Homework 2-2: Parallelizing a Particle Simulation with MPI

Andrew Chen, Xuan Jiang, and Brian Park

UC Berkeley, Computer Science 267

March 2022

1 Introduction

We used variety of techniques to optimize the problem of parallelizing n-body particle simulation on the Cori KNL node. We used the $O(n)$ solution from our previous homework to accelerate it with MPI (Message Passing Interface) [1].

2 MPI

2.1 Data Structures

Used the same binning idea that we derived from homework 2-1. To be more specific:

- `typedef vector<particle_t*> bin_t`
  This holds pointers to all of the particles that lie within a bin.

- `bin_t* bins`
  This holds the vector of bins stored in row-major order.

2.2 Design Choice

For MPI, it was much harder to implement due to explicit communication. We mainly took use of communication primitives such as `MPI_Isend` and `MPI_Irecv`. We only used this to explicitly send or receive neighboring particles when computing `apply_force()`. Note that we only focused on moving between rows in a 1D layout. We found moving bins by a grid, in a 2D layout is much harder to implement. Given more time, we would've tackled the 2D layout. For rebinning, we used `MPI_Allgather` and `MPI_Allgatherv`. We didn't want to deal with implementing additional data structure such as creating a ghost buffer or halo zone. Rather, we kept it simple and implemented All-to-All communication so that we can just rebin. We would need to synchronize before moving to the next step anyways, so this was the safest approach as well in order to achieve correctness. This is actually the same approach we took in OpenMP solution, but due to communication bottleneck in a distributed system, implementing explicit send and receive primitives may have been the more appropriate and tedious programming model. Due to the time it takes to debug and the performance we would achieve, we just stuck to this implementation. For `gather_for_save()`, we implemented `MPI_Gather` and `MPI_Gatherv`. The reason why we used two gather version is because the sizes to collect are not uniform, so first we must calculate the size so that we can concatenate them properly in the second call to gather that uses the `MPI_Gatherv` version, indicating variable length buffer. This is a many-to-one collective communication primitive and was necessary to communicate everything to the driver node to check for correctness when we save to output. The most difficult part of implementing gather was getting displacement right and how to collectively communicate data from different processes.

We make sure that particles are stored in bins. The number of bins is fixed, which is why we use an array of bins. The bins themselves change in size each simulation as they move around, which is why we use `std::vector`.

3 Experimental Results

We evaluate the results of n-body particle simulations with random seed set to 42 to make results reproducible.

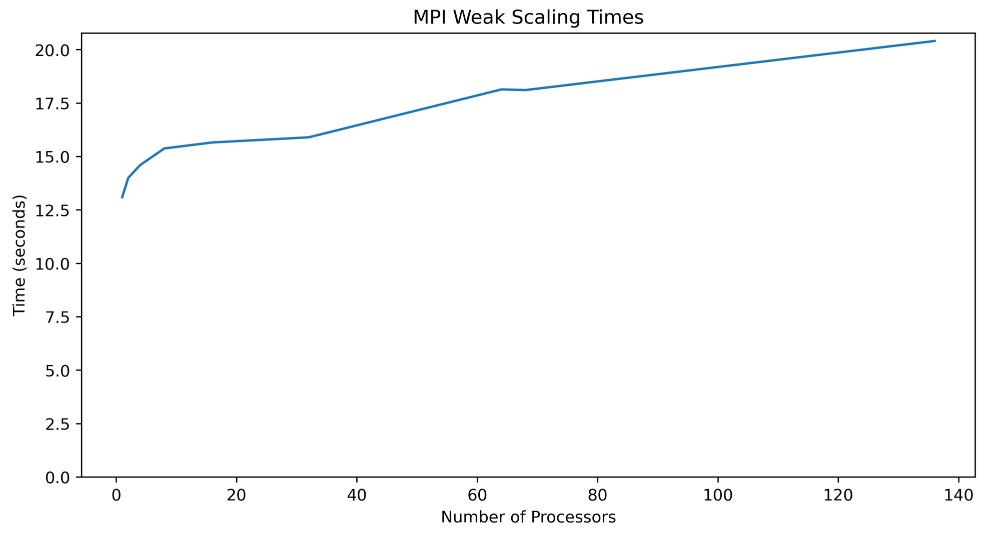


Figure 1: MPI Weak Scaling

Figure 1 shows the weak scaling of MPI as it scales with 10000 particles per processor. We extend this to two nodes when running 136 processors (2 nodes $\times$ 68 processors).

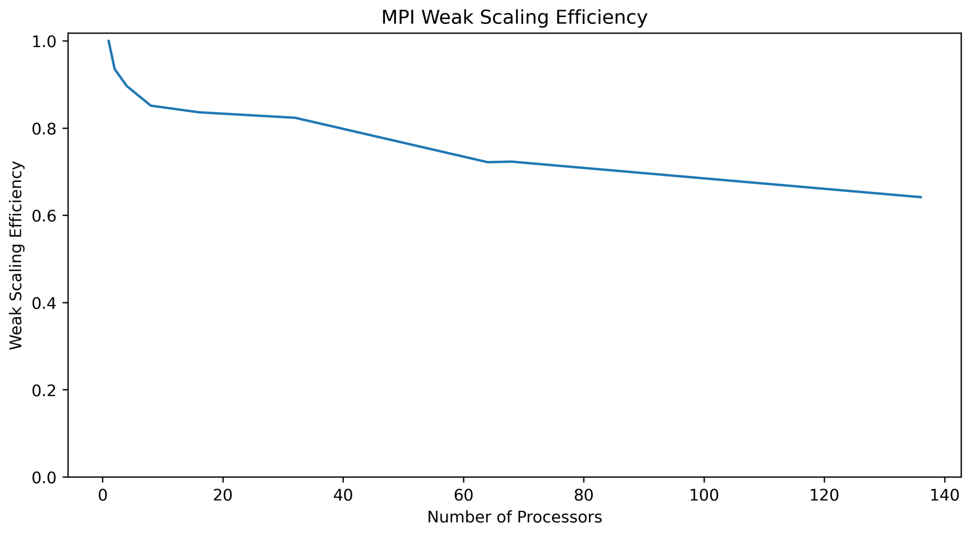


Figure 2: MPI Weak Scaling Efficiency

Figure 2 shows the weak scaling efficiency of MPI. We see that it somewhat sustains performance. When jumping to two processors, it has 93.6% efficiency and then gradually drops to 64.2% efficiency when scaled up to 136 processors.

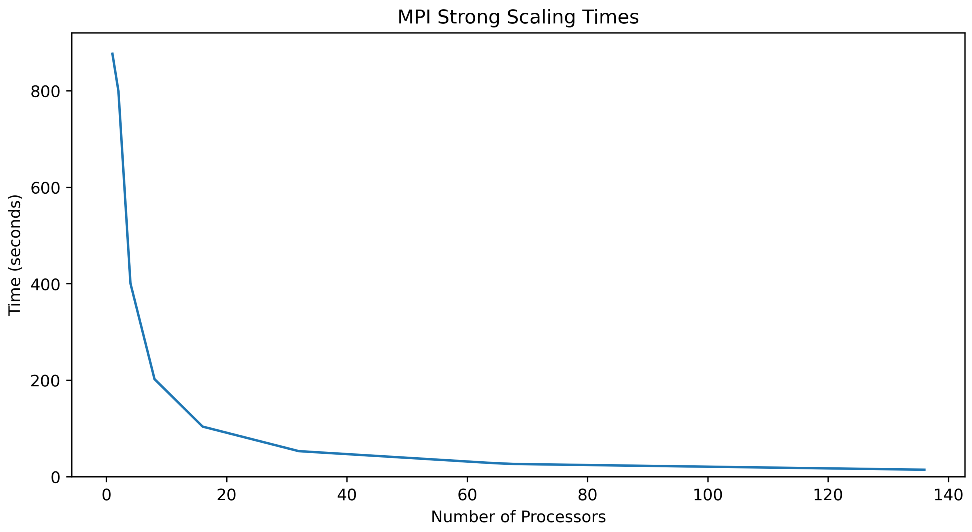


Figure 3: MPI Strong Scaling

Figure 3 shows the strong scaling of our simulation with MPI for 1 million particles. We see that it decreases as expected. It reaches 14.8 seconds when all of the 136 cores are utilized.

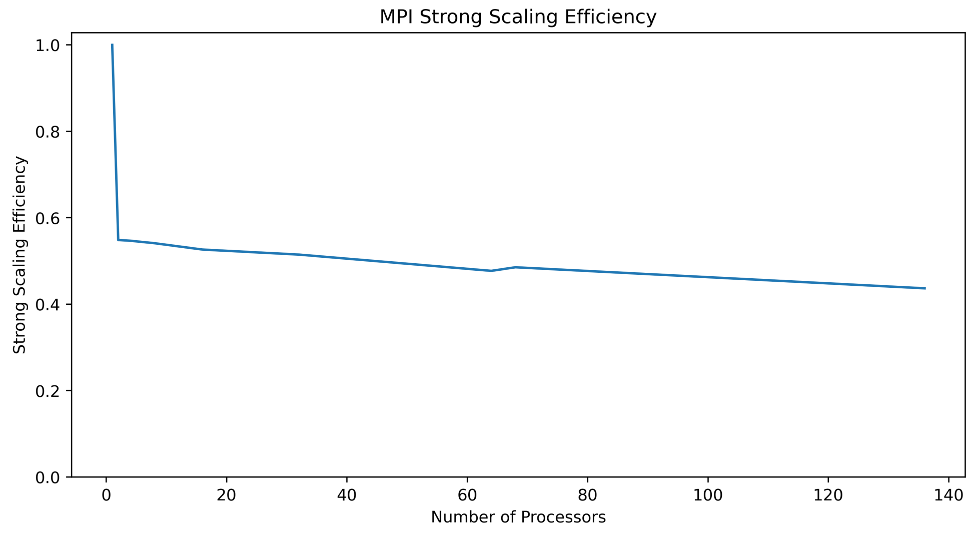


Figure 4: MPI Strong Scaling Efficiency

To see the results better, Figure 4 shows the strong scaling efficiency. There is a large dropoff in efficiency to 2 processors, giving 54.8% efficiency. It sustains performance down to 43.7% efficiency when all 136 cores are used. This is pretty low effiency, but considering that it is sustained, we deemed it scalable. This huge dropoff must have been due to the high communication overhead added on top of our synchronization overhead. We see later in our performance breakdown, that MPI communication overhead is indeed the main cause.

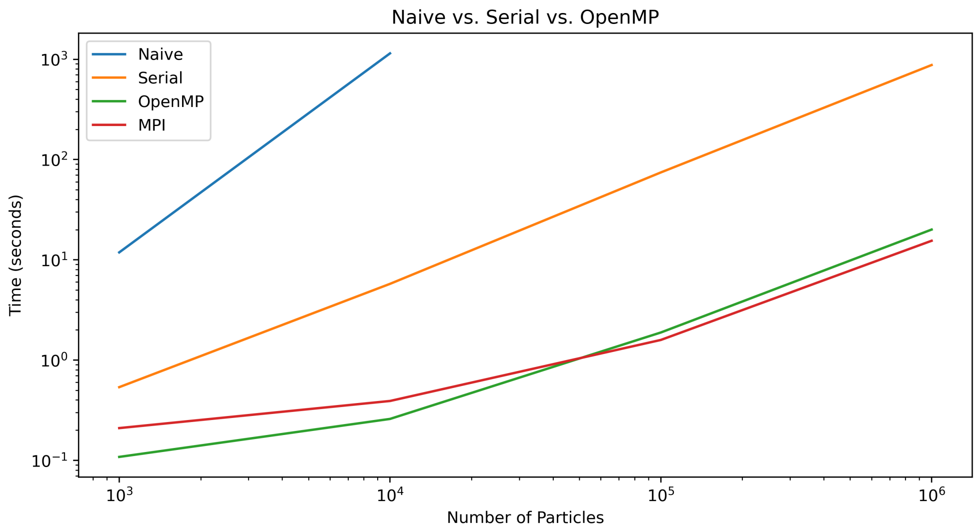
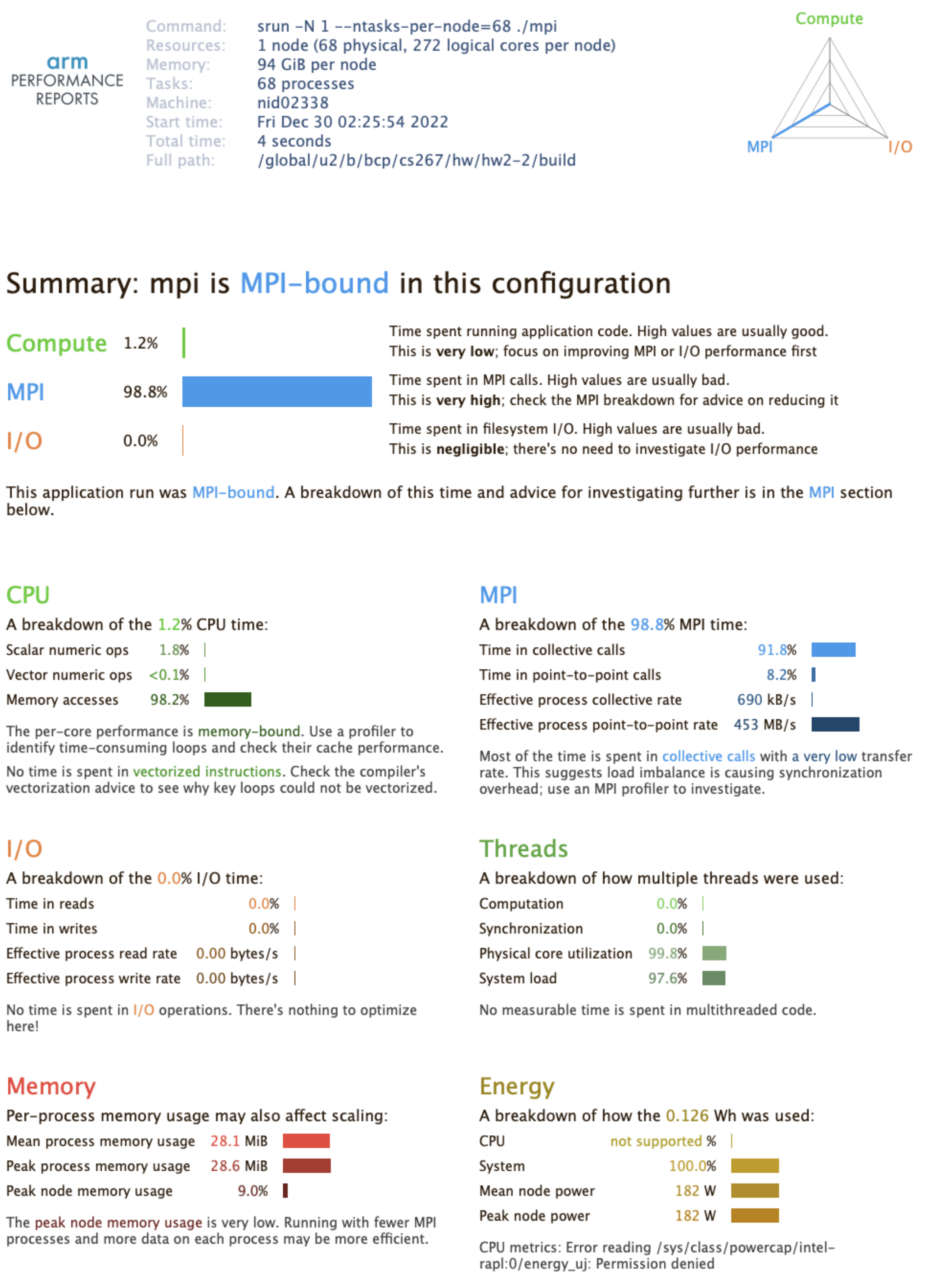


Figure 5: Naive vs. Serial vs. OpenMP vs. MPI

Figure 5 shows the performance comparison between all of the other programming systems we have implemented so far for the n-body simulation.

3.1 Performance Breakdown

Profiling with MPI was difficult, as some of the profilers did not work smoothly with multiple nodes. We were able to successfully profile with Arm Performance Reports with limited configurations. Figure 6 shows a sample output of the trace. Embarrassingly, it shows that 98.8% of our code is MPI bound, telling us that most of the time is spent in communication overheads. Due to the difficulty of debugging with MPI and working with distributed systems, we wish to improve this more if given more time and experience with MPI.



Command:	srun -N 1 --ntasks-per-node=68 ./mpi
Resources:	1 node (68 physical, 272 logical cores per node)
Memory:	94 GiB per node
Tasks:	68 processes
Machine:	nid02338
Start time:	Fri Dec 30 02:25:54 2022
Total time:	4 seconds
Full path:	/global/u2/b/bcp/cs267/hw/hw2-2/build

Summary: mpi is MPI-bound in this configuration

Compute	1.2%	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	98.8%	Time spent in MPI calls. High values are usually bad. This is **very high**; check the MPI breakdown for advice on reducing it
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was MPI-bound. A breakdown of this time and advice for investigating further is in the MPI section below.

CPU

A breakdown of the 1.2% CPU time:

Scalar numeric ops	1.8%
Vector numeric ops	<0.1%
Memory accesses	98.2%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

A breakdown of the 98.8% MPI time:

Time in collective calls	91.8%
Time in point-to-point calls	8.2%
Effective process collective rate	690 kB/s
Effective process point-to-point rate	453 MB/s

Most of the time is spent in collective calls with a very low transfer rate. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.

I/O

A breakdown of the 0.0% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

Threads

A breakdown of how multiple threads were used:

Computation	0.0%
Synchronization	0.0%
Physical core utilization	99.8%
System load	97.6%

No measurable time is spent in multithreaded code.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	28.1 MiB
Peak process memory usage	28.6 MiB
Peak node memory usage	9.0%

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of how the 0.126 Wh was used:

CPU	not supported %
System	100.0%
Mean node power	182 W
Peak node power	182 W

CPU metrics: Error reading /sys/class/powercap/intel-rapl:0/energy_uj: Permission denied

Figure 6: Arm Performance Reports MPI Profile Trace

4 Can We Do Better?

Some design choices we made were mainly simple, to ease debugging and development. Along with the grid partitioning, we also had to be mindful of how we think about programming a MPMD (multiple program multiple data) model. Because the exact same code is being run at the same time, we had to make sure we also synchronize and think creatively how to distribute computation across partitions of data. The use of `rank` was useful, and we figured out a way to make this scalable with various number of nodes and processes that MPI and the KNL nodes provide. We partitioned it such that each process or rank will be allocated a row of the grid space data structure. If we have less ranks than rows provided by the working set, we spill over and have ranks allocate more than one row for computation, denoted by `leftover_processes`. Not proceeding with a rank to 2D mesh allocation may have affected performance, as load balancing could improve. With 2D, you could have more ranks do less computation, thus also potentially minimizing communication bottleneck between proceses.

Like the previous project, we computed the theoretical peak of our system on 2 nodes. The theoretical peak defined for our simulation can be derived as:

$$\frac{1.4 \times 10^9 \text{ instructions}}{\text{second}} * 136 \text{ cores} = 190.4 \text{ GFLOPS}$$

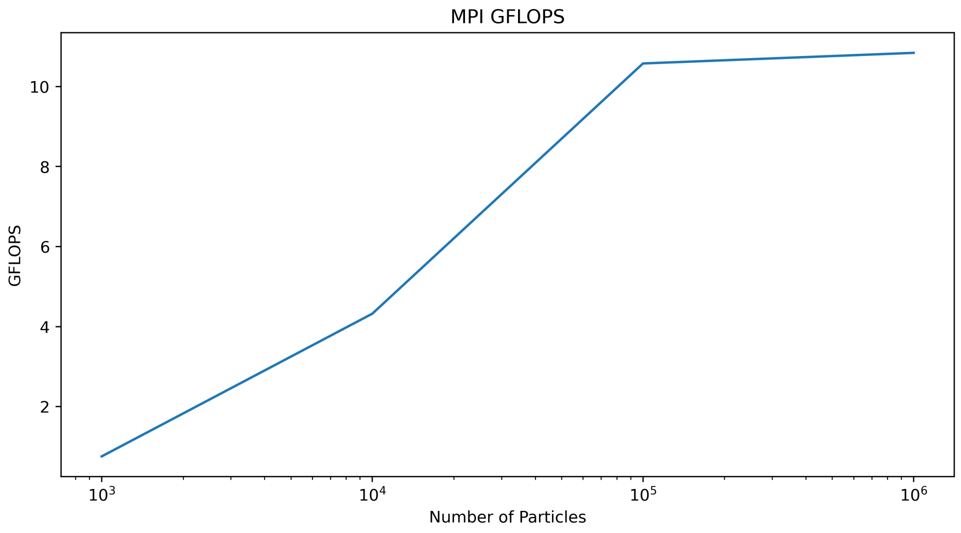


Figure 7: MPI FLOPS

When computing the performance, we see that we reach a peak of 10.84 GFLOPS, which is 5.7% of theoretical peak. This is around 2× worse than our OpenMP solution. Trying to make our simulation communication optimal was very difficult. In order to do substantially better, we have to consider not only synchronization and compute efficiency, but also communication bottlenecks as well. The only benefit that our code has over other methods, such as OpenMP, is that we can scale it beyond a single node. Our performance plot also shows that our simulation in MPI hits a “roof” at around 11 GFLOPS. The slant indicates potentially our memory bound. When debugging performance, we could try to do higher particle count and aim to raise that roof as high as we can. The key and difficulty in this assignment seemed to be trying to hide the communication latencies, as we see from our embarassingly high MPI overhead.

5 Contributions

Everyone started off the exploration of MPI together. Andrew tried to divide columns of bins among each process while Brian and Xuan used a row approach. Brian was able to figure out how to add MPI directives to optimize code. Brian also figured out how to solve segmentation fault errors and finally put everything in working and presentable condition. In the end, Brian’s approach was used as the final solution.

References

[1] “Mpi: A message passing interface,” in *Supercomputing ’93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pp. 878–883, 1993.