

Assignment 3 - MPI

Zirong Cai, Phuong Nguyen

5. Dezember 2021

1. Amdahl's Law(1P)

1.

$$\begin{aligned}
 Eff &= \frac{Sp}{p} \\
 &= \frac{1}{sp + (1 - s)} \\
 &= \frac{1}{0.1p + 0.9} \geq 0.7
 \end{aligned}$$

Solve the inequality above we have $p \leq 5.2$, since p must be a integer, we have $p_{max} = 5$.

2. For a given Program, if the non-parallelizable part of a program accounts for 10% of the runtime ($s = 0.9$), we can get no more than a 10 times speedup, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units. On the other hand, if we want to achieve 70% efficiency, There's an upper bound of the number of processes p , meaning, the more processes we use, the worse the efficiency becomes.

Amdahl's law only applies to cases where the problem size is fixed. In practice, as more computing resources become available, they tend to get used on larger problems (larger datasets), and the time spent in the parallelizable part often grows much faster than the inherently serial work. In this case, Gustafson's law gives a less pessimistic and more realistic assessment of parallel performance [1]:

$$S_{latency}(s) = 1 - p + sp$$

2. Process Pinning within a Compute Node (1P)

a. Task 1 - 2 MPI and 4 OMP

```

di29waj@i22r07c05s03:~/hpclab/A3/src/pinning$ OMP_NUM_THREADS=4 \
    LMPI_PIN_DOMAIN=omp LMPI_PIN_ORDER=scatter \
    LMPI_PIN_CELL=core mpirun -np 2 ./pinning
rank=0; thread=0; cpu id=0
rank=0; thread=1; cpu id=1
rank=0; thread=2; cpu id=2
rank=0; thread=3; cpu id=3
rank=1; thread=0; cpu id=14
rank=1; thread=1; cpu id=15
rank=1; thread=2; cpu id=16
rank=1; thread=3; cpu id=17

```

b. Task 2 - 4 MPI and 2 OMP

```
di29waj@i22r07c05s03:~/hpclab/A3/src/pinning$ OMP_NUM_THREADS=2 \
KMP_AFFINITY="explicit,proclist=[0,6,7,13,14,20,21,27]" \
IMPLPIN_ORDER=spread mpiexec -n 4 ./pinning 2> /dev/null
rank=0; thread=0; cpu id=0
rank=0; thread=1; cpu id=6
rank=1; thread=0; cpu id=7
rank=1; thread=1; cpu id=13
rank=2; thread=0; cpu id=14
rank=2; thread=1; cpu id=20
rank=3; thread=0; cpu id=21
rank=3; thread=1; cpu id=27
```

c. Optional task - pinning for a pure MPI job

When running a job on a cluster, besides pinning MPI processes through environment variables, one can also achieve the same thing through `srun` variables. For example, to get the same pinning as in the task 1 and task 2, we need the following variables:

- `-n` `j`: number of MPI tasks
- `-mpi=pmi2`: specify which PMI library to use
- `-cpu-bind=map_cpu:j`: list of cores
- `-hint=nomultithread`: disable hyperthreading

The MPI pinning in task 1 and task 2 (without OpenMP) can be achieved as follows:

```
di29waj@i22r07c05s02:~/hpclab/A3/src/pinning$ srun -n 2 --mpi=pmi2 \
--cpu-bind=map_cpu:0,14 --hint=nomultithread ./pinning
rank=0; thread=0; cpu id=0
rank=1; thread=0; cpu id=14

di29waj@i22r07c05s02:~/hpclab/A3/src/pinning$ srun -n 4 --mpi=pmi2 \
--cpu-bind=map_cpu:0,7,14,21 --hint=nomultithread ./pinning
rank=0; thread=0; cpu id=0
rank=1; thread=0; cpu id=7
rank=2; thread=0; cpu id=14
rank=3; thread=0; cpu id=21
```

3. Analyzing an HPC Network (2P)

a. General information about OmniPath

Omni-Path Architecture (OPA) is a Intel's high-performance communication architecture which is used for as interconnects for communications in a some supercomputers, for example SuperMUC-NG, CoolMUC-3. The target of Ommin-Path is lower communication latency, lower power consumption, and a higher throughput. It is claimed by Intel as a better alternative to Nvidia's Mellanox Infiniband for communications in big-scale applications. The special features of OPA for traffic management referred from [2] are:

- Adaptive Routing: selects the least congested path to balance the packet load.
- Dispersive Routing: distributes packets across multiple paths to reduce congestion.
- Traffic Flow Optimization: diminishes the variation in latency for high priority traffic.

- Packet Integrity Protection: allows for rapid, transparent recovery from transmission errors.

Explain why the bandwidth depends on the message size According to [3], the Omni-Path HFI has different hardware support for different send modes which are Programmed I/O (PIO) send and send direct memory access (SDMA). The PCI Express (PCIe) of PIO hardware blocks has the bandwidth of 10.6 GBps while the one of SDMA has the bandwidth of 12.5 GBps. The paper also provided the information that the sending small messages usually involves PIO send contexts, whereas SDMA engines are used for large message transfers. Thus, the bandwidth depends on the message size, the bandwidth is larger when sending big-size messages since SDMA hardware is used for Verbs transaction and SMMA blocks have higher maximum bandwidth.

The paper [4] also measure how the bandwidth depends on the message size, as presented below: In Fig. 1, when transferring messages with size of 2^{14} , the bandwidth is only around

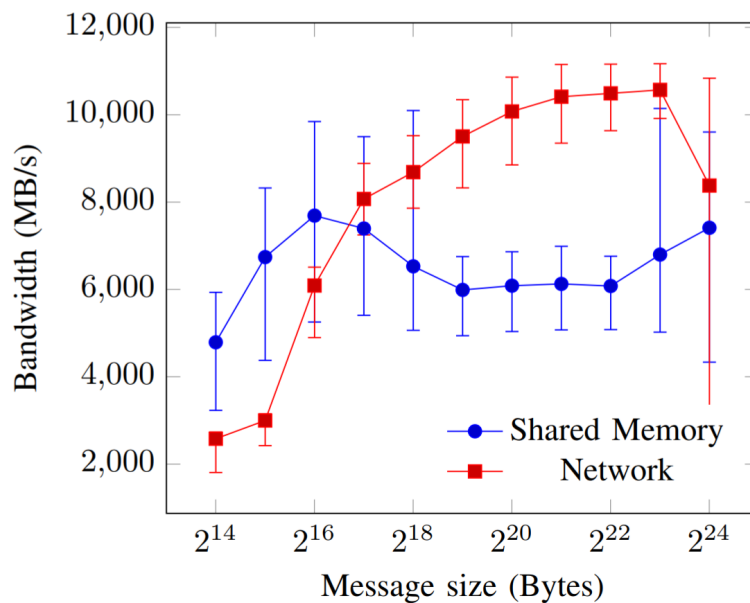


Fig. 2. Mean achieved throughput with OSU bidirectional point-to-point bandwidth benchmark [26] when two MPI ranks run on the same compute node and transfer messages via shared memory (blue) or on different compute nodes and transfer messages through the network (red). Error bars indicate the minimum and maximum values observed over approximately 100 trials.

Abbildung 1: Measurement of bandwidth respect to message sizes [4]

2500 MB/s. The bandwidth increases when the transferred-message sizes are increased, reaching the peak of 11000 MB/s at the message size 2^{20} . This confirmed out previous explanation.

What latencies and peak bandwidths can you expect? For peak bandwidths, one can expect the maximum bandwidth that PCIe of SDMA can handle which is 12.5 GBps. For latency, according to [2], it can be as small as 1 microsecond for small messages, and up to milliseconds for large messages.

b. Network Topology

The topology for provided information is presented in the Figure 2. Since we have 10 leaf

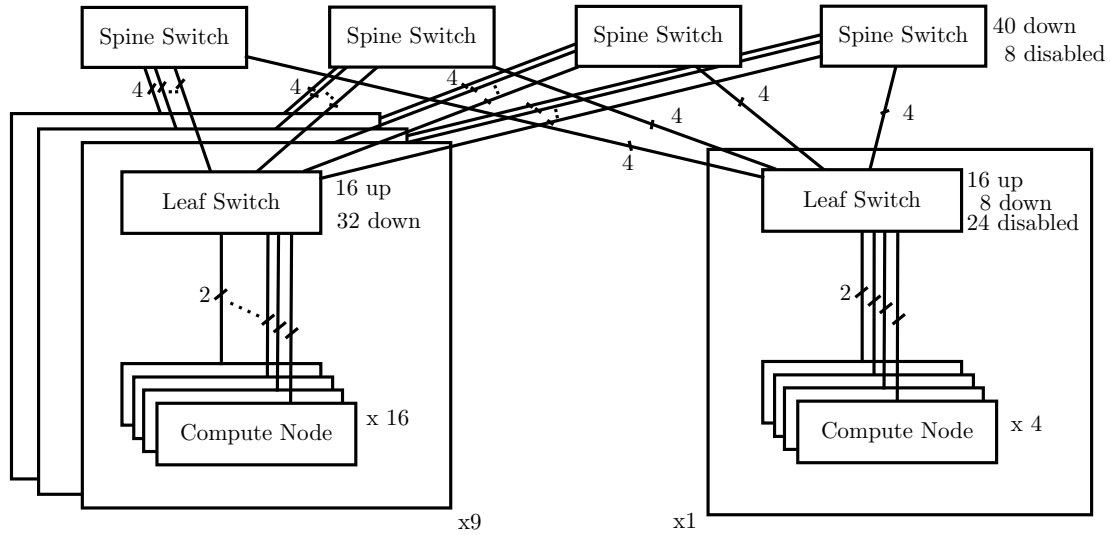


Abbildung 2: Network topology for CoolMUC-3

switches, each switch have 32 ports for connecting to nodes, and each node has 2 links to fabric, thus we can connect each leaf switch to 16 nodes. With 148 nodes, we can have 9 leaf switches which have 16 nodes/switch, and 1 leaf switch which has 4 nodes/switch (total node = $9 \cdot 16 + 1 \cdot 4 = 148$ nodes). Each leaf switch, besides 32 ports for nodes, has 16 ports for the spine switches. Each leaf switch connects to 4 spine switches, thus using 4 ports/spine switch. The ratio of ports for lower level and upper level is 32:16 thus 2:1. For the spine switches, each of them connects to 10 leaf switches, so using $4 \cdot 10 = 40$ ports. Thus, there are still 8 unused ports. The information about how spine switches are connected are not provided in the description, thus it is not clear enough for us to illustrate it in the figure.

c. Bisection width and bisection bandwidth

32 nodes Since we have 10 nodes per leaf switch, then 32 nodes are connected to two leaf switches. Each leaf switch connect to 4 spine switches with 4 links per spine switch. Thus bisection width is $4 \cdot 4 = 16$ links which is $16 \text{ links} \cdot 100 \text{ Gbit/link} = 16000 \text{ Gbit}$.

16 nodes All nodes are connected to a single switch. Since we don't know how they are routed inside the switch, we assumed 2 following cases:

- Assume that all nodes are connected by a single point inside the switch, then the bisection width is the number of links of 8 nodes which is $8 \cdot 2 = 16$ links. The bisection bandwidth is then 16000 Gbit.
- Assume that all nodes are connected pairwise inside the switch. Then we have C_2^{16} connections between nodes. To separate these nodes to 2 domains, one needs to cut $C_2^{16} - 2 \cdot C_2^8 = 64$ connections. If each connection is a single link, then the bisection

width is 64 whereas the bisection bandwidth is 64000 Gbits. If each connection is a double link (the same bandwidth as the bandwidth between nodes and the leaf switch), then the bisection width is $64 \times 2 = 128$, while the bisection bandwidth is 128000 Gbits.

4. Broadcast (3P)

a. Code

Trivial Version(root sends messages to all other processes):

```
void my_bcast(void *data, int count, MPI_Datatype datatype, int root,
             MPIComm communicator)
{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);

    if (world_rank == root)
    {
        int i;
        for (i = 0; i < world_size; i++)
        {
            if (i != world_rank)
            {
                //root sends message to all other processes
                MPI_Send(data, count, datatype, i, 0, communicator);
            }
        }
    }
    else
    {
        MPI_Recv(data, count, datatype, root, 0, communicator,
                MPI_STATUS_IGNORE);
    }
}
```

Tree Topology:

```
void broadcast_tree(void *data, int count, MPI_Datatype datatype,
                  int root_proc, int end_proc, MPIComm communicator)
{
    if (root_proc == end_proc)
    {
        return;
    }
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int receive_proc = root_proc + (end_proc - root_proc + 1) / 2;
    if (rank == root_proc)
    {
        MPI_Send(data, count, datatype, receive_proc, 0, communicator);
    }
    else if (rank == receive_proc)
    {
        MPI_Recv(data, count, datatype, root_proc, 0, communicator,
                MPI_STATUS_IGNORE);
    }
    broadcast_tree(data, count, datatype,
                  root_proc, receive_proc - 1, communicator);
}
```

```

        broadcast_tree(data, count, datatype,
                      recieve_proc, end_proc, communicator);
    }

```

Buffer Optimization:

```

void broadcast_buffer(void *data, int count, MPI_Datatype datatype, int root,
                    MPIComm communicator)
{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);

    int buffer_size = 262144;
    int packet_num = (count / buffer_size);
    int remainder = count % buffer_size;
    if (world_rank == root)
    {
        for (size_t j = 0; j < packet_num; j++)
        {
            MPI_Send((double *)data + (j * buffer_size), buffer_size,
                    datatype, (world_rank + 1) % world_size, 0, communicator);
        }
        //remainder
        MPI_Send((double *)data + (packet_num * buffer_size), remainder,
                datatype, (world_rank + 1) % world_size, 0, communicator);
    }
    else
    {
        for (size_t j = 0; j < packet_num; j++)
        {
            MPI_Recv((double *)data + (j * buffer_size), buffer_size,
                    datatype, (world_rank + world_size - 1) % world_size, 0,
                    communicator, MPI_STATUS_IGNORE);
            MPI_Send((double *)data + (j * buffer_size), buffer_size,
                    datatype, (world_rank + 1) % world_size, 0, communicator);
        }
        //remainder
        MPI_Recv((double *)data + (packet_num * buffer_size), remainder,
                datatype, (world_rank + world_size - 1) % world_size, 0,
                communicator, MPI_STATUS_IGNORE);
        MPI_Send((double *)data + (packet_num * buffer_size), remainder,
                datatype, (world_rank + 1) % world_size, 0, communicator);
    }
}

```

b. Evaluation

1) Optimize the bandwidth

From the table 1, we see that the communication volume w.r.t. message size of the entire broadcast routine of trivial and tree are actually the same (in the assignment it's indicated that the communication volume of tree algorithm should be higher than trivial algorithm which really confusing me) while the message transformed in critical path of tree algorithm is much less than in trivial algorithm (\log_2^n vs. n). This can also be observed from the execution time. Also we see that when the process number is 8, the execution time of MPI_Bcast is almost the same as in trivial algorithm, but as the process number increases, the performance of MPI_Bcast becomes better than tree algorithm. This may mean,

MPI_Bcast has different algorithm implemented regarding to different process number.

Topology	NUM_PROC	Critical Path		Entire Broadcast	Execution Time
		MPI-Msg	Amount of Data	Msg-Size	
Trivial	8	7	$7 * 16KB$	$7 * 16KB$	4.67928e-05
	14	13	$13 * 16KB$	$13 * 16KB$	6.7117e-05
	28	27	$27 * 16KB$	$27 * 16KB$	0.000120425
Tree	8	3	$3 * 16KB$	$7 * 16KB$	2.04132e-05
	14	4	$4 * 16KB$	$15 * 16KB$	1.99638e-05
	28	5	$5 * 16KB$	$27 * 16KB$	3.4531e-05
Buffer	8	7	$7 * 16KB$	$7 * 16KB$	8.91956e-06
	14	13	$13 * 16KB$	$13 * 16KB$	5.6849e-05
	28	27	$27 * 16KB$	$27 * 16KB$	7.08333e-05
MPI_Bcast	8	Unknown	Unknown	Unknown	6.53468e-05
	14				9.25825e-06
	28				1.41854e-05

Tabelle 1: Broadcast using 3 different topology with array size $n = 2048$

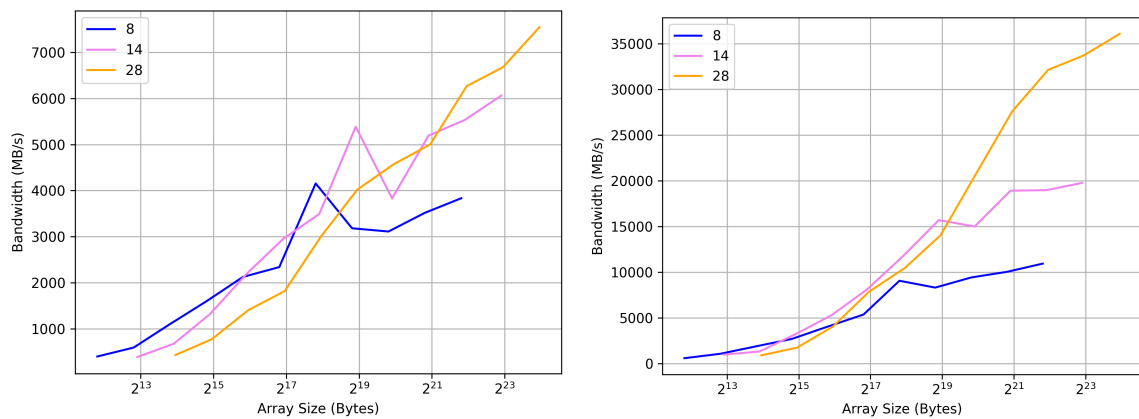


Abbildung 3: Collective Network Bandwidth (left: Trivial, right: Tree)

Figure 3 shows the collective network bandwidth of trivial and tree algorithm. I expected it to converge when the data size grows (means the channels are fully used). And it seems like not the case in the figure. I am not sure if it's because the data size we choose didn't reach the upper bound of the bandwidth or some other reason. Also it's notable that there's a big drop of process 8 and 14. The reason of this could be that MPI uses different protocols regarding to different data size. There are two protocols of MPI_Send and Recieve:

- Eager protocol

Send messages directly, suitable for short messages

- Rendezvous protocol

Send header to retrieve address, then send message, suitable for long messages

So instead of sending message directly, rendezvous protocol wait until the receiver told sender the address where he should write the data to, and this will result in extra time

cost and thus reduces the bandwidth.

2) Optimize the buffer

The idea of optimizing the buffer usage is that when the size of data go beyond the buffer size, we may need to transmit multiple times for one message. While rank 0 is transmitting data to rank 1, rank 1 need to wait until all messages are received and then start transmitting to next rank. But this don't have to be like this. We can split the data into multiple small packets, After rank 1 has received the first packet, it then start to transmit the packet to next rank while still waiting for the second packet from rank 0. The process can be seen below in 5. Instead of transmitting $(p - 1) * (n/buffer_size)$ packets in critical path, now we only need to transmit $(n/buffer_size) + p - 2$ packets. One problem remained is how to set the buffer size since there is no documentation about the buffer size of MPI(or there is but we can't find), but according to our observation above, we decide to choose the buffer size where the bandwidth drops, so $n = 2^{18}$.

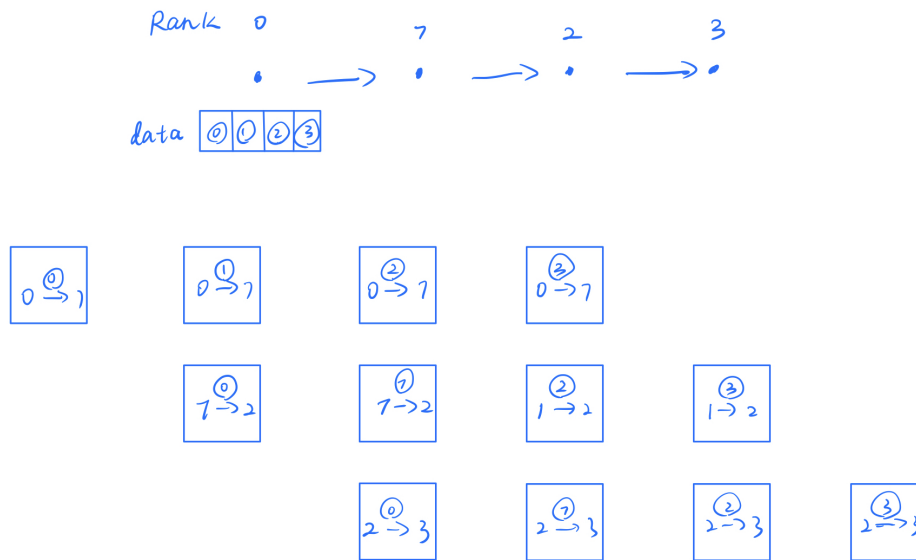


Abbildung 4: Buffer Optimization Algorithm

And the result is shown in 5. Comparing the performance of these three algorithm, we saw that the tree algorithm gives us the best performance and then the buffer one, the trivial one is with no doubt the worst one.

3) Hyper mode Since the tree algorithm optimizes the usage of bandwidth and the third algorithm optimizes the usage of buffer, combine this two algorithm may give us a better performance. But due to the time limit we didn't implement this.

In figure 6, we approximate the bandwidth using $B = arraySize/time$. MPI_Bcast shows an amazing performance. Thinking of the perfect network with M nodes, where every node has a dedicated wire to each other node. Then when we double the process number while keeping the data size, the execution time of our application should stay the same, and thus the bandwidth.

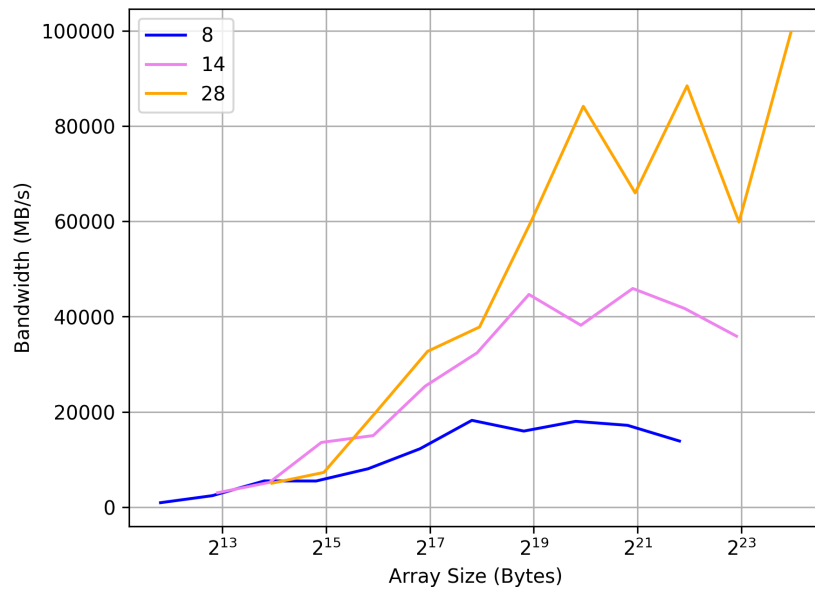


Abbildung 5: Collective Network Bandwidth(buffer optimize)

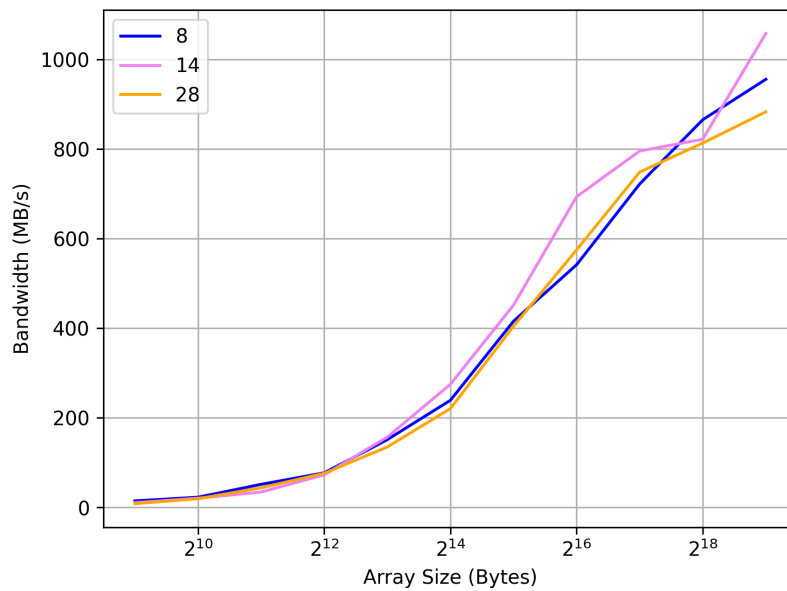
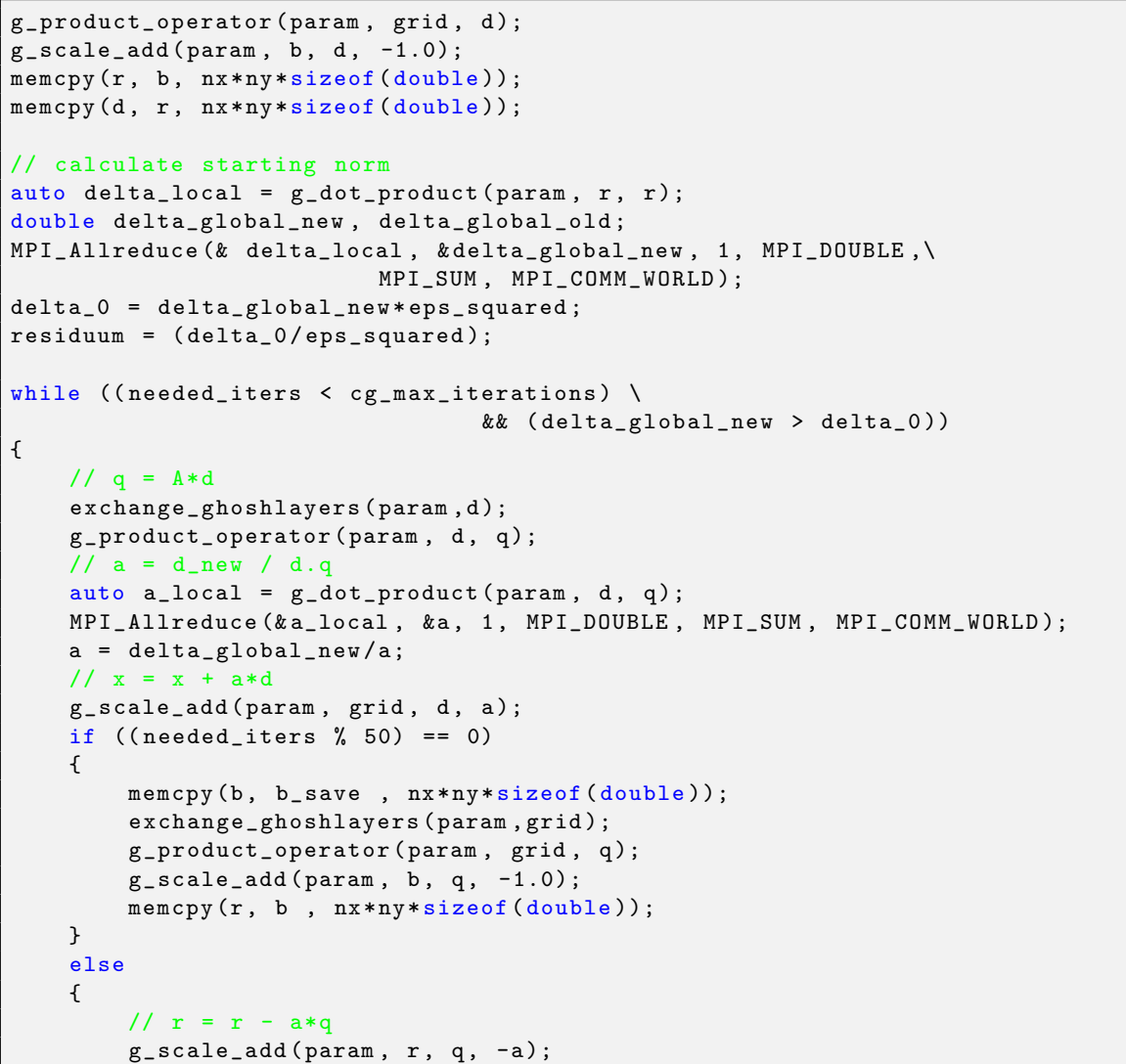


Abbildung 6: Approximated Bandwidth(MPI_Bcast)

5. 5 Parallel CG (3P)

a. Parallelism Strategy

Input and output To avoid storing the whole data in 1 nodes, and avoid the cost of broadcast and gathering, we decided to parallelize input and output processes of the given code. For input, each process initializes each own data and stores it in a local storage. Then the program called the solver. At the end, each node write its own data into the output files. Since we don't have parallel IO yet, we write in serial, each process hold the file handler once, and write to the file. The lines in the resulted output files do not keep the same order as the output in the serial case, but it does not affect to the visuallisation, since for visuallisation we plot for each value of a point (x,y).

Parallel Solver We adjusted all helping functions so that each rank only iterates in their local domains. Besides, each the solver function, we need to have 3 communication points which call *MPI_Allreduce*. The idea is we calculate all vector by vector locally, and broadcast and sum up the resulted double. This helps in calculating the global *delta* and global *a*. Besides, we also need to exchange ghosh cells to update informations of the grids every time before we applies 5-points stencil to the grids. The implementation of the parallel solver is presented below. 

```
g_product_operator(param, grid, d);
g_scale_add(param, b, d, -1.0);
memcpy(r, b, nx*ny*sizeof(double));
memcpy(d, r, nx*ny*sizeof(double));


// calculate starting norm
auto delta_local = g_dot_product(param, r, r);
double delta_global_new, delta_global_old;
MPI_Allreduce(& delta_local, &delta_global_new, 1, MPI_DOUBLE, \
              MPI_SUM, MPI_COMM_WORLD);
delta_0 = delta_global_new*eps_squared;
residuim = (delta_0/eps_squared);

while ((needed_iters < cg_max_iterations) \
        && (delta_global_new > delta_0))
{
    // q = A*d
    exchange_ghoshlayers(param,d);
    g_product_operator(param, d, q);
    // a = d_new / d.q
    auto a_local = g_dot_product(param, d, q);
    MPI_Allreduce(&a_local, &a, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    a = delta_global_new/a;
    // x = x + a*d
    g_scale_add(param, grid, d, a);
    if ((needed_iters % 50) == 0)
    {
        memcpy(b, b_save , nx*ny*sizeof(double));
        exchange_ghoshlayers(param,grid);
        g_product_operator(param, grid, q);
        g_scale_add(param, b, q, -1.0);
        memcpy(r, b , nx*ny*sizeof(double));
    }
    else
    {
        // r = r - a*q
        g_scale_add(param, r, q, -a);
    }
}
```

```

}
// calculate new deltas and determine beta
delta_global_old = delta_global_new;
delta_local = g_dot_product(param, r, r);
MPI_Allreduce(&delta_local, &delta_global_new, 1, MPI_DOUBLE, \
              MPI_SUM, param->comm_2d);
beta = delta_global_new/delta_global_old;
// adjust d
g_scale(param, d, beta);
g_scale_add(param, d, r, 1.0);
residuum = delta_global_new;
needed_iters++;
}

```

Ghosh cell exchange Each rank has 4 neighbors, when we want to exchange ghoshcells, the rank needs to communicate with these fours. Communications with top and bottom neighbors are pretty simple since the data is contiguous in memory. But, for the case with left and rights, since we need to to send columns data but data was stored in row-major, we need to define an additional data type with *MPI_Type_vector*. The ghosh cell exchanges is presented in the code below. 

```

// Exchange top most row to the top neighbors:
MPI_Sendrecv(grid + (ny-2)*nx, nx, MPI_DOUBLE, param->topo_top, toptag,
             grid, nx, MPI_DOUBLE, param->topo_bottom, toptag,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Exchange row most row to the bottom neighbors:
MPI_Sendrecv(grid + nx, nx, MPI_DOUBLE, param->topo_bottom, downtag,
             grid + (ny-1)*nx, nx, MPI_DOUBLE, param->topo_top, downtag,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

// Data type for sending columns
MPI_Datatype column_type;
MPI_Type_vector(ny, 1, nx, MPI_DOUBLE, &column_type);
MPI_Type_commit(&column_type);
For that
// Exchange left most column to the left neighbors:
MPI_Sendrecv(grid + 1, 1, column_type, param->topo_left, lefttag,
             grid + nx - 1, 1, column_type, param->topo_right, lefttag,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// Exchange right most column to the right neighbors:
MPI_Sendrecv(grid + nx - 2, 1, column_type, param->topo_right, righttag,
             grid, 1, column_type, param->topo_left, righttag,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```

b. Results and Discussions

Validation The program was ran with gridsize = 0.001 and eps = 0.0000001 on the local computer. The output then visualized with gnuplot. Resulted picture is as blow:

The number of required iterations and final residum remained unchanges when we ran the program with different number of cores and different MPI topology (Table ??). The Figure 7 showed the expected solution. Thus this parallel solver is reliable.

Performance of different process-grids We tested the program on 1 node of CM2 with 28 MPI processes, 1 process/core. The gridsize is 0.0005 and eps is 0.0005. The table below presented the performance in respect to different MPI topology.

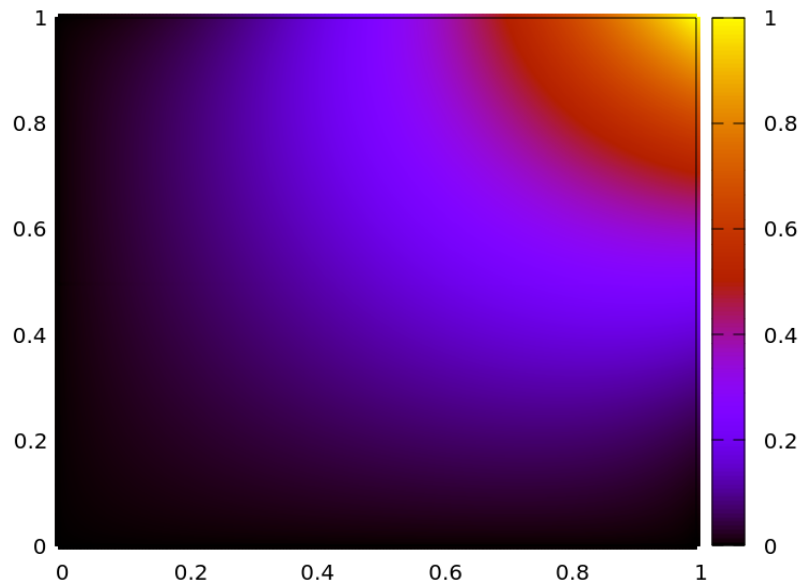


Abbildung 7: Visualization of solution with parallel solver

nMPI	TopoX	TopoY	nIterations	Residum	Time
28	1	28	1717	1.24881e-17	5.41372
28	2	14	1717	1.24881e-17	5.30194
28	4	7	1717	1.24881e-17	5.30789
28	7	4	1717	1.24881e-17	5.36031
28	14	2	1717	1.24881e-17	5.43706
28	28	1	1717	1.24881e-17	5.72815

table:topo1)

Regarding the runtime, there is no significant differences between different topology choices. The (2x14) and (4x7) seems to have the best performance. Normally, we would expected the topology which comes near to the shape of a square will performance the best, since it's optimal for the amount we need to exchange in each boundary, at the same time optimal for collective operations. In our case, since we have 28 cores, (4x7) is the most close to a square that we could get.

Performance of different grid-sizes We performed the test with different grid sizes. For this we used 1 nodes which has 28 tasks, 1 task/core. The results are presented in Figure 8 and 9.

From these two figures, we can see that the run time is increased proportionally to the number of required iterations so that the simulation converged. Besides, both runtime and number of required iterations seems to increase exponentially when we decrease grid sizes. When we decreased the grid size by 2, the runtime increased about 4-8 times depends on the problem sizes. 4 times would be a reasonable number since when we decrease the grid size by 2, the number of grid points in each dimension increases by 2, thus the number of total grid points increase by 4. However, this performance is also affected by cache-effects. When we have big grid sizes, we have small number of total grid points, this the data still can fit into caches. It's not the case when we have smaller grid sizes. Thus the time can increase more than 4 times.

For the final residuals, it decrease proportionally in respect to the grid sizes. This is due to the criteria we set as a stopping condition.

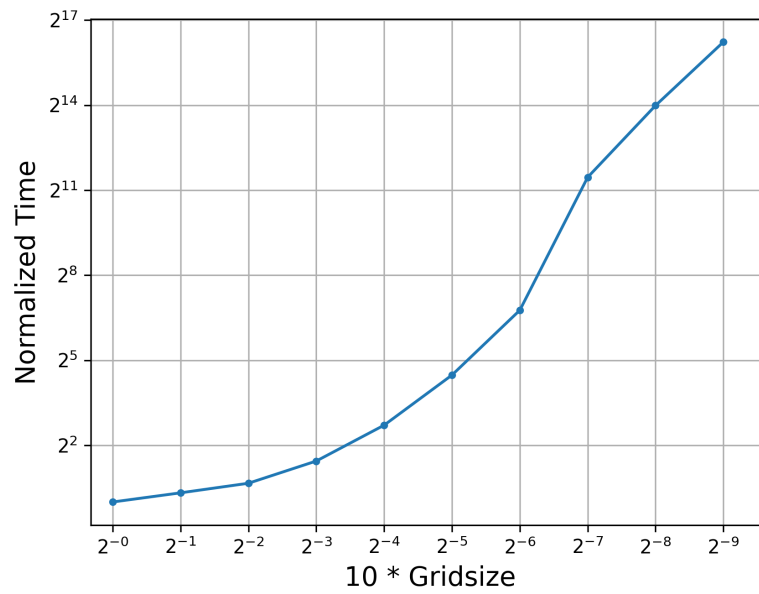


Abbildung 8: Normalized runtime of different grid sizes

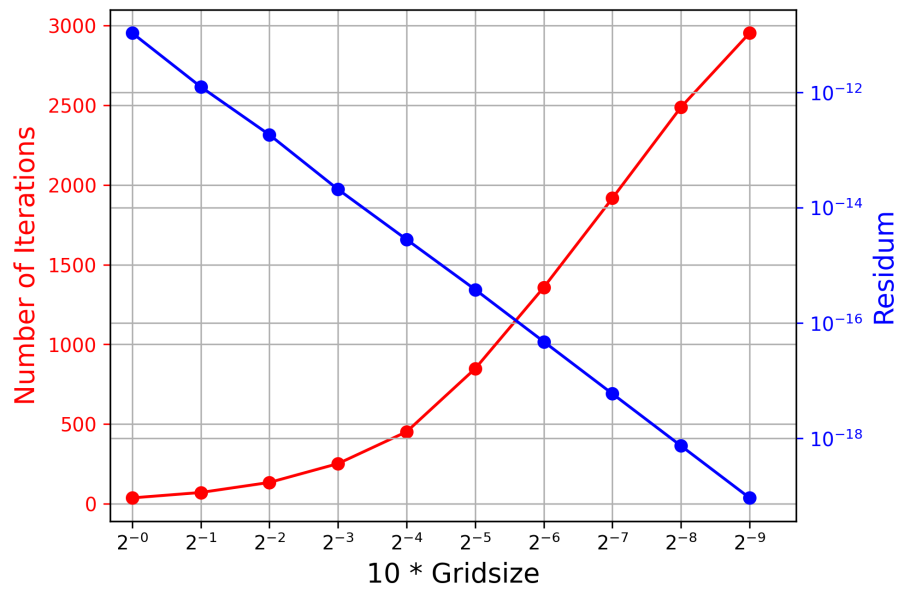


Abbildung 9: Number of iterations and final residual vs. different grid sizes

Strong-scale test in 1 node We use number of tasks ranging from 1 to 28, each task binds to 1 core. The result is in Figure 10

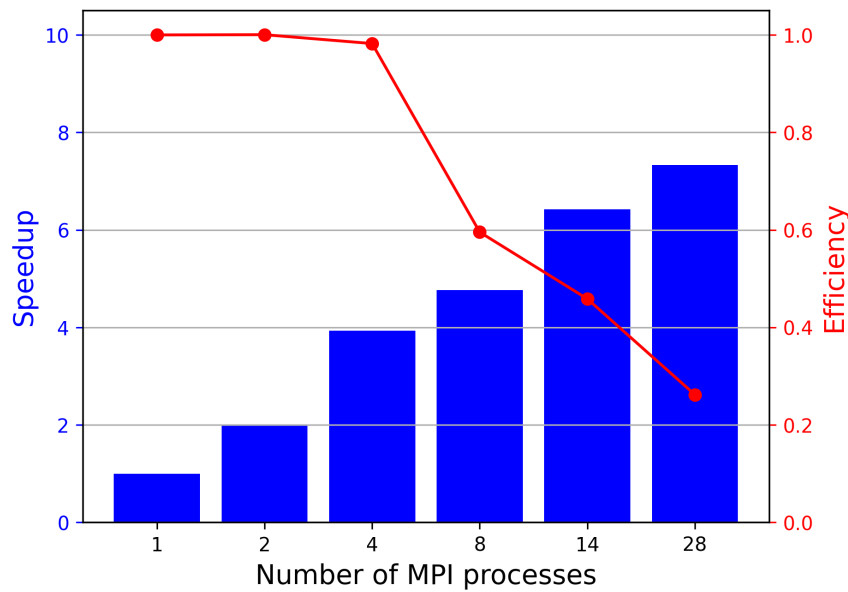


Abbildung 10: Strong-scale performance in 1 node, grid size = 0.00025

With a fix problem size (grid size = 0.00025), from 1 MPI task to 4 MPI tasks, we observed a linear-scaling which is good. However, when the number of MPI processes goes over 8, the speed up is not that good anymore. This trend is also revealed by the Efficiency line, while it kept steady at 1 from 1-4 processes, and dropped down when having more processes.

This trend is reasonable, since when we use 28 MPI processes in 1 node, the cost of communications overshadows the computation that can be done in parallel. Besides, such communications can be limited by the bandwidth and latency of the communication networks (even in side of 1 node).

Strong-scale test in 4 nodes We kept the same problem size as the above test (grid size = 0.00025) and increases number of using cores beyond a node. The results are shown in the Figure 11

Interestingly, when going beyond 1 node, we observed a super-linear scaling, especially in the test with 224 MPI processes on 4 nodes, the speedup is 242 times, and the efficiency is 1.08. This might be due to the reason that we a fixed problem size, when we have more cores, then the sub-problem can be fit into the processes caches, thus the efficiency is even better then the single core test.

To verify this, we increased the problem size further by decreasing the grid size to 0.0001. We tested with 4 nodes and the normalized speedup is presented below in Figure 12.

Here the super-linear scaling effect is disappear. The program scales nicely in respect to number of nodes and the efficiency remains almost constant around 1. This means our prediction for super-linear scale for the previous test is correct.

c. Future Optimisation

Due to limitation in time, we could not manage to optimize the code further. Possible optimizations are listed below:

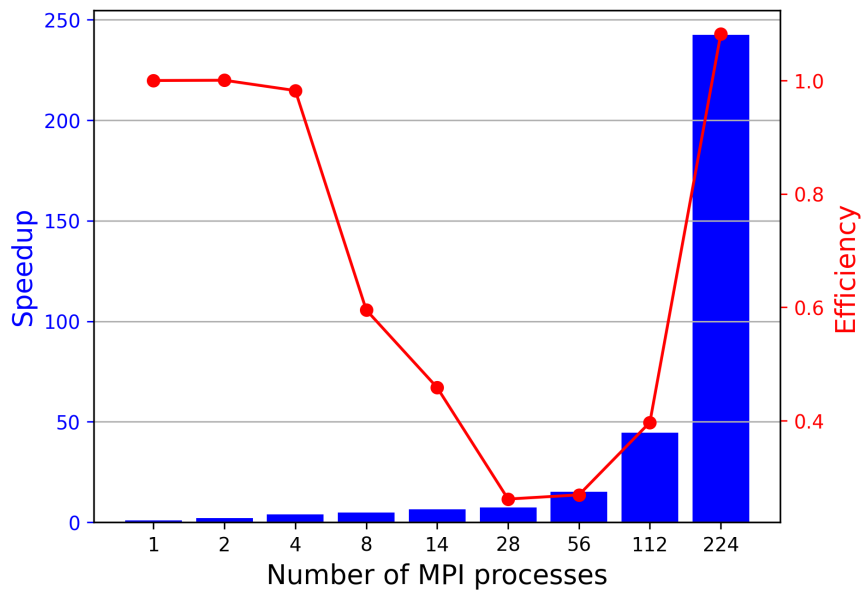


Abbildung 11: Strong-scale performance in 4 node, grid size = 0.00025

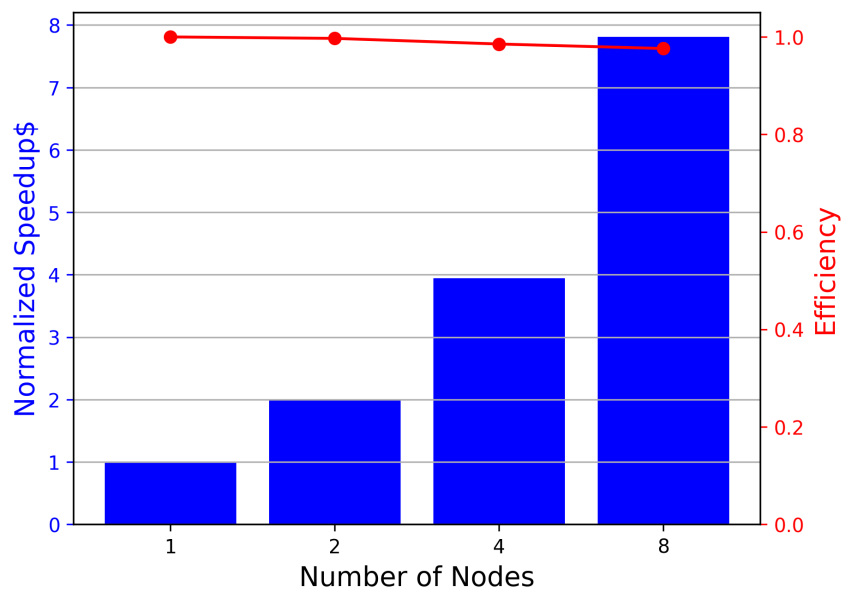


Abbildung 12: Strong-scale performance in 4 node, grid size = 0.0001

- Parallelizing output with MPI-IO (*MPI_File*)
- Overlapping communication and computation.

Literatur

- [1] Gustafson's law. https://en.wikipedia.org/wiki/Gustafson%27s_law.
- [2] Intel® opa performance. <https://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-performance-overview.html>.
- [3] Mark S. Birrittella, Mark Debbage, Ram Huggahalli, James Kunz, Tom Lovett, Todd Rimmer, Keith D. Underwood, and Robert C. Zak. Enabling scalable high-performance systems with the intel omni-path architecture. *IEEE Micro*, 36(4):38–47, 2016.
- [4] Philip Taffet, Sanil Rao, Edgar León, and Ian Karlin. Testing the limits of tapered fat tree networks. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 47–52, 2019.