02616 Large Scale Modeling

Project 1

Alexander Elbæk Nielsen
s214724

Junrui Li
s242643

Philip Korsager Nickel
S214960

Technical University of Denmark

October 19, 2025

All members contributed equally to the creation of this report.

Contents

The Mandelbrot Set

This project regards the implementation of the Mandelbrot set calculation and parallelization of this using MPI. For a given pixel, the Mandelbrot set can be calculated by

$$p_0 = x + iy$$
$$p_{n+1} = p_n^2 + p_0$$

where x and y are the image coordinates and i is the imaginary number. The iterations are truncated when $|p_n| > 2$ or $n > n_{max}$, at which point the value n is stored for that particular pixel.

In this project, we are given an implementation of the Mandelbrot set calculation. The goal of the project is to extend the implementation to use MPI for parallelization of the calculations, i.e. split the image into chunks, and let the chunks be processed independently by different ranks. To obtain this, we implement and test two different MPI implementations; a blocking send/receive code, and a non-blocking send/receive code. Furthermore, we implement and test both a static scheduling strategy and a dynamic scheduling strategy for the distribution of chunks, meaning we will test a total of 4 different implementations.

Implementation

2.1 Repo structure

Quick overview of the repo structure:

2.2 Scheduling

The Mandelbrot set calculation is inherently load-imbalanced, as there are areas of the image needing far more operations than other areas. This means certain chunks of the image will take longer to compute. To examine this, we implement and test two different load balancing strategies; static and dynamic scheduling. Static scheduling should have small communication overhead, but will most likely be imbalanced due to chunks being assigned to each rank before the work starts. Dynamic scheduling sees ranks requesting more work when they have processed a chunk - this means the ranks processing low effort chunks will get more work than those processing high effort chunks - but with more communication overhead.

Static Scheduling

The static scheduler simply assigns each chunk to a rank in a deterministic round-robin style, meaning every rank gets one chunk, then every rank gets a second, etc. until there are no more chunks. The implementation is made by defining target ranks for each chunk as

```
for chunk_id in range(self.config.total_chunks):
    rank = chunk_id % self.world_size
    self.assignments[rank].append(chunk_id)
```

i.e. for each chunk_id assign the chunk to the rank resulting from integer division between the chunk_id and the total number of ranks. This means, for an image of size 1000×1000, a chunk size of 100, and 4 ranks, the distribution will be

- rank 0: chunks 0, 4, 8
- rank 1: chunks 1, 5, 9
- rank 2: chunks 2, 6
- rank 3: chunks 3, 7

Here, ranks 0 and 1 will process 1 more chunk than ranks 2 and 3, regardless of which chunks are the computationally easiest, meaning we might see rank 2 or 3 finish entirely before ranks 0 and 1 start on their last chunk.

Dynamic Scheduling

The dynamic scheduler attempts to fix the imbalanced structure of the static scheduler, by having ranks request another chunk to process when they are finished processing. This works by letting rank 0 be a 'master' rank, meaning this rank does not process any chunks itself, but rather handles the distribution of work, while the rest of the ranks, which we denote 'worker' ranks, are doing the processing The workflow is as follows:

1. Each rank except the master rank is assigned 1 chunk.
2. Whenever a rank is finished processing a chunk, it sends the processed chunk to the master rank and then sends a message requesting more work. In the blocking send/receive code, this is done sequentially, while the non-blocking send/receive code handles all communication simultaneously.
3. The master rank assesses if there is more chunks to be processed, and then either assigns a chunk to the requesting worker, or sends a message that no work needs to be done.
4. In the end, the master rank gathers all chunks into a full image.

This dynamic scheduling strategy should even out the total processing times across ranks, as the ranks that get assigned computationally heavy chunks does not need to process as many chunks as others. However, the larger need for communication means we have to assign one rank as a master rank, which does not process anything. This means the dynamic scheduling will be slower than the static scheduler for a low number of ranks, as the difference between 3 and 4 working ranks is much bigger than that of 15 or 16 working ranks, for instance. An alternative to this would be to have a sub-process for the master rank, in which it could process chunks while simultaneously distributing work to the worker ranks, but this would require a large expansion of the implementation, as well as some tweaking of how large chunks the master rank would be able to process, without it influencing the distribution of work.

2.3 Communication

The communication implementation in mpi4py differs fundamentally between static and dynamic scheduling. Static scheduling uses an all-to-one gather pattern after all computation completes, while dynamic scheduling implements a continuous master-worker request-response cycle throughout execution.

Message Protocol

The implementation uses distinct message tags for different communication purposes: `META_TAG` and `DATA_TAG` for chunk metadata and pixel data, `REQUEST_TAG` and `ASSIGN_TAG` for dynamic work distribution, `COUNT_TAG` for static chunk counting, and `DONE_TAG` for worker completion signals.

Metadata and data are transmitted separately because `Irecv` requires pre-allocated buffers of exact size. The metadata `[start_row, end_row]` provides chunk dimensions, enabling proper buffer allocation before data reception.

Blocking Communication

Static Scheduling Workers compute all assigned chunks locally, then send their chunk count followed by metadata and data for each chunk. The master receives from workers sequentially, allocating exact-sized buffers after receiving each metadata message.

Dynamic Scheduling The master uses `Probe(ANY_SOURCE, ANY_TAG)` to handle messages from any worker. Upon receiving a work request, the master assigns the next available chunk (or -1 to signal termination). Workers operate in a request-compute-send cycle until receiving the termination signal.

Non-blocking Communication

Static Scheduling: Three-Phase Design The non-blocking static implementation uses a three-phase protocol to address the buffer allocation challenge:

Phase 1: The master collects all metadata from workers using blocking receives to determine exact buffer sizes.

Phase 2: With buffer sizes known, the master posts all `Irecv` operations simultaneously, enabling concurrent data reception from all workers.

Phase 3: The master waits for all receives to complete and assembles the final image.

Workers implement a corresponding two-phase protocol: first sending all metadata, then using `Isend` for concurrent data transmission followed by `Waitall`. This design trades one additional metadata communication round for full parallelism during data transmission.

Dynamic Scheduling: Pre-posted Buffer Pools The dynamic non-blocking implementation pre-posts `Irecv` operations for all workers using fixed-size buffers. The master maintains two pools: one for work requests and one for results (metadata and data).

The master continuously tests both pools for completed receives. Upon detecting a completed request, it assigns work and immediately re-posts a new `Irecv` with a fresh request object (completed requests cannot be reused). Result buffers are allocated at maximum chunk size, with metadata specifying the actual valid data region.

Workers use `Isend` for non-blocking transmission and maintain an `active_sends` pool to ensure buffers remain valid until transmission completes. The implementation periodically tests and removes completed sends to prevent memory growth.

2.4 Timing Measurement

Implementation

We measure three primary timing metrics using `MPI.Wtime()`: wall-clock time (end-to-end execution), computation time (time spent in `compute_chunk()`), and communication time (time in MPI send/recv operations).

Our timing implementation tracks the following communication operations: `REQUEST_TAG` for worker task requests (dynamic only), `ASSIGN_TAG` for master task assignments (dynamic only), `DATA_TAG` for chunk data transfer (both modes), and `gather()` for statistics collection (excluded from wall-clock time).

Computation Timing

We wrap each chunk computation and record per-chunk timing:

```
comp_start = MPI.Wtime()
start, end, chunk = compute_chunk(config, chunk_id)
comp_time = MPI.Wtime() - comp_start
stats["comp"] += comp_time
chunk_details.append({rank, chunk_id, start, end, comp_time})
```

These per-chunk records allow us to analyze load balancing across ranks and identify computational hotspots.

Communication Timing

We wrap each MPI operation with timing measurements:

```
t0 = MPI.Wtime()
comm.send(payload, dest=worker, tag=ASSIGN_TAG)
stats["comm_send"] += MPI.Wtime() - t0
```

We split communication time into `comm_send` and `comm_recv` to analyze directional overhead.

Scheduling-Specific Patterns

Static and dynamic scheduling have different timing characteristics due to their communication patterns.

In static scheduling, computation and communication are separated. All ranks compute their assigned chunks first, then send results in a dedicated gathering phase:

```
for cid in chunk_ids:
    results.append(compute_chunk(...))
image = gather_results(...)  # separate communication phase
```

In dynamic scheduling, computation and communication are interleaved. Workers continuously request tasks, compute, and send results:

```
while True:
    chunk_id = comm.recv(source=0, tag=ASSIGN_TAG)
    compute_chunk(...)
    comm.send(result, dest=0, tag=DATA_TAG)
```

Aggregation

Each rank collects local statistics during execution. After computation finishes, we gather all timing data to rank 0:

```
all_times = comm.gather(rank_times, root=0)
all_chunks = comm.gather(chunk_records, root=0)
```

Rank 0 then computes aggregate metrics: `comp_total` (sum of all ranks' computation time) and `comm_total` (sum of all ranks' communication time).

We measure wall-clock time from the start of computation until just before the final `gather()` operation that collects statistics. This excludes the overhead of metadata collection itself and focuses on the actual computation and communication work.

Experiments

3.1 Correctness of solution

Before jumping in to the experiments and benchmarkings, we would like to verify that the resulting images from our 4 implementations are actually correct - i.e. that the chunks are processes correctly and assembled correctly to a full image. To do this, we use the resulting image from the reference implementation `Mandelbrot.py` along with the testing package `pytest-mpi` to verify that the images from each implementation are correct. This is done by adding the decorator `@pytest.mark.parametrize()` to a test function, which uses `numpy.testing.assert_allclose()`. Using this, we have verified that our 4 implementations produce correct results for all combinations of the configurations:

- Ranks: 1, 4
- Chunk-size: 10, 15
- Image size: 15×15, 20×20
- Image limits $[-2.2, 0.75]$, $[-1.3, 1.3]$ and $[-2.2, 0.75]$, $[0, 1.3]$

With these checks in order, we feel confident in our implementations and ready to move on to the testing and benchmarking.

3.2 Experiments and Results

Quick optimizations using Numba

Declorating the function performing the Mandelbrot calculation with '@njit' and optionally 'parallel=true' yields significant gains without spending time hypertweaking the code. This can be observed in figure 1. Enabling threaded parallelization by changing 'for i in range' to 'for i in prange' . We include these optimizations in all the following experiments.

Load balancing

Next, we would like to explore the effect of the chunk size for the different implementations. This should emphasize the differences between the different implementations, as small chunk-size should mean the computationally heavy sections should be split into different chunks, leading to the static scheduler being more balanced, while the dynamic scheduler should experience larger communication times. For larger chunk-sizes, we should see

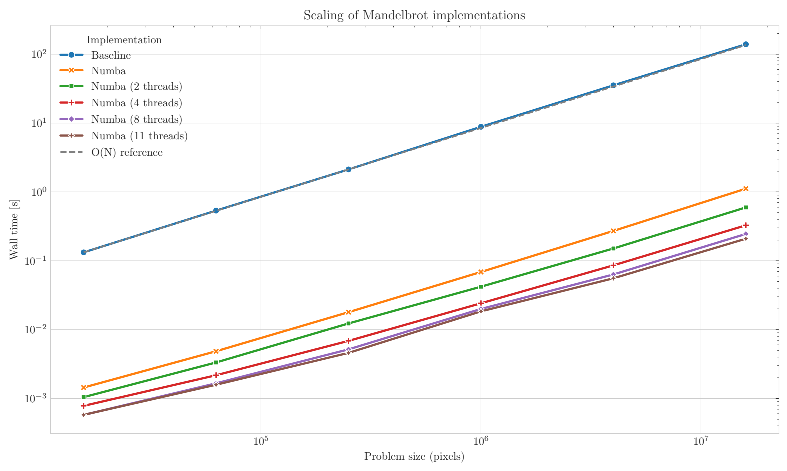

Figure 1: Comparing the provided baseline Mandelbrot calculation with minimal Numba enhancements

the dynamic scheduler perform better, as the difference in computation times should vary more from chunk to chunk. To test the effect of the chunk-size, we conduct the following:

- 8 ranks
- Image size 10000×10000
- Chunk-sizes $50, 100, 200, 250, 500, 750, 1500$
- All 4 implementation combinations
- Image limits $[-2.2, 0.75]$, $[-1.3, 1.3]$ and $[-2.2, 0.75]$, $[0, 1.3]$

The result of this test can be seen in figure 2 and figure 3. Here, we see that the dynamic scheduling strategy is slower for small chunk-sizes, but faster for large chunk-sizes, compared to he static scheduling strategy. This is exactly as expected: since we only have 8 ranks here, the dynamic scheduling will be slower, since there is only 7 working ranks, but when the chunk-size gets large, the benefit of assigning chunks intelligently can be seen clearly, as the total runtime per rank is smaller than static scheduling, and there are less differences in computation times between ranks. In this experiment, we see little to no difference between the blocking and non-blocking implementations, as well as little to no difference between the symmetric full image with limits $[-2.2, 0.75]$, $[-1.3, 1.3]$ and the asymmetric image with limits $[-2.2, 0.75]$, $[-1.3, 1.3]$, with the latter being a sign that the implementations are somewhat load-balanced.

In figure 4, we show the computation time per chunk, dependent on the chunk-size. Chunk-ID simply denote a specific chunk, with max(Chunk_ID) being the total number of chunks in that configuration. From this plot, it is clear to see the imbalance in computation times for different chunks. It is most clear for the very large chunk-sizes, where some chunks

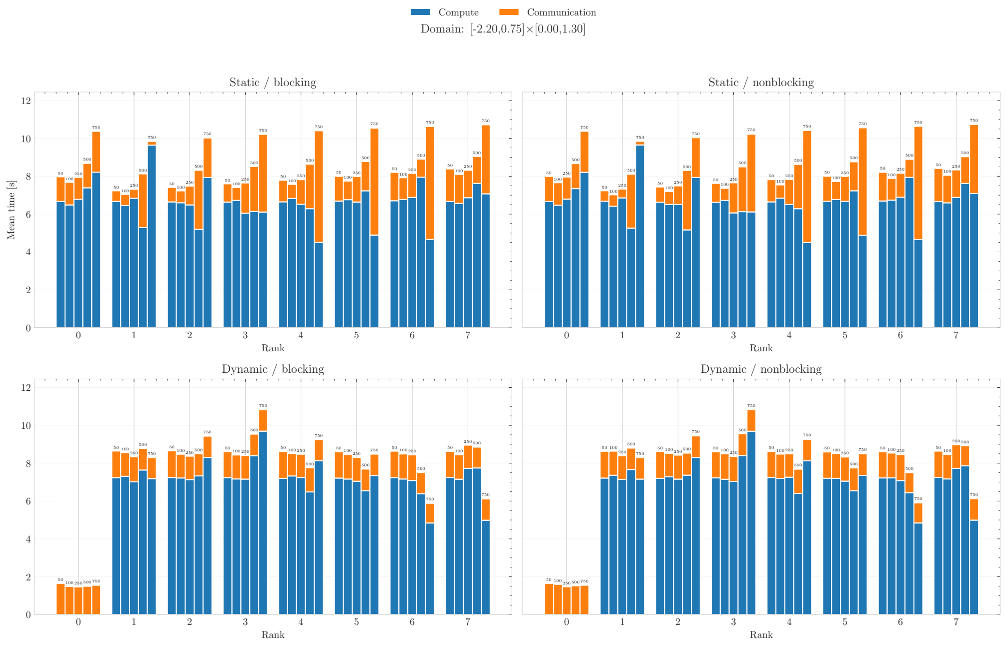

Figure 2: Rank-based timings on asymmetrical domain

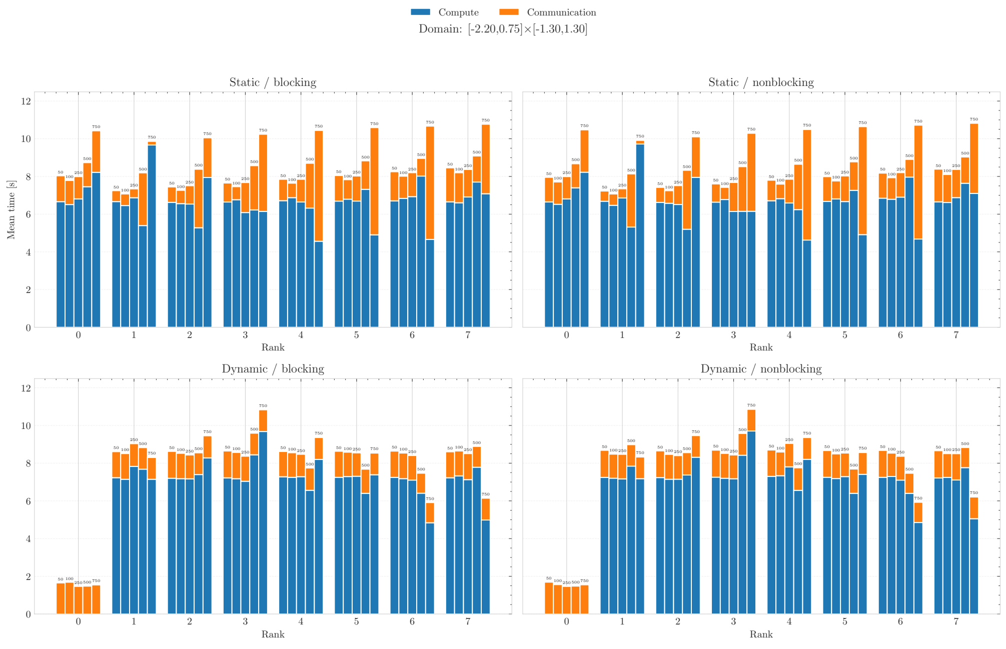

Figure 3: Rank-based timings on symmetrical domain

take upwards of 15 seconds to compute, but it can also be seen for smaller chunk-sizes that there are chunks in the middle to end part of the computations, which take longer to compute than the rest of the chunks.

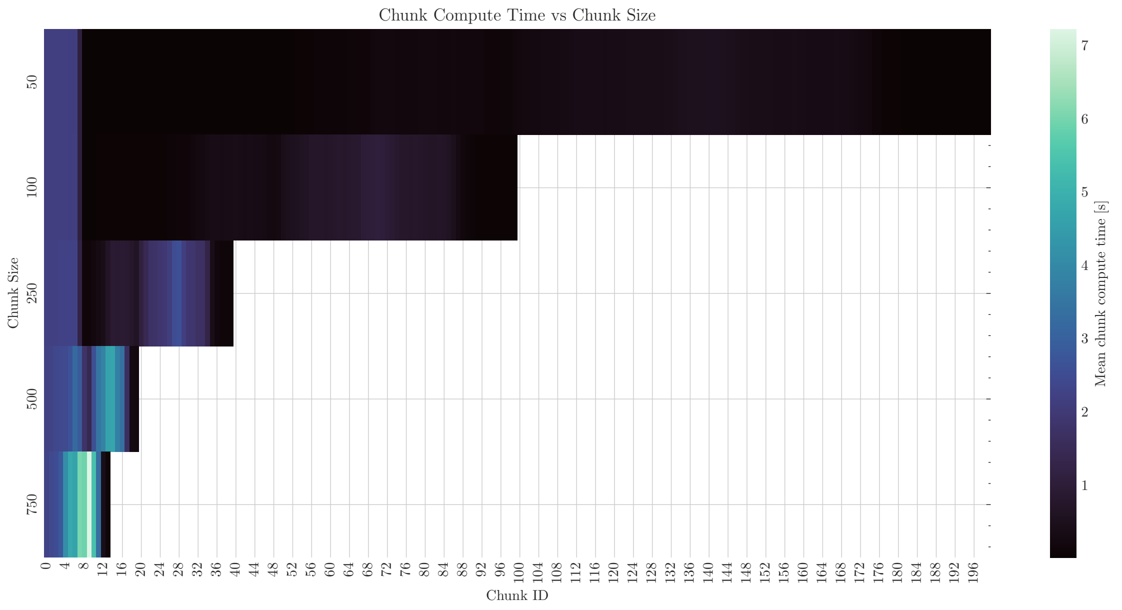

Figure 4: Heatmap of chunk computation times dependent on chunk-size.

Increasing Problem Size

To see how the implementations scale with problem size, i.e. the number of pixels to process in the image, we look at the following experiment setup:

- 16 ranks
- Chunk-size 100
- Image sizes 1000×1000, 5000×5000, 10000×10000, 20000×20000, 30000×30000 and 40000×40000
- All 4 implementation combinations
- Image limits $[-2.2, 0.75]$ and $[-1.3, 1.3]$

With this experiment, we should see how each implementation scales to larger problems - the expectation is that this scaling is linear w.r.t to the number of pixels processed, given that our implementations are correct. In figure 5, the results of this experiment can be seen, and we do indeed see a close to linear relationship between number of pixels and total runtime.

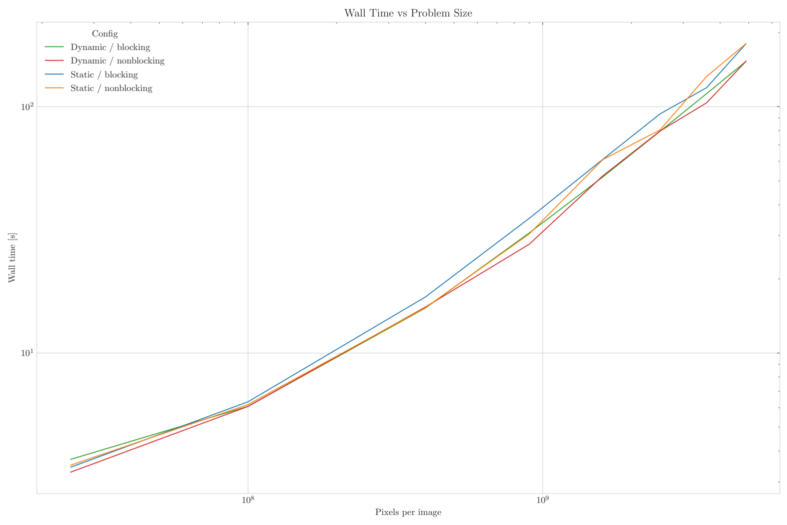

Figure 5: Scaling with increasing problem size for each implementation.

Increasing Number of Processes

When increasing the number of processes, the communication time becomes increasingly large, especially for dynamic scheduling and blocking send/receive communication. However, we could also reach a point where each rank only process very few chunks due to the amount of ranks, meaning it is especially beneficial to have the quickest finishing ranks process more chunks, in order to get the best computation times. To test how computation time and communication time develops when increasing number of ranks, we setup the experiment

- 1, 4, 8, 10, 16, 20, 24, 30, 40, 50, 60, 70, 80, 90 ranks
- Chunk-size 100
- Image-size 20000×20000
- All 4 implementation combinations
- Image limits $[-2.2, 0.75]$ and $[-1.3, 1.3]$

Having just 1 rank is of course equal to the initial Mandelbrot set calculation script, before adding MPI parallelization (but with Numba optimization). In figure 6, the wall time is plotted versus number of ranks for all 4 implementations. The results here are pretty much as expected - the most important thing to note is the sudden change in runtimes around 24-30 ranks, where we actually see an increase in runtime when moving past 24 ranks. This is due to the limitations of the system, as we only run on a single host up until 24 ranks, after which we must use multiple hosts to have sufficient resources available. This means the implementation suddenly use more communication time, to send data across several hosts. When increasing the number of ranks further, the computation speed-up dominates this effect, hence why we only see a bump right around 24-30 ranks.

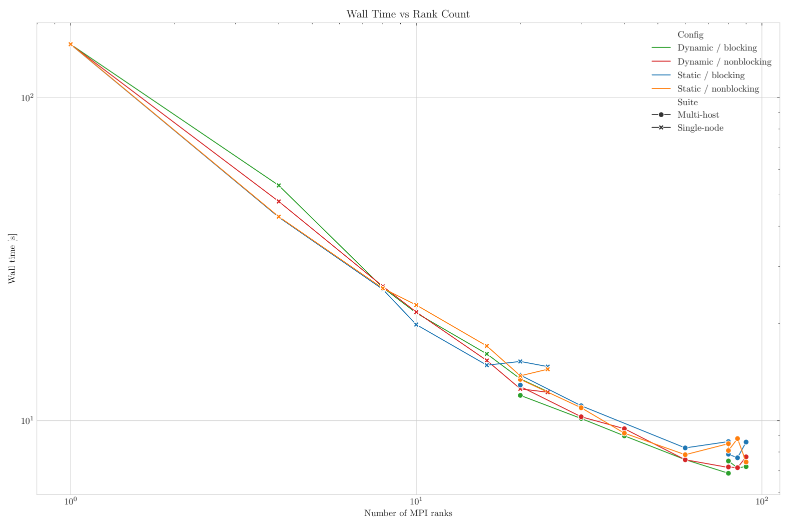

Figure 6: Scaling up to 24 nodes on a single host and up to 90 on multiple hosts.

Discussion

4.1 Discussion and Analysis

Effect of Chunk Size

The chunk size parameter directly influences the trade-off between load balancing and coordination overhead. In our implementation of the dynamic scheduler one rank is dedicated as a master, distributing work to other ranks. This reduces available workers from 8 to 7 in the 8-rank experiments. For small chunk sizes (50-100 pixels), computational variance between chunks is naturally limited as we observe in Figure 4, making the 12.5% loss in computational capacity more costly than the imbalance penalty in static scheduling. For large chunk sizes (500-750 pixels), computational variance reaches 15$\times$ between heaviest and lightest chunks. Static scheduling assigns chunks deterministically, creating situations where some ranks process multiple heavy chunks while others finish quickly and idle. Dynamic scheduling redistributes work continuously, ensuring ranks that complete easier chunks immediately receive additional work.

The crossover point where dynamic overtakes static depends on when imbalance penalty exceeds resource loss. With more ranks, this occurs at smaller chunk sizes because the fractional cost of the master rank decreases ($\frac{1}{N}$), while the probability of poor static assignment increases with more workers competing for a fixed pool of chunks.

Communication Cost and Message Patterns

Communication time in dynamic scheduling remains essentially constant across chunk sizes despite order-of-magnitude differences in message count. Chunk size 50 generates 200 work request-response cycles compared to 13 cycles for chunk size 750, yet total communication time changes negligibly. This indicates that MPI communication cost is dominated by data transfer rather than message overhead. Each rank must ultimately receive the same number of pixels regardless of how the image is partitioned. The latency of individual request-response messages contributes negligibly to total communication time compared to bandwidth-limited pixel data transfer. This cost structure makes fine-grained dynamic scheduling practical. Increasing message frequency by 15$\times$ does not increase communication overhead proportionally, allowing dynamic scheduling to use small chunks without prohibitive communication penalties.

Scaling with Problem Size

All four implementations exhibit linear scaling with problem size, as demonstrated in figure .5. The consistent spacing between implementation curves indicates that relative performance characteristics remain stable across image sizes from 10^6 to 1.6×10^9 pixels. Dynamic scheduling shows consistently higher absolute runtime than static scheduling in these experiments because it uses 15 worker ranks compared to 16 in static scheduling. At 16 total ranks with chunk size 100, this 6.25% reduction in computational resources outweighs load balancing benefits for the tested problem sizes.

The linear scaling confirms that communication overhead grows proportionally with data volume without introducing nonlinear bottlenecks. This validates the implementation correctness and indicates that the communication pattern does not create unexpected performance degradation at large problem sizes.

Scaling with Number of Ranks

The four implementations show distinct scaling behavior as rank count increases, as shown in figure .6. Below approximately 40 ranks, static scheduling outperforms dynamic scheduling across both blocking and non-blocking variants. Static scheduling benefits from having one additional worker rank, and at moderate parallelism, the computational imbalance is insufficient to justify the coordination overhead of dynamic scheduling. Around 40-50 ranks, dynamic scheduling overtakes static scheduling. The master rank overhead becomes fractionally smaller (2-2.5% of total ranks), while computational imbalance becomes more problematic in static scheduling. With many ranks and fixed chunk size, each rank processes fewer chunks on average. A single heavy chunk assigned to a rank in static scheduling creates proportionally more impact on total runtime when other ranks finish their work and idle. Beyond 60-70 ranks, all implementations plateau. With 200 total chunks and 70 workers, average chunks per rank drops below 3. The discrete nature of chunk distribution limits parallel efficiency - perfect load balancing cannot eliminate idle time when work granularity is insufficient relative to worker count. Non-blocking communication shows negligible advantage until very high rank counts (beyond 70 ranks). Computation time per chunk (seconds) dominates communication time (milliseconds) by three orders of magnitude, leaving minimal opportunity for overlap between computation and communication. Only at extreme parallelism, where chunks become very small relative to communication frequency, does non-blocking provide measurable benefit.