Neoverse Knights

Alex
Orry
Ethan

May 4$^{\text{th}}$ 2024

Contents

1 HPL 3
1.1 Methodology . . . 3
1.2 Expected development process . . . 3
1.3 Initial compile . . . 3
1.4 Compile-time optimization . . . 4
1.5 Testing and tuning methods . . . 5
1.6 Results . . . 6

2 NAMD 8
2.1 STMV . . . 8
2.2 Compiling NAMD . . . 8
2.3 Running NAMD . . . 9
2.4 Results . . . 9
2.5 Alternative Input Comparison . . . 10

3 MiniVite 11
3.1 Input Files and Louvain Algorithm . . . 11
3.2 Compilation and Parameter Selection . . . 12
3.3 Results . . . 13
3.4 Strong-Scaling Graphs . . . 15

4 OSU 17
4.1 Compilation . . . 17
4.2 Choosing the binaries . . . 17
4.3 Production Runs . . . 17
4.4 Optimisations . . . 17
4.5 Results . . . 18

5 STREAM 22
5.1 Initial compilation . . . 22
5.2 Run validity issues . . . 22
5.3 Results . . . 23
5.3.1 STREAM for MPI . . . 23
5.3.2 Final valid non-MPI run . . . 24

1 HPL

The High Performance Linpack benchmark is arguably the staple of all things HPC. It is an incredibly popular tool in benchmarking a system's compute power in terms of the number of floating point operations it can do per second (FLOPS), a number frequently used to rank the world's fastest computers.

1.1 Methodology

In brief words, the HPL benchmark solves a dense system of linear equations of the form $Ax = b$, where A is a matrix of size $N \times N$, x is the vector of "variables" and b is the result vector.

HPL uses Gaussian elimination in a series of row operations to reduce A to an upper triangular form (where it becomes a trivial substitute-to-solve computation). It is a highly parallelizable computation, well-suited to being divided into smaller independent tasks that can be distributed across multiple nodes and cores.

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Figure 1: Matrix equation in upper triangular form

One can see that immediately, $6z = 3 \implies z = 0.5$. We perform a cascade upward, substituting the value for z into $4y + 5z = 2$, solve for y, and then do the same for the top row. This can be done as a series of row operations too, similar to the Gaussian elimination process.

Despite that it needs significant memory and inter-node and core communication, the HPL benchmark is ultimately compute-bound. We can work out to what extent this is the case with the Arm Performance Reports [1] (included with Arm Forge), but it isn't installed on the cluster quite yet. As a future addition, it could aid in the parameter-tuning process.

1.2 Expected development process

The first hurdle to overcome is to simply get a working version of the algorithm, followed by optimising it, and getting it to run on all cores and nodes. The final portion of difficulty comes from parameter fine-tuning. There is a useful website for this [7], of which all you need to do for a decent HPL input file is input number of nodes in cluster, cores per node, memory per node, and finally block size (but this is easily tune-able anyway). In the end, it turned out that the file produced by this was surprisingly optimal, after a variety of parameter tuning attempts.

1.3 Initial compile

Since we had already completed this as a small assignment on our personal computers, we were able to gather our collective assignments to share knowledge for as smooth and painless an implementation onto the cluster as possible. In the end, we decided to use the Clang compiler for its better compiler messages.

Most of the difficulty for this was in setting correct paths to libraries and includes. For example, the MPI dependency wanted a library (ending with ".so, .a"), but also needed to have its include folder specified too, as well as ("lib" directory) being added to the "LD_LIBRARY_PATH" environment variable.

This led to a smooth compile and basic run on the login node. However, we needed to scale.

1.4 Compile-time optimization

Firstly, this involved switching from Clang to Armclang, a compiler specific to the architecture. To make this easy for ourselves, we adjusted our "~/.bashrc" scripts as follows, to avoid having to constantly write out full file paths.

```
    # User specific environment
    if ! [[ "$PATH" =~ "$HOME/.local/bin:$HOME/bin:" ]]
    then
        PATH="$HOME/.local/bin:$HOME/bin:$PATH"
    fi
    PATH="/apps/modules/compilers/armclang/arm-linux-compiler-24.04
       _RHEL-8/bin:/apps/modules/libraries/openmpi/5.0.3/armclang
       -24.04/bin:$PATH"
    export PATH

    LD_LIBRARY_PATH="/apps/modules/libraries/openmpi/5.0.3/armclang
       -24.04/lib"
    LD_LIBRARY_PATH="/apps/modules/libraries/arm_performance_libraries/
       armpl_24.04_gcc/lib:$LD_LIBRARY_PATH"
    LD_LIBRARY_PATH="/apps/modules/libraries/arm_performance_libraries/
       clang-18/lib:$LD_LIBRARY_PATH"
    export LD_LIBRARY_PATH
```

Following this, we could also integrate a few more key compilation flags, which were tested intermittently to ensure performance actually improved with inclusion of new flags and libraries. A particular contender for making sure it definitely improved the performance of HPL was the Arm Performance Libraries, of which we found in multiple other benchmarks harmed performance to some varying extents.

A small snippet of the Makefile for HPL looks roughly like, with the main flags included and no file paths to avoid clunkiness:

```
    armclang $(HPL_DEFS) $(LIBRARIES) $(ARMPL) -fopenmp -fomit-frame-
       pointer -O3 -funroll-loops -larmpl -Ofast -funsafe-math-
       optimizations -mcpu=native -ffast-math ...
```

Originally, we used "march=armv8.2-a", but found that the "mcpu" command did a marginally better job for compute speeds. We discovered that this was because the mcpu command optimised exactly for the Arm Neoverse V1 architecture, and was able to make use of all its best features, whereas the march command was more generic to all Arm CPUs of that type.

All the flags above in combination have been thoroughly tested (on small problem sizes) to ensure that they individually (and when working together) maximise performance.

1.5 Testing and tuning methods

This was a tediously time-consuming process, involving runs of numerous sizes and varying parameters. Runs did not necessarily have the same best parameters when we increased size, but we did notice a pattern to speeds as we changed them.

The exact parameters we chose to change were chosen after reading documentation and discussion on what the tuning site [7] we used earlier might not have necessarily gotten correct. For our recorded smaller problem size runs (each took roughly 5 minutes or less):

```
N        NB       P        Q        Depth    Align    Time     GFLOPS
64512    128      4        16       1        8        134.3    1332.3
64512    192      4        16       1        8        135.2    1323.5
64512    128      8        8        1        8        135.5    1320.9
64512    64       4        16       1        8        137.0    1305.6
64512    192      8        8        1        8        137.7    1299.5
64512    192      8        8        1        8        138.1    1295.5
64512    64       8        8        1        8        138.5    1291.9
64512    128      16       4        1        8        158.6    1128.3
64512    64       16       4        1        8        159.4    1122.5
64512    192      16       4        1        8        159.9    1119
64512    192      16       4        1        4        161.3    1109.6
64512    192      16       4        1        8        162.7    1099.6
64512    192      16       4        1        16       178.8    1001
```

Figure 2: HPL runs of small problem size, sorted by compute speed - descending.

We can observe that the highest speeds are achieved by a 4×16 process grid - depth was not tested at this stage,but we did briefly skim over a few varying block sizes, scaling in multiples of 64, as well as differing aligns. The result of our findings here indicated that the default align of 8 was best, but that we should also see to a block size of 128 as well as 192. Comparing this to the fastest final runs, this is an interesting result (we will see in more detail later).

Looking at some medium-sized runs, we got:

```
N        NB       P        Q        Depth    Align    Time     GFLOPS
91392    128      4        16       1        8        370.7    1372.6
91392    192      4        16       1        8        371.6    1369.3
91392    128      8        8        1        8        374.2    1359.9
91392    128      8        8        1        8        375.2    1356.0
91392    256      4        16       1        8        381.2    1335
144768   192      16       4        1        8        1538.6   1314.5
144768   192      16       4        1        4        1556.1   1299.8
```

Figure 3: HPL runs of medium size, ordered in each set descending.

What's interesting here is that we found that even for a significantly larger problem size, a process grid of 16×4 performed significantly worse than one for 8×8 or even 4×16.

1.6 Results

```
N        NB      P       Q       Depth   Align   Time    GFLOPS
204672   192     8       8       1       8       3918.3  1458.8
204672   192     8       8       2       8       3924.2  1456.5
204672   192     8       8       2       8       3924.8  1456.3
204672   192     4       16      2       8       3941.8  1450.0
204672   192     4       16      1       8       3947.2  1448.0
204672   128     4       16      1       8       3975.5  1437.8
225792   192     8       8       1       8       5228.8  1467.6
225792   192     8       8       0       8       5328.7  1440.1
226944   192     8       8       1       8       5336.2  1460.3
```

Figure 4: Largest HPL runs done

Our final results are as follows, with the **highest speed at 1467.6** GFLOPs. This was attained after an additional flag, and maxing out problem size to 119GB worth of memory (out of 120GB allowed by Slurm). We did do one run with the full 120GB, using the same parameters as the best run of 119GB, but it ended up marginally slower.

One trend that we noticed coming from all results collated was the distinctly positive correlation between increasing problem size and compute speed. This will be because the more we increase the problem size, the more we need to compute in comparison to the memory we need to move.

Note a few key takeaways for future tuning:

1. Higher problem size generally indicates a higher performance
2. Square process grids typically work better for larger runs, but smaller runs prefer a 4×16 flattened grid. Surprisingly 16×4 grids did the worst.
3. Lookahead depth of 1 was optional, 2 being decent, and 0 being the worst.
4. For lower problem sizes, they were more lenient toward varying block sizes, but for the larger runs, we needed a block size of 192 to reach highest performance.

As to why we chose to show results in a table rather than a graphical way, given the amount of tuneable parameters in HPL, it was inappropriate to have that many graphs looking at correlations, especially given the limitations on compute time feasible. HPL runs, especially the most performant ones, require around an hour and half to run. With the compute nodes becoming especially busy over the past few days, it is just not possible to test meaningfully.

We even saw with a smaller amount of results that what worked on a small problem size did not correlate well to what worked on a larger problem size, so in some sense, there is an element of luck involved in this. In an ideal world, we would've done a significantly larger number of high problem size runs to really see the effectiveness and correlations between different parameter changes.

Below, we list all the results, sorted in descending order of GFLOPs for a four-node Arm Neoverse V1 cluster.

```
N         NB        P         Q         Depth     Align     Time      GFLOPS
225792    192       8         8         1         8         5228.8    1467.6
226944    192       8         8         1         8         5336.2    1460.3
204672    192       8         8         1         8         3918.3    1458.8
204672    192       8         8         2         8         3924.2    1456.5
204672    192       8         8         2         8         3924.8    1456.3
204672    192       4         16        2         8         3941.8    1450.0
204672    192       4         16        1         8         3947.2    1448.0
225792    192       8         8         0         8         5328.7    1440.1
204672    128       4         16        1         8         3975.5    1437.8
91392     128       4         16        1         8         370.7     1372.6
91392     192       4         16        1         8         371.6     1369.3
91392     128       8         8         1         8         374.2     1359.9
91392     128       8         8         1         8         375.2     1356.0
91392     256       4         16        1         8         381.2     1335
64512     128       4         16        1         8         134.3     1332.3
64512     192       4         16        1         8         135.2     1323.5
64512     128       8         8         1         8         135.5     1320.9
144768    192       16        4         1         8         1538.6    1314.5
64512     64        4         16        1         8         137.0     1305.6
144768    192       16        4         1         4         1556.1    1299.8
64512     192       8         8         1         8         137.7     1299.5
64512     192       8         8         1         8         138.1     1295.5
64512     64        8         8         1         8         138.5     1291.9
64512     128       16        4         1         8         158.6     1128.3
64512     64        16        4         1         8         159.4     1122.5
64512     192       16        4         1         8         159.9     1119
64512     192       16        4         1         4         161.3     1109.6
64512     192       16        4         1         8         162.7     1099.6
64512     192       16        4         1         16        178.8     1001
```

Figure 5: Sorted (descending - GFLOPs) table of all results attained.

2 NAMD

This application is a high-performance simulation of large biomolecular systems. It produces simulations of interacting molecules over a series of discrete time iterations. We have been asked to complete runs of NAMD using the STMV input file.

2.1 STMV

The input file for a NAMD run modelled a Satellite Tobacco Mosaic Virus (STVM) in water[5]. The visualisation of this virus can be shown in Figure 6 which was created in VMD using the provided pdb file. This is shown in the visualisation of the virus in

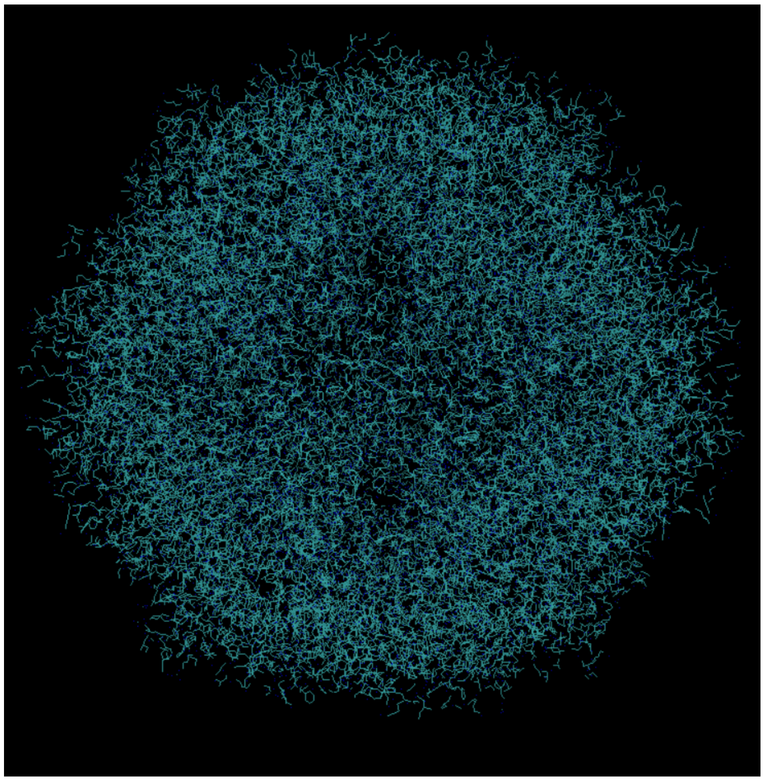


Figure 6: A visualisation of the STMV molecule with the water molecules removed.

We can then state that our runs of NAMD were simulating how a single STMV molecule interacted with a group of water molecules. This simulation ran over 500 time iterations.

2.2 Compiling NAMD

To compile NAMD we first needed to compile and install charm, tcl, and fftw.

Charm has a variety of different flavours it can be compiled as. In our testing we evaluated 'mpi-linux-arm8-mpicxx' with and without shared memory parallelism. We also specified the flags '–with-production -march=native -DCMK_OPTIMIZE' to get the most optimised build we could.

TCL was a straightforward download and compile with no special flags as it doesn't directly impact performance. The main issue we encountered was trying to use pre-built binaries that didn't work due to compiler incompatibilities.

For fftw we tried to use the module on the cluster but found that it was a static library rather than the shared library that NAMD wanted by default. Rather than try to work around that we elected to compile our own optimised version. We also found that the module and our own fftw were fftw3 which NAMD needed an extra flag to attempt to use, we do expect that this lead to a small performance improvement over fftw2.

Once all the pre-requisites were ready we compiled NAMD with the following command, telling the compiler to optimise as much as possible for the architecture. ARM specifically recommends using '-mcpu=native' as opposed to -march and -mtune because it leads to a better optimised binary that doesn't have to support different CPU models within the architecture family. We also tried using -fexpensive-optimisations and found it gave a marginal improvement.

```
./config Linux-ARM64-g++ --charm-arch $BUILD_CHARM_ARCH --with-fftw3 --
   cc-opts '-O3 -mcpu=native ' --cxx-opts '-O3 -mcpu=native '
```

We tried to use the provided ARM performance library fftw binary, however found that it actually reduced the performance of our binary by a small amount. We believe this is potentially to do with the build being equally as optimised as our custom build but having to link against a much larger library as there is no way to only link against the fftw part of the armpl library.

We run out of time but would have been interested in testing the performance of a TCP build of NAMD as given that our cluster has a relatively slow interconnect between nodes we do not expect that we would see a large performance penalty.

2.3 Running NAMD

Our sbatch was fairly simple, loading in the required modules, setting needed libraries into the LD_LIBRARY_PATH, and then running NAMD with mpirun.

We tried various combinations of both the SMP and non-SMP builds of NAMD throughout our testing. SMP is shared memory parallelism and for each spawned mpi process will attached several PE 'worker threads'. This leads to reduced memory usage due to cores being able to share memory (not an issue for us due to the amount of memory in cluster) but normally comes with a performance penalty due to having to use one core per PE group primarily for communication.

We found that we got our best results when using a hybrid approach, having one mpi rank and 15 PE ranks running per node. The increase in performance of this was only minor though as despite having gains due to less MPI overhead on each core we sacrificed 1 core per node to the previously mentioned communication thread. We believe this increase in performance was due to less overhead and more opportunity for cores to exploit shared cache. It should be noted however that SMP builds took significantly longer to initialise meaning that although the ns/day was higher the wall time was noticeably longer than non-SMP builds.

SMP also caused us some initial pain when trying to do a variety of hybrid runs when it pinned multiple processes to the same cores, overriding slurms allocations.

2.4 Results

To get our results we looked through each out our output logs and found the peak ns/day output. We then used that to generate a scaling graph that shows that we have not yet started to see diminishing returns from increasing the number of processes. Our strong scaling graph contains results from pure MPI runs for cores 16-48 and a hybrid run for 64 cores as these were the best results we got.

Our highest overall result was 1.03938 ns/day with a hybrid run, using 1 MPI rank and 15 PE worker threads per node. It's interesting to note that even a single modern GPU would be able to crush this score using the GPU enabled build of NAMD.

Given more time we believe we could have got better results by running the simulation for more time steps. This is because NAMD continually improves it's ns/day throughout a run by running auto optimisations at set intervals.

Given that we have yet to see diminishing returns from adding more cores we believe that in our runs NAMD was primarily compute bound.

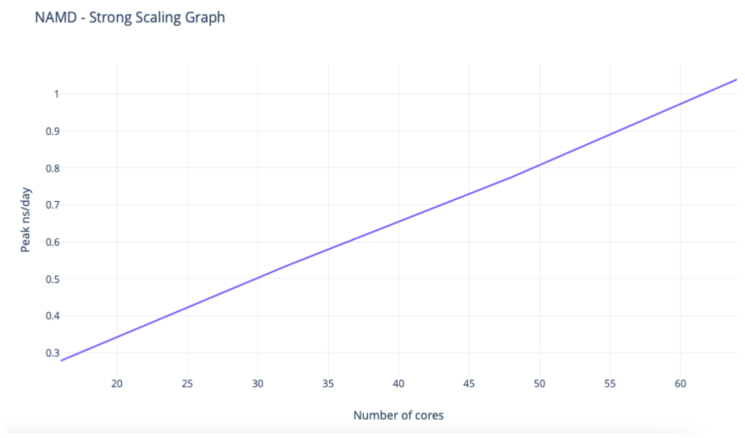


Figure 7: NAMD peak ns/day scaling graph

Cores	ns/day	Run Type
16	0.278032	MPI
32	0.534473	MPI
48	0.7748	MPI
64	1.03938	Hybrid

2.5 Alternative Input Comparison

We didn't see the bonus task on running an extra NAMD input until 22:15 on submission day so this run may or may not make it in :). We decided to try and run the ApoA1 input file as it is a universal standard for benchmarking NAMD.

The ApoA1 benchmark models a protein (specifically apolipoprotein A1, a protein involved in lipid metabolism) and how it behaves when it is submerged in a solvent (in this case water). Given that stmv has 1066628 atoms and 769956 bonds vs apoa1 having 92224 atoms and 70660 bonds we are expecting that the apoa1 benchmark will take significantly less computation time to complete and also result in a higher peak ns/day score.

If the apoa1 run completes (queue is busy as of writing) you will find the output in our github results repo, otherwise there will be a rather sad Orry.

3 MiniVite

MiniVite is a simplified version of Vite that does not apply heuristics to improve the performance of the Louvain algorithm. Both versions implement the distributed Louvain algorithm created by Ghosh et al. which can cluster vertices into communities within a graph containing highly related vertices [3]. The goodness of communities is measured by determining the modularity of the communities; this measures the difference between the fraction of edges within a community compared to the expected fraction that should exist on a random graph with identical vertices and distribution of degree.

$$Q = \frac{1}{2m}\sum_{i=1}^{N}\sum_{j=1}^{N}(A_{ij} - \frac{k_i \cdot k_j}{2m})\delta(c_i, c_j) \tag{1}$$

Where Q is the measure of modularity in the graph, m is the sum of all edge weights, k_i is the weighted degree of vertex i, c_i is the community that contains vertex i and $\delta(c_i, c_j) = 1$ if $c_i = c_j$. The value of modularity ranges between 0 and 1; a value of 1 being the best possible modularity score possible.

As it is a simplified version, that does not use heuristics, this means that it provides a baseline parallel implementation of the Louvain algorithm allowing comparisons to be made between different runs and data. This is important as we want to compare the application's performance across multiple nodes on the cluster.

3.1 Input Files and Louvain Algorithm

We were asked to run the MiniVite application on two different graph input files. These are the LiveJournal dataset [2] and the Pokec dataset [6] which are both graphs of social media applications. The differences between these two graph datasets are provided in Table 1 which looks at the properties of the graph.

Property	LiveJournal	Pokec
Number of nodes	4847571	1632803
Number of edges	68993773	30622564
Number of nodes in largest weakly connected component	4843953	1632803
Number of edges in largest weakly connected component	68983820	30622564
Number of nodes in largest strongly connected component	3828682	1304537
Number of edges in largest strongly connected component	65825429	29183655
Estimate of Average degree per node	14.23	18.75
Average cluster coefficient	0.2742	0.1094

Table 1: A table comparing the different properties of the graph datasets+

With these results, we can see that the LiveJournal data set has around three times as many nodes and twice as many edges in the graph compared to the Pokec dataset. They both represent sparse matrices since the average degree of each node is 14.23 and 18.75 respectively which is far below the $O(V)^2$ number of edges that would be expected in a dense graph. Moreover, we can state that the graph is well connected since 79% of all nodes in both graphs belong to the largest strongly connected component, informing us that most nodes have a direct path between each node $(u, v), (v, u)$. We can hypothesise that most nodes will belong to one large community due to the large number of nodes in the largest strongly connected component with there likely to be many smaller communities between nodes that have fewer connections.

The simple Louvain algorithm works in phases. Each phase starts by trying to assign every vertex v to the communities of its neighbouring vertices u and keeping the change that provides the greatest increase in modularity score. Once all vertices have been checked then all vertices in a community are merged into a meta-vertex (a vertex that shares the same outside connections as all nodes in the community) and then the next phase begins on the new graph. These phases repeat until the change in modularity is below a threshold τ.

3.2 Compilation and Parameter Selection

To compile MiniVite serious modifications were completed to the MakeFile provided in the download. These changes were made manually to the file since no configuration file was provided that could create the required MakeFile. The main dependency for MiniVite was MPI and OpenMP. Compiling with these dependencies was straightforward as OpenMP comes standard with most compilers and requires a flag to be passed in and MPI requires a specialised compiler that can use MPI. As we are using Arm Neoverse CPUs in the cluster we selected armclang++ as our compiler using the specific MPI version MPIC++.

There are four different MPI communication primitives that we could select to compile our code with that would affect the performance.

1. MPI Collectives: -DUSE_MPI_COLLECTIVES
2. MPI Send-Receive: -DUSE_MPI_SENDRECV
3. MPI RMA: -DUSE_MPI_RMA (using -DUSE_MPI_ACCUMULATE additionally ensures atomic put)
4. Default: Uses MPI point-to-point nonblocking API in communication-intensive parts.

Due to time constraints, we could not test the different communication primitives in compilation, so we stuck with the default. Selecting the correct communication primitive is very important; communicating the vertex-community information per iteration is the most expensive step of the distributed Louvain implementation, as mentioned in the implementation of Ghosh et al. [3]

Many different parameters can be passed into the executable at runtime which modify the application's behaviour and the code's performance. We have listed below the important parameters that can affect the code's performance on real-world data. The parameter values selected relied on previous knowledge of the problem within the team and the results were compared to a baseline run on four nodes.

Parameter Name	Flag	Description
Equal Distribution of Edges	-b	This parameter can only be used on real-world inputs and attempts to distribute edges evenly over processes. This can cause an irregular number of nodes across processes.
Algo for random edges	-l	Uses the LCG for choosing random edges over C++ standard algorithm for choosing random numbers
Threshold	-t	The threshold for exiting an iteration of the Louvain algorithm
Number of ranks	-r	This controls the number of aggregators in MPI I/O and is meaningful for input binary files

Table 2: A table showing the different input parameters that can affect the

3.3 Results

As we mentioned above we used the LiveJournal dataset to complete parameter value selection, due to it being the larger graph that was more sparse. We compared the results achieved in the baseline run, looking specifically at the MODS score produced which is the product of the average total time taken multiplied by the final modularity score achieved by the algorithm. As a result of the modularity score, $Q \in [-\frac{1}{2}, 1]$, the lower the value of MODS the better the performance of the code since the average time to compute the communities will be lower. However, the final value for modularity must be taken into consideration as we do not want an exceptionally low MODS score achieved by having a final modularity of 0.01.

Run Type	Final Modularity	MODS
Baseline	0.642421	238.869
Optimised	0.548243	22.0143

Table 3: A table comparing the baseline results of miniVite to the optimised results.

All of the results obtained in Table 3 were from a 4-node cluster with 64 cores in total, running on all nodes. The table compares the baseline run which only passed in the input binary file and the required parameters for MPI with a fully optimised run that used the following parameter values: rank was set to four, the threshold was set to $\tau = 0.01$, we distributed edges evenly across processes. Finally, bonded MPI to sockets and mapped to nodes. We can see that for a small decrease of 0.1 in the modularity of the results we achieved a reduction in the MODS score by a factor of 10. These parameters selected provided a suitable speed increase for the time spent on the problem.

The slurm batch script that was used to create jobs

```
    #!/bin/bash
    #SBATCH --job-name=gotta_go_fast # create a short name for your job
    #SBATCH --nodes=4                # node count
    #SBATCH --ntasks-per-node=1      # tasks per nodes
    #SBATCH --cpus-per-task=16          # cpu-cores per task (>1 if multi
    -threaded tasks)
#SBATCH --time=00:30:00          # total run time limit (HH:MM:SS)

module purge
module load compilers/armclang/24.04
module load libraries/openmpi/5.0.3/armclang-24.04

OMP_NUM_THREADS=16
OMP_PLACES=cores
OMP_PROC_BIND=close

# Baseline run
# mpirun -np 4 $MiniVitePath -f $LiveJournalPath
# Pokec dataset
# mpirun -np 4 --bind-to socket --map-by node $MiniVitePath -f
    $PokecPath -b -r 2 -t 0.01
# LiveJournal
mpirun -np 4 --bind-to socket --map-by node $MiniVitePath -f
    $LiveJournalPath -b -r 2 -t 0.01
```

In the end, we used most of these parameters with only the rank being changed; it was decreased to 2 from 4, due to the 48 core runs across three nodes running out of memory. We had to ensure that all runs used the same parameters to allow direct comparison between the number of cores used. Below in Table 4 are the results of performing this variable core runs on the two different inputs. Using this table we will be able to plot strong-scaling graphs for each input to visually see how the MODS score changes across the number of threads.

Input	Number of cores	Final Modularity	Average Time	MODS score
LiveJournal	16	0.548241	8.25128	4.52369
LiveJournal	32	0.548241	39.843	21.8436
LiveJournal	48	0.548241	34.8863	19.1261
LiveJournal	64	0.548241	39.8533	21.8492
Pokec	16	0.577765	1.95167	1.12761
Pokec	32	0.577765	20.0917	11.6082
Pokec	48	0.577765	14.2057	8.20757
Pokec	64	0.577765	17.1818	9.92704

Table 4: A table showing comparing the results of varying the number of cores used on the two input files.

From the table, we can then plot strong-scaling graphs which show how the MODS score varies over the number of cores that have been used.

3.4 Strong-Scaling Graphs

We have been asked to provide strong-scaling graphs for each of the input files we were asked to run. All these runs must use the same input size and optimisation parameters, allowing comparison only about core and node numbers to be drawn. Below are the two graphs, we provide a plot of the MODS score against the number of cores used. This distributivity was achieved using MPI to communicate between the nodes and OpenMP to communicate between the different node's cores.

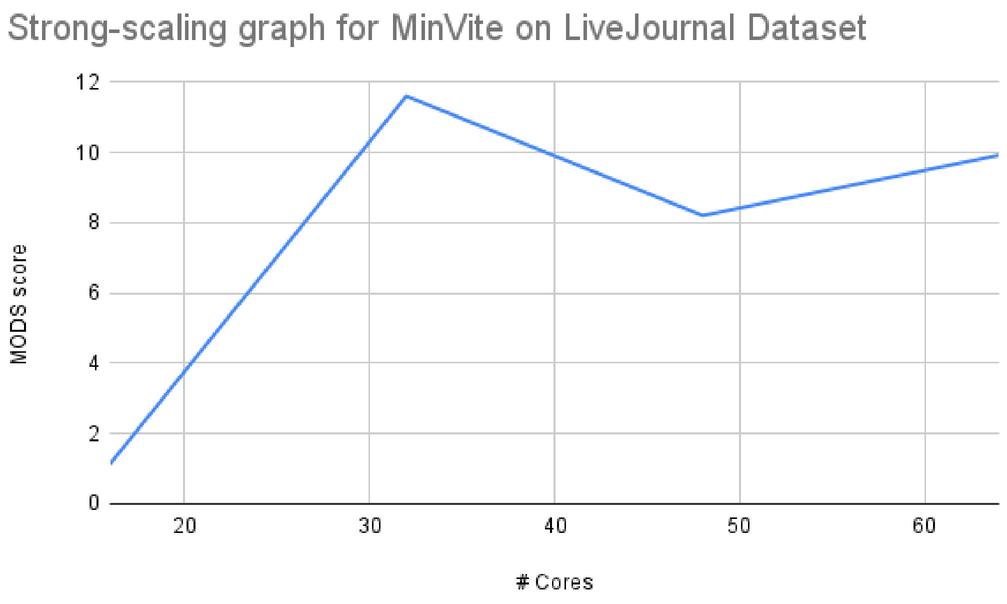


Figure 8: A plot of the strong-scaling graph for the LiveJournal graph dataset, comparing 16, 32, 48, 64 cores.

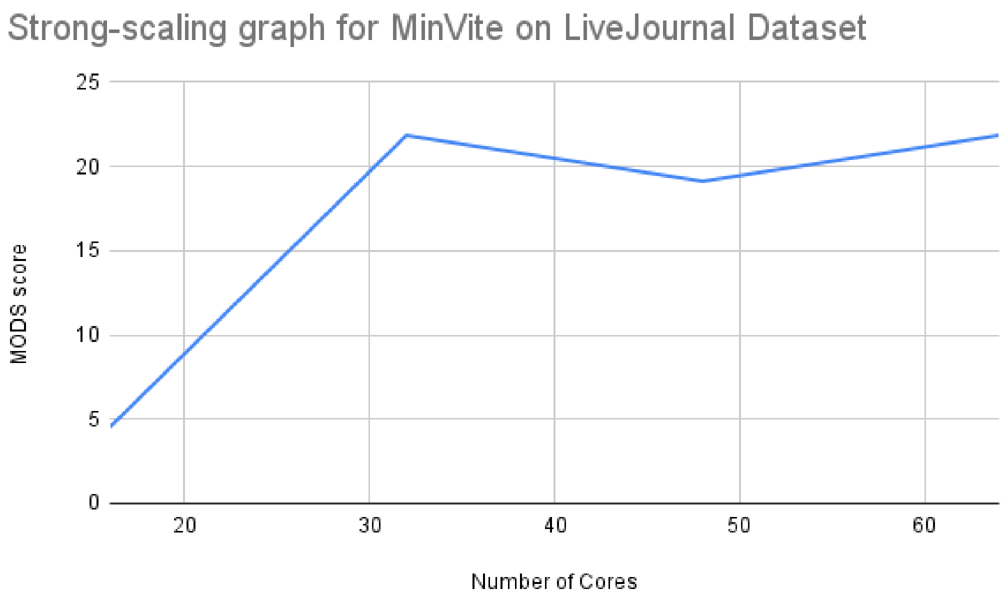


Figure 9: A plot of the strong-scaling graph for the Pokec graph dataset, comparing 16, 32, 48, 64 cores.

The graphs in Figures 8 and 9 show the MODS score increasing as the number of cores increases with some fluctuation between the 32, 48 and 64 core runs. We want to achieve as low a MODS score as possible, due to it encapsulating the average running time, which means we can state that performance decreases as the number of cores increases. Since a large part of the algorithm is spent communicating information between adjacent nodes this makes sense as the more cores that are utilised the longer this communication can take. Using this information and the information provided in the implementation by Ghosh et al. we can conclude that the application is communication bound [3]. Consequently, our scores will not be as impressive as the cluster is already heavily communication-bound due to the 15Gb network connection that the cluster has.

4 OSU

The OSU Micro-Benchmarks suite is a set of benchmarks developed for assessing the performance of communication operations on HPC. This gives cluster admins a standardised toolset to evaluate the performance of different combinations of hardware and library versions, helping them identify bottlenecks and optimise settings for best performance. We focused our benchmark on the point-to-point set of benchmarks which evaluate the performance between two given nodes.

4.1 Compilation

Compiling OSU is very straightforward as it is only dependent on MPI. Our build script simply configured it to use the mpi compilers and then compiled it using make.

4.2 Choosing the binaries

Because we were tasked with finding both the bandwidth and latency between every pair of nodes we elected to use the osu_latency and osu_bw binaries.

To get all the pairwise latencies we could have chosen to use osu_multi_lat but since it evaluates all the latencies at once this could potentially harm performance (with multiple different node pair tests sharing interconnect).

4.3 Production Runs

We decided we needed to script the runs for each unique pair of nodes, rather than trying to do it manually each time. Unfortunately because srun was not properly configured this was quite difficult (otherwise a simple batch script queuing each task up with srun would have worked).

Instead as a workaround a template sbatch file was made for each binary, another batch script was then used to modify the script to use the correct nodes and then queue it using sbatch.

The templates sbatch script is very simple and simply assigns a nodelist for slurm to allocate, also setting –exclusive so that it wasn't sharing the node with any other jobs.

4.4 Optimisations

Because OSU is communication bound, performance is largely not improved by compiler optimisations. Instead, results are mostly dependent on hardware and the mpi library. Because we only had one option for an mpi library (openmpi) and there was no choice on the network interface we were using we chose to instead focus on optimising the usage of the software.

The primary focus for optimisation was getting the MPI processes to run on the same core that the IRQ interrupts were being issued to so there was as minimal cache transfer around the CPU as possible. To do this we need to look at the interrupt information using 'cat /proc/interrupts' and then interpret in a way that could be used to pin the processes to that core.

The following bash will get the core assigned to the first eth0 IRQ interrupt on a given host:

```
INTERRUPT_CORE=$(cat /proc/irq/`grep -m 1 "\eth0." /proc/interrupts |
   awk -F ":" '{print $1}' | awk '{$1=$1;print}'`/smp_affinity_list)
```

In an ideal world, we would also disable the irqbalance daemon while doing this core pinning to avoid the system moving the interrupts to different cores during load balancing, this wasn't done out of concern for impacting other teams results. Given that all compute nodes had the same cores assigned for each queue when we inspected we believe that this load balancing was not particularly active and just kept all queues on the algorithms default cores.

In our first initial attempt to pin cores we made a mistake where instead of binding to the core handling eth0 interrupts (the primary network interface), we instead bound to the core handling the server management port interrupts which resulted in latency instability (see 'bandwidth-pinned' latency graph).

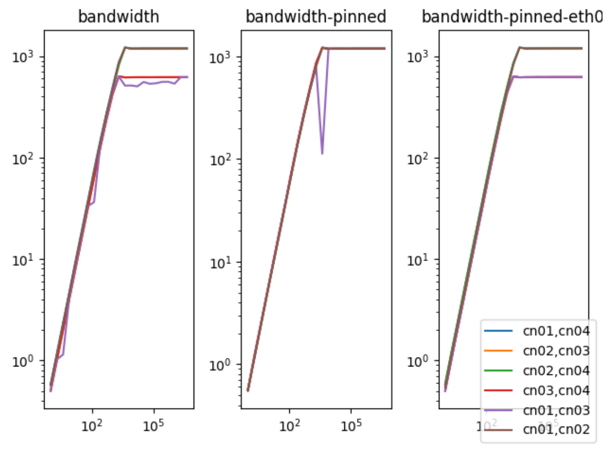
4.5 Results

To analyse our results we wrote a Python script that would collate all the log files and condense them into two CSV files, one for bandwidth and another for latency. We then wrote another script that would create two kinds of graph matrices. The first has a graph per node pair, overlaying each different run on top of each other allowing us to identify the optimisations that worked or didn't work. The other creates a graph for each run and overlays each node pair onto the same graph, allowing for identification of nodes that are outliers.

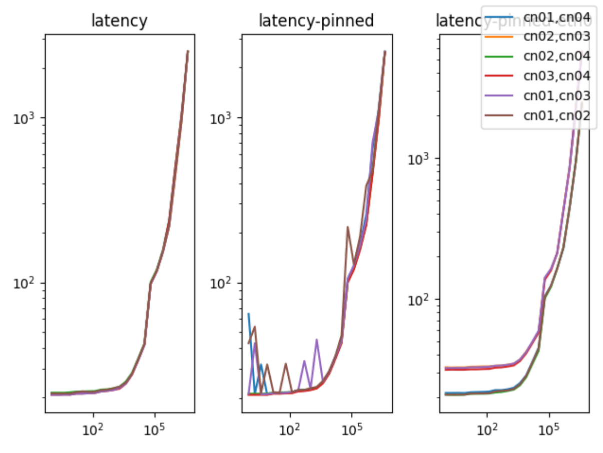
For bandwidth, we can immediately see that we were managing to saturate the network up to around 1200 Mb/s (9.6 Gb/s) with packets of size 4096 and above. This matches up perfectly with the rated 'up to 12.5 Gb/s' network speeds of our cluster nodes (assuming we won't reach peak performance). We could however see that there were several instances where links would only reach half of that at 620 Mb/s, interestingly this was always cn03 but only for 2 of the 3 runs so it appeared to be a momentary network issue rather than a fundamental architecture problem.

For latency we were able to maintain 21us on most connections up until we started sending packets of size 65536 and above, after this point we see exponential growth in latency as then packet sizes grow exponentially. Most interestingly we could see huge amounts of instability at low packet sizes when we pinned the MPI processes to the cores handling management ports. Interestingly, once again cn03 got worse results consistently, but only when pinned to the IRQ core. We believe this isn't due to the core pinning but likely because of the intermittent issue also observed in the bandwidth tests.

These results surprised us because we expected the IRQ core pinning to have a much more significant result than it did. The fact that we got no significant gains from using it indicates that mpirun likely already takes into account IRQ allocations when selecting cores during run time. Potentially if we reran our tests we would not see the intermittent issues with cn03 again but we thought the fact that they appeared was more interesting to include.

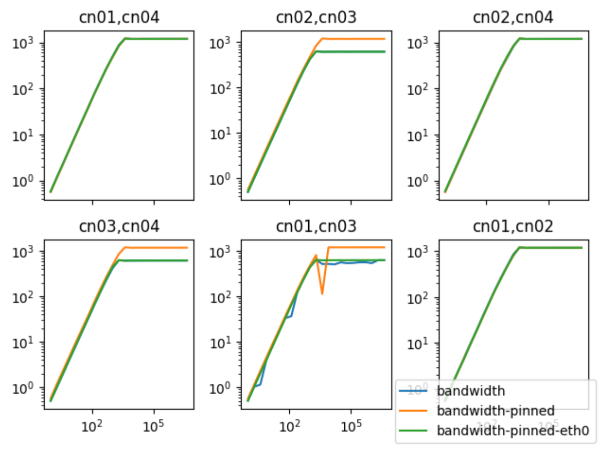


(a) Bandwidth

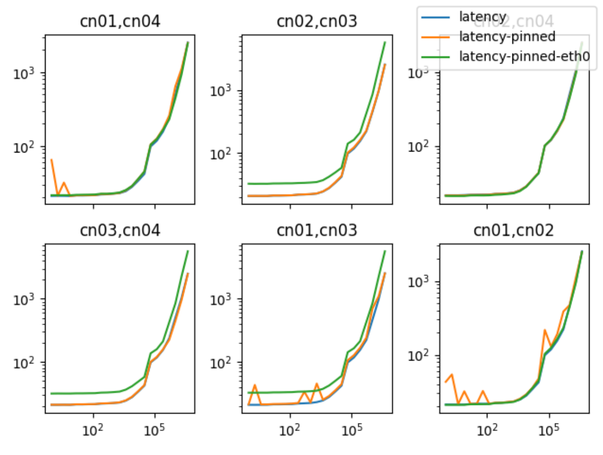


(b) Latency

Figure 10: Different Core Pinning Strategies



(a) Bandwidth



(b) Latency

Figure 11: Different Node Pair Performance

Bandwidth (Mb/s) - pinned to eth0 IRQ

Size	cn01,cn02	cn01,cn03	cn01,cn04	cn02,cn03	cn02,cn04	cn03,cn04
1	0.54	0.50	0.57	0.51	0.59	0.50
2	1.12	1.01	1.18	1.02	1.19	1.02
4	2.25	2.01	2.32	2.04	2.38	2.03
8	4.32	3.99	4.68	4.07	4.74	4.07
16	8.80	8.04	9.39	8.09	9.48	8.15
32	17.48	16.04	18.74	16.18	18.74	16.21
64	35.46	31.90	37.09	31.72	37.30	32.00
128	69.76	63.89	73.88	64.12	74.97	65.01
256	137.47	124.52	141.88	126.44	145.72	125.69
512	252.78	236.91	273.62	240.77	277.04	244.46
1024	464.95	432.51	490.21	428.09	495.66	453.77
2048	821.99	631.82	852.34	630.37	840.96	627.83
4096	1214.29	616.51	1210.32	616.37	1209.06	616.54
8192	1187.38	618.55	1187.49	618.61	1187.43	618.62
16384	1189.23	619.44	1189.12	619.55	1189.10	618.00
32768	1190.21	620.03	1190.12	620.06	1189.49	620.07
65536	1188.94	619.48	1188.97	619.45	1189.11	619.49
131072	1190.19	620.02	1190.18	620.02	1190.21	620.08
262144	1190.69	620.30	1190.67	620.30	1190.73	620.31
524288	1190.93	620.43	1190.93	620.44	1190.96	620.44
1048576	1191.07	620.50	1191.07	620.50	1191.07	620.50
2097152	1191.13	620.53	1191.13	620.54	1191.14	620.54
4194304	1191.17	620.55	1191.16	620.55	1191.16	620.55

Latency (us) - pinned to eth0 IRQ

Size	cn01,cn02	cn01,cn03	cn01,cn04	cn02,cn03	cn02,cn04	cn03,cn04
1	20.92	32.31	21.52	32.68	20.92	31.39
2	20.92	32.30	21.51	32.55	20.86	31.48
4	20.93	32.35	21.54	32.64	20.87	31.43
8	20.95	32.35	21.48	32.61	20.89	31.43
16	21.28	32.66	21.76	32.97	21.23	31.63
32	21.30	32.69	21.83	33.01	21.27	31.64
64	21.38	32.86	21.89	33.15	21.24	31.77
128	21.42	32.98	22.00	33.17	21.32	31.85
256	22.06	33.52	22.54	33.52	21.67	32.48
512	22.27	33.66	22.53	33.71	21.85	32.66
1024	22.44	34.08	22.85	34.13	22.22	33.11
2048	22.98	34.78	23.43	34.76	22.72	33.74
4096	24.81	37.16	25.31	37.17	24.40	36.27
8192	28.44	42.00	28.74	42.30	27.82	41.08
16384	35.35	49.78	35.83	49.35	34.54	48.58
32768	44.52	59.55	45.02	58.44	42.70	57.63
65536	102.88	141.37	104.11	140.67	100.98	136.65
131072	123.98	163.38	124.08	162.07	121.70	158.93
262144	164.12	211.38	164.82	211.34	164.08	211.39
524288	231.72	423.35	228.83	423.28	231.45	423.43
1048576	443.41	846.53	442.18	846.41	443.47	846.57
2097152	935.31	2225.94	935.73	2213.28	937.77	2282.99
4194304	2462.14	5605.34	2452.70	5612.80	2441.52	5676.78

5 STREAM

Stream measures memory bandwidth in 3 different ways, using a basic copy operation, (scalar) multiplication operation, addition operation, and a ”triad” operation, which is a combination of the three above. By default, STREAM allows us to run with OpenMP support for multithreading, but not OpenMP. We'll talk about that later. It uses mass vectorization to perform operations on large arrays, but these arrays must exceed cache size by a significant portion to properly measure bandwidth and not just cache speed.

5.1 Initial compilation

The first step is simple:

```
    gcc stream.c -o stream
```

This basic run puts us in the 10's of thousands of MB/s for bandwidth, done just for the sake of knowing the script works - obviously from here, we can add a lot to improve.
Since this was done after HPL, we managed to gain an immense amount of experience in the best compiler flags and libraries to use. One of the optimisations was to use ”armclang” instead of standard GCC/clang compilers. This was surprisingly a painless experience, as a lot of it was copy and paste.

Originally, we also tried STREAM linking with the Arm Performance Library, but this ended up being slower than without. Therefore, we didn't use it.
Next, since STREAM is configured for OpenMP naturally, the first real optimized compile command was:

```
    armclang -fopenmp stream.c -o stream
```

5.2 Run validity issues

Our first full run, compiled with the following script:

```
    #!/bin/bash

    # Standard further optimisation flags
    OPTIMS="-Ofast -funsafe-math-optimizations -mcpu=native -ffast-math
        -flto"
    CCFLAGS="-fopenmp -fomit-frame-pointer -O3 -funroll-loops"

    # Size of manipulated array of doubles
    STREAMSIZE=500000

    # Arbitrary
    N_TIMES=100

    armclang $OPTIMS $CCFLAGS -DSTREAM_ARRAY_SIZE=$STREAMSIZE -DNTIMES=
       $N_TIMES -O stream.c -o stream
```

This outputted at ≈ 600GB/s. This is an insane (and unrealistic) number. After brief discussion, we decided that it was due to this testing cache mostly rather than memory. After increasing the “STREAMSIZE” variable to $1e8$, which was calculated to suitably exceed cache size on one node, we got an output of 214GB/s, which was a much more believeable result.

Running this on a single compute node was easy, and produced the same result as when ran on the login node (expected since they are of the same specification).

5.3 Results

5.3.1 STREAM for MPI

We would've called it a day there, but then stumbled across a version of STREAM for MPI, also written by the University of Virginia, in the Versions folder of the repository [4]. Reading the documentation, we were expecting to see results scale approximately linearly to number of nodes used (given that the nodes use the same hardware). The script we used to compile was:

```
    #!/bin/bash

    OPTIMS="-Ofast -funsafe-math-optimizations -mcpu=native -ffast-math
        -flto"

    # standard further optimisation flags
    CCFLAGS="-fopenmp -fomit-frame-pointer -O3 -funroll-loops"

    MPdir="/apps/modules/libraries/openmpi/5.0.3/armclang-24.04"
    MPinc=-I$MPdir/include
    MPlib=-L$MPdir/lib/libmpi.so

    STREAMSIZE=200000000
    N_TIMES=100
    mpicc $OPTIMS $CCFLAGS -DSTREAM_ARRAY_SIZE=$STREAMSIZE -DNTIMES=
       $N_TIMES -O stream_mpi.c -lmpi -o mpi_stream
```

Across the 4 nodes, we worked out at 3× cache size across all nodes total, with the specific formula:

$$4 \times (16 \times (L1 + L2) + L3)$$

For the Arm Neoverse V1 architecture, where L1 and L2 are independent for cores, and L3 is shared, then multiplied by 4 for the 4 running nodes, which results in a minimum size, after multiplying by 3, of approximately 3×10^8 array size for elements of 8 bytes. For safety and to guarantee validity of results, we set array size to 2×10^9 (we describe the details of cache sizes in 5.3.2).

The Slurm batch script is as follows:

```
    #!/bin/bash
    #SBATCH --job-name=stream4
    #SBATCH --nodes=4
    #SBATCH --ntasks-per-node=1
    #SBATCH --cpus-per-task=16
    #SBATCH --mem=120G
    #SBATCH --time=00:05:00
    #SBATCH --output=new_mpi_build/mpistream4_%j.out

    module load compilers/armclang/24.04
    module load libraries/openmpi/5.0.3/armclang-24.04

    OMP_NUM_THREADS=16

    # All nodes, all cores
    mpirun -np 4 -map-by node ./STREAM/mpi_stream
```

Explaining this briefly, we want Slurm to run the MPI-enabled STREAM executable on 4 nodes. There are 2 ways to go about this:

1. Let MPI handle all multithreading. We do this by setting nodes to 4, tasks per node to 16, and cpus per task to 1. Then, for the run command:

   ```
           mpirun -np 64 --map-by ppr:16:node --bind-to core ./STREAM/
               mpi_stream
   ```

   This uses thread pinning to ensure 16 processes per node, where each process is associated with a single core (compiled without -fopenmp flag).

2. Secondly (and what we ended up doing), we could use a combination of MPI and OpenMP. Since the STREAM file was configured to use both MPI and OpenMP here, its usually better to use a combination because OpenMP lets you use shared memory parallelism, whereas MPI does not (meaning that even if 2 cores (and MPI processes) were on the same CPU, they would have to communicate via MPI). We ran with the Slurm batch script specified above, where each of the 4 tasks are split to 16 CPUs per task, and we map each task to a node, and set thread count to number of cores, OpenMP will automatically distribute threads to unique cores.

The second of the two is both theoretically and practically better (not by that much, but still nicer). Looking at the scaling results across a number of different nodes, to show validity:

	1 node	2 nodes	3 nodes	4 nodes
Triad (MB/s)	214153.4	411671.9	636297.4	846718.5

Taking into account general error for the fact that STREAM runs do vary naturally by a good margin on occasion, results show that scaling is approximately linear, which is what we expect.

5.3.2 Final valid non-MPI run

In the end, after observing the code for the MPI implementation of STREAM, it was decided that this benchmark wasn't suitable for the purposes and intents of STREAM. Getting into the details of cache, there is 64KB of data and instruction cache (each) for L1 per core, L2 is configurable at 512KB-1MB per core, we will assume upper bound here, and L3 is up to 32MB, again assuming upper bound. L3 is shared across cores, and then we add $16 \times 1MB(L2) + 16 \times 128KB(L1)$, to get 50MB of cache per node. Then, multiplying by 4 to make sure run is valid, we get 200MB of array size required, which is an array of size 25M doubles. Rounding up for good measure to 17M, we will use this in runs.

More rigorous testing of this setup indicated some key facts:

1. Optimisations like -O3, or loop unrolling didn't make a big difference to recorded bandwidth
2. The unsafe math flags did the most in speeding up computation

3. When we move closer and closer to the cache size of the cluster, computation gets much faster. In that sense, worrying about particulars with small increments in problem size is a bit pointless, as one could easily get higher numbers just by decreasing problem size (and increasing cache hits consequently).

4. Results vary significantly between run-to-run anyway, especially if the number of repeats isn't large.

Our final run ended up at 227.7GB/s, a reasonable and expected result.

Overall, there isn't much else to say about STREAM given that our end goal is to simply test the memory bandwidth on a single node, giving us a decent expected figure. For a problem size significantly larger than the 25M we used, we noticed a large tail-off in performance down to about 214GB/s. Although this was initially surprising, it was likely due to the problem size becoming so big that performance became compute-bound, rather than the intended memory-boundness.

References

[1] Arm. Arm Forge: Performance Reports. `https://developer.arm.com/documentation/101136/22-1-3/Performance-Reports`.

[2] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.

[3] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarrià-Miranda, Arif Khan, and Assefaw Gebremedhin. Distributed louvain algorithm for graph community detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 885–895, 2018.

[4] John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. In *Proceedings of the 8th International Conference on High-Performance Computing*, pages 19–25, 1997. `https://www.cs.virginia.edu/stream/FTP/Code/Versions/stream_mpi.c`.

[5] Alexander McPherson and Fei Sun. Molecular dynamics of viruses, 2013.

[6] Lubos Takac and Michal Zabovsky. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*, volume 1, 2012.

[7] Advanced Clustering Technologies. How do i tune my hpl.dat file? `https://www.advancedclustering.com/act_kb/tune-hpl-dat-file/`.