Assignment 3 - CS671

Message Passing Interface

- Ashish Jindal (AJ523)

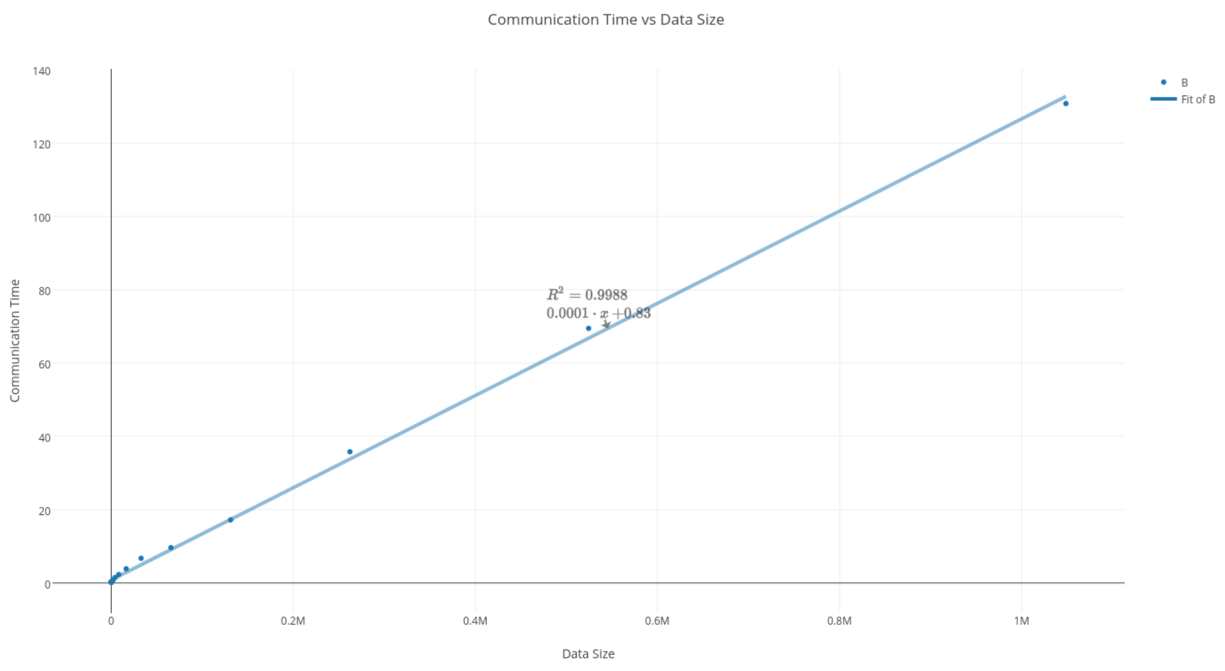
Q 1 -

1. Code - q1/main.c
2. Script to run on caliburn and generate stats - q1/run.sh
3. Data - q1/stats.csv (Check main.c for data interpretation)

Solution:

T_{Comm} = Data_Size/ BW + T_{start}
Equation if of the form **$y = mx + b$**, where **$m = 1/BW$** and **$T_{start} = b$**

Plot -



Using the least square fit solution from the above data, we get the equation as -
Y = 0.0001258 * x + 0.834496

Hence -
1/BW = 0.0001258
=> BW = 7949.12559618 bytes / us
=> **BW = 7.949 GB/sec**

T_{start} = 0.834496 us

Average Cost of computation = 0.00082359 us
We do a floating point multiplication for large number of values stored in an array and then divide the time taken on total computation by number of computations done.

We Also take into account the loop overhead and MPI_Wtime() overhead in above calculations, as explained below -

Average Loop Overhead - 0.00030339 us per iteration
To ensure that compiler doesn't optimize an empty loop, we use an "asm("")" instruction inside the loop, which is treated as volatile asm instruction by gcc as it has no operands and thus is not optimized out by the compiler plus there is no overhead.

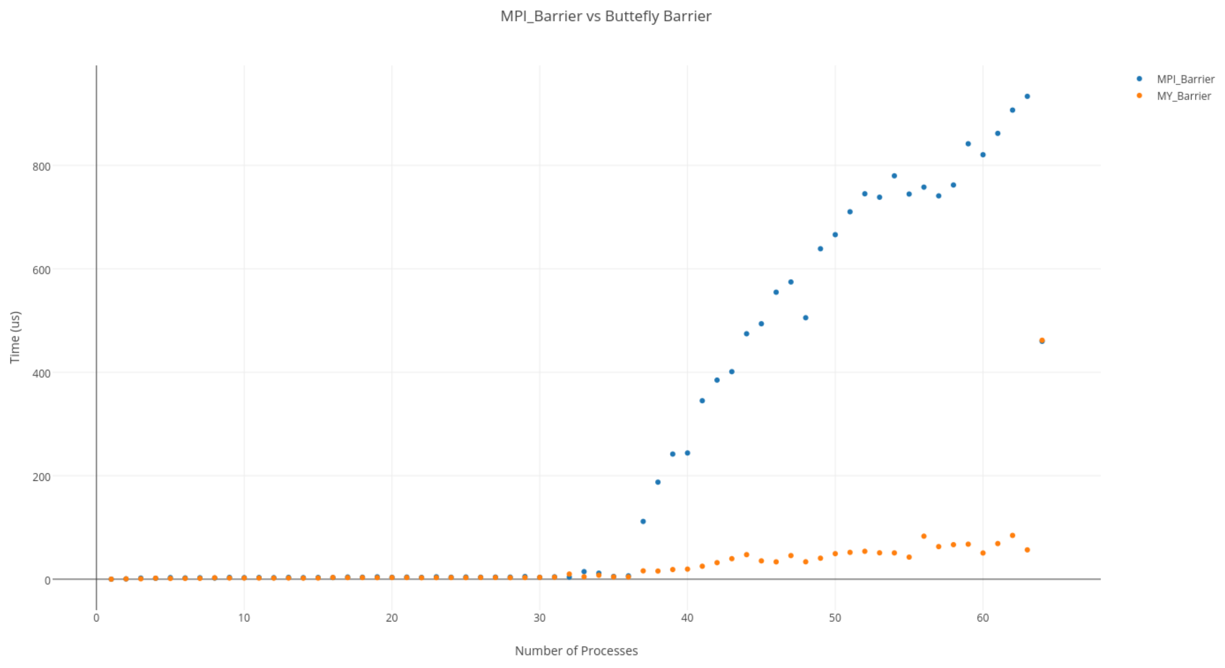
MPI_Wtime() Overhead - 0.03009717 us
Using two simultaneous calls to MPI_Wtime() help us calculate the number of CPU clock cycles between successive MPI_Wtime() calls. We do this multiple times inside a loop to get an accurate average value**.**

Variation among ping-pong message times for a fixed message size -
Yes there was a large variation in timings, one reason I can think for it is interference from other processes being scheduled by OS.

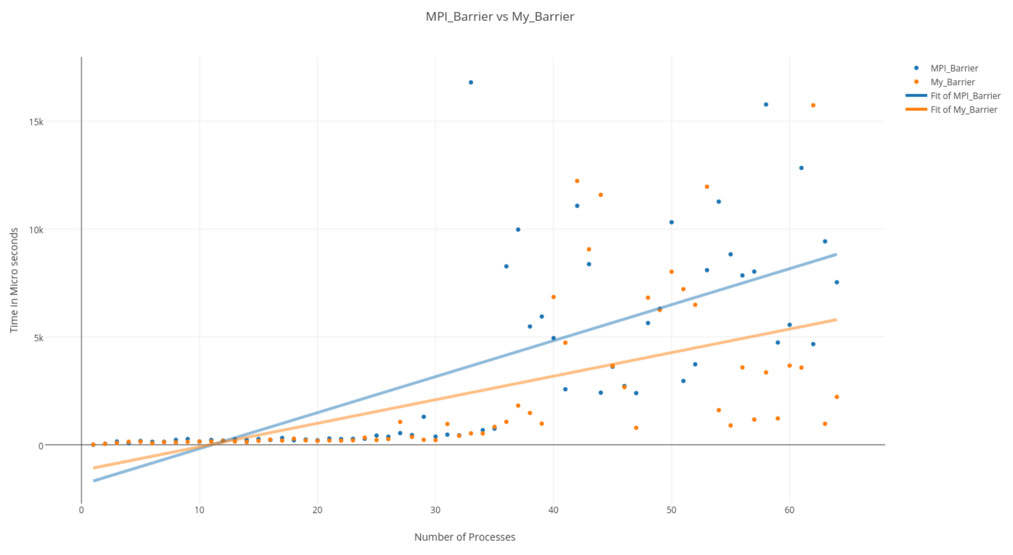
Q 2 -

1. Code - q2/main.c
2. Script to run on caliburn and generate stats - q2/run.sh
3. Data - q2/stats_mpi_barrier.csv (Min/Max/Avg time for MPI_Barrier())
4. Data - q2/stats_my_barrier.csv (Min/Max/Avg time for My_Barrier())

Performance comparison (Averaged over 1000 barriers):



Performance comparison (single barrier):

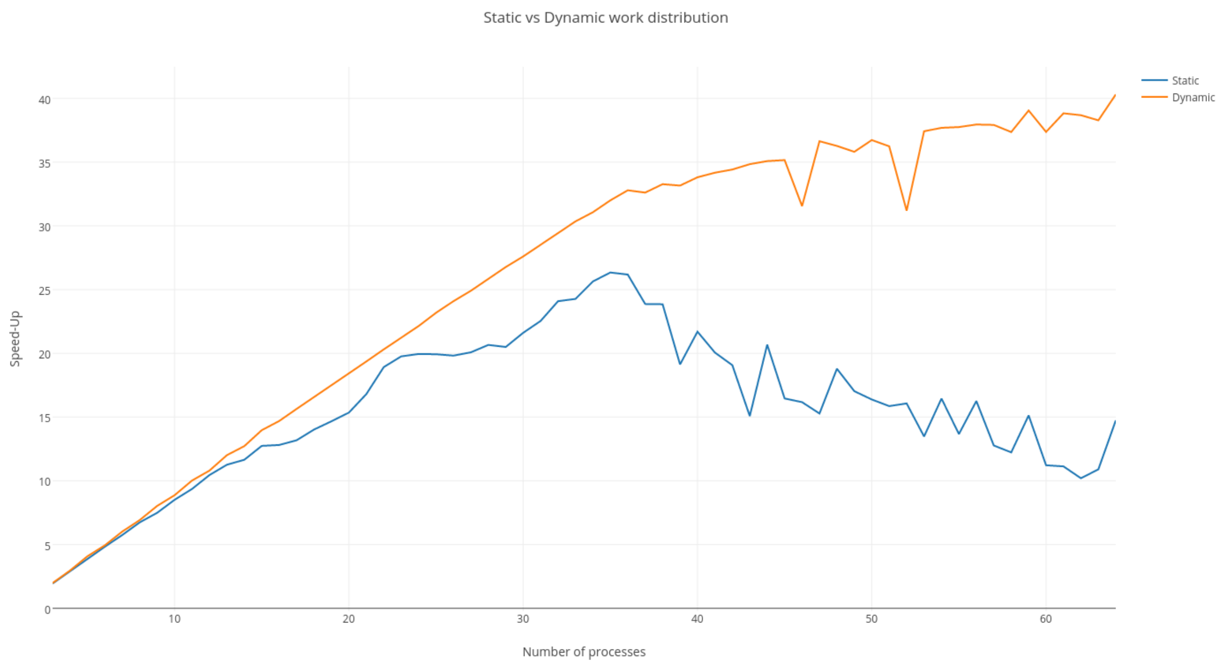


Q 3 -

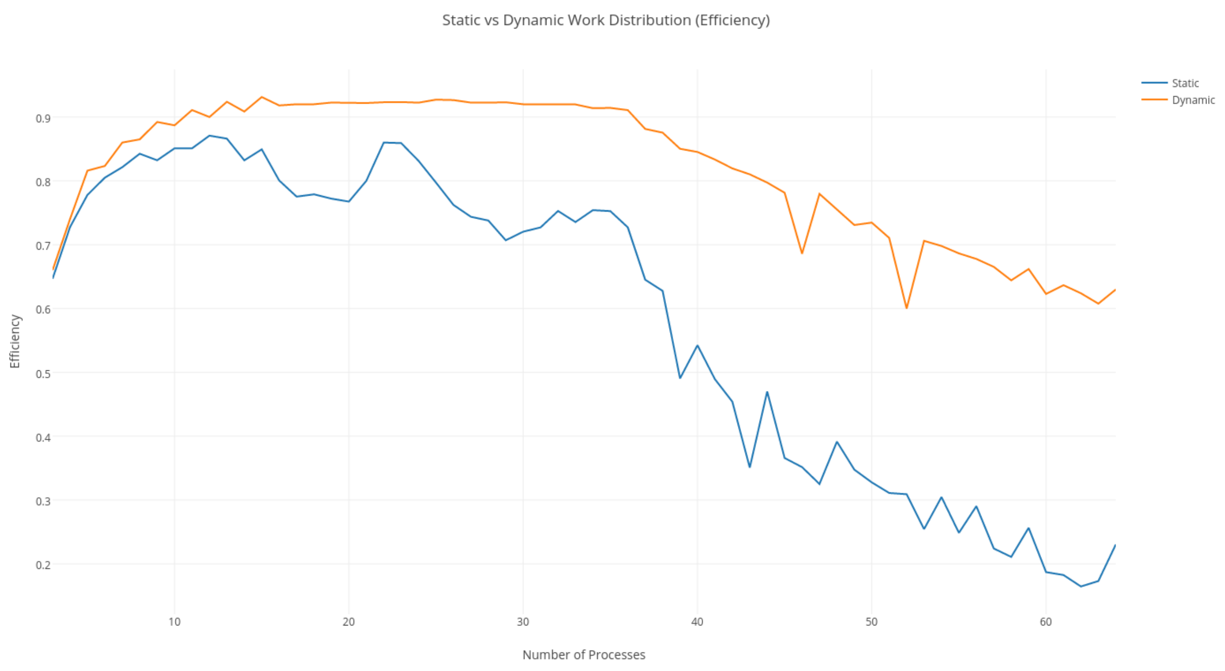
1. Code - q3/mandelbrot/main.c, q3/mandelbrot/mandelbrotThread.c
2. Script to run on caliburn and generate stats -q3/mandelbrot/run.sh
3. Data - q3/mandelbrot/stats_mpi_madelbrot_static.csv (Speedup stats for static work distribution)
4. Data - q3/mandelbrot/stats_mpi_mandelbrot_dynamic.csv (Speedup status for dynamic work distribution)
5. Data - q3/mandelbrot/stats_mpi_mandelbrot_static_chunk_*.csv (Stats for varying chunk size in static work distribution with constant number of processes)

Solution:

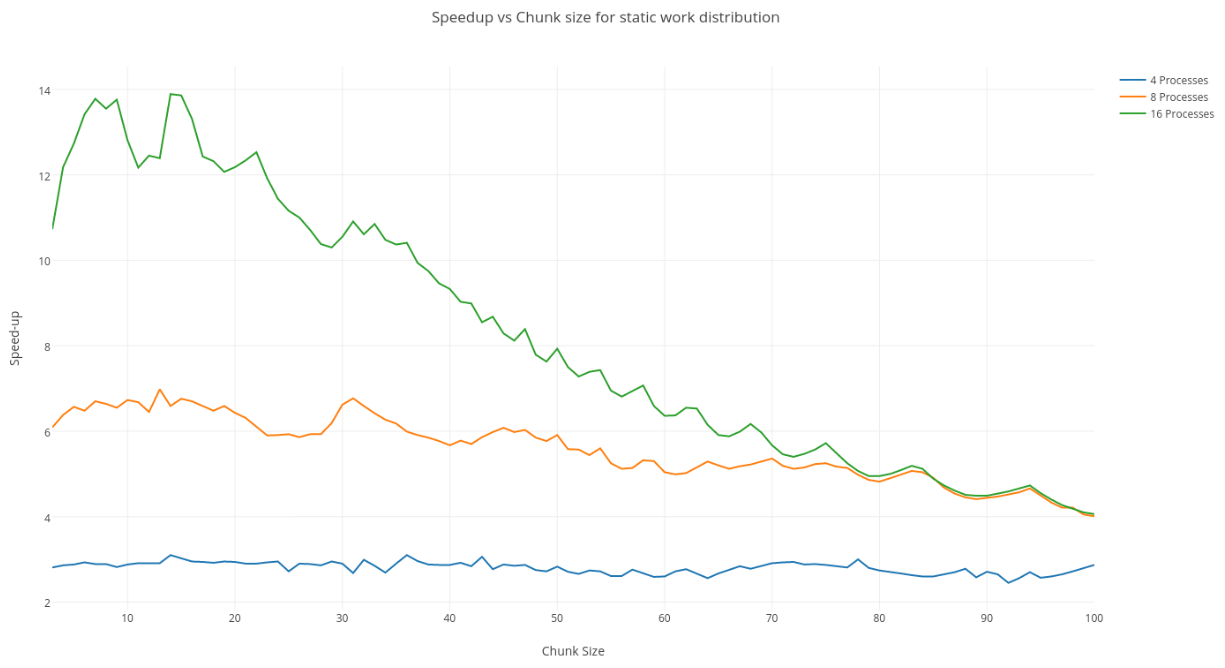
Speedup comparison - Static vs Dynamic



Efficiency Comparison - Static vs Dynamic



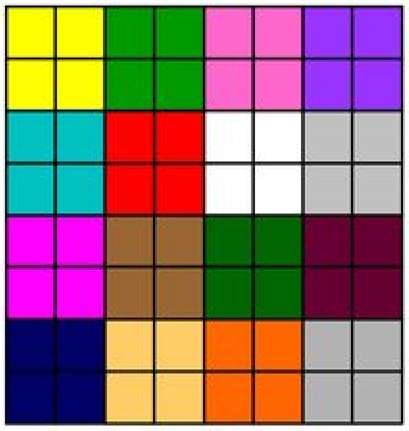
Speedup variation with chunk size for static work distribution -



A) Code in file q3/mandelbrot/mandelThread.c - masterStatic() and workerStatic()
B) Code in file q3/mandelbrot/mandelThread.c - masterDynamic() and workerStatic()
C) See the plots above for speedup and efficiency comparisons. Following are some parallelization overheads in my implementation -
    a) Memcpy - Copying result from worker's buffer to master buffer
    b) 1 worker processes 1 complete row. The row itself can be broken into smaller chunks which may increase parallelism although it will cost in more number of messages.
    c) There is only one master which has the task of aggregating the results from all other threads. Multiple processes might have to wait if master is kept busy. If we somehow increase the number of master processes then we can handle more processes without incurring wait time for worker threads.
D) Report -

<u>Static work distribution:</u>
We keep one thread as master thread and rest as worker threads. Master's job is to aggregate the incoming data into the final array. Workers divide the array into 5x5 blocks (Configurable clock size) and then each thread works parallely on designated blocks - which are offsetted using the thread rank.
Whenever a thread finishes its work on the block, it sends the result to master thread witch some additional information like the block dimension and location the result array. Worker threads keep repeating the above process till they run out of blocks.

Following diagram shows an example scenario in which worker threads are working on a block of 2x2 dimension inside a parent array of size 8x8.



We run the above algorithm using multiple block sizes ranging from 2x2 to 100x100 and number of processes kept constant to 4/8/16. The plots for same are above. It can be observed that, we get maximum speedup when we keep block size in the range 10x10 to 15x15. And it decrease as we go above it (Reduced parallelism).

Dynamic Work Distribution:
In this implementation we have one master and rest as worker threads. Master's work is to provide more work to threads which are finished with their computation and simultaneously keep aggregating the result in the result array.

- Initially master process allocates each worker with one row.
- When a worker is done with its work, it sends the result to master and then master allocated the next available row to that worker.
- We keep track of number of rows that remain to be processed and also the number of pending rows from worker threads.
- When number of pending operation become zero and the master has no more work to assign, it sends stop message to the workers so that they can quit waiting for more messages from the master.

The idea is to keep assigning more work to threads which are able to finish their computation faster, this way we reduce the ideal time in threads.
It can be observed from the plots that dynamic work distribution scales well with increased number of processes as compared to static distribution.