Assignment 4 report

1. A detailed discussion of the design and rationale behind your approach to parallelizing the algorithm. Specifically try to address the following questions:

High-level design

- I implemented a software-ring to pass passages across all the processes. The messages are the latest routes updated by each processes.

Task scheduling and load-balancing

- To parallelizing the algorithm, the first problem is about how many work each process will handle. In my code, **each process handles some different wires**.
- Since the **workload for each task is not changing and can be calculated before the computation starts**, I use **static assignment**.
- To make the processes load-balanced, I calculated the **computation cost (how many `for-loop` cycles)** before assigning the same amount of work to each process evenly. It works well in 1,4,16 cores.
- However, when having 64 or 128 cores, although all the processor has the same amount of work, load is not fully balanced. **When using many cores, process creation and management may needs more time than computation.**
- I **`random_shuffled` all the wires** before task assignment and assign each process same amount of wires. This time, workload is much balanced among all the 64/128 cores. However, it seems that the input data itself has some spatial locality. **Although random_shuffle can help load-balancing, it will make the quality worse more than the performance increase**.

Version 1: Software ring(fully asynchronized), message: routes

- A software ring is like:

> Proc 0 (isend)->(irecv) proc 1
>
> Proc 1 (isend)->(irecv) proc 2
>
> Proc 2 (isend)->(irecv) proc 3
>
> ……

- Message content is the all the routes. Every process update its routes in the message body.
- Using `Isend` and `Irecv`. Both sending and receiving are asynchronized.
- Because there are no `wait` or any other synchronization, both performance and quality are really good if there are not many processes working toghether. (Have better performance then the reference and high quality(no decrease) when using 1,4,16 cores).

- However, when using 64 or 128 processes, there too less messages sent and received by each processes. For example, when using 128 processes, every process have too less computation work. Every process will ended with 0.5s and send/receive less than 5 messages. The quality would be really bad.
- The performance is too good(hard 128 cores within 1s) but the quality is too bad(useless result). So I think this is not a good way. There should be some synchronization within the

Version 2: Software ring(sync) , message: routes

- Since the quality in 64 and 128 process cases are really bad, I think some synchonization is needed.
- So I make the receiving action synchronized. Using `MPI_Isend` with `MPI_Recv` now. This time, **quality is much better** but the performance is slower.
- Using `perf record`, I can know that most of the time is used in waiting the message(MPI_Recv).

Version 3: Optimization: parallelize `MPI_recv` - using many messages

- In Version2, there is only one message in the ring. Since I the message content is all the routes, I can **make several messages passing parallelly in the ring**. This will **reduce the waiting time** for each processes.
- I used 8 messages passing in parallel when using 128 processes. The program **became 2x faster and quality remain the same.**

Version 4: Optimization: reduce message size(change data type)

- To make the message shorter and faster to pass, I think message content should be minimized.
- Max cost should be lower than 255. I changed cost_t from `int(4B)` to `uint8(1B)` and point coordinates's datatype from `int(4B)` to `int16_t`.
- This will **reduce the communication time**.
- What's more, since `perf record` said 90% of the computation(not waiting) is about the matrix `cost_t *costs`, the program will also benefit from less memory access, since the cacheline can now fetch 4x costs items now during continuous memory access. It is more **cache-friendly**.
- **Increased the performance by 50%**.

Version 5: Optimization: less computation - transposed cost matrix

- After changing cost_t datatype, I am thinking about whether there are methods to do better in cache-friendly. Now each cacheline can contain 4x cost_t items. It is great when accessing items one by one. That is to say, **when accessing `cost_t *costs`, horizontal lines are friendly but vertical lines are not**.
- So I made a transposed copy of the costs matrix. Walk horizontal lines with `costs` and vertical lines with `costs_Transpose` **inprove the performance by 30%**.

Version 6: Software ring(sync) , message: costs matrix

- Using `perf record`, I noticed that the bottleneck in version 3(4,5) is that everying time a process receives a new message(all the routes), it need to uses the routes to re-calculate the new costs matrix(**which will cause its next process waiting for `MPI_recv`**). This will uses 85% of the time. I think most of the calculation is redundant -- when using 128 cores, every core need to repeat calculating the costs for other 127 cores again and again.
- So passing costs matrix as the message may help. Costs matrix is now 1024*1024 byte = 1MB. All the routes are about 500 KB. Costs matrix is more data but we can avoid many calculations.
- However, after I implemented this version I noticed that the it is **slower** than the version 5. I think one reason is that the **message size is 2x larger** and there is **only one message in the ring**.

Answers to questions

- What approaches have you taken to parallelize the algorithm? What efforts did you make to profile your code and identify bottlenecks, and how did this data affect your designs?

  > See above.

- What information is communicated between MPI processes, and how does this affect performance?

  > See above.

- How does communication scale with the number of processes?

  > In my implementation using a software-ring, communication does not scale with the number of processes. **But more and more communication is needed to maintain the same quality when using more cores, which will slow down the program.** I chose different sweet spots with different number of processes.

- At high process counts, do you observe a drop-off in performance and/or solution quality? If so, (and you may not) why do you think this might be the case?

  > Yes. My performance and/or(depending on trade-off) quality will drop-off. I `perf record` some process and noticed that most of the time is used for **waiting for the message(MPI_Recv)**. So every process should be really quick when processing each wires. **Communication itself takes 15% of the time of waiting(implicit synchronization).** I should find some way to reduce the waiting time.

- Why do you think your code is unable to achieve perfect speedup? (Is it workload imbalance? communication? synchronization? data movement? etc?)

> My code is unable to achieve perfect speedup is because of the synchronization.
>
> I can make the workload really balanced by randomized the data, which will not help.
>
> I optimized the communication messages and it takes 15% of the time of waiting(implicit synchronization). So I think the problem is synchronization. I also tried the fully-async message ring, which is really fast but cannot work in 64/128 cores.(See above)

Easy (different sweet spots for different num of proc)

Processors	Computation time	Max Cost	Sum of Squares Cost	speedup
1	7.2	19	98938390	1
4	4.3	25	99777324	1.67441860465116
16	4.4	29	100586936	1.63636363636364
64	2.3	36	109625760	3.1304347826087
128	1.2	54	107042972	6

Medium (different sweet spots)

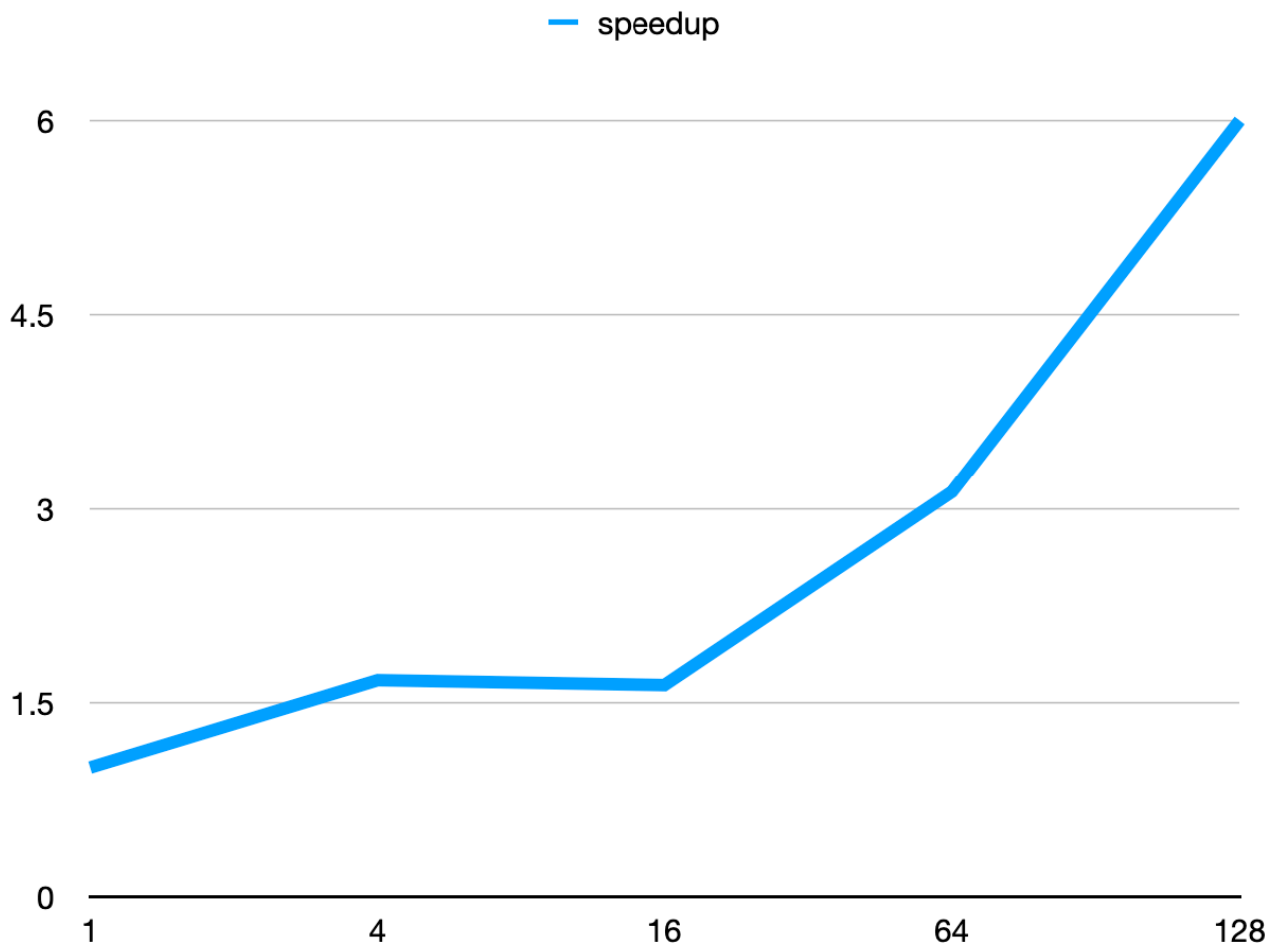
Processors	Computation time	Max Cost	Sum of Squares Cost	speedup
1	27	37	488474700	1
4	14	38	489733232	1.92857142857143
16	15	41	490648772	1.8
64	7.2	52	502848560	3.75
128	4.5	63	529662388	6

Hard (different sweet spots)

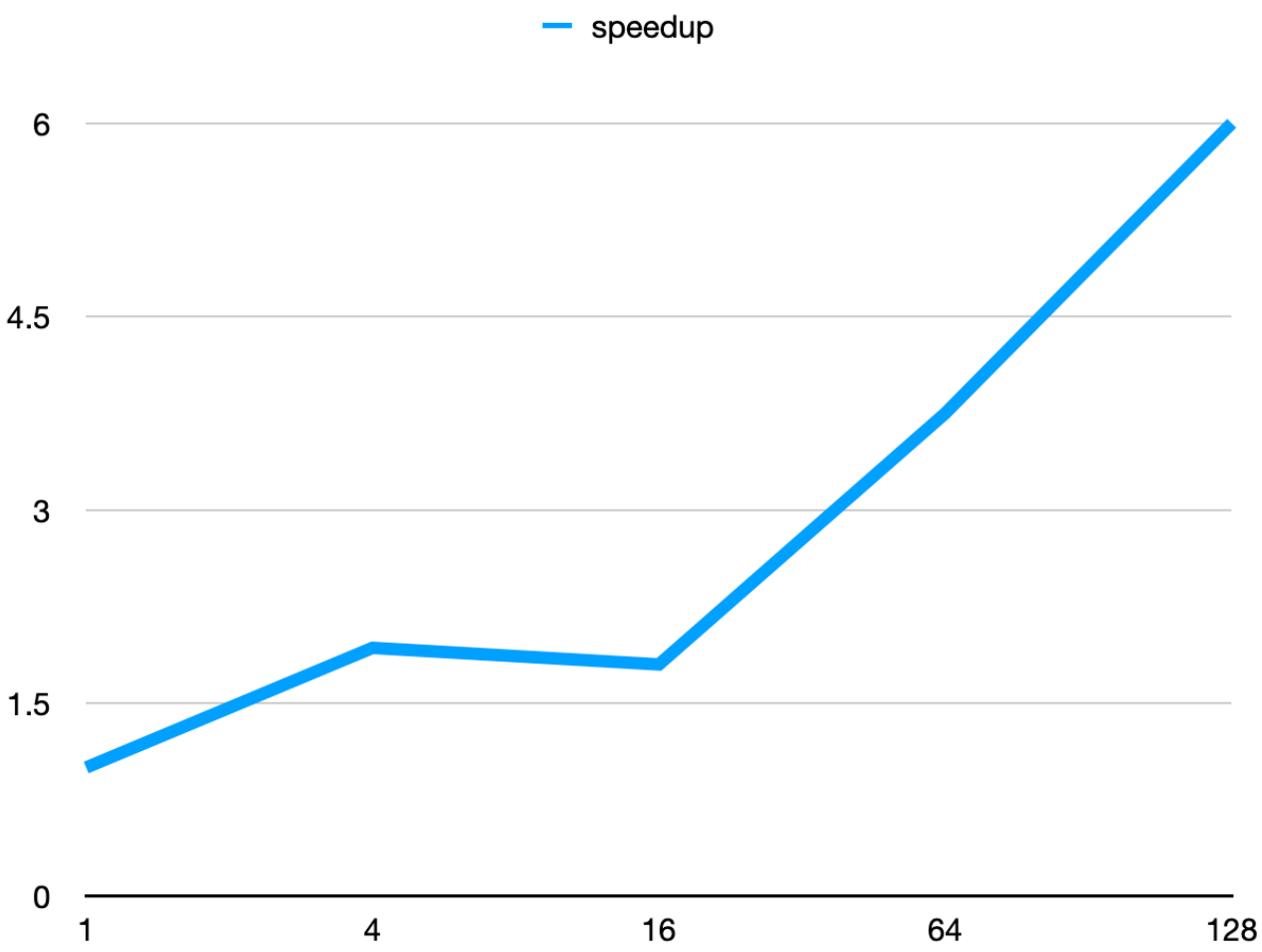
Processors	Computation time	Max Cost	Sum of Squares Cost	speedup
1	30	41	625679889	1
4	18	44	627526921	1.66666666666667
16	17	47	628650409	1.76470588235294
64	9.2	65	653760129	3.26086956521739
128	7.3	64	648128111	4.10958904109589

2. A plot of the computation speedup (metric 1) vs. the number of processors sampled at 1, 4, 16, 64, and 128 processors.

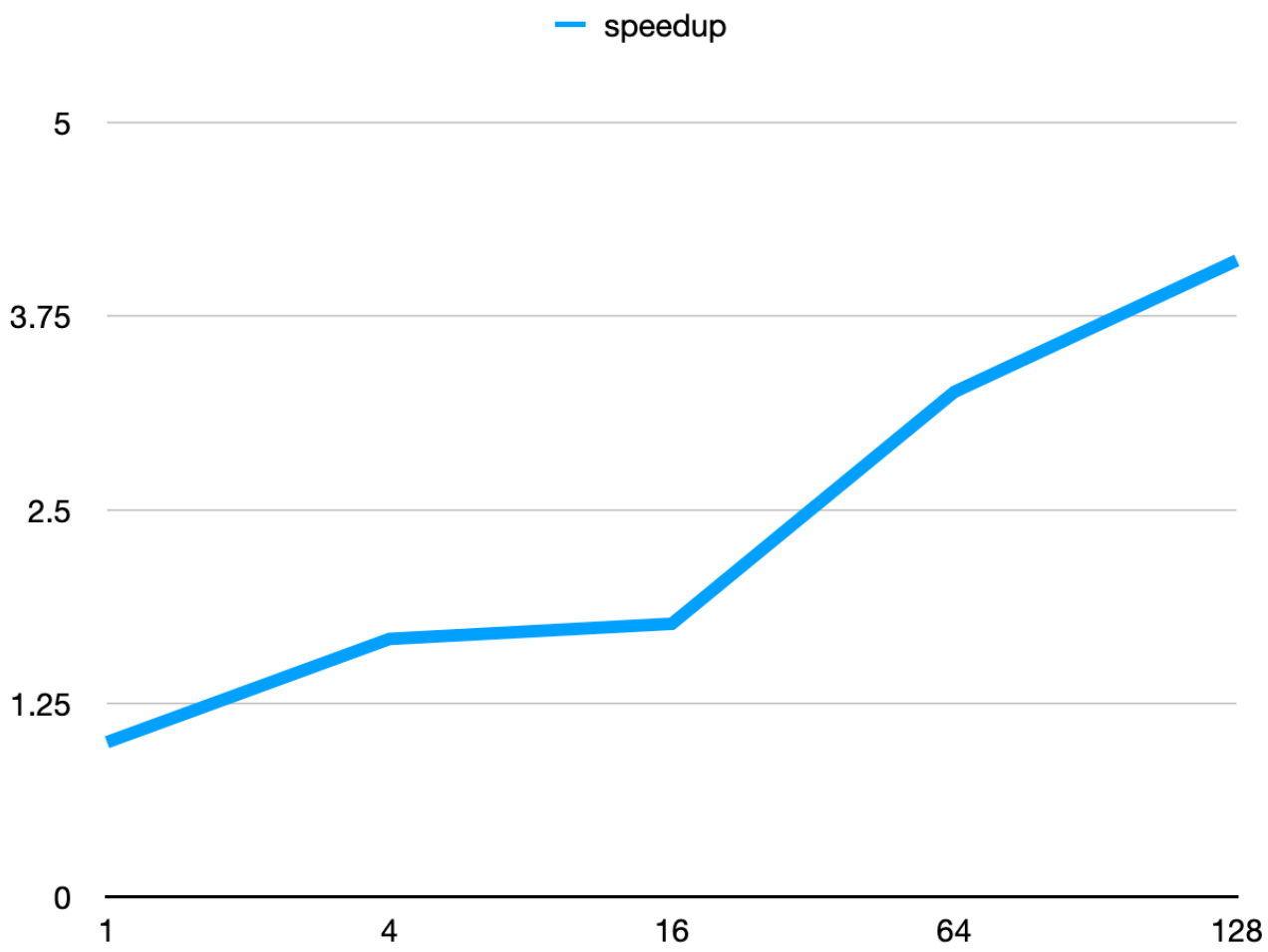
Easy



Medium



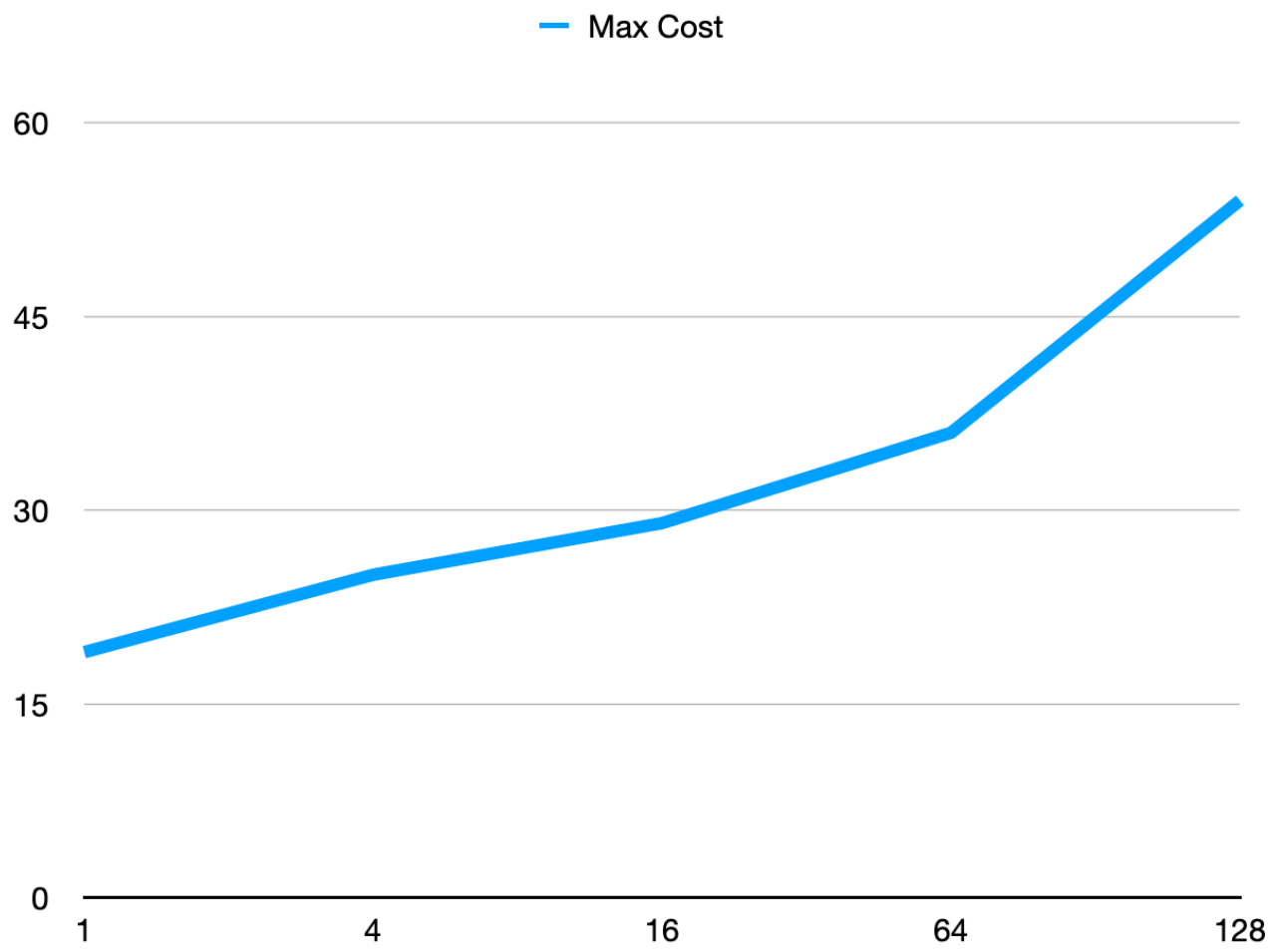
Hard



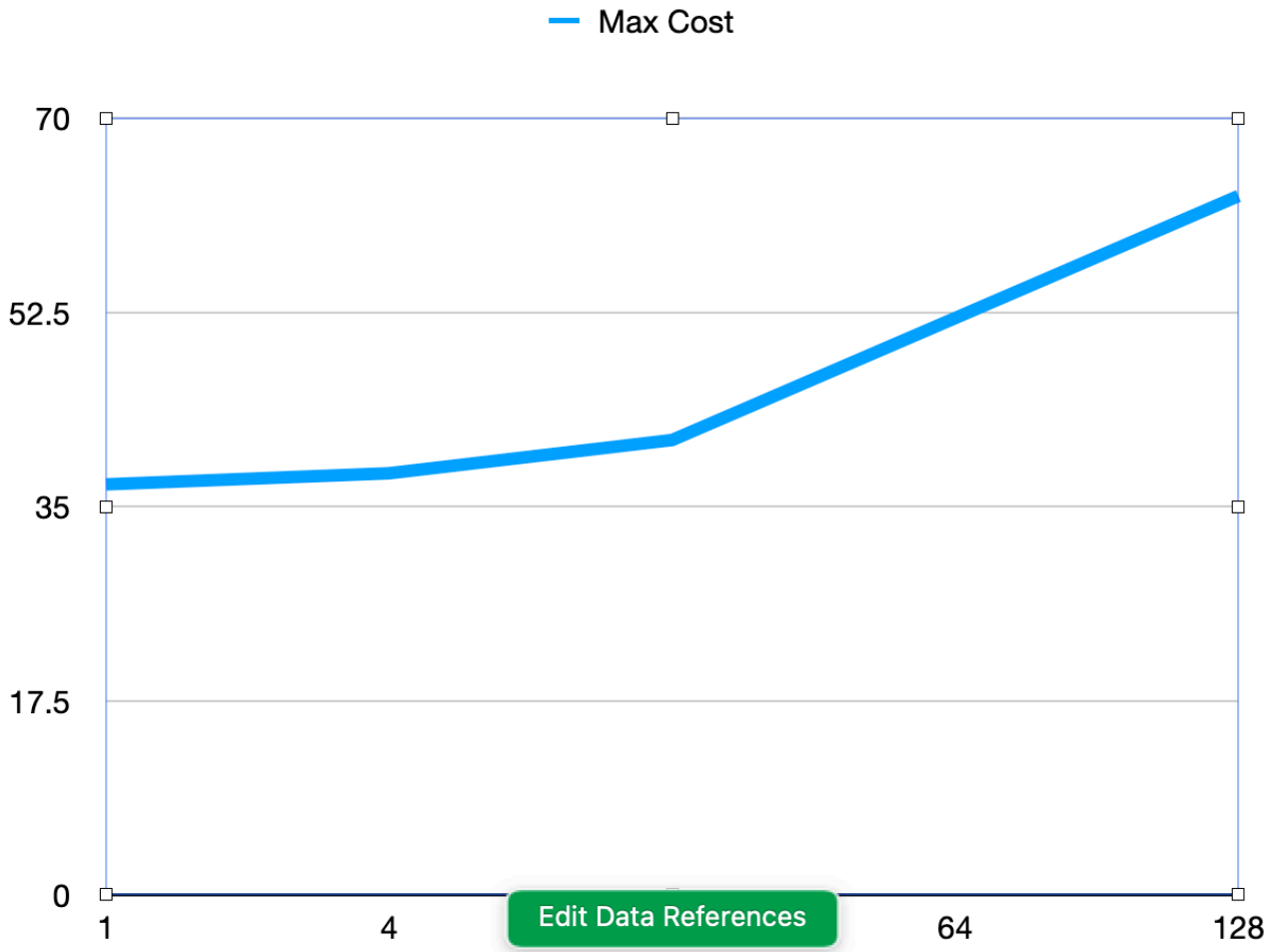
3. A plot of the max cost (metric 2) vs. the number of processors. Please explain your results and how the time for P processors. it's affected by your design.

> Quality is worse when I use more cores. To solve the problem, I tried to use different sweet spots in different processes. About 10x messages was sent when using 128 cores compared to using 1 core. This will take many more time, which results in the speedup is not good. I can make the max_cost stay the same when using 128 cores. However, this will take lots of time and make 128 core computation time more then 1 core.

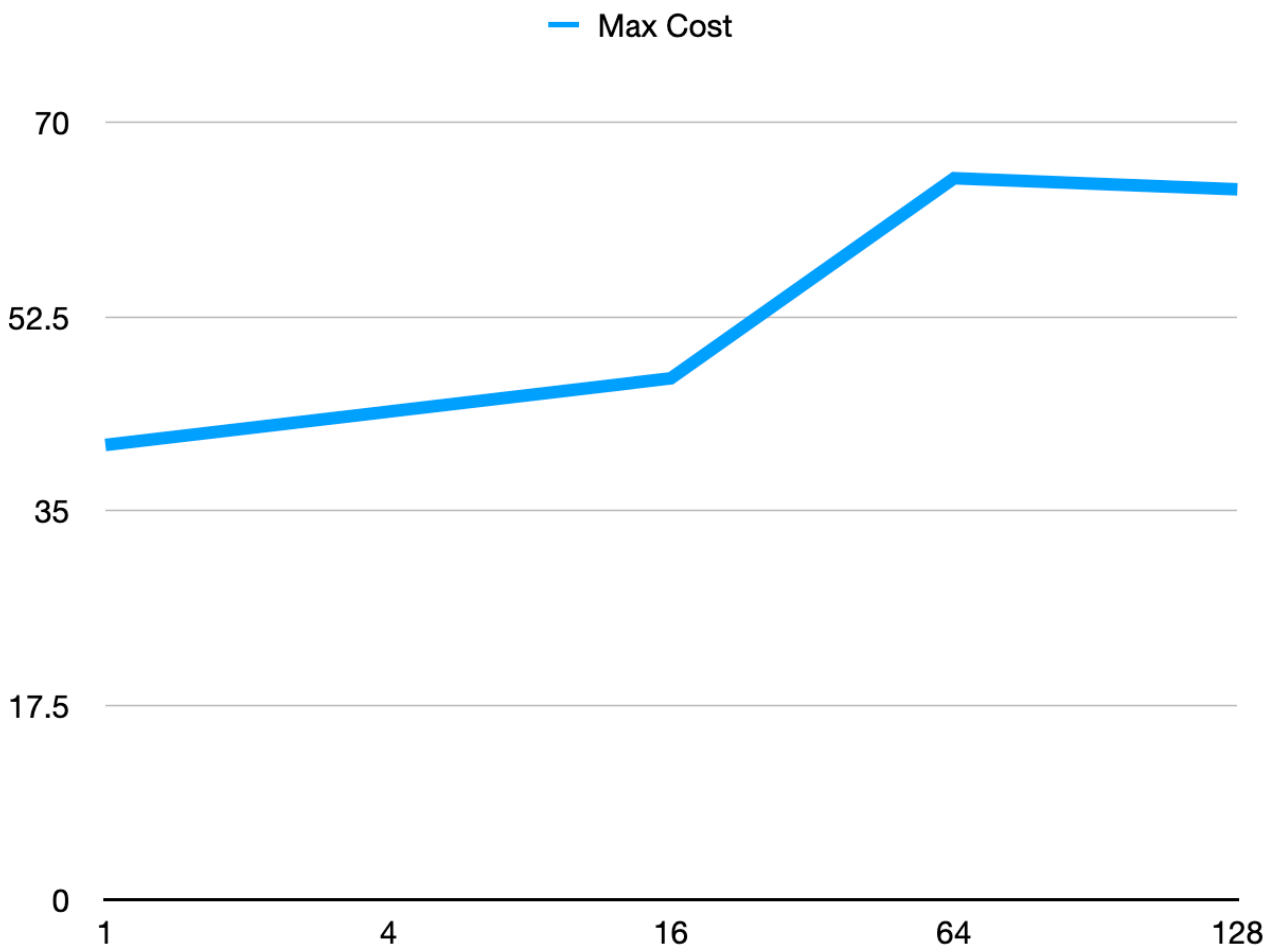
Easy



Medium



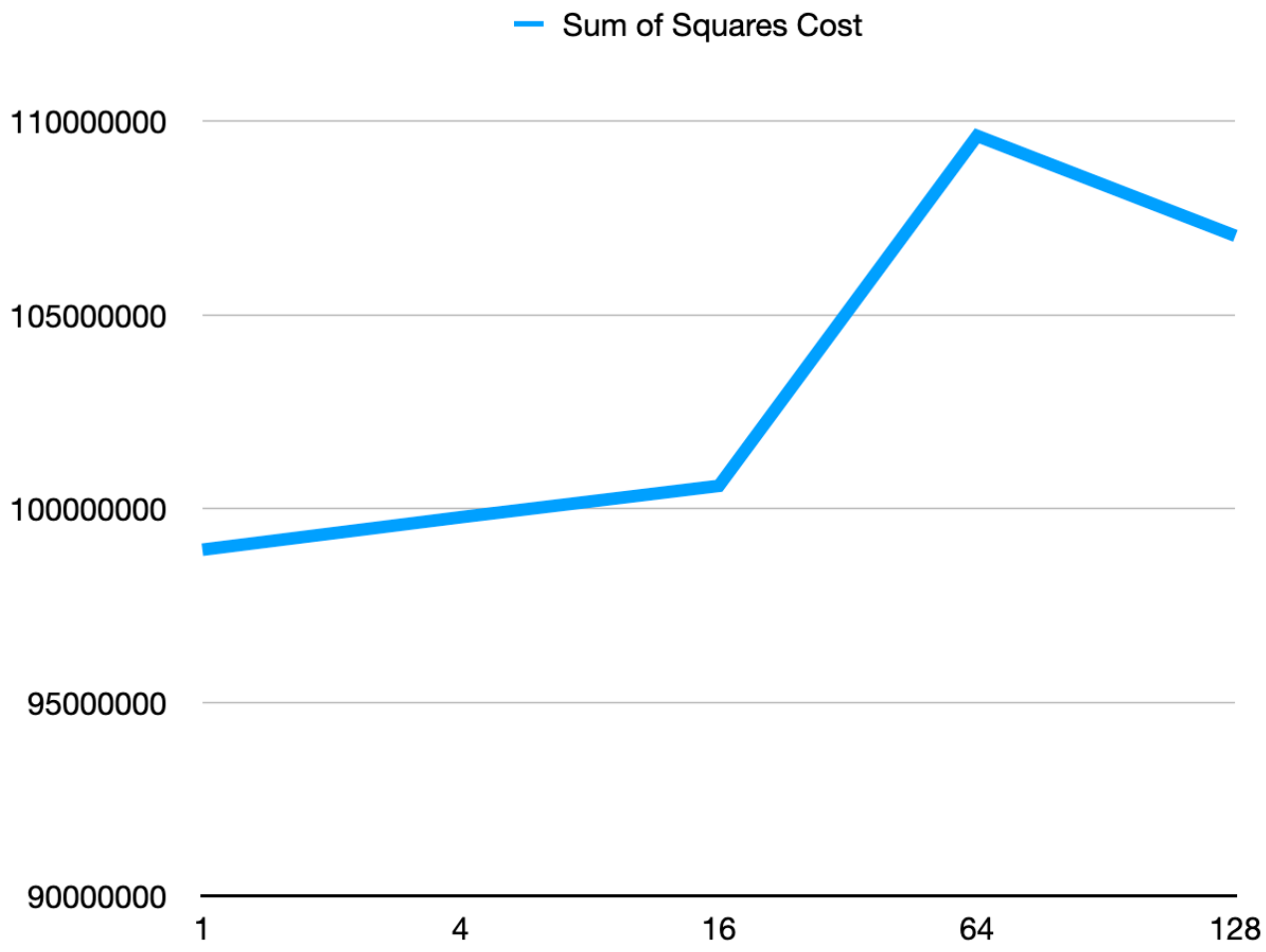
Hard



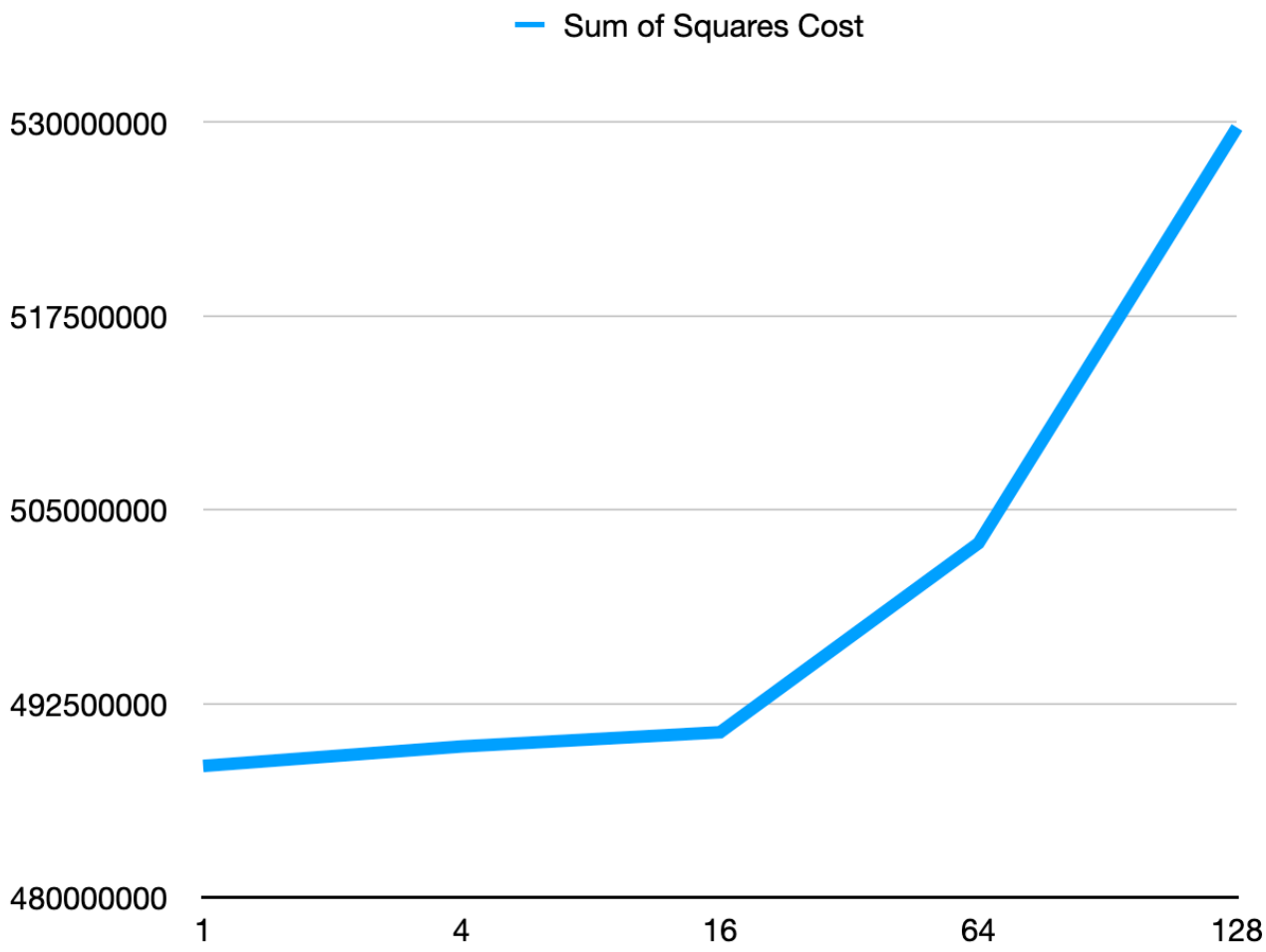
4. A plot of the sum of squares cost metric (metric 3) vs. the number of processors. Please explain your results and how it's affected by your design

> Similar to question 3. Quality is worse when I use more cores. To solve the problem, I tried to use different sweet spots in different processes. About 10x messages was sent when using 128 cores compared to using 1 core. I can make the sum of squares cost stay the same when using 128 cores. However, this will take lots of time and make 128 core computation time more then 1 core.

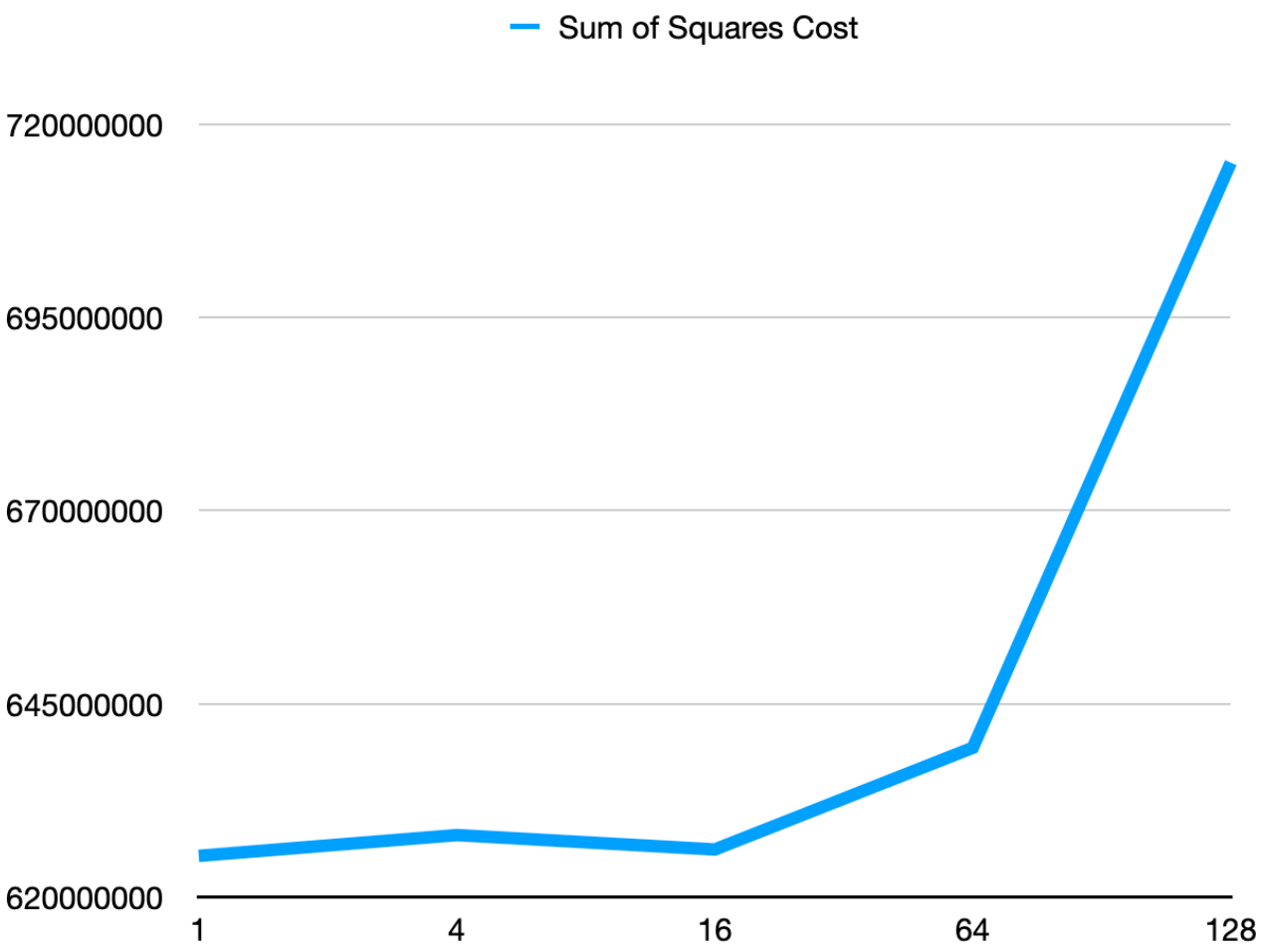
Easy


Sum of Squares Cost
110000000
105000000
100000000
95000000
90000000
1
4
16
64
128

Medium


Sum of Squares Cost
530000000
517500000
505000000
492500000
480000000
1
4
16
64
128

Hard


Sum of Squares Cost
720000000
695000000
670000000
645000000
620000000
1
4
16
64
128