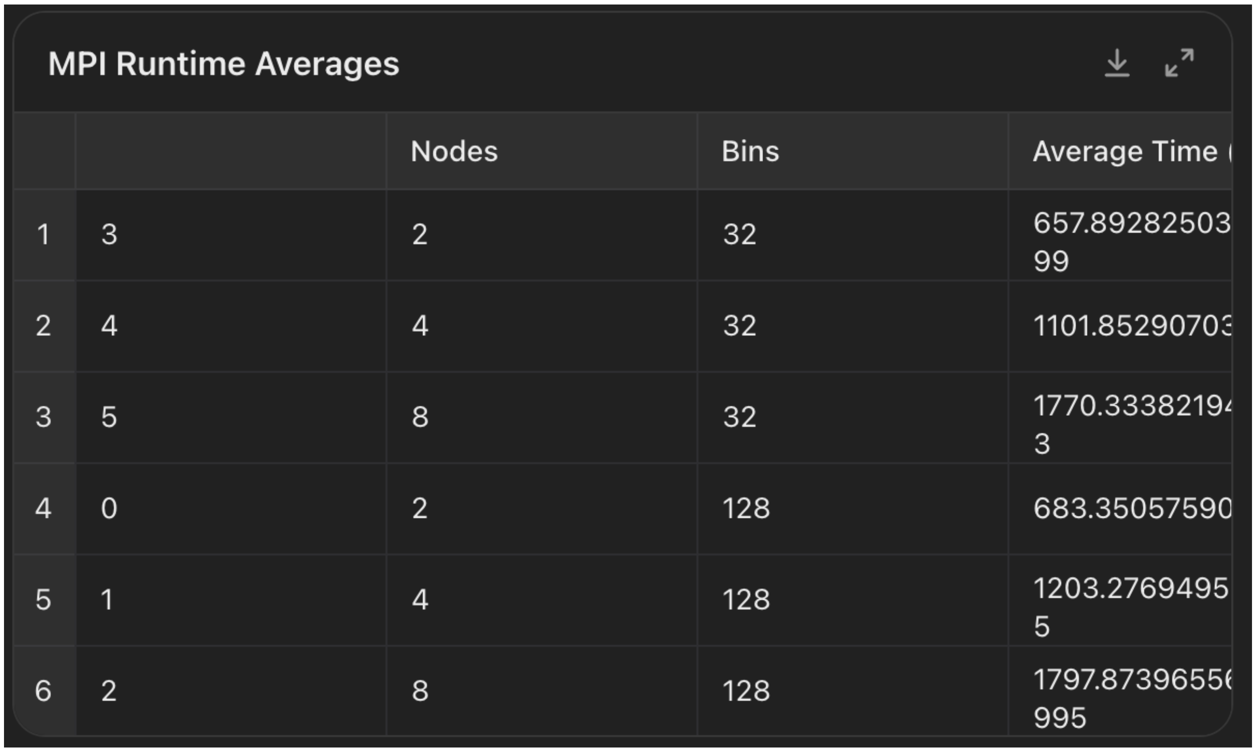
Problem 1:

Run the following commands and observe the output in the corresponding output files:

```
Module load OpenMPI/4.1.6
sbatch run_count_mpe.sh

sbatch run_count_dec_mpi.sh
```

Problem 2:



MPI Runtime Averages

		Nodes	Bins	Average Time (
1	3	2	32	657.8928250399
2	4	4	32	1101.8529070
3	5	8	32	1770.333821943
4	0	2	128	683.35057590
5	1	4	128	1203.27694955
6	2	8	128	1797.873965569 95

In order to modify the number of nodes open run_histogram_mpi.sh and modify the following line:
#SBATCH --nodes=8

A: Results and Discussion

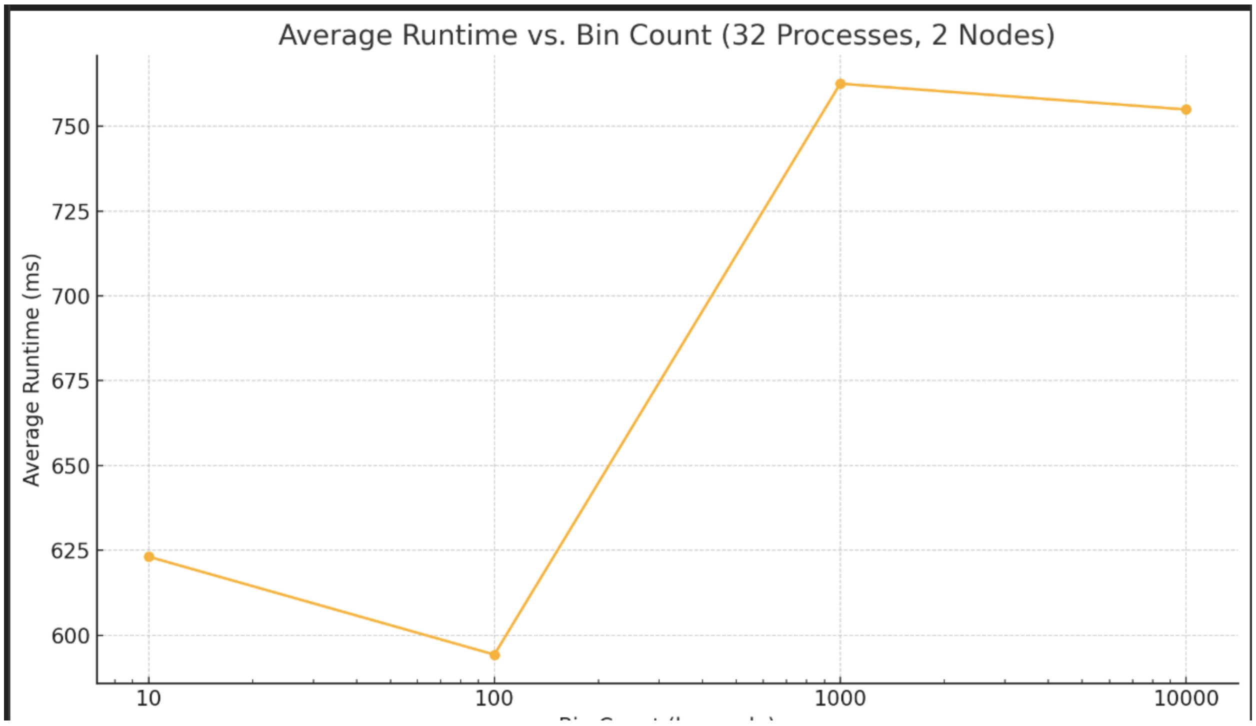
Based on the above when less nodes are used the average runtime is much lower. This indicates that this problem is communication bound rather than compute bound. The additional overhead of adding more nodes and communicating amongst them add much more time than the benefit from splitting the work up into smaller sizes. If there was a larger problem size, then this approach would make more sense because the communication would be a smaller proportion of the overall computation.

B: Results and Discussion

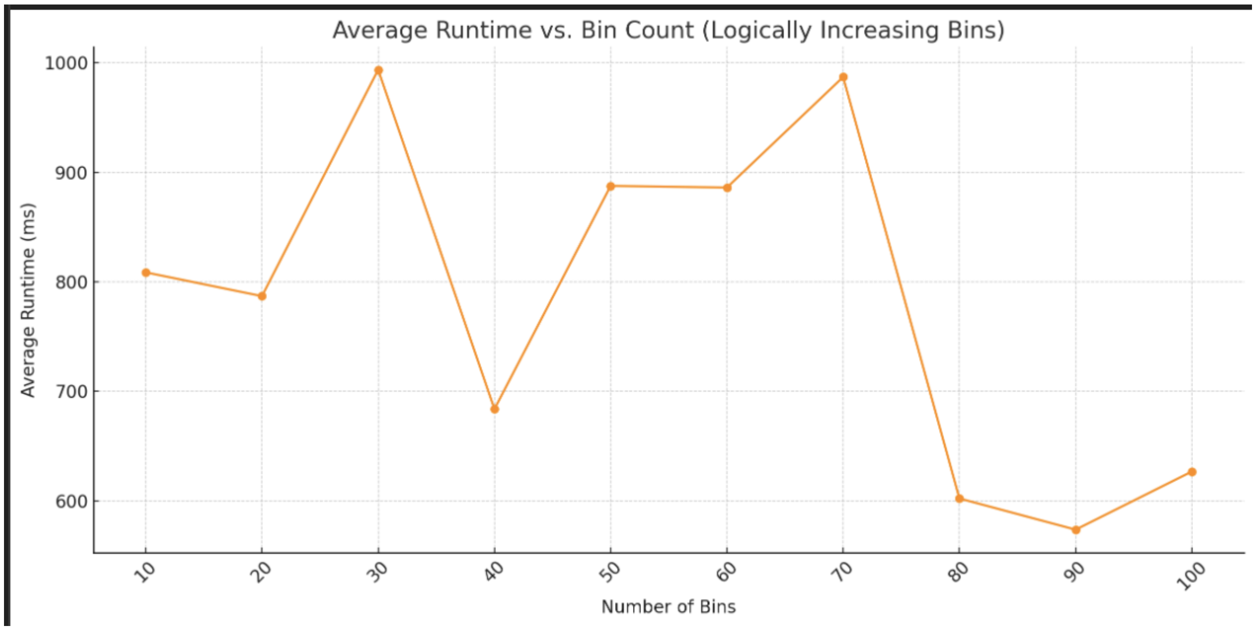
The same story can be said when there are less bins. The communication still takes more time than the computation itself leading to slower runtimes when more nodes are used. This also spawns more processes that then also must be merged in at the end to get the result. In both this case and the previous case, the overhead of adding more nodes and processes far outweighs any benefits

C: Discussion

On average the runs with less bins were slightly faster than the ones with more bins. This is because there is less synchronization overhead when merging all the results back, as there is a much smaller array that needs to be merged.



Based on the results of my experiment of using constant number of nodes and processes I found that there seems to be a sweet spot for bin count. Upon further analysis it seems this is due to the tradeoff between underutilization of the bins by the processes and the overhead of merging.



The runtime does not grow linearly with the number of bins, as previously believed. The best results are obtained with bin counts between 80 and 90. This suggests that performance is influenced by the distribution of the workload and the effectiveness of synchronization in addition to the number of bins.

Problem 3

Scalasca

Main Use Cases

- Primarily developed as a scalable performance analysis tool for High Performance Computing systems with millions of cores. Focused on MPI and OpenMP applications.
- Useful for detecting wait states and communication inefficiencies thus identifying bottlenecks and imbalances in large scale programs.
- Offers summary reports that give a quick overview of possible areas that are causing bottlenecks, this is meant to be used to strategically use the trace reports.
- Offers trace reports that give extremely in depth information about bottlenecks and inefficiencies allowing programmers to optimize those areas.
- Offers a visualization tool to determine where bottlenecks are occurring.

Strengths

- Handles large systems with large numbers of cores very well as it was designed for these types of systems.

- Integrated with other performance analysis and instrumentation tools which allows for ease of use.
- Offers the option to be selective about instrumentation to filter out areas that are not of interest.
- It is easy to use in environments with resource managers like slurm because of its easy to use cli tools.
- The cube viewer that it uses shows a lot of information about the distribution of work across ranks and functions.

Weaknesses

- Its difficult to start using this tool because it requires prerequisite knowledge of other tools such as Score-P and Cube.
- If proper filtering isn't implemented the trace size will become absolutely huge and unusable.
- It can only do post processing, and trace reports can be pretty slow.
- Doesn't really support things like CUDA and OpenACC because it is highly optimized for OpenMP, MPI, and Pthreads.

Sources

https://www.scalasca.org/scalasca/about/about.html

https://apps.fz-juelich.de/scalasca/releases/scalasca/2.3/docs/UserGuide.pdf

mpiP

Main Use Cases

- Designed for people who are trying to identify MPI usage patterns with minimal overhead.
- Works well for quick identification of hotspots in MPI programs, captures aggregated and task local statistics about MPI function usage.
- Generates concise and readable reports.

Strengths

- Very lightweight and doesn't require user to modify source code.
- Only needs debug symbols turned on to work.
- Easy to use due to its lightweight nature and simple command and output structure.
- Can identify time spent in functions vs computation and can break down MPI usage to a fine degree.
- Reports call stack traces to help identify hotspots.

Weaknesses

- Only works for MPI, does not support analysis for OpenMP, Pthreads, or CUDA
- There is no visual output, all information is expressed in the terminal.
- Does not provide insight on inter process communication.
- It works by aggregating statistics rather than tracing through the program which makes it harder to find the root cause of issues using mpiP.

Sources

mpiP Development Team. *mpiP: A Lightweight MPI Profiler*. Version 3.5, Lawrence Livermore National Laboratory. Available at: https://github.com/LLNL/mpiP

https://software.llnl.gov/mpiP/

Problem 4:

"MPI on Millions of Cores" VS "FFT, FMM, and Multigrid on the Road to Exascale: performance challenges and opportunities"

Communication Bottlenecks

The 2010 study "MPI on Millions of Cores" mainly examined communication obstacles from the perspective of MPI. The authors highlighted the drawbacks of collective operations such as Allreduce and Bcast, which had poor scalability because of delay that increased with the number of processes and synchronization overhead. Furthermore, it was noted that one of the factors aggravating collective communication delays was the network topology, particularly torus or mesh arrangements. How to effectively scale the MPI infrastructure to accommodate millions of ranks was the main issue at the time. The 2018 work, on the other hand, focuses on the communication behavior incorporated into full-scale scientific algorithms rather than the MPI primitives. It investigates the various demands that FFT, FMM, and Multigrid place on the interconnect. For example, global all-to-all communication patterns are problematic for FFT and become unworkable at scale. FMM, on the other hand, is commended for its localized, hierarchical communication style, which inherently stays clear of synchronization bottlenecks. Halo exchange patterns are a problem for multigrid (MG), especially at coarser grid levels. With this modification, communication is no longer seen as a library-level problem but rather as an algorithmic design problem that affects scalability at exascale.

Concurrency and Load Balancing

Concurrency was mostly addressed in the 2010 study in relation to managing static workload distributions and launching a large number of MPI ranks. The imbalance brought on by uneven partitioning or poorly dispersed communication overheads was one of the main bottlenecks found. Scalability was supposed to depend on minimizing idle time brought on by processor load imbalance. By 2018, the analysis has evolved into something far more complex. The more recent study delves deeply into the algorithms' internal handling of concurrency. For instance, because of the mathematical structure of its decomposition, FFT has scalability limitations and can saturate beyond $\sqrt{P}$ processors. At the coarsest grid settings, where the issue size is too small in relation to the number of processors, multigrid has trouble with concurrency. Although it still needs to be carefully adjusted, FMM manages concurrency more gracefully because of its hierarchical tree structure. This change demonstrates a deeper comprehension of how concurrency and load balancing at large sizes are impacted by fundamental algorithm structure, not merely task parallelism.

Resilience and Fault Tolerance

While fault tolerance was mentioned in the 2010 article, it was not thoroughly examined. It was generally believed that algorithm-level resilience was not necessary and that system-level checkpoint/restart methods would be adequate to handle failures. The likelihood of a system failure was still controllable at petascale levels at the time. However, the 2018 publication adopts a completely different position. The likelihood of component failures increases with system scale, making resilience an inevitable bottleneck. In order to evaluate how susceptible various algorithms are to data corruption, the study presents new metrics, such as the Data Vulnerability Factor (DVF). For example, FMM's reliance on intricate tree structures makes it especially vulnerable, as a single incorrect pointer could cause significant computation disruptions. This illustrates a paradigm shift where resilience is now deliberately addressed by algorithm developers as a design constraint rather than merely a system level concern.

Memory and Cache Behavior

Memory performance was only mentioned in passing in 2010. The study did not pay much attention to cache or memory bandwidth constraints and instead assumed homogeneous memory structures. The majority of performance issues were expressed in terms of processor counts and communication. On the other hand, memory and cache bottlenecks were the main focus by 2018. The more recent study offers a thorough examination of the interactions between each algorithm and the memory subsystem. Strided memory access patterns impair cache performance and have an impact on FFT. Due to pointer-heavy data structures, FMM has erratic memory access, which reduces GPU efficiency and locality. Sparse matrix-vector multiplications (SpMV), a set of operations infamous for having very low arithmetic intensity and being memory bandwidth-bound, dominate multigrid. This progression demonstrates how memory access comprehension and optimization became just as important as compute or communication optimization as architectures became more hierarchical (including NUMA, cache levels, and accelerators).

Energy Efficiency

Energy efficiency received very little attention in 2010. The authors did not treat energy as a first-order bottleneck, although acknowledging that communication requires more energy than computing. The 2018 study, on the other hand, views energy usage as a crucial constraint to exascale computing. The authors compare the energy-per-operation profiles of FFT, FMM, and Multigrid and find that high arithmetic-intensity algorithms, such as FMM, are better suited for using less energy. The high communication needs of FFT are penalized, and the low computational intensity and high memory bandwidth demand of MG's sparse linear algebra algorithms lead to poor energy efficiency. This expanded focus reflects a larger trend in exascale research that recognizes that sheer speed is useless if it cannot be attained within a strict energy budget and instead correlates performance with performance per watt.

Heterogeneity and Accelerator Suitability

The systems in question were mostly homogeneous clusters of CPUs in 2010. The performance bottleneck analysis did not focus on the difficulties of programming for heterogeneous systems, such as GPU-accelerated nodes. However, by 2018, heterogeneity had become commonplace, and the performance of each algorithm on accelerator-based systems is explicitly evaluated in the article. FFT is comparatively simple to port and benefits from well-established GPU libraries like cuFFT. However, because of the bandwidth-intensive nature of SpMV operations and inadequate data reuse, MG suffers from GPU inefficiencies. Pointer-based structures and erratic memory access provide problems for FMM since they don't translate well to SIMD or GPU paradigms. This illustrates a significant change in perspective from optimizing for a large number of cores to optimizing for a variety of computing components with wildly disparate performance attributes.

Conclusion

The evolution of the field and the evolving character of exascale computing issues are demonstrated by the change in performance bottlenecks between 2010 and 2018. The 2018 study adopts a more comprehensive perspective that takes memory hierarchy, energy, resilience, and heterogeneity into account, whereas the 2010 paper concentrated on the scalability of MPI as a middleware layer. The development emphasizes the value of co-design, which involves optimizing algorithms in concert with the architectures they operate on. These days, scaling involves more than just adding more cores; it also entails doing so in a way that is portable across a variety of hardware platforms, robust, memory-aware, communication-efficient, and energy-efficient.