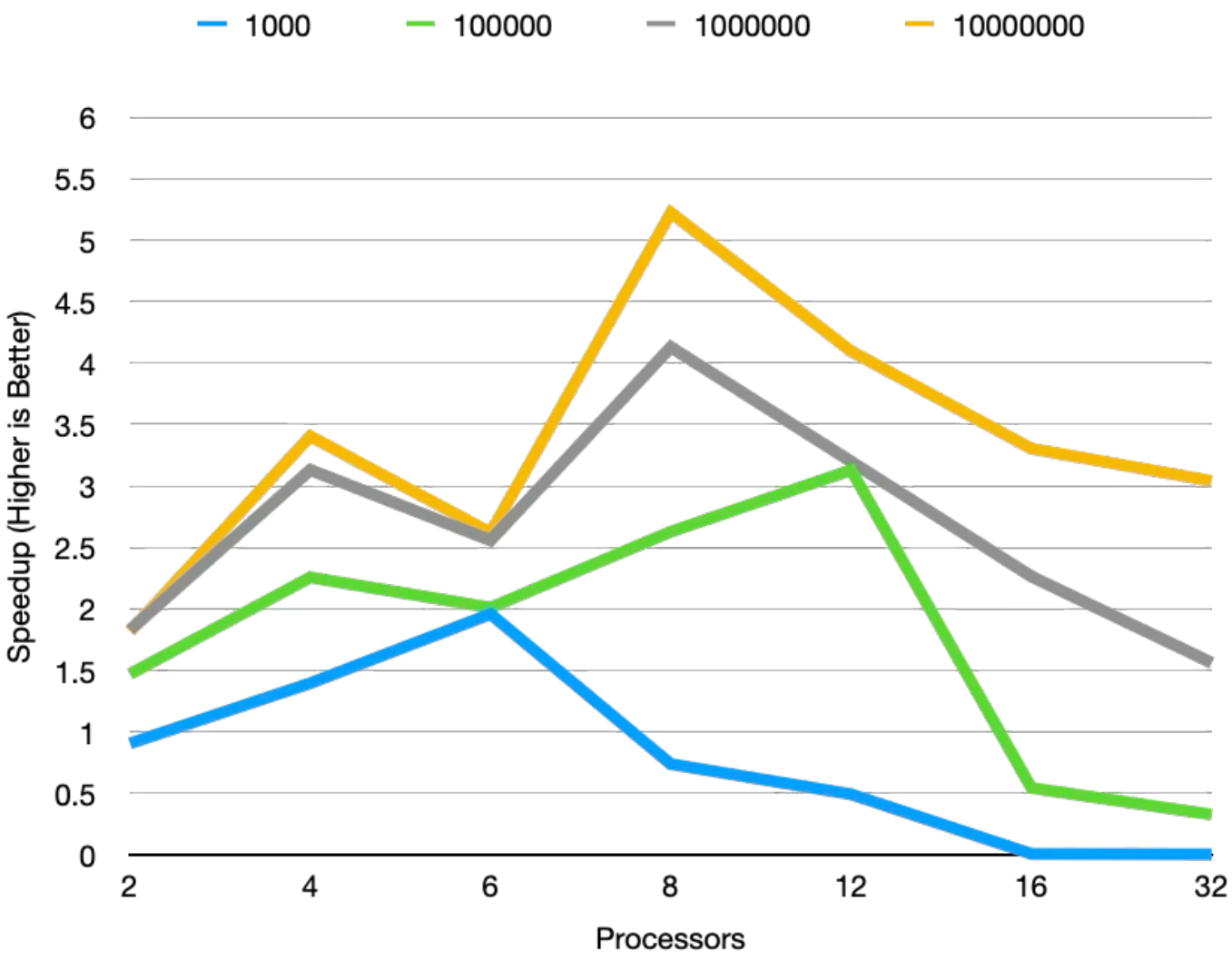
Processes	Sequential (sec)	Parallel (sec)
2	19.6263688802719	10.86131095886230000
4	24.1833556890488	7.04071180025737000
6	47.1475319862366	18.83759641647340000
8	31.7704610029856	5.92974758148194000
12	38.893313964208	9.73377132415771000
16	19.1572391986847	5.79862300554911000
32	19.1857340335846	6.31074174245199000



Speedups

Processes	1000	100000	1000000	10000000
2	0.906044571314326	1.47029988571308	1.82922615484337	1.8183814

Processes	1000	100000	1000000	10000000
4	1.39613402935936	2.2567802850276	3.13130559625513	3.4052587
6	1.95876270216829	2.01071724417144	2.55800930778038	2.6199139
8	0.737964230315567	2.62818759581881	4.13029567041373	5.2233862
12	0.492703566436405	3.12965456803108	3.1995530697803	4.0968080
16	0.00735008100152974	0.544218168659359	2.26446470854849	3.3039398
32	0.00383214378195467	0.326625338113886	1.56216298678984	3.0403454



Discussion

Pool

Slightly easier to code compared to MPI. Best at 8 processes for large N. Small N has too much overhead.

MPI on M1

Peaks at 8 processes (as that's the maximum processes the Apple M1 chip has) with ~6.9x speedup for 10^7. Good in-memory communication compared to Windows as I have compared to my friends.

MPI Cluster

Extra network delays limit speedup (~4.10× at 12 processes).

Limitations

The results maybe due to the server cluster being busy or network delays, where the server cluster being the two computer servers, server02 operated by Mangorangca and Badron, and server01 operated by Abdulmanan and Bucay.

Conclusion

My experiments show some trade-offs between ease of implementation, available hardware, and performance.

- **Shared-memory (Pool on M1)** is the simplest to code and deploy. It provides reliable 3–3.5× speedups for large workloads with 4–8 worker processes, but diminishing returns beyond that due to task-management and merge overhead.
- I did my research outside of this experiment, and my readings aligns with my experiment in that, **Distributed-memory MPI on a single M1** fully leverages the chip's fast, unified memory fabric. It delivered the highest acceleration, nearly 7× for N=10^7 with 8 ranks, possible due to low-latency communications (everything is on one chip, unlike AMD and Intel chips).
- **MPI across servers** gains access to more CPU cores but incurs network latency, capping speedups at around 4× with 12 ranks.