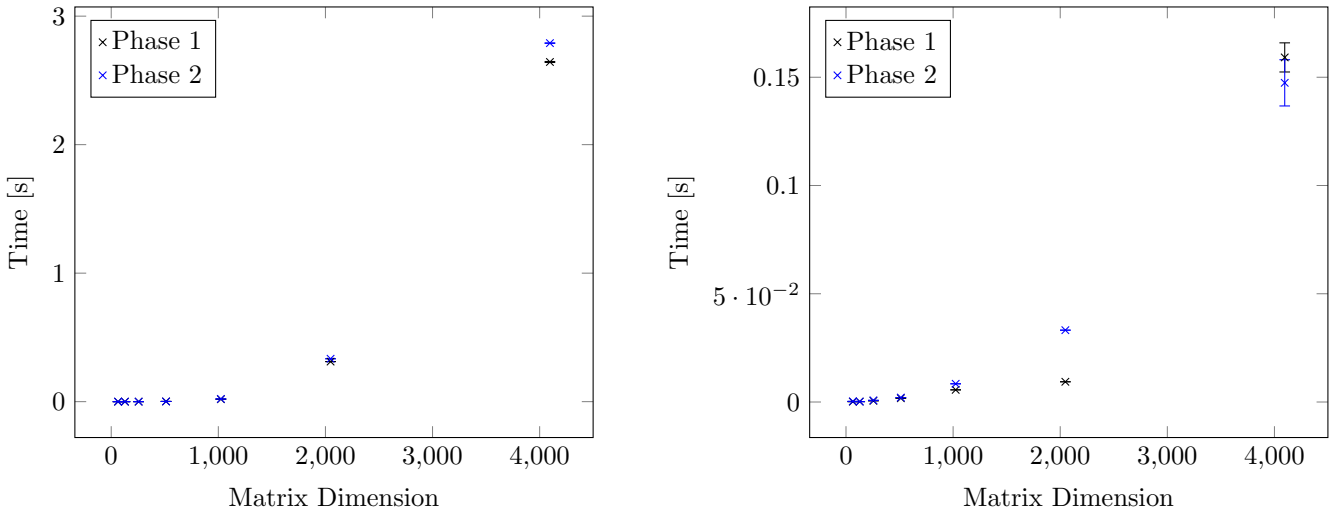
Assignment 3:
MPI Point-to-Point and One-Sided Communication

Emily Bourne (emily.bourne@tum.de)
Abraham Duplaa (abraham.duplaa@tum.de)
Jiho Yang (jiho.yang@tum.de)

January 11, 2018



(a) Computational time baseline

(b) Computational time baseline

Figure 1: Baseline

1 Bug Fixing and Baseline

1. **What was the bug in the provided implementation?**
   The setup step for Cannon's algorithm was missing. In this step the starting block matrices are rotated such that the processor with block (i,j) receives block (i,mod(j+i,n_columns)) from matrix A and block (mod(i+j,n_rows),j) from matrix B where i=0,...,n_rows-1, and j=0,...,n_columns-1.

2. **Where was the code modified to fix the bug?**
   After line 183, the following code was added:

```
// Skew matrices:
for (int i=0; i<coordinates[0]; ++i) {
    // rotate blocks horizontally
    MPI_Sendrecv_replace(A_local_block, A_local_block_size, MPI_DOUBLE,
            (coordinates[1] + sqrt_size - 1) % sqrt_size, 0,
            (coordinates[1]+1) % sqrt_size, 0, row_communicator, &status);
}

for (int j=0; j<coordinates[1]; ++j) {
    // rotate blocks vertically
```

```
    MPI_Sendrecv_replace(B_local_block, B_local_block_size, MPI_DOUBLE,
            (coordinates[0] + sqrt_size - 1) % sqrt_size, 0,
            (coordinates[0] + 1) % sqrt_size, 0,
            column_communicator, &status);
}
```

Line 280 was also modified to allow for numerical errors. The line was replaced with the following code:

```
if(fabs(temp - C[i][j])>1e-15){
```

3. **Why was the bug not detectable with the provided input set?**

   Without the setup step the matrix multiplication gives the following values:

$$R_{ij} = \sum_k A_{i,\ \bmod (j+k,n)} B_{\ \bmod (i+k,n),j}$$

   where R is the false result of the broken Cannon algorithm.

   The files named NxN-2.in all have the same values in the columns. This means that $A_{ik} = A_{jk}\ \forall i,j$. This means that :

$$\begin{aligned} R_{ij} &= \sum_k A_{i,\ \bmod (j+k,n)} B_{\ \bmod (i+k,n),j} \\ &= \sum_k A_{i,\ \bmod (j+k,n)} B_{\ \bmod (j+k,n),j} \\ &= \sum_l A_{i,l} B_{l,j} = C_{i,j} \end{aligned}$$

   Where C is the correct result of the matrix multiplication.

   The files named NxN-1.in all have repeating values in the columns and rows. Thus unless a block size of less than 8x8 is used all blocks are identical. This results in a similar situation to the files named NxN-2.in except that the block can play the role of A or B.

4. **How does compute time scale with fixed 64 processes and varying size of input files?**

   On both types of nodes the compute time scales quadratically with fixed 64 processes and varying size of input files. This makes sense as the problem size also scales quadratically with the matrix dimension.

5. **How does MPI time scale with fixed 64 processes and varying size of input files?**

   On Sandy Bridge MPI time scales linearly for sizes less than 1024 but loses this linearity for 2048x2048 matrices. On Haswell MPI time scales quadratically.

6. **Are there any scalability differences between the Sandy Bridge and Haswell architectures?**

   The scalability of the MPI time is different between the Sandy Bridge and Haswell architectures. On Haswell MPI time scales quadratically. On Sandy Bridge MPI time scales linearly for sizes less than 1024, however for 2048x2048 matrices the results once more resemble those achieved on Haswell. This seems surprising but it should be noted that Sandy Bridge has a better total Linpack performance [1] than Haswell, so it makes sense that it gives somewhat better results.

7. **Other than the bug, does the provided implementation have any other practical limitations?**

   The code can only accept matrices where both dimensions are a multiple of the square root of the number of processors used. The matrices are then split into blocks. The size of a given dimension of these blocks is equal to the size of the dimension on the full matrix divided by the square root of the number of processors used. This means that matrix multiplications where one of the dimensions is a prime number cannot be parallelised and nor can matrices whose dimensions do not have a common factor other than one.

   In cases where one dimension is much larger than the other it could be preferable to only split the matrix into blocks along one dimension. This is also not possible.

2 MPI Point-to-Point Communication

2.1 Introduction to Non-Blocking Point-to-Point Communication

Unlike MPI Point-to-Point *blocking* operations which only terminate when the construct (eg. MPI_Send, MPI_Recv) buffer can be reused, MPI Point-to-Point *non-blocking* operations terminate immediately after the MPI request is created. This allows for local computation to be overlapped with communication, which should hopefully provide a decrease in total computing time. In order to understand how to use non-blocking operations, many non-blocking operations were investigated. Here are a couple common non-blocking operations:

- MPI_Isend
- MPI_Irecv
- MPI_Issend
- MPI_Irsend

The operations used in the code specifically were MPI_Isend and MPI_Irecv:

- **MPI_Isend**: This operation starts a standard-mode, nonblocking send. Nonblocking calls allocate a communication request object and associate it with the request handle (the argument request). This request handle is what is later checked to ensure the send has been fully executed. When a nonblocking send call is done, this is when data may start being copied out of the send buffer. The send buffer should only be accessed once the send completes.
- **MPI_Irecv**: This operation starts a nonblocking receive and returns a handle that is also used later on in the code to check the status of receiving the message. The receive buffer should only be accessed when the receive is complete.

Although nonblocking operations can reduce overall time by overlapping local computation with MPI communication, there must be a synchronization point shortly after the local computation to ensure the communication has completed and the buffer is re-available. Only after this MPI *'send/receive complete'* operation is it safe to access the send and receive buffers once again. There are many MPI operations that can be used to check whether the send and receive have completed. In order to choose the right one, the situation must be evaluated (amount of sends and receives, type of send and receive, etc.). Here are a few of those operations:

- MPI_Wait: This operation blocks until the request was executed.
- MPI_Waitall: This operation blocks until all requests are executed. We used this MPI blocking operation for synchronization within every loop.
- MPI_Test: Tests for the completion of a request

In order to be able to ensure the blocks have been sent and received before the upcoming iteration, we have used MPI_Waitall (as shown above). We chose MPI_Waitall because it can handle multiple requests from multiple MPI operations. Since there are two MPI_Isend and two MPI_Irecv, MPI_Waitall will block until all four requests have been executed.

2.2 Performance Analysis of Non-Blocking Point-to-Point Communication

The relationship between compute time and MPI time change as the matrix dimension (N) continues to increase, as shown in Figure 2. As expected, the compute time increases cubicly with increasing matrix dimension size. Compute time is initially less than MPI time for both Haswell and Sandybridge but as matrix dimension increases, compute time and MPI time cross at around matrix dimension N=512. This crossing of MPI time and compute time is very significant. From this point on, the program becomes **compute bounded**.

At N = 1024, MPI time decreased on both Haswell and Sandybridge, which may be caused by efficient memory usage. N = 1024 seems to be a critical point for MPI time, because MPI time increases at a faster rate after this point, for N = 2048 and N = 4096. It is important to note that non-blocking point-to-point communication performed well on both Haswell and Sandybridge, where as this was not observed when using single-point communication.

Once the program becomes highly compute bounded, overlapping should also increase. As seen in table 1, the difference between MPI time and total time are significant for non-blocking operations. In order to calculate

MPI time accurately, Intel Trace Analyzer was used to break down the computation and record MPI times. For the largest case (N = 4096), the trace analysis on the Sandybridge (refer to table 1) shows that only 0.61% of the total time was spent on MPI. Haswell also displayed significant overlap with 2% of the total time being spent on MPI.

When analyzing Figure 7 and Figure 8, we can visually see the differences between non-blocking and blocking operations. The black lines represent messages sent between processes. These lines connect sending and receiving processes. The messages are sent from one MPI section (in red) to another, and it is important to note that the send and receive occur during the application time, and then the MPI section (in red) is when MPI_Waitall is checking whether the requests were executed.

Although the overlap is high, especially for large matrices, this did not necessarily correlate to a better speedup. For Sandybridge, the reverse was seen with speedup. The speedup was high for smaller matrix sizes, and high once again for N = 1024. Haswell had a speedup of < 1 for smaller matrices, but then experienced a large speedup at N = 1024.

2.3 Code Discussion and Future Suggestions

In order to be able to send and receive at the same time, we created two additional buffers to hold the values being sent. As seen in the code, we create two buffers, A_buf_new and B_buf_new. These new buffers are the address spaces to which MPI_Isend sends the local block to another rank.

With each rank sending their respective block to the new buffer addresses, each rank should now contain four blocks, two of which are old and two of which are new in the additional buffers. MPI_Waitall is then called to ensure all requests have been executed. Finally, the old buffers (A_local_block and B_local_block) are replaced with the values in the new buffers and now the loop starts over.

In order to optimize the code further, the first and last iterations could be peeled off to facilitate request management and minimize the amount of branches.

Q1) Which non-blocking operations were selected by your group?
The non-blocking operations selected were MPI_Isend and MPI_Irecv. We chose these operations since MPI_Isend and MPI_Irecv satisfied the needs of the code and seemed to be able to provide the most amount of overlap.

Q2) What is the theoretical maximum overlap that can be achieved?
Assuming there is negligible overhead when using non-blocking operations, the theoretical maximum overlap would be 100%. For a compute-bounded problem, the MPI communication would theoretically be able to send and receive entirely during the computation time. For larger matrix sizes where the cannon algorithm becomes compute-bounded, we should theoretically have 100% overlap meaning communication would be happening during computation and end before the computation has finished.

Q3) Was communication and computation overlap achieved?
Yes, there was in fact overlap, as shown by the apparent speedups and also by using Trace Analyzer to analyze the sending and receiving of data (Please see Table 1 and Figure 7, Figure 8). However, it is important to note that architecture and the amount of computation necessary greatly varies the overlap. 100% overlap was almost achieved when using SandyBridge on the largest matrix.

Q4) Was a speedup observed versus the baseline?
Speedup here is defined as $time(baseline)/time(non-blocking)$.
Yes, speedup was observed. As shown in fig. 4, The blue points display the speedup of non-blocking on both sandybridge and haswell.
For Haswell nodes, nonblocking performs slower than the baseline for small matrices but as matrix size increases, there is an insignificant yet recordable speedup. This is shown in the from matrices of size 512 to size 4096.
The opposite occurs in Sandybridge. The total time speedup is greater for smaller matrices but then speedup decreases and hovers at around 1 as matrix size increases.

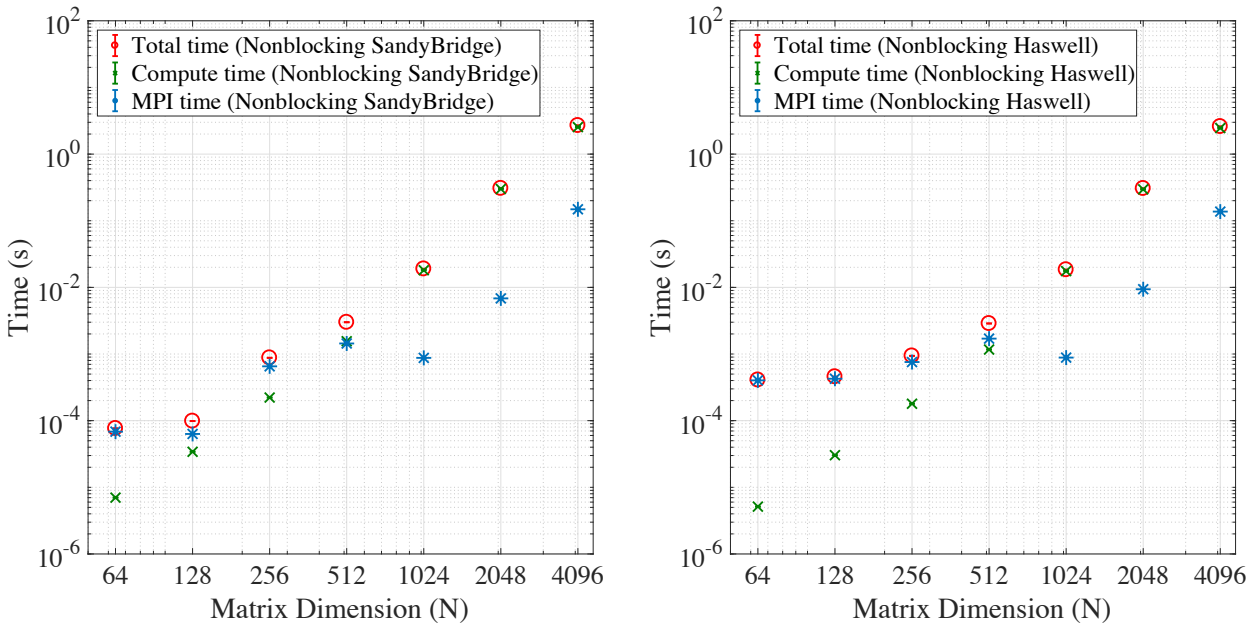


Figure 2: Total, MPI, and compute times measured from Non-blocking communication using Sandybridge (left) and Haswell (right). Times are measured exclusively for the cannon's algorithm block. Total time measures the whole time from the entry to the cycle loop to the end of the loop. MPI times are measured and accumulated exclusively when MPI operations are called. Compute times are measured and accumulated around the computation loop. It was executed 15 times and average, and variance were computed, and added to the plots.

Q5) Were there any differences between Sandy Bridge and Haswell nodes?
There were differences between both. As described in the previous question, the nodes had opposite speedup patterns with matrix size.

Furthermore, Sandybridge on average outperformed Haswell. The average total-time speedup on Sandybridge for all the matrices was 1.369, whereas the average total-time speedup on Haswell was 0.999.

Also, Sandybridge showed better overlap in the large matrix case. This could be attributed to Sandybridge architecture or because of having more cache memory. Furthermore, Sandybridge performed better on the linpack [1] than Haswell, which could be a further point to Sandybridge generally performing better when using non-blocking MPI operations.

3 MPI One-Sided Communication

One-sided communication is a very powerful feature supported by MPI in order to reduce communication time, achieve communication-computation time overlap, and reduce synchronisation costs and intermediate buffering. It supports non-blocking communications, but unlike point-to-point communication, one-sided communication is based on Remote Memory Access (RMA). The basic idea of one-sided communication models is (other than decoupling data movement with process synchronisation) to make each process expose a part of its memory to other processes, such that other processes can *directly* read from or write to this memory. This has low protocol overhead since no message is being received, but a process has access to memory address of other processes to which it wishes to write/read memory on/from. This mimics the shared memory system, therefore requires synchronisation to guarantee the communications were executed correctly. First, a memory block intended to be shared with other processes is allocated locally for each process (this works like buffers from non-blocking point-to-point communication). Then these memories are grouped into a *Window*, so when doing one-sided communication, it is sufficient that the user specify the rank to which it wishes to send to and the window. Once the windows are created (and hence memory blocks are allocated to the window), these memory spaces become remotely accessible. One-sided communications are required to be executed between synchronisation (e.g. MPI_Win_Fence). First synchronisation allows other processes to remotely access the shared blocks, and

the second synchronisation works as a barrier. Instead of sending/receiving messages, one-sided communication *puts*/*gets* the data to/from the *shared* memory block.

3.1 List of One-Sided Operations Used

This section summarises the function description of the one-sided operations used for code implementation. For details of the code please refer to Section. 3.2.

- MPI_Win_create : Create an MPI Window object for one-sided communication. It must specify the memory space intended to be remotely accessible, and window to where it is to belong. After the creation windows become ready to be remotely accessible, but only after MPI_Win_fence they will be accessible.
- MPI_Win_free: Frees the windows created
- MPI_Win_fence: Synchronises windows. It must specify the window the synchronise. Every one-sided operations must be performed in between this function. The first fence call initiates the remote access to the windows, and the second fence call synchronises the window.
- MPI_Put: Remotely writes the block of memory (local) to the remotely accessible memory spaces within window (shared). Rank and window must be specified. Since the memory space created for shared use is grouped by the window, and rank is specified, rank and windows are sufficient to locate the *shared* memory space.

3.2 Cannon's Algorithm with One-Sided Communication

Blocking communications were replaced by one-sided communications. MPI_Win_create was used to create the windows and set the memory blocks remotely accessible. MPI_Win_fence was used to start the remote memory access. MPI_Put was used to *put* the local memory to target ranks' remotely accessible memory space (one could also use MPI_Get). While the processes are *putting* their data to target ranks' memory, the computation loop is executed. Once the computation is done, MPI_Win_fence is used again to synchronize the processes and also to stop remote memory access, and then the data from shared memory block is retrieved to the local memory block for next loop (note these are in the same physical memory). For details, please refer to the attached cannon_onesided.c. This is a very straight-forward implementation of one-sided communication, and considering the purpose of the assignment, this approach was taken. However, in order to improve the performance, one could consider using two shared blocks alternately instead of one. Two windows could be created for each matrix, hence two shared memory blocks per matrix. After sending data to both of the shared memory blocks, from the main iteration block use the loop index to swap between the windows, such that it makes second one always one step ahead than the first. This way it will be possible to fence the individual windows less frequently, hence allow more time for communication completion. Another approach would be to use local group based synchronisation. Since MPI_Win_fence requires the synchronisation of the entire communicator (MPI_Comm_World) [2], this becomes a complete RMA synchronisation operation over all ranks. Despite using one-sided commnication, global locking from MPI_Win_fence slows everything down significantly. Instead of using global synchronisation (MPI_Win_fence), one could consider other APIs such as MPI_Win_post, MPI_Win_start, MPI_Win_complete, etc, such that only local synchronisation is performed. However, this would require significant optimisation and code refactoring.

3.3 Results and Discussion on One-sided Communication

Table. 1 provides a summary of application and MPI times retrieved from Intel Trace Analyzer for matrix dimension N=4096. The data is taken from a segment of the code where cannon's algorithm is executed. Application time refers to the time taken to do operations other than MPI communication (this includes compute time), MPI time refers to the communication time, and total time is the sum of these two. In order to quantitatively check for overlap, the percentage of MPI time with respect to the total time is observed, since the less time you spend on MPI time the more time is spent on other useful operations, hence achieve overlap. While this percentage was 4.25 and 7.06 % on Sandybridge and Haswell, respectively for the baseline, for the one-sided communication it took 2.97 and 10.21 % on Sandybridge and Haswell, respectively. For Sandybridge, it clearly shows that there has been a good overlap. However, for Haswell node, poorer overlap performance was achieved. Despite this, it is difficult to state that one-sided communication version did not achieve overlap (and besides, good overlap was obtained from the Sandybridge node) since we are using proprietary MPI and the only change is the use of different APIs in MPI. Hence, it would be more appropriate to state that using one-sided communication achieves good overlap although the one-sided MPI APIs used for our implementation somehow affect negatively on Haswell (and again, it is difficult to figure out what's causing this). It's worth noting that

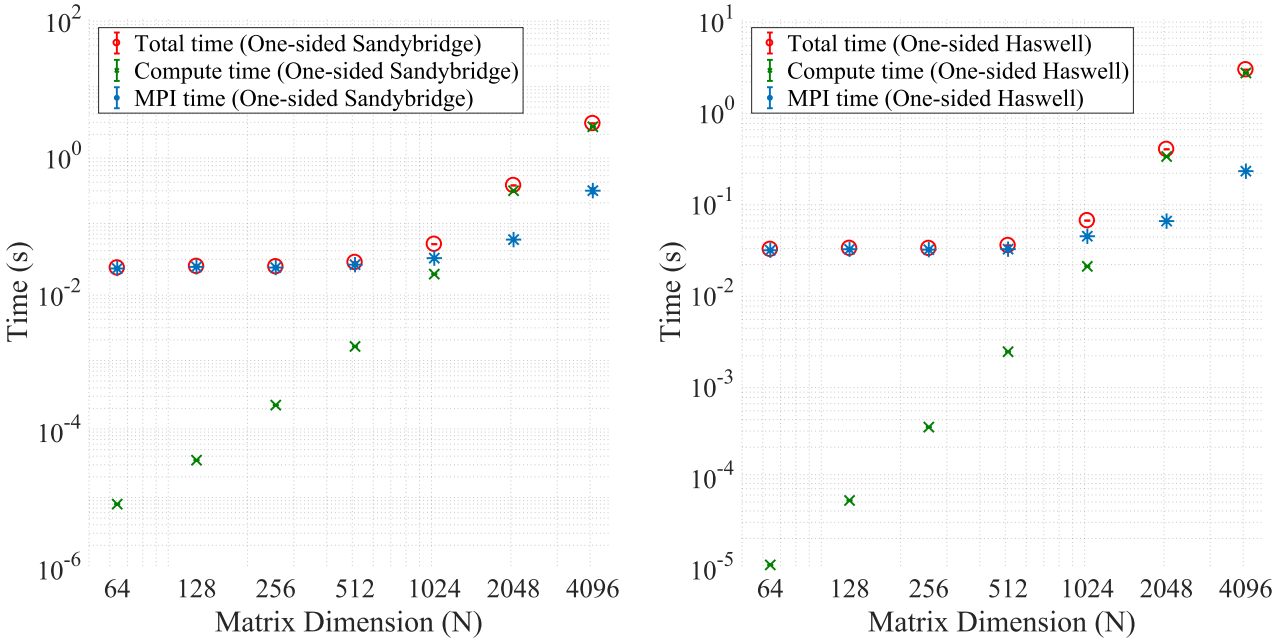


Figure 3: Total, MPI, and compute times measured from One-sided communication using Sandybridge (left) and Haswell (right). Times are measured exclusively for the cannon's algorithm block. Total time measures the whole time from the entry to the cycle loop to the end of the loop. MPI times are measured and accumulated exclusively when MPI operations are called. Compute times are measured and accumulated around the computation loop. It was executed 15 times and average, and variance were computed, and added to the plots.

the total time increased for one-sided version on both of the nodes. As mentioned previously, this is likely to be caused by the global locking from MPI_Win_fence. Figure. 5 and 6 depict the comparison of trace analysis from the baseline and one-sided communication. Figure. 4 depicts the speedup of total and MPI times from one-sided and non-blocking communication on Phase 1 and Phase 2 nodes. Compute time was excluded for this study since for non-blocking operations while doing the computation MPI communication may still happen at the background. Therefore, if compute time is dominant on total time, compute time is not suitable to study how non-blocking operations affect the performance. Total time, on the other hand, since the purpose of using non-blocking communication is to improve the overall performance must be studied in order to see how non-blocking communication improved the performance. MPI time is considered (note, this is measured exclusively around MPI functions), to check how the overheads differ from blocking and non-blocking MPI operations. From the results we can observe one-sided communication slowed down the overall performance of the code, although it achieved good overlap, and also one-sided operations cause more overhead than blocking operations. However, as matrix dimension increases compute time becomes more dominant than MPI time, hence the speedup tends towards 1. In comparison to non-blocking communication, it is also clear that one-sided communication gives worse performance than non-blocking communication. However, it may be possible to gain better performance than non-blocking communication by implementing an optimised version as described above. Figure. 3 depicts the total, compute, and mpi times taken from one-sided communication version of the code on both Sandybridge and Haswell (please note variance is included in the plot, but the values are very small). While compute time increases almost cubically MPI time maintains the same until N=1024, hence a critical point is observed at this point. As matrix dimension increases compute time becomes more dominant, hence becomes compute bound. Overall, Haswell and Sandybridge nodes performed similarly, but Haswell performance very slightly poorer than Sandybridge. Considering that Haswell performed significantly better than Sandybridge on previous worksheets, this could be due to Linpack performance (affects linear algebra operations performance) difference [1], which Sandybridge holds better performance.

The following block provides a summary/short answers to the questions. For details, please refer to previous sections on one-sided communication.

Q1) Which one-sided operations were used?MPI_Win_create, MPI_Win_free (creation and freeing of windows), MPI_Win_fence (synchronisation), and MPI_Put (Remotely write memory) were used.
Q2) Was communication and computation overlap achieved? Overlap was achieved on Sandybridge but not on Haswell. However, since we are using proprietary MPI and the only change is in MPI APIs, it is difficult to study what's causing the problem, although it seems one-sided communication APIs

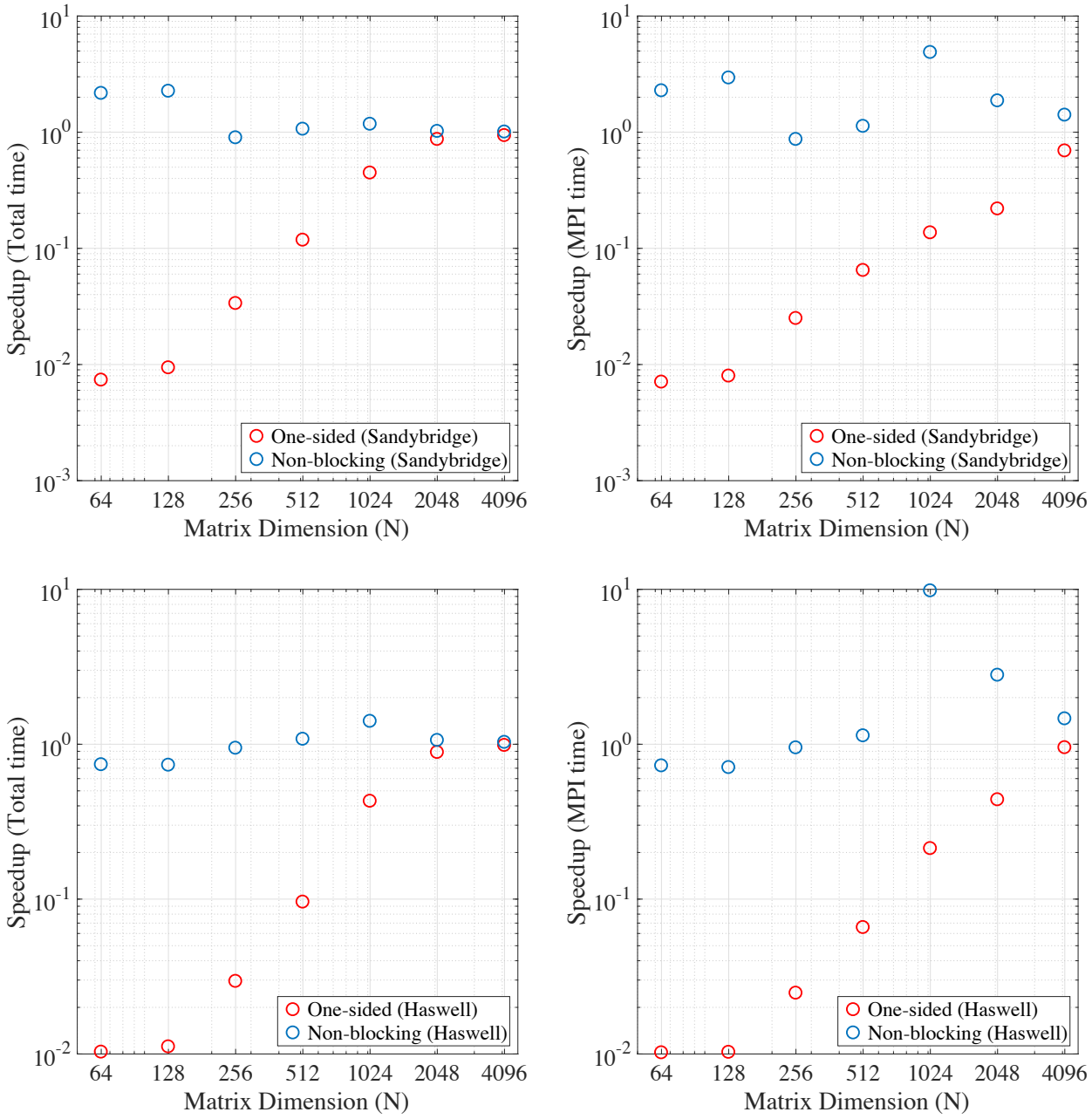


Figure 4: Speedup plot for total (left) and MPI time (right) from one-sided and non-blocking operations compared to the baseline on Sandybridge(top) and Haswell (bottom). Speedup here is defined as time(baseline)/time(non-blocking).

Implementation	Total (s)	Application (s)	MPI (s)	MPI / Total Time (%)
Baseline (Sandybridge)	478.02	457.69	20.33	4.25
Non-blocking (Sandybridge)	576.19	572.67	3.52	0.61
One-sided(Sandybridge)	527.14	511.48	15.66	2.97
Baseline (Haswell)	491.61	456.92	34.69	7.06
Non-blocking (Haswell)	507.77	518.11	10.35	2.00
One-sided (Haswell)	594.4941	533.769	60.7251	10.21

Table 1: Summary of trace analysis results for matrix dimension N = 4096. Time values are taken from the region of interest, where cannon's algorithm is performed.

are affecting negatively on Haswell. Anyhow, percentage of MPI time compared to total time was less using one-sided communication on Sandybridge (2.97%), compared to baseline (4.25%). This means more time was spent other than communication when using one-sided communication, hence it is reasonable to state that good overlap was achieved.

Q3) Was a speedup observed versus the baseline? Speedup here is defined as time(baseline)/time(one-sided). Although good overlap was achieved, significant performance slow down was obtained with one-sided communication. This is likely due to global locking of MPI_Win_fence, and extensive code optimisation using local locking (e.g. MPI_Win_post, MPI_Win_start, MPI_Win_complete, etc) may improve the performance.

Q4) Was a speedup observed versus the non-blocking version? No, one-sided communication performed significantly poorer than non-blocking communication version.

Q5) Were there any differences between Sandy Bridge and Haswell nodes? Sandybridge and Haswell nodes performed roughly the same, but Haswell performed slightly poorer. This, however, considering that Haswell node performed significantly better than Sandybridge on previous worksheets, this could be due to Linpack performance (affects linear algebra operation performance) difference [1], which Sandybridge holds better performance, and we are doing matrix-matrix multiplication for this exercise.

References

[1] SuperMUC Petascale System Description
https://www.lrz.de/services/compute/supermuc/systemdescription/

[2] MPI One-sided Communication: Virtual Workshop
https://cvw.cac.cornell.edu/MPIoneSided/pscw

Figure 5: Comparison of trace analysis between baseline (top) and one-sided communication (bottom) on Sandybridge node

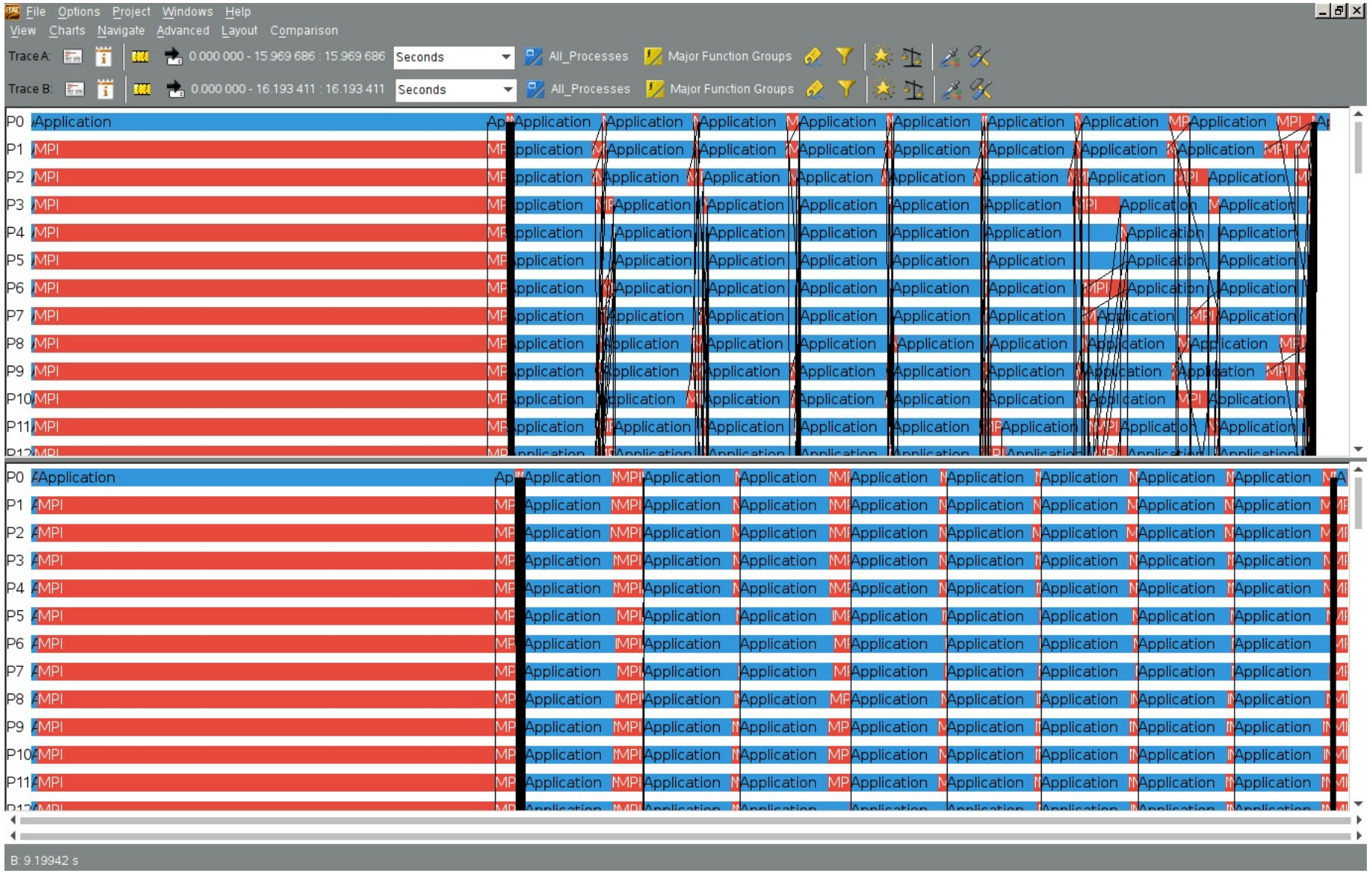


Figure 6: Comparison of trace analysis between baseline (top) and one-sided communication (bottom) on Haswell node

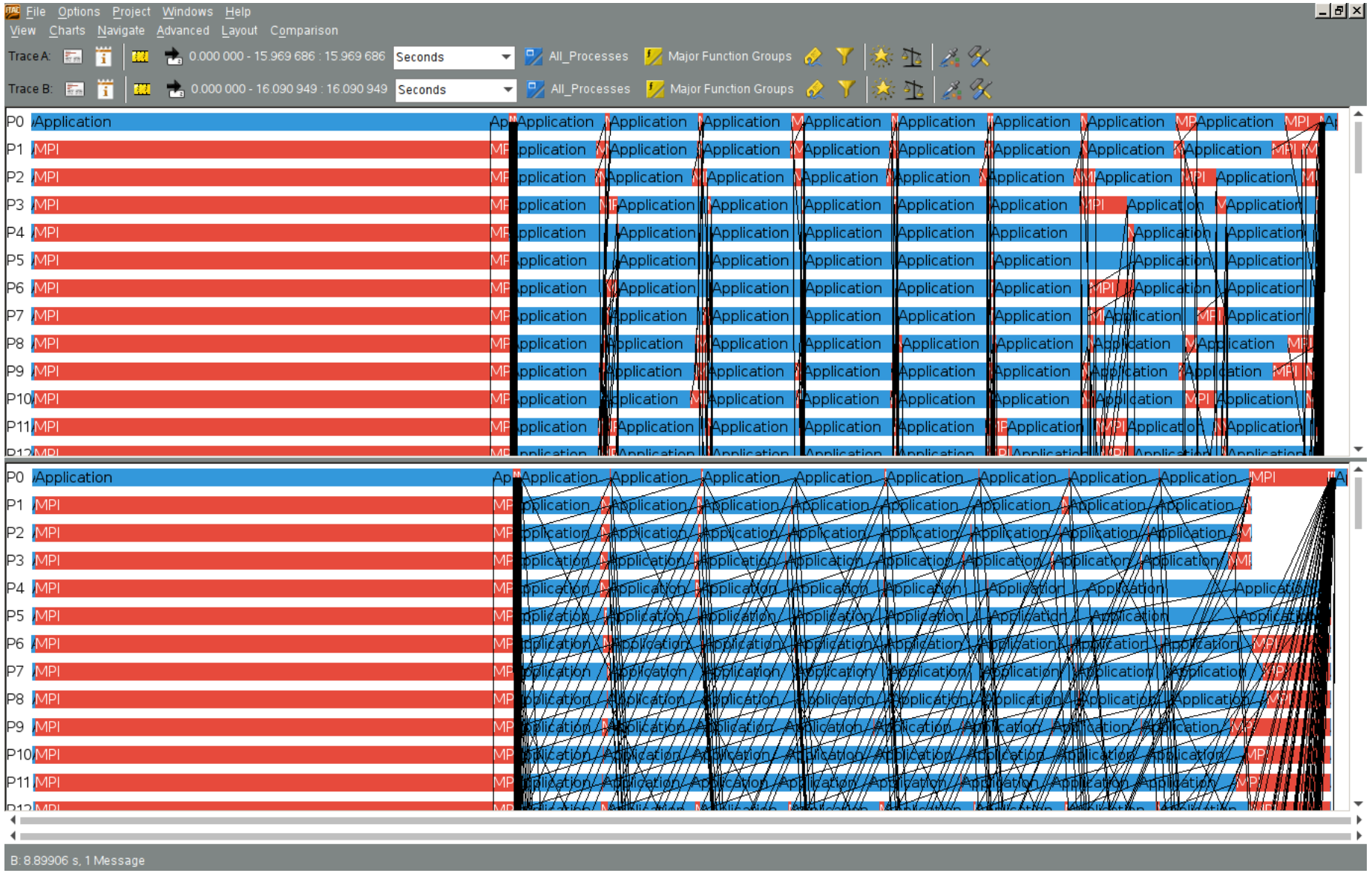


Figure 7: Comparison of trace analysis between baseline (top) and non-blocking communication (bottom) on Haswell node

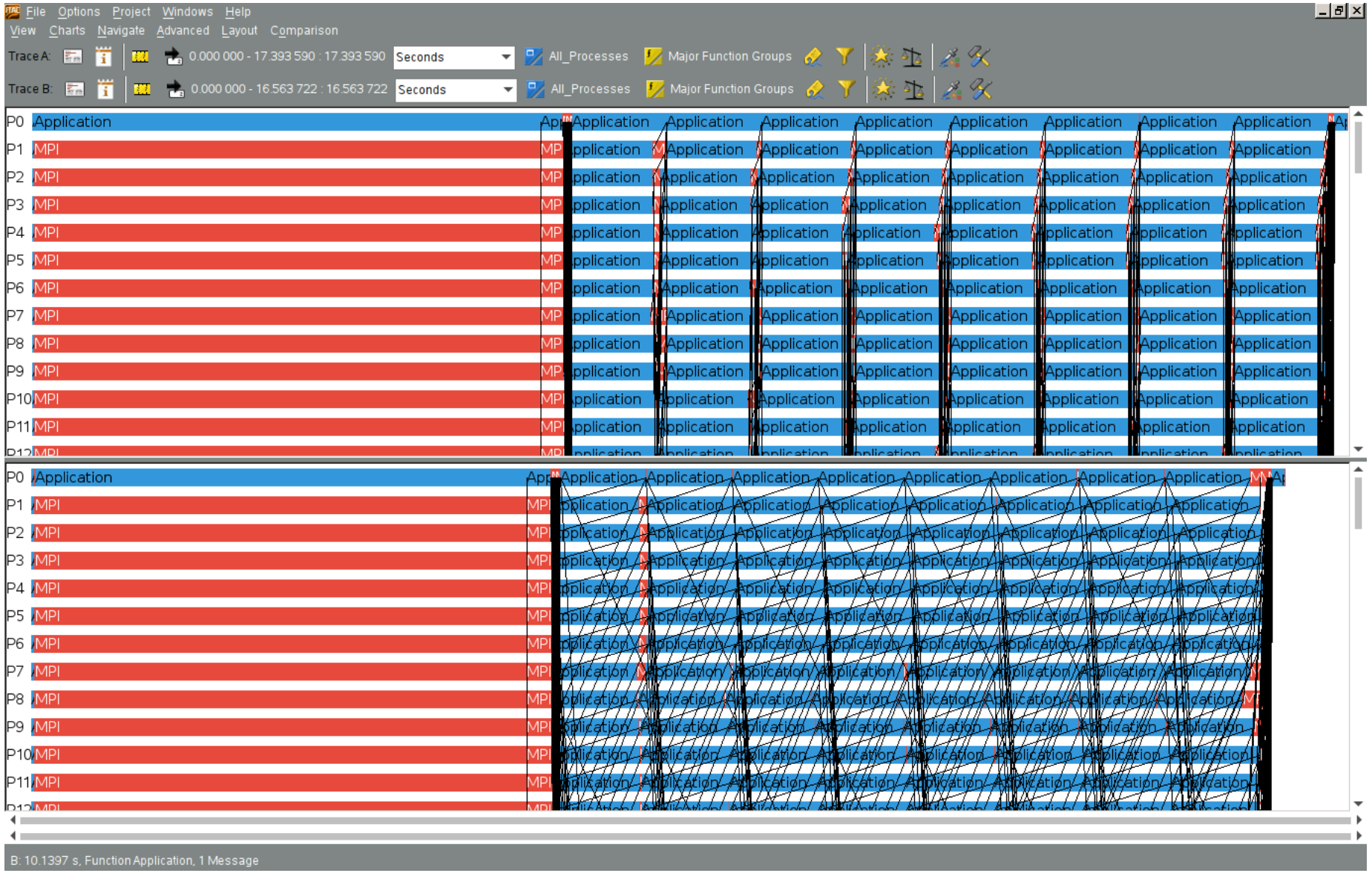


Figure 8: Comparison of trace analysis between baseline (top) and non-blocking communication (bottom) on Sandybridge node