WORKSHEET 4: MPI COLLECTIVES AND MPI-IO

Programming of Supercomputers [IN2190]

Group 09: Gerasimos Chourdakis, Nathaniel Knapp, Walter Simson

February 5, 2016

Fakultät für Informatik - Technische Universität München

AGENDA

Catching up

Task 2: MPI Collectives

Task 3: MPI Parallel IO

Conclusions

CATCHING UP

PROVIDED IMPLEMENTATION

- Cannon's algorithm, patched.
- MPI-activated, P2P blocking communication.
- I/O handled by rank 0. (big overhead)
- Rank 0 uses blocking Send/Recv to distribute and collect data. (big overhead)
- Extended to store C in the same way as A, B and to show the input/output time (including distribution/collection).

TRACING WITH SCORE-P AND VAMPIR (BASE CASE)

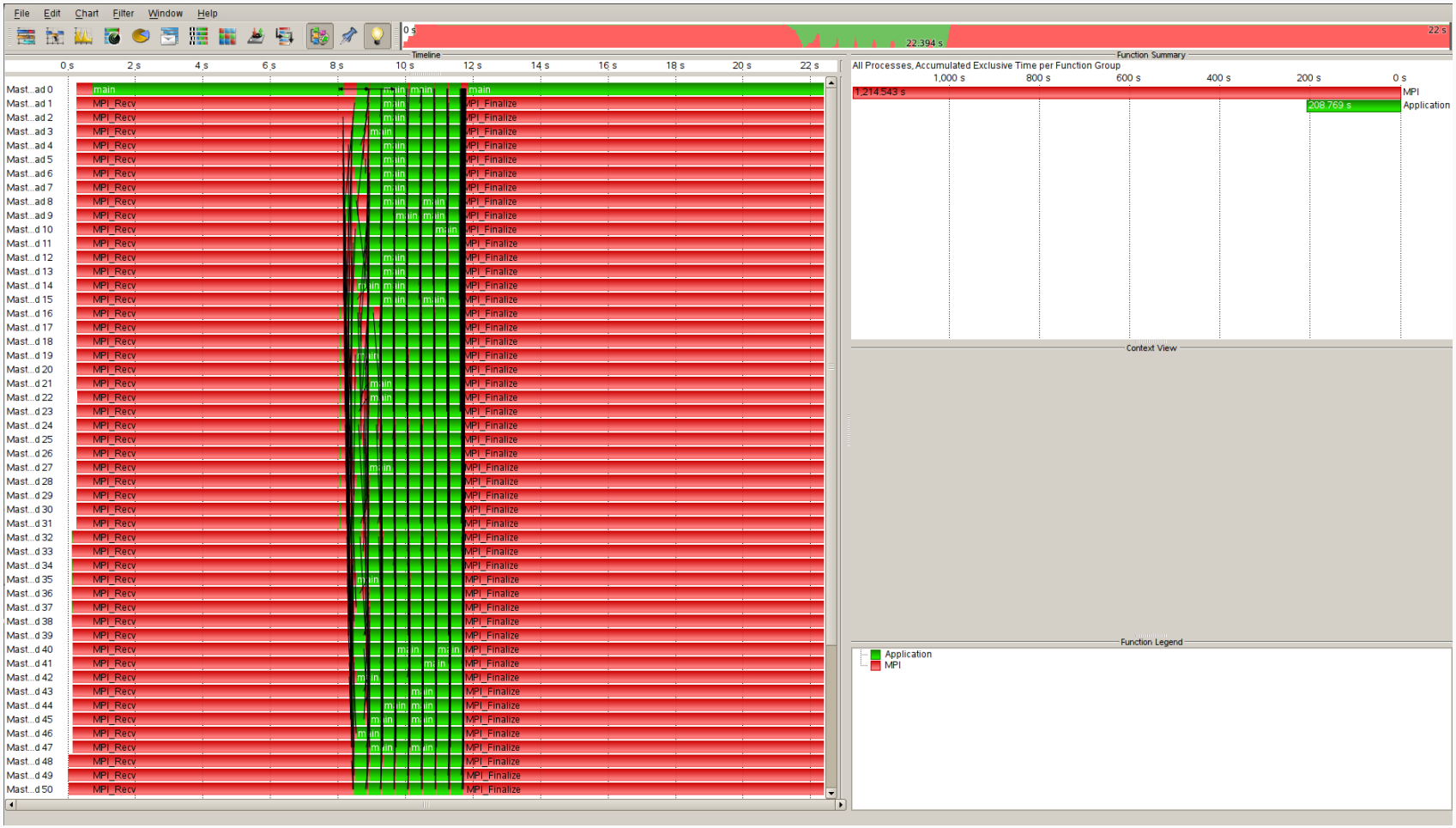


Figure: Tracing for the base case, N=4096, Sandybridge. Note the large waiting times for I/O and the non-synchronized start of the main loop.

Task 2: MPI Collectives

MPI COLLECTIVES

- The P2P blocking communication for distributing and collecting the dimensions and the blocks was replaced by calls to the `MPI_Bcast()`, `MPI_Scatter()` and `MPI_Gather()` functions.
- We didn't use non-blocking collectives since there was not any strong potential for performance gain. The code complexity would increase though.
- Some performance/scalability gain can be expected due to hardware optimizations and it is observed.
- The resulting code is much easier to understand, since the code is shorter and its purpose clearer. Also, shared code for every process.

TRACING WITH SCORE-P AND VAMPIR (COLLECTIVES)

Figure: Tracing for the collectives case, N=4096, Sandybridge. Note the cleaner traces and the synchronized/shorter start of the main loop.

Task 3: MPI parallel IO

TERMINOLOGY

”Better one and big request, rather than many and small”. The I/O time is dependent both on the latency and the bandwidth.

- **Data Sieving**: In order to make less requests, the whole file is read and then filtered. This reduces the latency but is not well scalable.
- **Two-phase I/O**: Each process needs to read many small parts of the files. For better scalability, each process reads a contiguous part of the file and then the processes rearrange the parts that they need.

The original implementation reads the whole file in one process. This is bounded by the total memory and the smallest bandwidth.

The MPI provides I/O operations that allow to read different parts of the same file in a scalable way, transparently.

SCALABILITY (BASE CASE)

The time for IO in the base case increases logarithmically. There is a memory restriction, as the Rank0 loads the complete matrices.

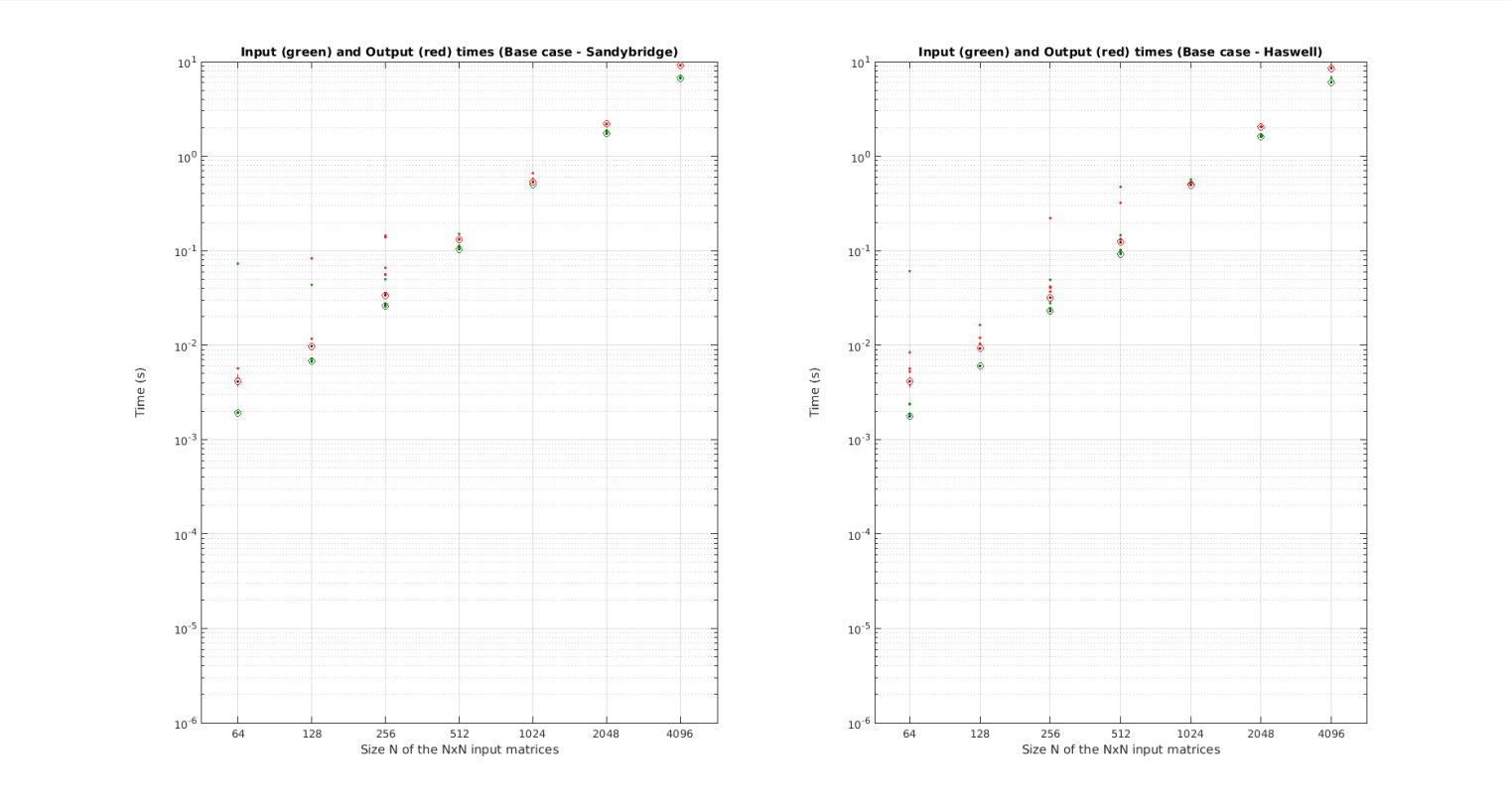


Figure: Input time (green) and output time (red) of Rank0 for Sandybridge (left) and Haswell (right). The time for distribution/collection is included. The points are medians of 30 samples. Same plot scale as in WS3.

MPI PARALLEL IO

- As we need binary files to work with, we first convert the provided input files to binaries. For this, we still have some (deactivated) legacy code.
- We use the `MPI_File_read()` in all ranks to read the header of the input files and the `MPI_File_write()` in Rank0 to write the header of the output file.
- We set a displacement equal to the size of the header.
- We create a 2D subarray to read and write the local blocks. We could also use another type, but this is intuitive.
- We commit the type and set the view according to it. We use the ”native” representation for performance.
- We read and write the body of the files using the `MPI_File_read_all()` and `MPI_File_write_all()`. Collectives allow for performance optimization.
- We created a bash shell script to convert these files to human-readable form, using `hexdump`.

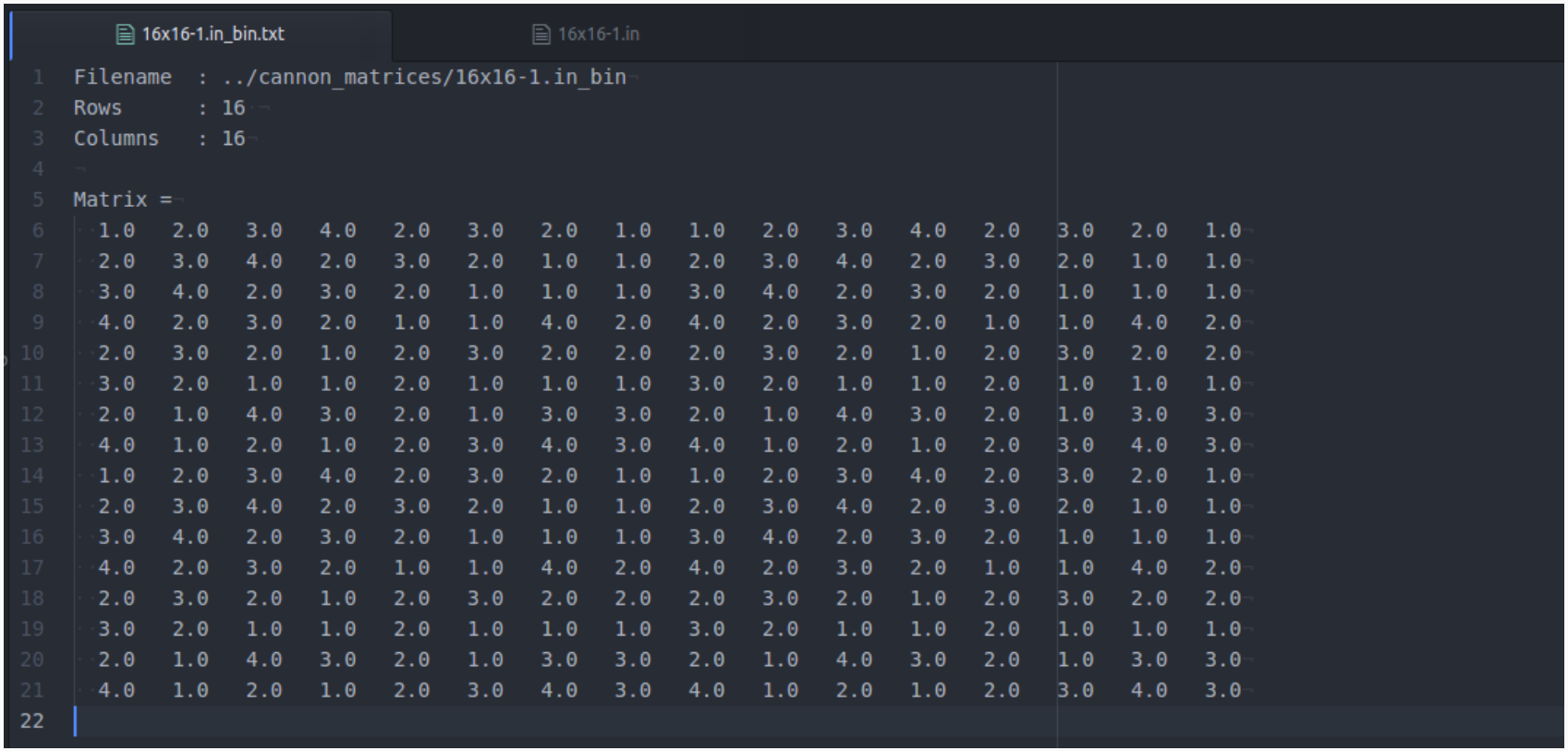
VALIDATION (MPI IO) - INPUT

16x16-1.in_bin.txt | 16x16-1.in

```
16 16
1.0  2.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0  2.0  3.0  4.0  2.0  3.0  2.0  1.0
2.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0  2.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0
3.0  4.0  2.0  3.0  2.0  1.0  1.0  1.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0  1.0
4.0  2.0  3.0  2.0  1.0  1.0  4.0  2.0  4.0  2.0  3.0  2.0  1.0  1.0  4.0  2.0
2.0  3.0  2.0  1.0  2.0  3.0  2.0  2.0  2.0  3.0  2.0  1.0  2.0  3.0  2.0  2.0
3.0  2.0  1.0  1.0  2.0  1.0  1.0  1.0  3.0  2.0  1.0  1.0  2.0  1.0  1.0  1.0
2.0  1.0  4.0  3.0  2.0  1.0  3.0  3.0  2.0  1.0  4.0  3.0  2.0  1.0  3.0  3.0
4.0  1.0  2.0  1.0  2.0  3.0  4.0  3.0  4.0  1.0  2.0  1.0  2.0  3.0  4.0  3.0
1.0  2.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0  2.0  3.0  4.0  2.0  3.0  2.0  1.0
2.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0  2.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0
3.0  4.0  2.0  3.0  2.0  1.0  1.0  1.0  3.0  4.0  2.0  3.0  2.0  1.0  1.0  1.0
4.0  2.0  3.0  2.0  1.0  1.0  4.0  2.0  4.0  2.0  3.0  2.0  1.0  1.0  4.0  2.0
2.0  3.0  2.0  1.0  2.0  3.0  2.0  2.0  2.0  3.0  2.0  1.0  2.0  3.0  2.0  2.0
3.0  2.0  1.0  1.0  2.0  1.0  1.0  1.0  3.0  2.0  1.0  1.0  2.0  1.0  1.0  1.0
2.0  1.0  4.0  3.0  2.0  1.0  3.0  3.0  2.0  1.0  4.0  3.0  2.0  1.0  3.0  3.0
4.0  1.0  2.0  1.0  2.0  3.0  4.0  3.0  4.0  1.0  2.0  1.0  2.0  3.0  4.0  3.0
```

Figure: Text input file for A (16x16), as given.

VALIDATION (MPI IO) - INPUT



16x16-1.in_bin.txt | 16x16-1.in

```
Filename  : ../cannon_matrices/16x16-1.in_bin
Rows      : 16
Columns   : 16

Matrix =
  1.0   2.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0   2.0   3.0   4.0   2.0   3.0   2.0   1.0
  2.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0   2.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0
  3.0   4.0   2.0   3.0   2.0   1.0   1.0   1.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0   1.0
  4.0   2.0   3.0   2.0   1.0   1.0   4.0   2.0   4.0   2.0   3.0   2.0   1.0   1.0   4.0   2.0
  2.0   3.0   2.0   1.0   2.0   3.0   2.0   2.0   2.0   3.0   2.0   1.0   2.0   3.0   2.0   2.0
  3.0   2.0   1.0   1.0   2.0   1.0   1.0   1.0   3.0   2.0   1.0   1.0   2.0   1.0   1.0   1.0
  2.0   1.0   4.0   3.0   2.0   1.0   3.0   3.0   2.0   1.0   4.0   3.0   2.0   1.0   3.0   3.0
  4.0   1.0   2.0   1.0   2.0   3.0   4.0   3.0   4.0   1.0   2.0   1.0   2.0   3.0   4.0   3.0
  1.0   2.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0   2.0   3.0   4.0   2.0   3.0   2.0   1.0
  2.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0   2.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0
  3.0   4.0   2.0   3.0   2.0   1.0   1.0   1.0   3.0   4.0   2.0   3.0   2.0   1.0   1.0   1.0
  4.0   2.0   3.0   2.0   1.0   1.0   4.0   2.0   4.0   2.0   3.0   2.0   1.0   1.0   4.0   2.0
  2.0   3.0   2.0   1.0   2.0   3.0   2.0   2.0   2.0   3.0   2.0   1.0   2.0   3.0   2.0   2.0
  3.0   2.0   1.0   1.0   2.0   1.0   1.0   1.0   3.0   2.0   1.0   1.0   2.0   1.0   1.0   1.0
  2.0   1.0   4.0   3.0   2.0   1.0   3.0   3.0   2.0   1.0   4.0   3.0   2.0   1.0   3.0   3.0
  4.0   1.0   2.0   1.0   2.0   3.0   4.0   3.0   4.0   1.0   2.0   1.0   2.0   3.0   4.0   3.0
```

Figure: Binary input file for A (16x16), after conversion.

VALIDATION (MPI IO) - OUTPUT

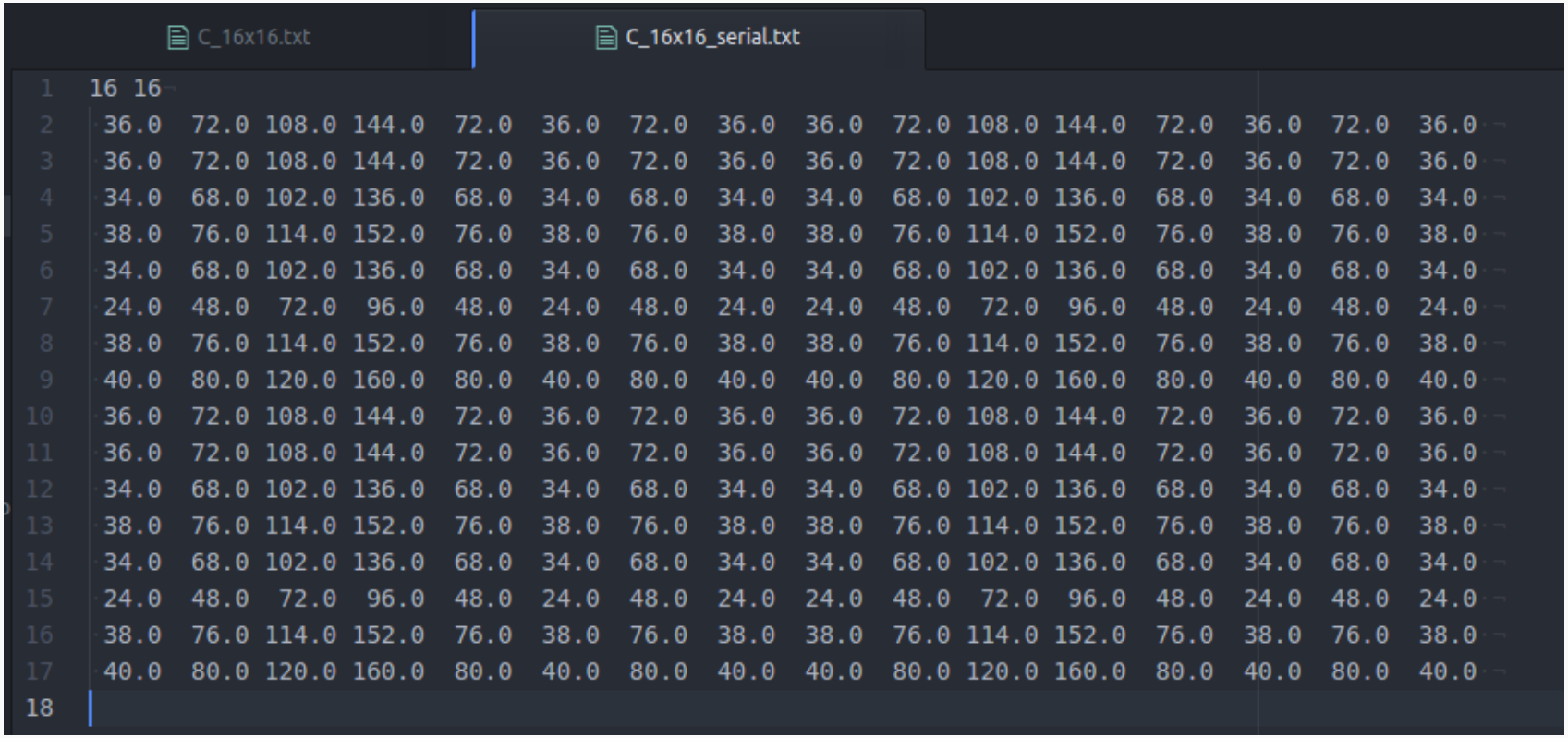
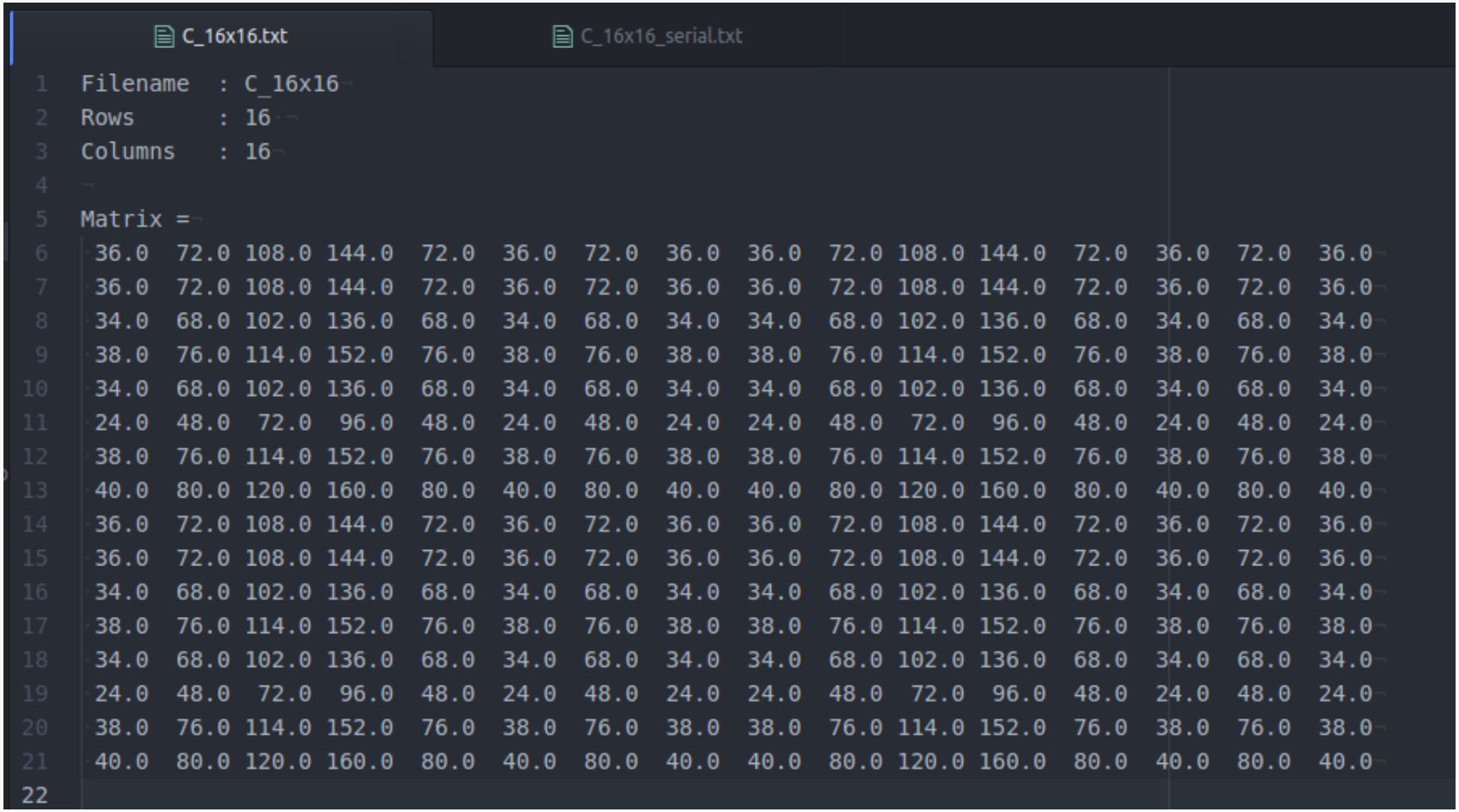


Figure: Text output file for C (16x16), from the base case.

VALIDATION (MPI IO) - OUTPUT



C_16x16.txt | C_16x16_serial.txt

```
Filename  : C_16x16
Rows      : 16
Columns   : 16

Matrix =
 36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0  36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0
 36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0  36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0
 34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0  34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0
 38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0  38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0
 34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0  34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0
 24.0  48.0  72.0  96.0  48.0  24.0  48.0  24.0  24.0  48.0  72.0  96.0  48.0  24.0  48.0  24.0
 38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0  38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0
 40.0  80.0 120.0 160.0  80.0  40.0  80.0  40.0  40.0  80.0 120.0 160.0  80.0  40.0  80.0  40.0
 36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0  36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0
 36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0  36.0  72.0 108.0 144.0  72.0  36.0  72.0  36.0
 34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0  34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0
 38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0  38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0
 34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0  34.0  68.0 102.0 136.0  68.0  34.0  68.0  34.0
 24.0  48.0  72.0  96.0  48.0  24.0  48.0  24.0  24.0  48.0  72.0  96.0  48.0  24.0  48.0  24.0
 38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0  38.0  76.0 114.0 152.0  76.0  38.0  76.0  38.0
 40.0  80.0 120.0 160.0  80.0  40.0  80.0  40.0  40.0  80.0 120.0 160.0  80.0  40.0  80.0  40.0
```

Figure: Binary output file for C (16x16), after conversion.

SCALABILITY (MPI IO)

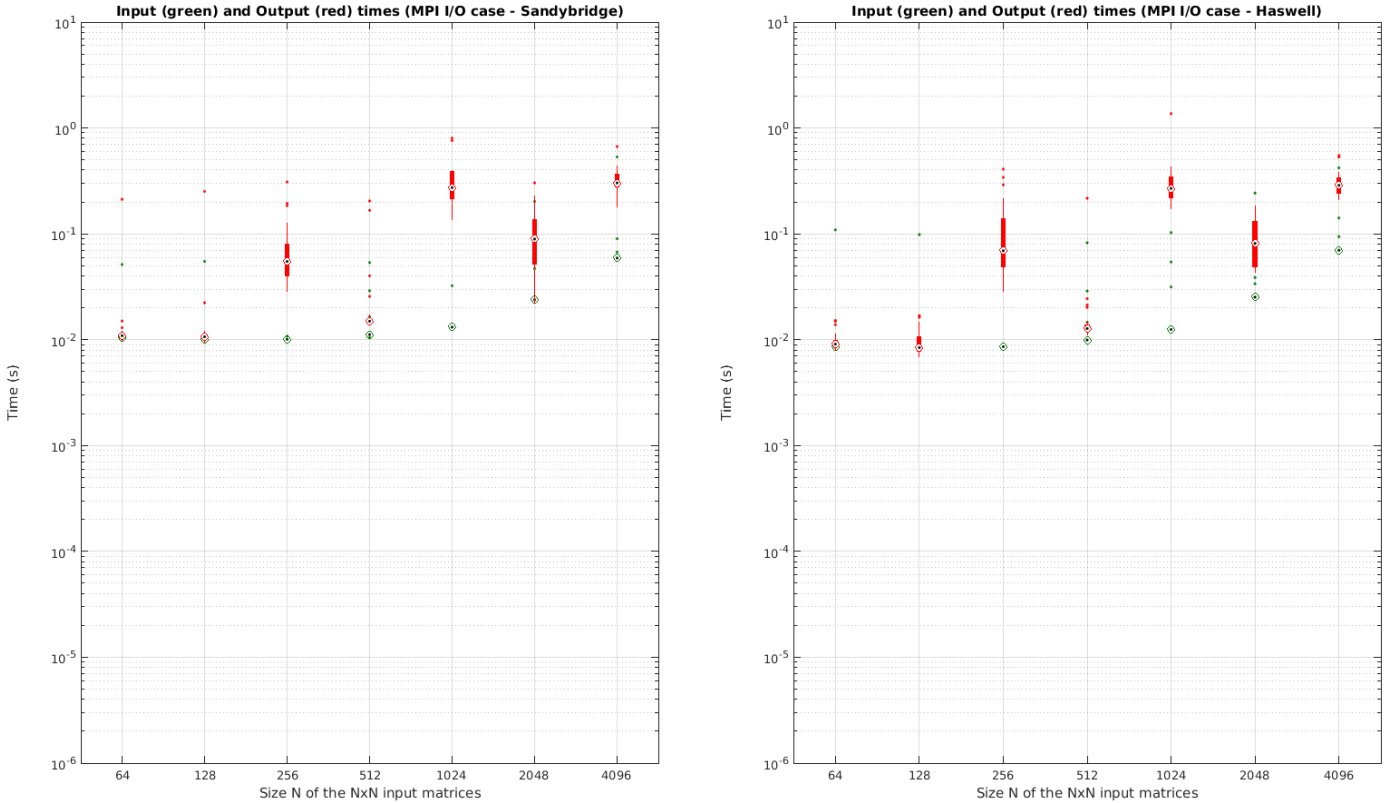
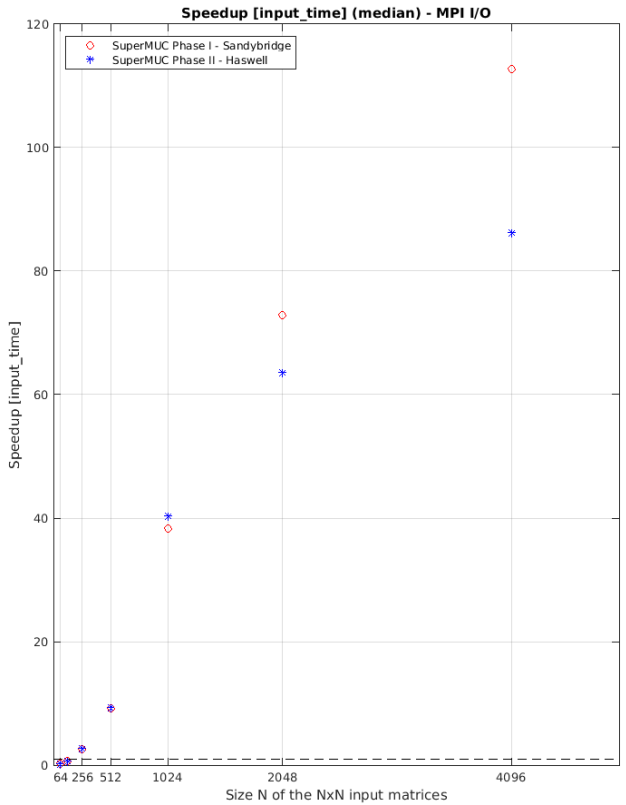


Figure: Input time (green) and output time (red) of Rank0 for Sandybridge (left) and Haswell (right). The points are medians of 30 samples. The reading time is smaller and more consistent than writing. It increases much slower than in the base case.

SCALABILITY (MPI IO)



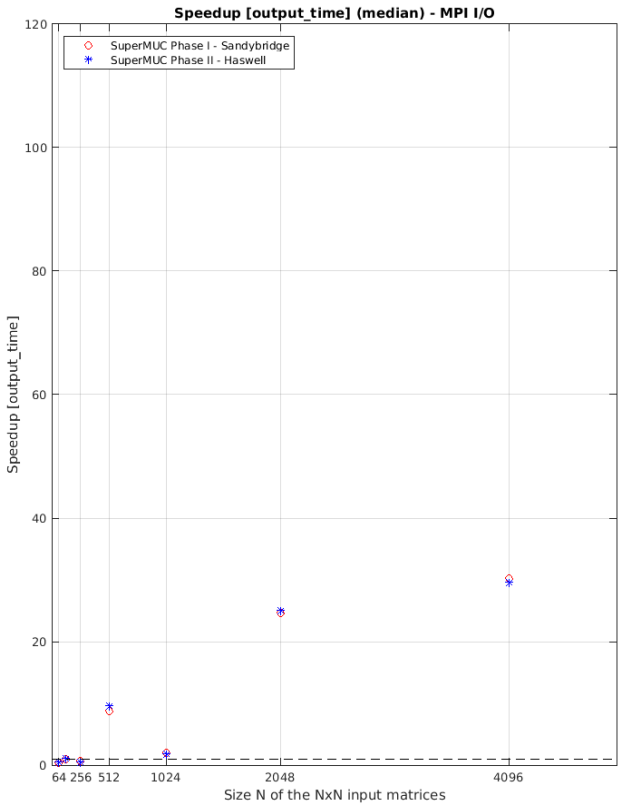


Figure: Speedup of the input (left) and output (right) time for Sandybridge (circles) and Haswell (stars). The points are medians of 30 samples. The reading time scales better. The writing time has some local extrema.

TRACING WITH SCORE-P AND VAMPIR (MPI IO)

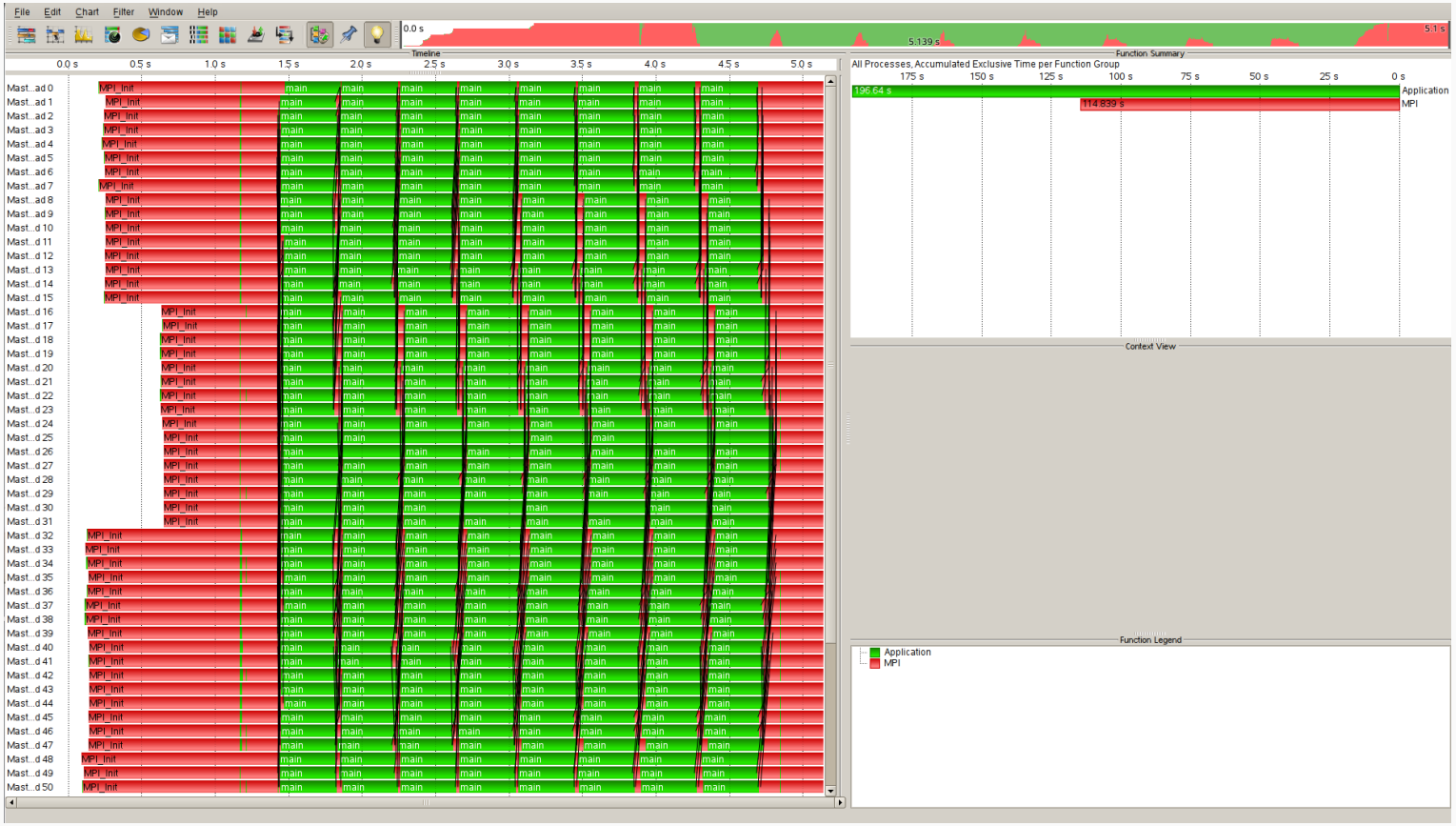


Figure: Tracing for the MPI IO case, N=4096, Sandybridge. Note the much smaller input, output and total time. Blocking P2P is used in the main loop. MPI time for IO before: 96% of the total MPI time, now: 38% (approx.).

CONCLUSIONS

CONCLUSIONS

- The MPI Collectives make the code clearer and can provide some performance and scalability improvement.
- The provided implementation is not well scalable and has big IO overheads.
- MPI IO can dramatically decrease these overheads and increase scalability.
- For smaller files, the provided implementation works better.
- Reading is faster, more time-consistent and better scalable than writing.
- SuperMUC Phase I and Phase II give very similar results.
- Is our implementation truly scalable up to truly big problems? Maybe a different approach (e.g. hierarchical) will be needed in that case.

Thank you very much!