FINAL PROJECT HPC - EXERCISE 1
PERFORMANCE EVALUATION OF OPENMPI COLLECTIVE OPERATIONS

Adriano Donninelli
Università di Trieste
adonnine@sissa.it

ABSTRACT

In the realm of high-performance computing, understanding the efficiency and performance of parallel processing libraries is crucial. This report delves into the assessment of the OpenMPI library's collective operations on the ORFEO cluster, specifically targeting the EPYC partition. OpenMPI provides a range of algorithms for collective operations, allowing users to optimize performance based on various parameters. The focus here is on the broadcast and scatter operations and the performance of a few of their possible implementations.

1 Introduction

The primary objective of this study is to evaluate the performance of the default OpenMPI implementations of broadcast and scatter operations, focusing on different algorithms, data sizes and number of processes. We analyzed the collective operations and algorithms using the OSU benchmark, executed on up to two nodes of the ORFEO cluster.

The analysis is performed on the EPYC partition which features 2xAMD EPYC 7H12 processors. These processors, based on the "Rome" architecture [1], house a total of 128 cores (2x64), each with a base clock of 2.6 GHz that can dynamically boost to 3.3 GHz. The interconnectivity of the system is facilitated by an infinity fabric with a theoretical bandwidth of 96 GB/s. Each processor is composed of eight Core Complex Dies (CCDs), the CCDs are interconnected through the infinity fabric, with three links at 16GT/s. Within each CCD, two Core Complexes (CCXs) reside, each containing four cores and 16 MB of L3 cache. This intricate architecture is designed to deliver high-performance computing capabilities.

2 Experimental Setup

To assess the performance of broadcast and scatter operations on the ORFEO cluster, we utilize two EPYC nodes, for a total of 256 cores. The OSU benchmark facilitated performance evaluations but we increased the number of iterations, especially warm-up ones to obtain more stable results.

The processes were distributed evenly across the two nodes, to ensure that the latency measured always considers inter node communications. Results for different data sizes are presented, enabling visualization and analysis of performance metrics.

To streamline the data collection process, a Bash script manages resource allocation, while a Python script, triggered by the Bash script, orchestrates the mpirun calls, reads and aggregates the results. This makes the testing process straightforward and easily replicable.

2.1 Algorithms

When approaching collective operations, the default OpenMPI implementation utilizes a heuristic to dynamically select the most optimal algorithm, considering aspects like cluster topology, data size, and the number of processes. In our assessment, we compare this default with a few of the possible implementations for broadcast and scatter operations. Here's a brief description (ref [2]) of some of the aforementioned algorithms we use:

- Binary Tree: Each internal process has two children with sequential dtaa transmission from each node to both children.
- Pipeline: Linear structure, unidirectional propagation from root to leaves [3].
- Basic Linear: Root node sends individual messages linearly to all participating nodes.
- Binomial: Employs a binomial tree structure for scatter operations.
- Non-blocking Linear: Overlapping receive and send operations, ensuring efficient communication.

These algorithms offer diverse approaches to collective operations, optimizing data transmission based on their inherent structures and characteristics.

2.2 Mapping

In the context of Open MPI, the map-by option is a feature used in job launch commands, to specify how processes should be mapped to the available hardware resources (e.g., cores, sockets, or nodes) in a parallel computing environment. In order to decide the most appropriate map-by setup for the experiments we conducted a few latency tests.

The goal was to optimize MPI broadcast operations by mapping resources efficiently. Inspired by the ORFEO Intel benchmark, we focused on reducing latency by strategically binding processes. To assess the possible improvements we tried to bind up to 16 cores belonging to the same NUMA region, specifically targeting cores closest to the InfiniBand card (cores 96-111). The osu latency benchmark was employed for two-process scenarios, and osu broadcast for larger process counts.

The latency between two processors was notably reduced when binding to the specified cores, especially for small message sizes. However, as the message size increased, the impact diminished as pure latency played a smaller role in the total timing. Moreover, testing with a larger number of processors demonstrated an increase in latency, even with small message sizes.

We also observed that the choice of the algorithm significantly influenced the optimal map-by strategy. For instance (Table 1), comparing Chain and Binary Tree algorithms for broadcasting revealed that the Chain algorithm's performance dropped when using map-by socket instead of map-by core. The effect on the Binary tree was notably less severe. This was attributed to the higher latency associated with distant communication paths between subsequent processors in the map-by socket scenario.

Table 1: Latency vs map-by comparison for Broadcast operation. 128 cores per Node

Test Type	Message Size	Avg Latency (us)
Map by socket - Chain Algorithm	1	32.10
Map by core - Chain Algorithm	1	7.77
Map by socket - Binary Tree Algorithm	1	6.70
Map by core - Binary Tree Algorithm	1	3.67

Distributing processors evenly among nodes, utilizing a map-by core strategy, proved effective in reducing overall latency, especially with a larger number of processors.

Considering these observations, it was decided not to create a custom rank file for the benchmark but to solely use the map-by option. Achieving an optimal solution for all algorithms is not trivial. Further tests confirmed that the map-by core policy consistently led to low latencies across all algorithms, establishing it as our preferred choice for subsequent experiments.

3 Experiments

In our experiments, we conducted a comprehensive evaluation of both broadcast and scatter operations, examining various parameters to assess their performance. The tests encompassed a range of core configurations, message sizes, and visualization techniques.

Test Parameters: Cores varied from 1 to 128 per node, totaling 256 cores. Message sizes ranged from 1 to 2^{19} bytes.

We provide two visualizations to compare the performances of the collective operations considered. First (Figure 1, 2), a 3D heatmap for each collective operations and its algorithms is presented, portraying latency against message size and the number of processes. Similar observations for the two operations can be drawn: Across different algorithms, similar trends emerged, with occasional spikes in default algorithms when heuristic decision functions changed. The spikes likely indicated suboptimal choices but they could possibly be attributed to occasional resources overload or failure.

Secondly (Figure 3, 4), for both collective operations we plot and compare latency against number of cores for the different algorithms using a fixed small message size of 4 bytes. The graph reveals some fluctuations in timings but overall the results demonstrated satisfactory patterns.

We also construct two distinct theoretical performance models, one model pertains to the Pipeline algorithm, while the other is for the Binary Tree algorithm (and Binomial in the scatter case). Further details on these performance models are provided in the subsequent subsection.

3.1 Performance Models

Developing performance models is challenging due to the myriad factors that need consideration. Recall that we consider a message size of 4 bytes and therefore we will simplify the calculations by assuming data transfer is instantaneous.

Various latencies in point-to-point communication between cores and nodes must be accounted for. To model the broadcast pipeline algorithm, we measured latency across different regions of an EPYC node, resulting in 5 distinct timings: for cores in the same CCX, in the same CCD but different CCX, in the same NUMA region, in the same SOCKET, and between different SOCKETS. Additionally, we measured latency for communication between different nodes.

Table 2: Latency Values for Different Regions (in seconds)

Region	Latency
Same CCX	0.15e-6
Same CCD, Diff. CCX	0.31e-6
Same NUMA	0.34e-6
Same SOCKET	0.36e-6
Diff. SOCKET	0.65e-6
Diff. NODE	1.82e-6

Since the pipeline algorithm exhibits linear, unidirectional propagation from root to leaves, left to right, we used Open MPI flag '–report-bindings' to verify that half the cores are subsequently instantiated on the first node and then the rest are distributed on the second node. The pipeline latency estimation is then modeled as the sum of point-to-point communications between subsequent cores, with latency (Table 2) based on the binding region between core i and core $i+1$ which is known since we adopt map-by core policy.

$$T_{pipeline} = \sum_{i=0}^{i=N-1} T_{pt2pt}(i, i+1).$$

Where N is number of cores and $T_{pt2pt}(i, i+1)$ is the latency in pt2pt communication between core i and core $i+1$.

As illustrated in Figure 3, this model reasonably approximates the empirical results, demonstrating underestimation primarily when the number of cores exceeds 150. Given the inherent fluctuations in point-to-point communications latency, it is expected that our simplified model, which overlooks these fluctuations, would experience underestimation as the number of cores grows significantly.

Similar to the pipeline algorithm, we developed a model for the binary tree broadcast. In this case, we calculated the tree's height and considered the sum of communication latencies of each layer. At each layer, point-to-point communications occur in parallel between different processes, and we account for this by taking the maximum point-to-point timing based on the bindings of the core in that layer.

However, this simplistic model consistently underestimates empirical results. The latency grows only when the number of cores increases by a power of two, indicating an obvious underestimation. Additionally, the doubling of point-to-point communications in each layer, assuming perfect parallelism, contributes to underestimation. To address this, we added two penalty factors, the first that accounts for the number of point-to-point communications for each layer and their latency. The second penalty factor, on the other hand, is based on the total number of processes used. While these scaling factors are derived from empirical results, they allow us to construct a more refined model consistent with timings.

$$T_{tree} = \gamma(N) + \sum_{i=1}^{i=H} T_{pt2pt}(i)\ (1 + f_{overlap}(i)).$$

Where N is number of cores, H is the height of the tree and $T_{pt2pt}(i)$ is the latency of the longest communication occurring on i-th layer of the tree. $f_{overlap}(i)$ and $\gamma(N)$ are penalty factors estimated empirically, the former depends on number of communications on layer i, the latter depends on total number of processes.

For the scatter binomial, we adopted the same binary tree model and associated factors, resulting in a reasonable estimation. We conclude that binomial tree scatter behavior is similar to broadcast binary tree in latency growth and that the factors estimated might be consistent among different tree based implementations.

In contrast, attempts to model the scatter linear algorithm yielded poor results. Initially assuming communication similar to a flat tree, where the root process scatters data to all children individually, led to a substantial overestimation. It is plausible that some optimization in the communication pattern is applied, resulting in more modest timings.

4 Conclusion

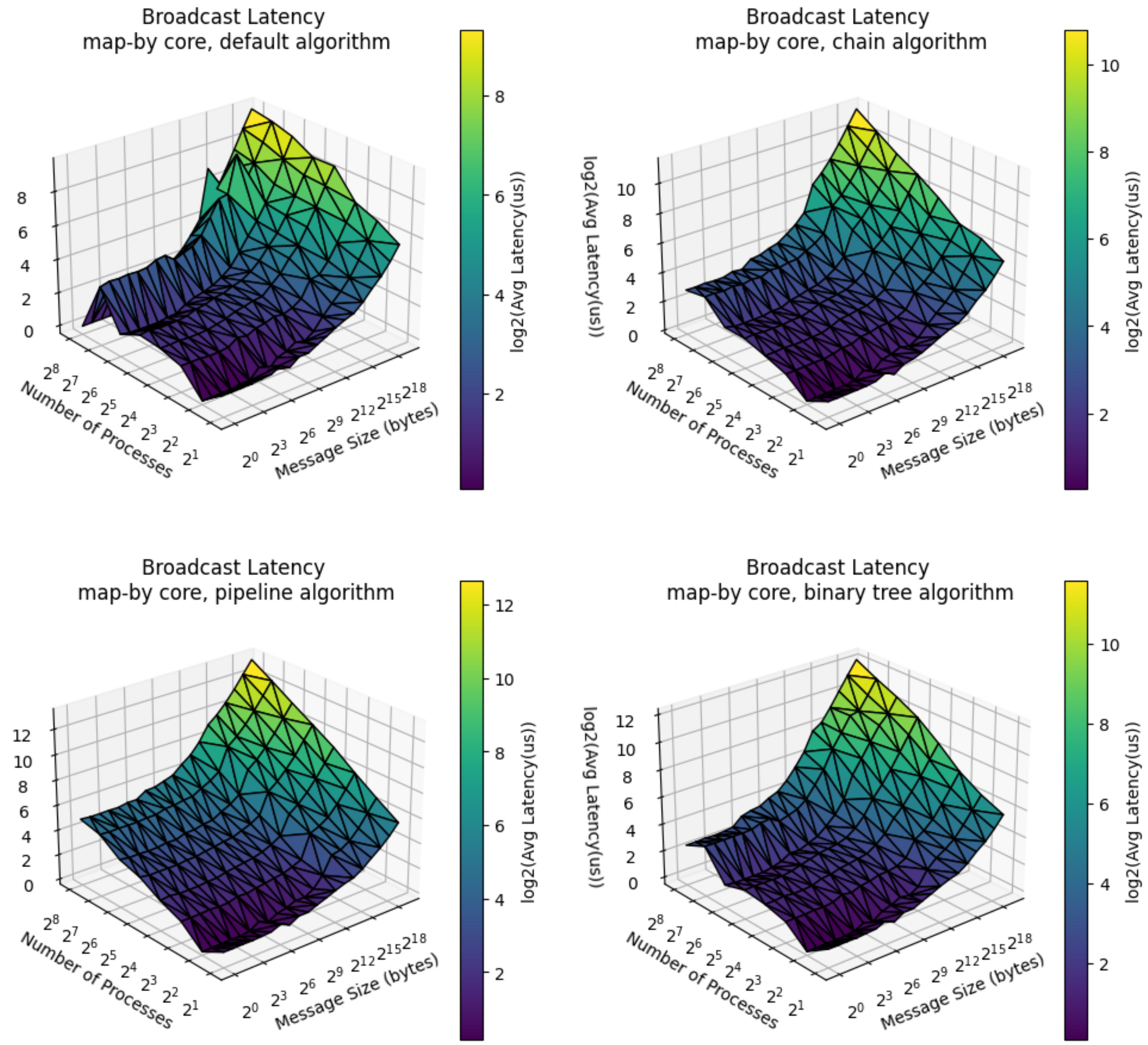
In summary, our evaluation of broadcast and scatter operations on the ORFEO cluster highlights the intricate dynamics of communication patterns, hardware, and algorithm choices within OpenMPI. The experimentation and modeling revealed complex latency behaviors, particularly exposing underestimations in simplified performance models. We observed the importance of considering factors like cluster topology, binding regions, point-to-point communication variations, and scaling effects for constructing accurate performance models.

References

[1] Markus Velten, Robert Schöne, Thomas Ilsche, and Daniel Hackenberg. Memory performance of amd epyc rome and intel cascade lake sp server processors. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22. ACM, April 2022.

[2] Emin Nuriyev, Juan-Antonio Rico-Gallego, and Alexey Lastovetsky. Model-based selection of optimal mpi broadcast algorithms for multi-core clusters. *Journal of Parallel and Distributed Computing*, 165:1–16, 2022.

[3] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, June 2007.

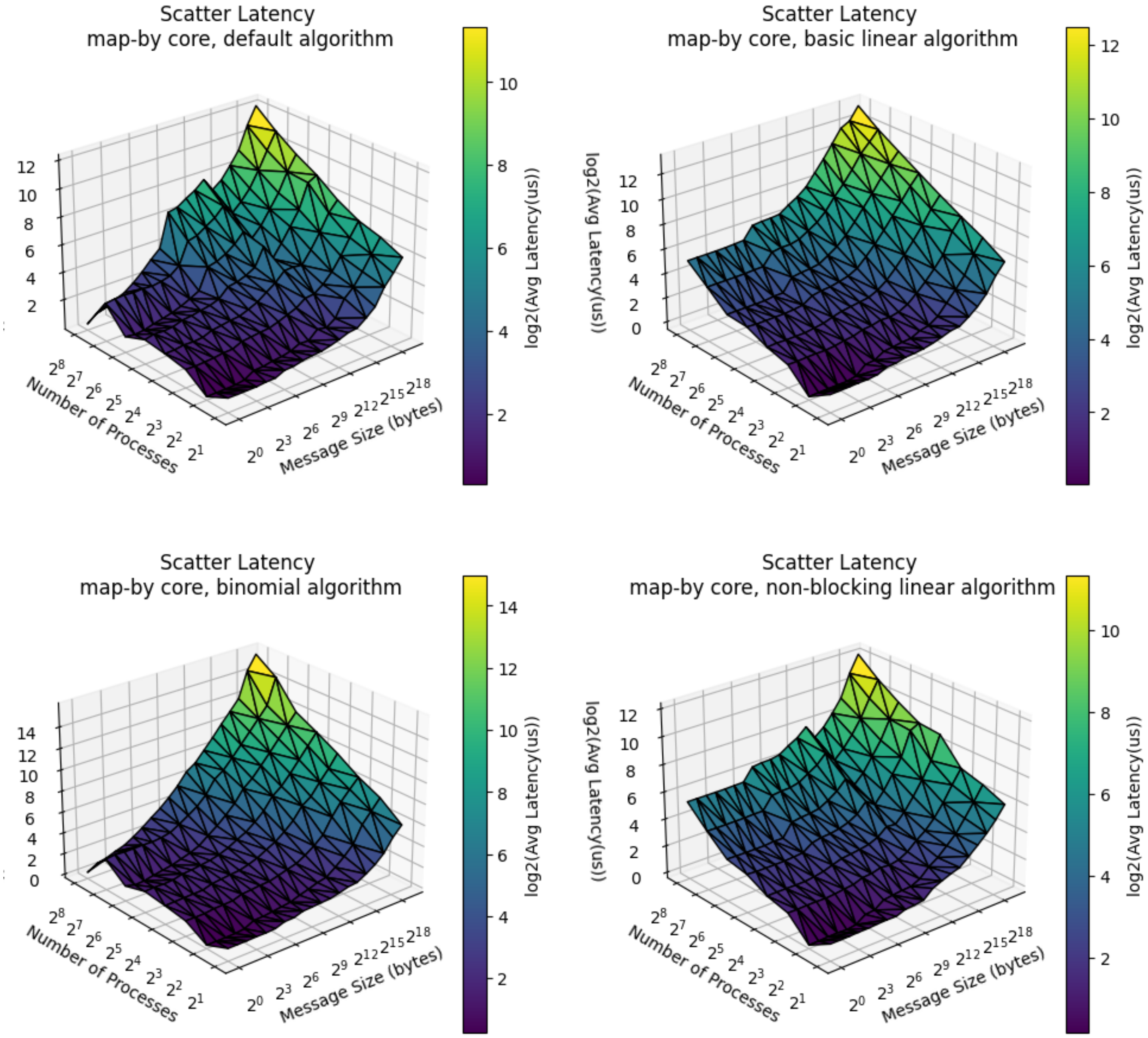

Broadcast Latency
map-by core, default algorithm

Broadcast Latency
map-by core, chain algorithm

Broadcast Latency
map-by core, pipeline algorithm

Broadcast Latency
map-by core, binary tree algorithm

Figure 1: 3D Heatmap for different algorithms of Broadcast operation.



Scatter Latency
map-by core, default algorithm

Scatter Latency
map-by core, basic linear algorithm

Scatter Latency
map-by core, binomial algorithm

Scatter Latency
map-by core, non-blocking linear algorithm

Figure 2: 3D Heatmap for different algorithms of Scatter operation.

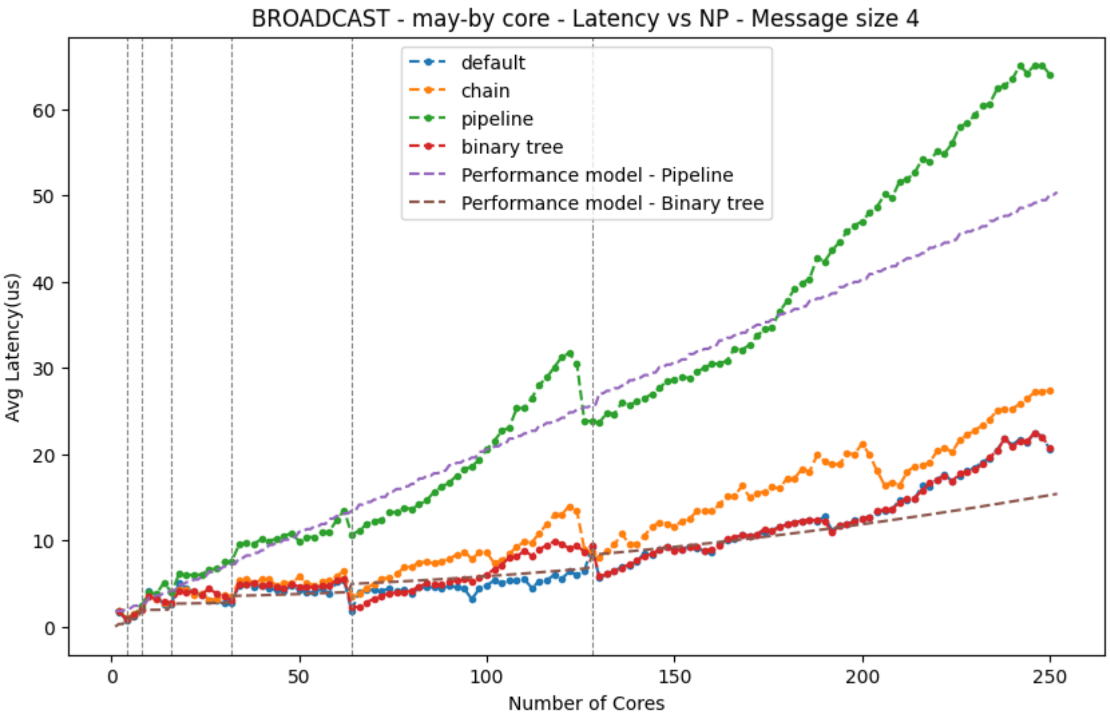


Figure 3: Plot of latency of different broadcast algorithms for fixed message size.

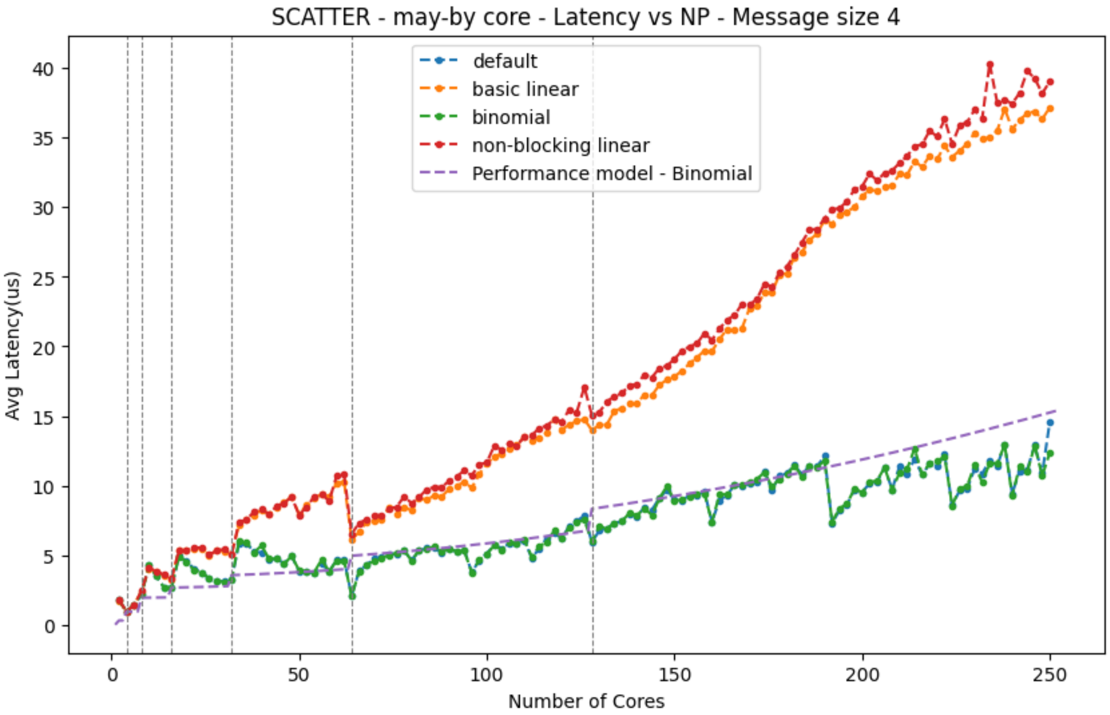


Figure 4: Plot of latency of different scatter algorithms for fixed message size.