Parallel Relaxation on a Distributed Memory Architecture

January 8, 2024

Contents

1 Introduction

Previously we have solved the problem of relaxation using the `pthread` API in C for a system configured with shared memory. Now we will try to solve the same problem again but with a different approach, using a distributed memory system. Last time we were about to achieve a 37× speedup on $10,000^2$ matrices, in theory we would want to make this look silly, and attempt to solve problems with $1,000,000^2$ matrices, however SLURM simply would not give us enough memory, at least if we were trying to initialise the memory on one process (this isn't strictly required, but it is what our implementation does).

Distributed memory systems are designed for solving significantly large problems where the cost of messaging is worth having nodes connected via a network, as opposed to memory buses. In terms of the structure for this report, it'll be more concise than the previous, and we'll briefly revisit any key ideas as they come across, but the focus will be on distributed memory.

2 Algorithm Design

2.1 Key Considerations

If we want to develop an amazing system capable of solving problem sizes over 4× that of our predecessor, we need to come up with an algorithm that appropriately fits the strengths and weaknesses of distributed memory.

In general, messaging is slow, and in general sending large messages is more efficient than sending many smaller ones due to the overhead (but this can vary). "Luckily" for us, the MPI will handle all the messaging for us, hopefully compiling down to shared memory constructs on nodes and nice efficient messaging across the distributed nodes.

To minimise the cost of messaging, we should use the maximum local processing power, and minimise messaging across nodes. As before, the iterations are dependant on each other, so this problem isn't pathologically parallel, and at least some messaging will be required.

2.2 Minimising Messaging

The method that we will go with aims to minimise the amount of messaging, to about two messages per node per iteration, plus a message that checks for convergence across nodes, and initial messages to distribute and collect up the work. MPI already implements these constructs for us, and as long as we use the semantically correct style of messaging, we should get an appropriate speedup.

In this scenario, semantically correct means not to use multiple sends when, for example, a scatter could be used. This can possibly help the MPI implementation to optimise how it distributes messages across the network. Previously with shared memory, we were able to divide up the amount of numbers exactly such that every thread got an exactly equal amount of numbers, plus or minus one each. This will be possible for distributed memory, but it would make our program far more complex than it needs to be. An equal amount of elements will still need to be sent, so we should keep our program simpler at the cost of a slightly more uneven workload.

To divide the initial matrix up, we can divide it into chunks where each chunk has a width, which will be the lower bound of distributing evenly, or one more than that, to account for other uneven problem sizes. See the example below, where a contiguous matrix has been split up into three chunks.

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 \\ 30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \end{bmatrix}$$

$$\begin{bmatrix} 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \\ 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 \\ 60 & 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 \end{bmatrix}$$

$$\begin{bmatrix} 70 & 71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 & 79 \\ 80 & 81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 & 89 \\ 90 & 91 & 92 & 93 & 94 & 95 & 96 & 97 & 98 & 99 \end{bmatrix}$$

This matrix is now split up, roughly evenly, and can be handled by three separate processes. Each process can now perform relaxation on it's own chunk, and we have an embarassingly parallel solution.

Of course, there are a variety of problems with this approach. We need to consider the fact that each process can't just see every other processes' elements, so processing the top row of the third chunk, for example, will be hard without that missing information.

$$\begin{bmatrix} 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \\ 50 & 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 \\ 60 & 61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 \end{bmatrix}$$

This chunk in the middle here requires two messages each iteration, the row from above, and the row from below it. It needs these each iteration as they are dynamic, and change throughout unlike the boundaries. Note that they only need the final row of the chunk above, and the first row from the chunk below, nothing more. In our program we can keep pointers to these rows as 'head' and 'tail', however as the head is the first element, that will already be stored. Our rows are of size n for an $n \times n$ matrix, so this is known at compile-time, as size is an argument to our program.

To balance the load, we can use some integer arithmetic, where the lower bound is $rows/processes$ and the amount remaining will be $rows \mod processes$, where the operations are restricted to the set of integers, and a division implies rounding down, such as in C. Distributing the remaining rows is then just telling the first m ranks that they get an extra row, and to make sure all memory is local, we do this before we attempt to split the load, by just keeping track of how many rows each process will be responsible for.

3 Implementation

3.1 Outline

Implementing this won't be as difficult as it appears, as MPI handles the work for us, given we use it's API correctly. This implementation will create the matrix on a single process, distribute the elements to each process, and then gather the result. Depending on what you wanted to do with the matrix, the distribution and gathering steps may be unnecessary, but we do them anyway, particularly for the case of generating random seeded matrices for our testing. Pulling of numbers from a single random stream should be ordered and consistent. For the case of the matrix with 1s on the north-west borders and 0s otherwise, this can be constructed completely distributed.

Some key considerations of this approach include how we would **print** out intermediate steps, if we don't gather on each iteration. There are a few approaches, but we can leave that discussion for the testing section, as intermediate step generation isn't a required part of our algorithm. The high level overview of what we need to implement is:

1. Generate a matrix on the root process
2. Distribute it evenly across processes
3. Perform relaxation on each chunk, with messaging for missing information
4. Check for convergence
5. Gather results to the root process if all nodes have converged

Overall this gives us a very weak producer-consumer pattern, where the producer produces the initial array, and the consumers (including itself) process on it until the producer gets it back. This problem could have really been solved without that pattern at all, if the matrix was only stored in distributed memory, which would let us solve much larger problems than the limit currently found at $40,000^2$ elements. The bulk synchronous parallel model of supersteps then a convergence check is still followed.

3.2 Non-Blocking Communication

Blocking communication is what can make distributed programming so slow. If we allow each chunk to actually have two matrices, then we get some bonuses. The first is that it makes convergence checking simple, we simply perform a lookup on a read-only version, and then check the difference. The second is that we can guarantee our read-only chunk won't change. This means we can presend the message, and not have to worry about waiting for the buffer to be copied out, as long as there is inter-iteration syncronisation.

If we then also start processing from the middle of our chunk, at row 1, instead of the head at row 0, we can also prerequest the missing information we need, and start processing right away. For this we can use `Isend` and `Irecv`. We could use the composition `Isendrecv`, but we'd rather process whether we've already sent or not, and would rather not wait on the sends.

3.3 The Code

To implement the approach above, we first use `MPI_Scatterv`, which allows us to scatter data across processes, where the distribution of data isn't perfectly equal. To accomplish this we have to create two arrays, one with the count of elements each process should get, and one with the offsets we should begin with in the matrix. We keep it fairly basic by just using the latest offset, plus the last count to get the new offset. The counts are balanced using the same integer arithmetic as before, and the offsets start at 0.

To then perform relaxation, we first check our global convergence flag, which each process sets locally, initially at 0. If we haven't converged we enter a while-loop. The loop prerequests information using `Isend` and `Irecv` separately, up to twice, leading to a total of four, non-blocking, messages of n `MPI_DOUBLE`s, where n is the size of one row in our matrix.

After sending our non-blocking requests, we proceed to process the middle of our chunk, if one exists. After the middle, we wait on the first row to come in, then we process our head, and finally we wait for the last row to come in, and process our tail. Edge cases are all fairly easily handled, where there is only one chunk, where the chunk is the first or last chunk, and so on.

Further optimisations can be made, such as rather than waiting for the head first, we could use something like `MPI_Waitany` to handle whichever comes first (if one hasn't come in yet) and then loop back and do it again only waiting for the second element. This could be useful, particularly between nodes that are uneven, as in our test cases, an uneven edge can be up to 40,000 elements (elements per row). However, this would make our inner loop even more complex, which is not needed given the state that it is in.

After processing our full chunk, we then use `MPI_Allreduce`, which will not only reduce all the convergence flags into one, but will distribute the result to all processes. This is part of the correct semantics, where we can encourage MPI to pick the right algorithm for our problem. If we had done a reduce followed by a broadcast, we may have created some inefficiency.

Finally, if we have converged, we perform an `MPI_Gatherv`, which works like scatter in the way data can be uneven. We put all our data into the resulting matrix on the root process, using the same counts and offsets as before. Finally, we release all local memory, and allow processes to cleanup and exit gracefully.

3.4 Memory Locality

There are more features from MPI that we could include into our code. One that could've been interesting to explore, is configuring a custom network topology. With a custom network topology we could encourage MPI to maximise data locality by putting chunks that message each other, close to each other, like a double linked list, where the links are memory buses as opposed to networks. Fortunately when printing out the processor and rank of each process (and then sorting the results of all 176 into order), we can actually see that, at least whilst I was running it, MPI already puts ranks on the same node by default, only changing when it runs out of space on a single node.

Something that is also guaranteed (as far as I can tell from my investigation) is that even nodes will be put in order of closeness, with my four nodes being B for ranks 0-43, D (44-87), E (88-131), F (132-171). Whilst it cannot (to my knowledge) be proven that B, D, E, F are in order of their memory locality, it would make for a reasonable naming convention.

4 Debugging

4.1 The Problem

Debugging the program was by far the most stressful and frustrating part of this assignment. Understanding how to use the API came with the implementation and a bit of time. Debugging required me to set up a completely new set of skills, and whilst painful, I am grateful as it's the kind of skills I think are transferable and useful to have as a computer scientist.

The first was that I had never had to debug something outside of an nice, visual IDE before, but as `mpicc` isn't itself built with a debugger, there wasn't a green little bug at the top of my screen locally that could just be pressed. To debug locally (after the not so fun installation process of OpenMPI), what had to be done was to setup my IDE to use the shell script command, much like on SLURM, and have separate build and run phases. Whereever a breakpoint was needed, this code would have to be inserted before it:

```
int i = 0;
while (!i) sleep(1);
```

This code ensures that when we go to attach the LLDB debugger (which is done after launching the processes), we aren't already too far into our program. After we attach the debugger, we can set `i` ourselves to end the loop and navigate through the code. This sounds like a lot when some print statements could be used, however the main issue I ended up facing was simply not solvable without being able to inspect all the variables and switching between processes, and moving each along the execution in the order I wanted.

4.2 Bugs

There were a few bugs faced that are common in MPI programs. The first was not setting the offsets correctly for `MPI_Scatterv` and `MPI_Gatherv`. This was mainly a logical bug and was quickly fixed when the output clearly came in the wrong order.

The second was a non-parallel bug where updating the local convergence flag could short circuit, leaving the second argument unevaluated (as a function call with side effects).

The main bug that was faced, was a bug where the program would segfault on a null pointer, during the call to `MPI_Finalise()`. This bug was extremely irritating, as there was print statements all throughout my program up until the very end where this was called. MPI also provided no relevant information either, besides the fact that it was during finalise and because the address at `0x0` (the null pointer) was null.

I went through my program many times adding barriers, print statements, and even stepped through the whole program on four parallel processes individually watching all variables to see if a null was being incorrectly set throughout. All the calls to `free()` were removed, and even my final solution was printing out correctly, with my data being sent to a variety of files.

Honestly, at some point I just considered something that in my opinion still isn't needed, but necessary only for the sake of MPI, however I may be wrong. The solution to my crash was to add the calls to `MPI_Wait()` not only for my receives, but my sends as well. To me this still doesn't really make sense, if it can be guaranteed that all non-blocking sends will be received (or the program will not terminate) then it shouldn't be the case that all sends are waited for as well, especially as each send has exactly one opposite receive.

5 Correctness Testing

5.1 Scripts and Unit Tests

The majority of the correctness testing will be done in a similar way it was done for the previous assignment, but with some more mathematical checks for sanity. We will start off by using a completely sequential version of the code, which was checked manually for the precision of 0.1. The sequential version also includes unit tests, such as performing relaxation on basic examples for 1 iteration, performing it on matrices that shouldn't change, and so on. In hindsight, these unit tests should have all been used to develop the parallel version too, however the parallel version came about before them, and it was much harder to go backwards from that compared to the double for loop of the sequential program.

The dataset tests were conducted on included large, varied matrices including ones generated using random numbers and fixed seeds. It did not include invalid inputs, and I'm reasonably sure my code won't work for silly configurations such as having 176 MPI processes for a 4×4 matrix (although it may do). The scripts for generating data for the scalability investigation were also reused, as all that had to be added to the code was outputting the final result to its own file, after the timings had been made. This was reasonable until the sizes of $10,000^2$ square matrices where the files would become too large to store on the cluster. The scripts used looked something along the line of "loop through these precisions, these sizes, and these process counts, running `mpirun` with the arguments" as a rough simplification.

5.2 Automated Checking

For the shared memory coursework, we used a Python script to performing a diffing algorithm on two pairs of results, however this time that was not possible as the results were only accurate up to the given precision. This could have been solved through a script that rounds and then handles the final the final digit accordingly, but why spend five hours automating something that can be done manually in five minutes?

5.3 Mathematical Results

This part really isn't even about testing, but more there are some cool mathematical test cases that can at least show that your program *isn't* working when it should be. The first is the case where you set each index of the matrix to the value of the index, like our matrices in the start of this report. This matrix should converge instantly after one step, as you would end up doing:

$$\frac{(n+1)+(n-1)+(n-size)+(n+size)}{4} = n \tag{1}$$

There are various patterns like this but they are in general less useful, as they only prove the program to be correct when no relaxation iterations are performed.

5.4 Lazy Programmers

Finally, whilst the program doesn't print each iteration by default, it was added in a variety of ways which were useful during the debugging. The most fun was to use a method inspired by sleep sort, where we just have each process call `sleep(chunk.rank)` before printing their chunk. Whilst this works locally with small process count and shared memory, this wasn't used on SLURM to avoid wasting the queue for others *(other users were noticeably less concerned about this)*. Instead we just gather each iteration and have the coordinator print out the matrix. This is removed in the submission as it slows the code down.

For testing excerpts, as before there are some files included with the submission (see `tests.zip`), and an example below. Included are `100.txt` which demonstrates a 100×100 element solution, `5k.txt` demonstrating a $5,000 \times 5,000$ element solution and `8x8-steps.txt` demonstrating the steps taken in solving an 8×8 matrix, on two nodes with two processes each. Each have a 0.001 precision, and were performed on the Azure machine, besides `5k.txt` as my copying commands weren't working, so this was done locally - arguably adding system independence to the tests. On the next page is the final output for the 8×8 matrix.

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.000000	0.952671	0.905663	0.856026	0.794278	0.696563	0.498382	0.000000
1.000000	0.905603	0.815005	0.725470	0.625833	0.494640	0.297547	0.000000
1.000000	0.855977	0.725516	0.607800	0.491723	0.360846	0.198403	0.000000
1.000000	0.794095	0.625636	0.491421	0.375344	0.260966	0.136521	0.000000
1.000000	0.696466	0.494572	0.360716	0.261078	0.174208	0.088410	0.000000
1.000000	0.498235	0.297342	0.198121	0.136374	0.088243	0.043947	0.000000
1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Table 1: The final result for an 8×8 relaxation, with precision 0.01

5.5 Untested Behaviour

As before there were some behaviours that haven't been fully tested. This includes negative numbers (if they converge), non-square matrices, and the case of having more processes available than rows. Unlike before, random numbers were tested this time, but they were still restricted to values less than one, as from just the algorithm, the convergence properties were hard to determine. In hindsight if this is used to solve differential equations I can imagine the convergence guarantee is fairly strong for a large range of numbers but this wasn't checked.

6 Scalability Investigation

Measuring the performance of our parallel code at scale is crucial for understanding whether we have actually made the process better, or worse. Large problems should be targeted, particularly in distributed memory computing as the cost of messaging needs to be worth it. To avoid rerunning the sequential version large problem sizes whilst the queue is busy, we will reuse the **single-threaded shared memory** results as our baseline. This is a fair test, as for a large size the single-threaded distributed memory test was ran and got very similar results of about 650 seconds (unoptimised), and starting zero pthreads should be about the same as doing no distributed work (with a reasonable enough MPI implementation).

There were 16 datasets created during this assignment, which can all be found in `datasets.zip` if interested. If there was more opportunity to test the completely sequential implementation (used for testing) against the single-threaded distributed memory version, it would be done, however hogging the machine just to run sequential code again feels particularly wrong.

6.1 Timing the Code

To make tests precise and meaningful, we time exactly the call to `relax()` on the root node. The timings are done using the `MPI_WTime()` function, which is one of the only functions that doesn't return an error code, but the value of the time passed.

What **is** timed:

- The scattering, and gathering, of the result matrix
- The creation of smaller, intermediate arrays, such as the local chunk copies, and arrays for offsets
- The bulk processing of the relaxation technique, including messaging

What **isn't** timed:

- Printing out intermediate steps (only for debugging and testing)
- IO operations such as writing to files, random number generation
- The allocation and initialisation of the initial matrix

Datasets Created (16 Included)

- Parallel execution (fixed size $(10,000^2)$ vs fixed number of processes (176))
- Implementations of MPI (NVIDIA, Ohio State University, OpenMPI) for both
- Compiler optimisations on and off for each of the above combinations

The Intel implementation wasn't tested as it didn't work with the filesystem or the file writing correctly.

6.1.1 Fixed Variables

We don't run the code varying the number of nodes (e.g. with 44 processes) or varying the precision. For the node count, OpenMPI is possibly already taking a large amount of our distributed memory code and making it shared, keeping the node count high restricts that. For the precision, not much really changes besides the time taken, we don't learn anything meaningful besides maybe at the precision limit of double values.

6.1.2 Starvation in the Real World

Whilst not making excuses for the results gathered, there is a point to notice about how SLURM seems to have a starvation problem relating to node counts. It will always run jobs that have a lower node count if they will fit, and much like the readers-writers starvation, this can starve jobs that have been waiting far longer that use four nodes, as long as, *hypothetically*, other students sent out automated job submissions with nodes counts less than four.

6.1.3 Compiler Optimisations

Much like before we test both with and without compiler optimisations. The impact of compiler optimisations can tell us two different stories. With compiler optimisations off, we limit the proportion of precompiled, already optimised code in our codebase, however at the cost of having statements that aren't necessary besides for readability not be optimised out. With compiler optimisations on we get to know really how fast our program goes, and what sizes it could handle in production.

6.2 Speedup

The speedup tells how much faster our parallel version is compared to our sequential version. We can plot speedup for a fixed number of processors, and a fix problem size. In theory, we want to be as close as possible to the Amdahl Limit, which would say that our speedup cannot exceed 176 when using 176 processors.

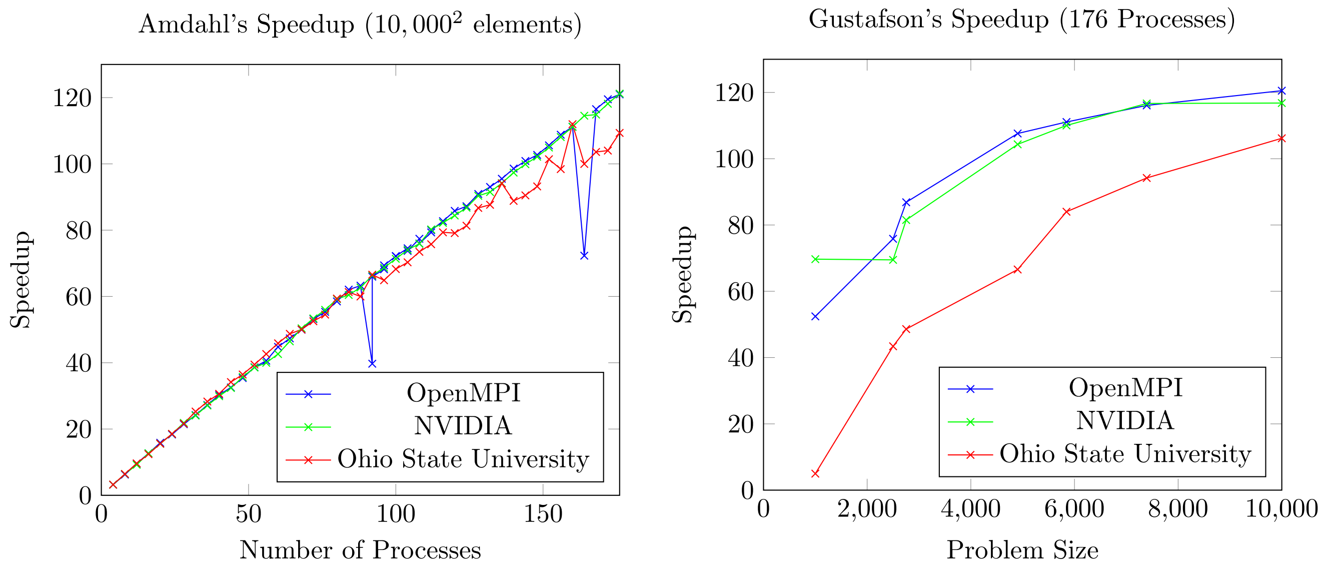

Figure 1: Comparing Unoptimised Implementations

Here the speedup is computed using the formula $\frac{T_{seq}}{T_{par}}$. We get a multiple of how many times faster our program went. We can see that our speedup went up fairly linearly with the number of processes for the unoptimised version and hits about 121× when using 176 cores. Initially, these results sound disappointing, but later we may reveal that there was simply not much else we could do.

Something interesting to note is that our speedup begins to fall off a bit as we process more and more data. This is not ideal and shows that our code has some inefficiencies, as we ideally want to get more speedup per problem size. Here Gustafson would tell us to add more CPUs as they become more and more important as the problem gets larger and larger.

In terms of implementation, it seems as if OpenMPI and NVIDIA's version were fairly competitive, whereas Ohio State University's simply lagged behind a bit. All competitors claim special support for InfiniBand networking, and NVIDIA even goes to say they are an optimised version of OpenMPI. NVIDIA do also claim to have support for what sounds like custom hardware "NVIDIA Quantum InfiniBand hardware-based networking acceleration engines" so it may be tailored for system with these involved, and in traditional marketing technology, who knows what quantum really means. It is also possible that our code was simply not complex enough in terms of MPI calls to be really tailored for a specific vendor.

There isn't too much to say about the optimised versions besides that the statistics look worse, as shown below in Figure 2. The implementation produced by Ohio State University also performs significantly worse than its counterparts here, at about a 3× less speedup, varying process counts. Note there was a large anomoly in the sequential testing for a specfic problem size, that graph should be treated as if the line was smooth.

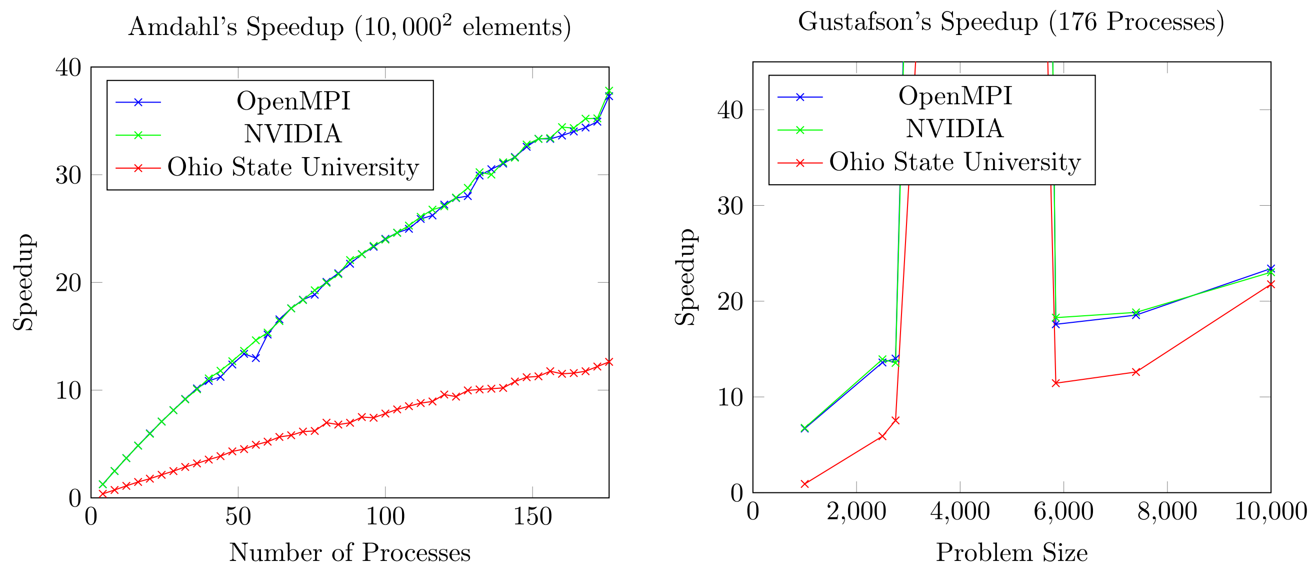

Figure 2: Comparing Optimised Implementations (O3)

Whilst the statistics look worse the optimised version actually runs about 2.5× as fast as the unoptimised version, so speedup clearly doesn't tell us everything. In reality, the true benefit of our implementation lies somewhere in between. It may be useful in the future to compare all levels of optimisation, where sequential 'noise' can be removed, such as using intermediate assignment rather than inline function calls, for example. For the rest of the investigation we will only look at unoptimised (O0) code.

6.3 Efficiency

The efficiency of our implementation tells us what proportion of the time our code was running completely in parallel. We want this to be as high as possible, as sequential code just means that we aren't making the most of all cores.

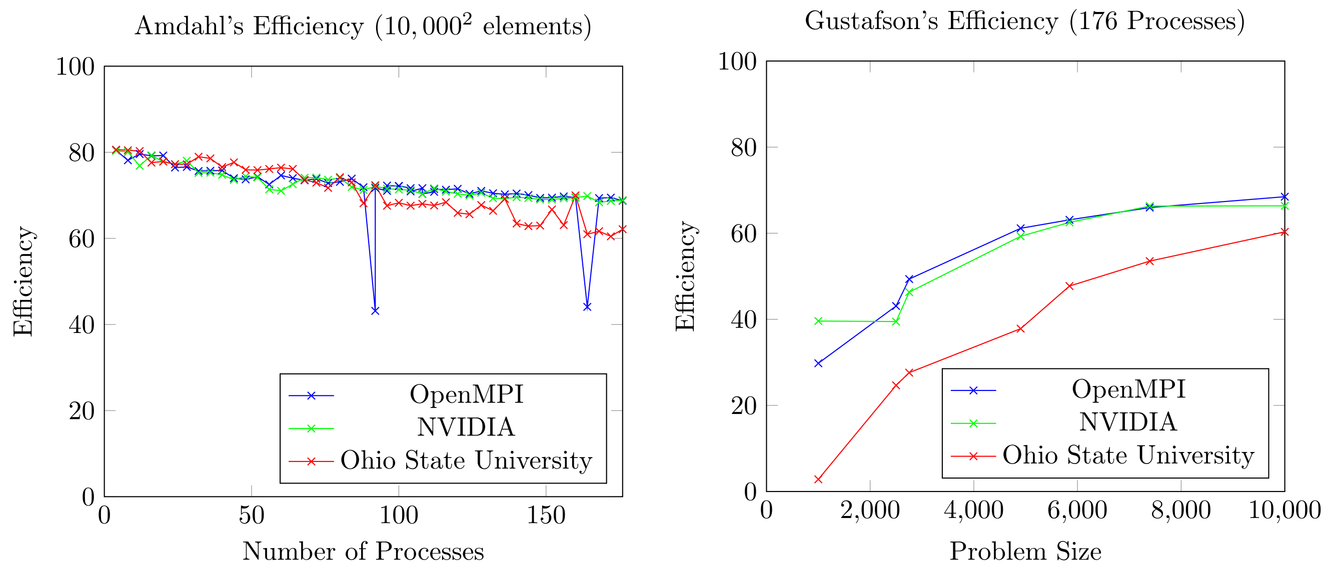

Figure 3: Comparing Unoptimised Implementations

Efficiency is computed directly from the speedup, so the graphs resemble the previous ones a lot. 80% efficiency is a strong metric, which decreases slowly to about 70% as processes are increased to 176. Something to note here is that as we measured in steps of 4 rather than individual processes, our efficiency should start at 100% and then quickly fall off. The efficiency relating to processes remains fairly constant which is great to see, as that explains our speedup increasing linearly.

This is unlike our shared memory implementation, where efficiency quickly fell off to about 60% at 44 threads. This is likely because the MPI constructs we are using, are designed to scale well themselves, and some slightly smarter choices were used. One improvement for this implementation is having a single local-convergence flag, rather than a sequential check across each element of each process. Each row in this implementation even had a local convergence flag so that improvement in efficiency is actually very clear. The shared memory version required checking every individual elements flag, which would take far longer and was completely sequential.

In terms of efficiency over problem size, we do get more efficient but again there is clearly a limit we are approaching. Some of our code is inherently parallel. Across implementations we see similar results and nothing unexpected based on the speedup. The OSU implementation appears to have very slightly better efficiency for processes counts less than 60, however this could be due to luck or randomness as the results are all extremely close. Repeated tests would be needed to plot the standard deviation.

6.4 Work Efficiency

If we want to compute the overhead of parallel processes (particularly messaging for MPI), we could time how long our parallel version would take running sequentially, by mulitplying the times by the process count. For this visualisation, we then divide through by the number of processes to represent the proportion of the parallel program that was overhead, and we get:

$$overhead = pT_{par} - T_{seq} \tag{2}$$

$$proportional\ overhead = T_{par} - \frac{T_{seq}}{p} \tag{3}$$

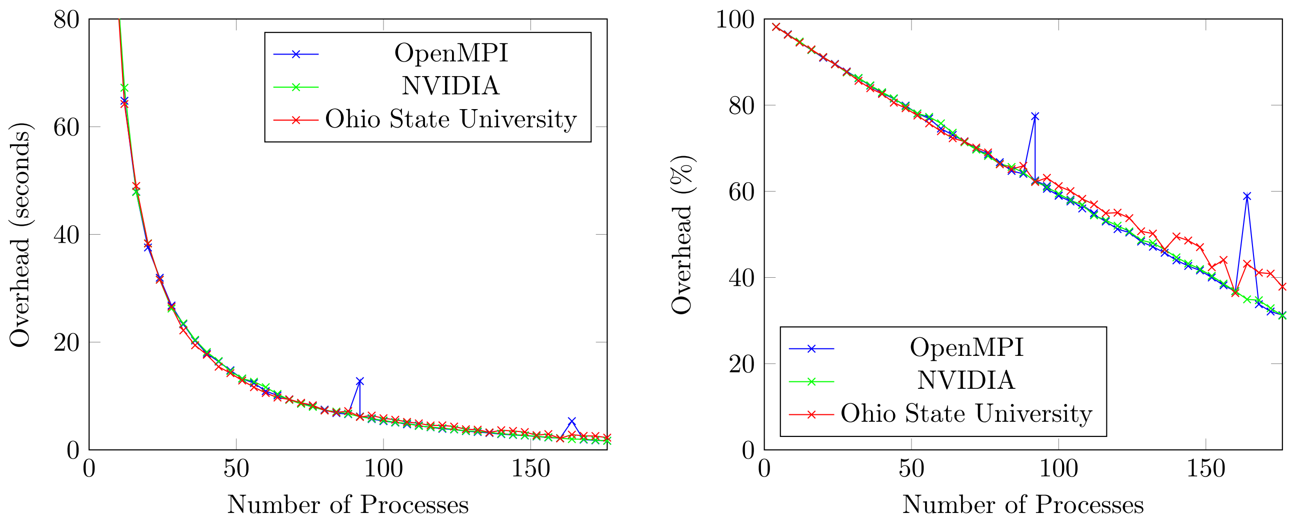

Figure 4: Measuring Overhead

These graphs are incredibly promising. The percentage overhead steadily decreases as we add more processes, and our realtime overhead in seconds gets smaller and smaller rapidly. This shows us that MPI is truly designed for scale, and whilst on small problems the cost of messaging is high, we can provide problems that are so large that the cost of messaging becomes a minute detail. As our problem size is restricted here to a $10,000^2$ matrix, we are also showing that this isn't just a theoretical feat, much like the set of "galactic algorithms" that only become more efficient on huge amounts of data. This implementation is extremely work efficient on a reasonable problem size that could be encountered in the right scenario.

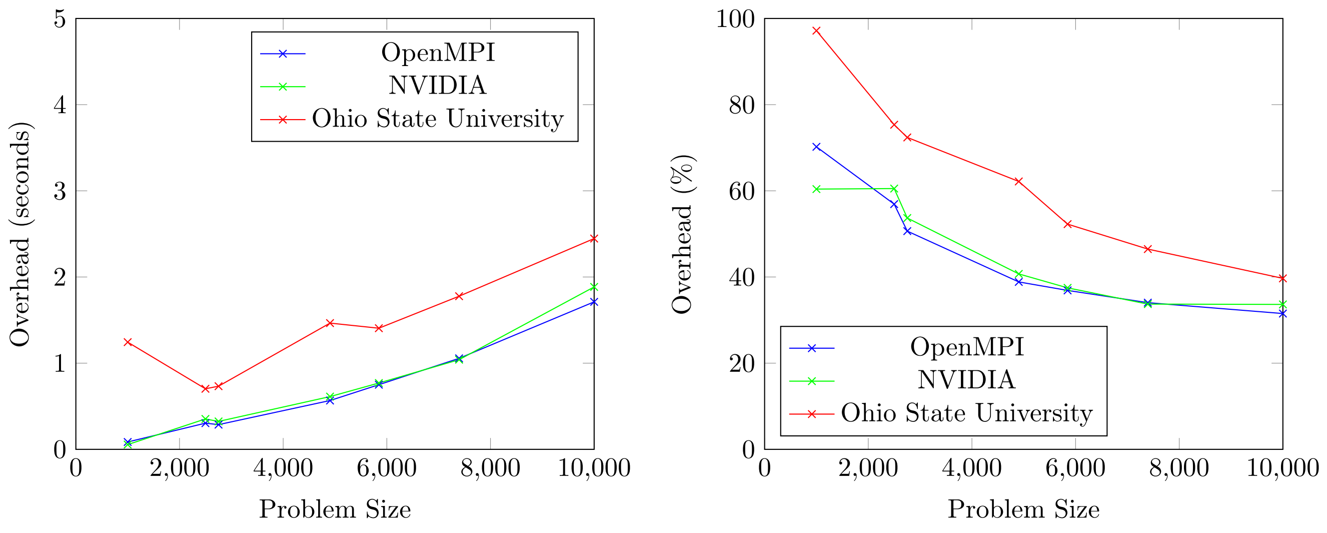

Figure 5: Measuring Overhead

In terms of problem size, we get a slightly less nice picture of the overhead. As our problem size increases, we get more and more overhead in seconds, however the the proportion of the program that is overhead still decreases. It seems to look as if it will converge to roughly 30% of our program being overhead. This doesn't sound amazing but it may just not be possible to do any better. Our program fundamentally is quite hard to split into purely independent chunks, and some some sequentially is an inherent part of our problem to solve.

6.5 Karp-Flatt

The final metric we will look at is the Karp-Flatt metric. This tells us how much of our program is inherently sequential. The smaller the value, the better, and it is related to overhead in that way.

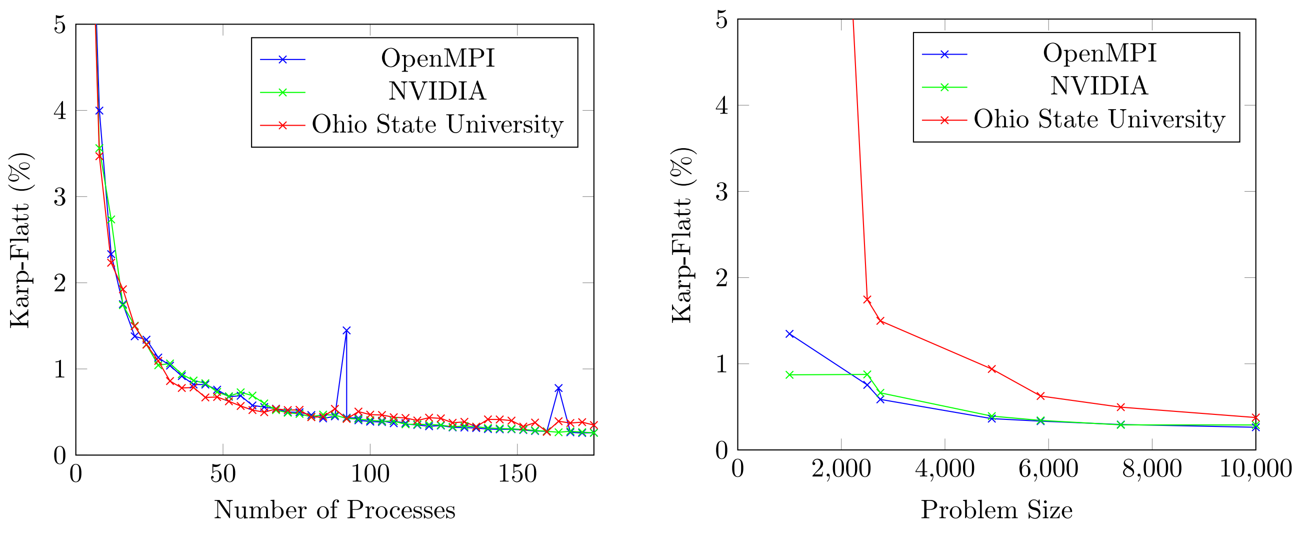

Figure 6: Measuring Karp-Flatt

The results are also very strong. Less than 1% of our program is inherently sequential for any reasonable problem size and process count. We see the same pattern where the OSU implementation is a little behind but catches up. This is better than the 2% we had previously for shared memory, and again is likely due to the far reduced parallel convergence check, of only checking m flags (processes) as opposed to n^2 flags (for an $n \times n$ array).

6.6 Isoefficiency

To explore how this program truly scales, we should look at the isoefficiency. This metric should tell us how much our problem size needs to grow by, in order to maintain it's current efficiency. The problem size should be plotted as a function of the number of processes, related by efficiency. However efficiency is a real number, to make any sort of reasonable plot we would need to group efficiencies into bands, such as Band 2 being 15% to 25% efficient, for example. As with the previous assignment, the plot we end up generating is extremely uninsightful, and so we don't include it.

7 Conclusion

Overall we have now developed parallel implementations for both a shared and distributed memory architecture. We have considered different styles of programming, the trade offs with using messaging, and developed highly-parallel solutions for a naturally sequential problem.

We compared different implementations of MPI, and determined that for standard use cases, it doesn't seem to matter a great deal. We measured our implementation across a variety of metrics and determined that for this implementation, the only real issue was the amount of memory, which is a good issue to have for a parallel solution. MPI provided us with a higher level of abstraction and a different way of thinking, and now it is clear why it is so popular in the world of data science.

A Data Processing

Listing 1: Data Processing Script

```
import os
import pandas as pd

def add_data(filename, seconds, sized, o3):
    par = pd.read_csv(filename)
    seq = pd.read_csv('data-sequential-O3.csv' if o3 else 'data-sequential.csv')

    par['speedup'] = (seq['seconds'] if sized else seconds) / par['seconds']
    par['efficiency'] = (par['speedup'] / par['processes']) * 100.0

    par['overhead'] = par['seconds'] - ((seq['seconds'] if sized else seconds) / 176)
    par['overhead_percentage'] = (par['overhead'] / par['seconds']) * 100.0

    numerator = 1.0 / par['speedup'] - 1.0 / par['processes']
    denominator = 1.0 - 1.0 / par['processes']
    par['karpflatt'] = (numerator / denominator) * 100.0

    par.to_csv(filename, index=False)


if __name__ == "__main__":
    optimised_seconds = 75.65775
    unoptimised_seconds = 654.502816

    for file in os.listdir('.'):
        if file.endswith('.csv') and file not in
            ['data-sequential.csv', 'data-sequential-O3.csv']:
            optimised = "o3" in file
            sizes = "sizes" in file

            add_data(
                file,
                optimised_seconds if optimised else unoptimised_seconds,
                sizes,
                optimised
            )
```