ESPM2: 2022 ACM/IEEE 7th International Workshop on Extreme Scale Programming Models and Middleware

Impacts and potential of using node resources for active middleware

Adrian Jackson, Senior Research Fellow, Edinburgh Parallel Computing Center (EPCC), UK

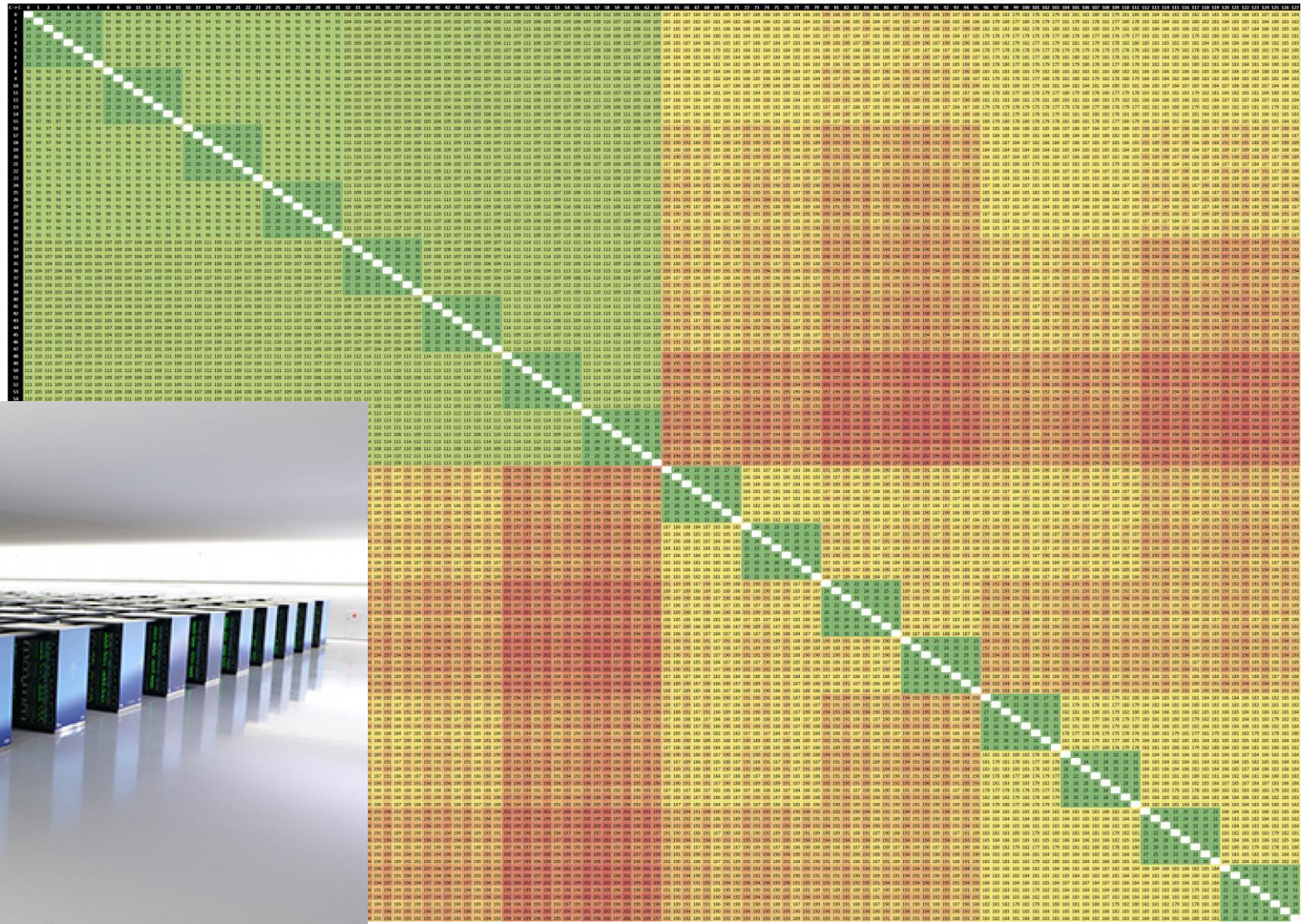
Adrian Jackson is a senior research fellow at EPCC, The University of Edinburgh. He has a long history of research in HPC, from optimising applications for specific hardware through to designing and developing middleware for new systems. He has spent recent years researching non-volatile memory and advanced data storage systems, and also heavily involved in hardware optimisation projects, such as the UK Excalibur FPGA testbed in the UK. Research experience across a range of different areas in high performance computing provides the basis for current work on compute node performance capacity and automatic communication libraries designed to exploit network capacity for improved application performance.

Impacts and potential of using node resources for active middleware

Adrian Jackson

EPCC, The University of Edinburgh

System trends



Exascale challenge

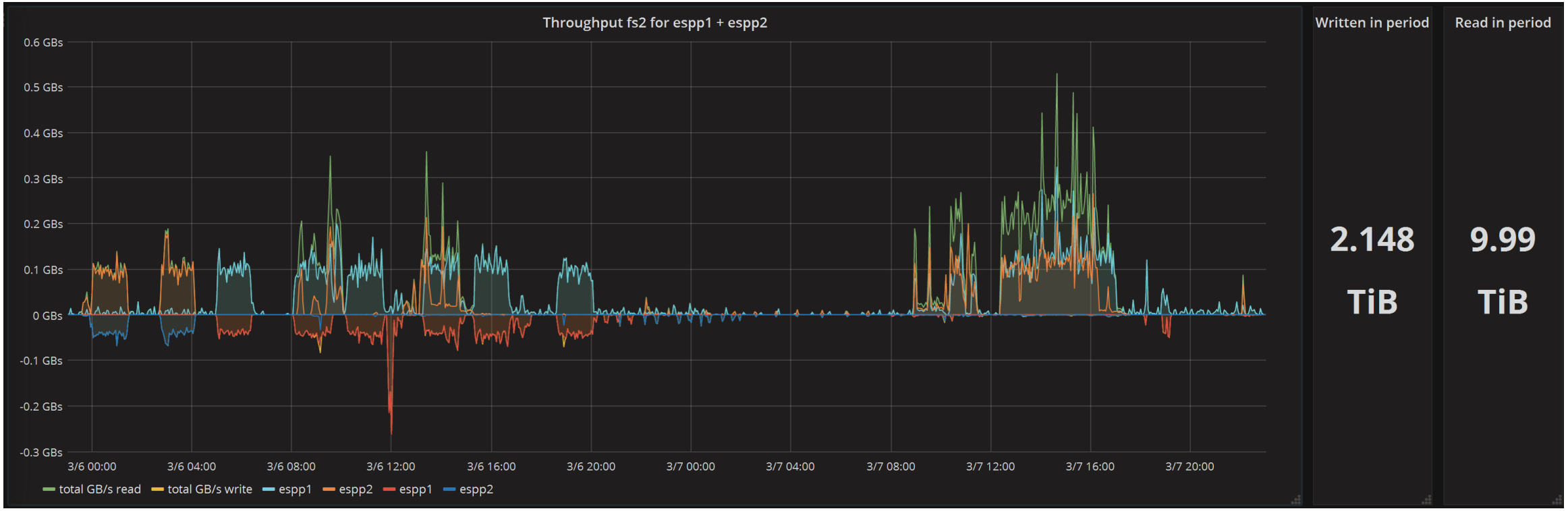
- Optimise overall parallel performance
  - Assuming computing free, memory expensive, communication very expensive
- Optimising overall resource usage
  - Work per energy expended
- Optimising users overall workflows

Exascale challenge

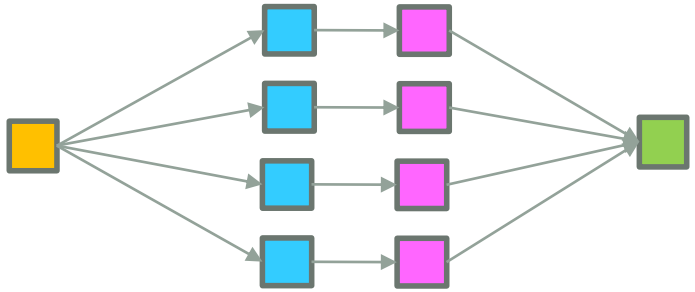
- Minimise movement
  - Keep data where it is
  - Key for maximising energy efficiency

- Minimise waiting on movement
  - Get data to where it needs to be before it is required
  - Key for maximising user efficiency

- Minimise the maximum hardware required


Throughput fs2 for espp1 + espp2
0.6 GBs
0.5 GBs
0.4 GBs
0.3 GBs
0.2 GBs
0.1 GBs
0 GBs
-0.1 GBs
-0.2 GBs
-0.3 GBs
3/6 00:00
3/6 04:00
3/6 08:00
3/6 12:00
3/6 16:00
3/6 20:00
3/7 00:00
3/7 04:00
3/7 08:00
3/7 12:00
3/7 16:00
3/7 20:00
total GB/s read
total GB/s write
espp1
espp2
espp1
espp2
Written in period
2.148
TiB
Read in period
9.99
TiB

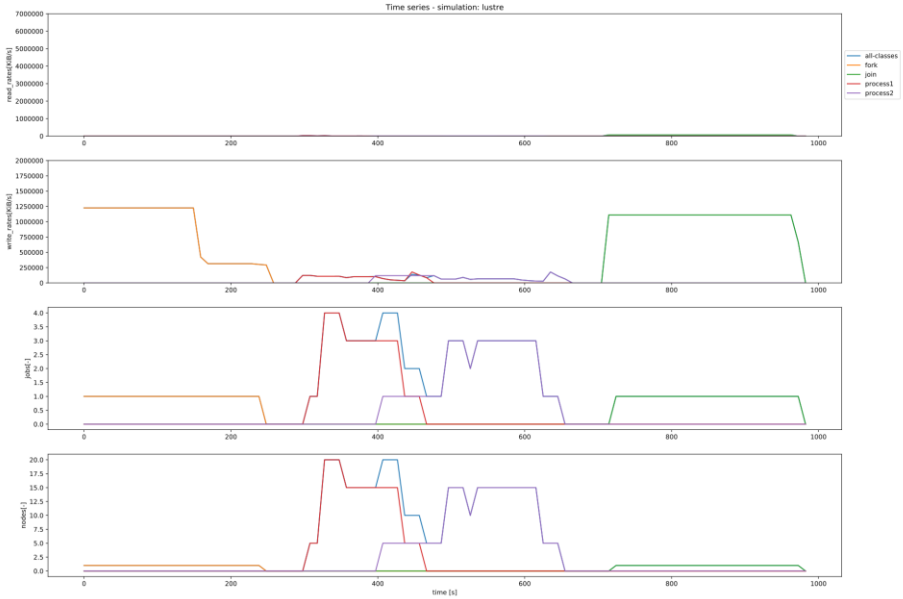
In-place data storage

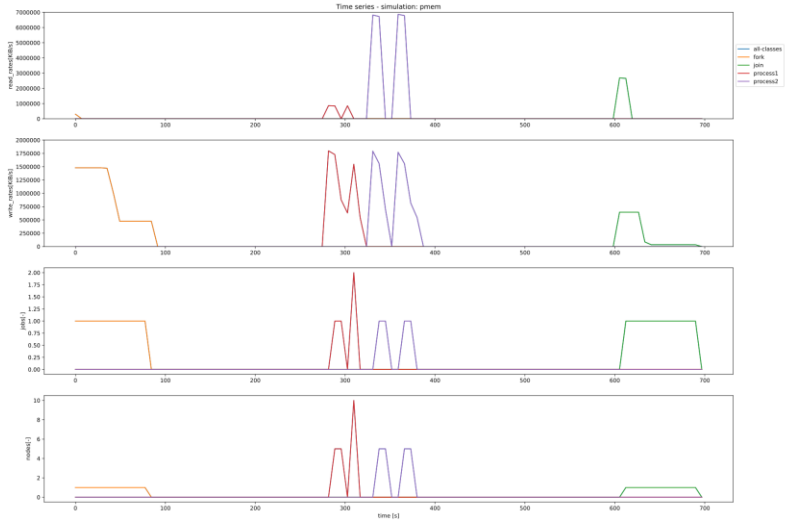


1 node
4 processes
4 files

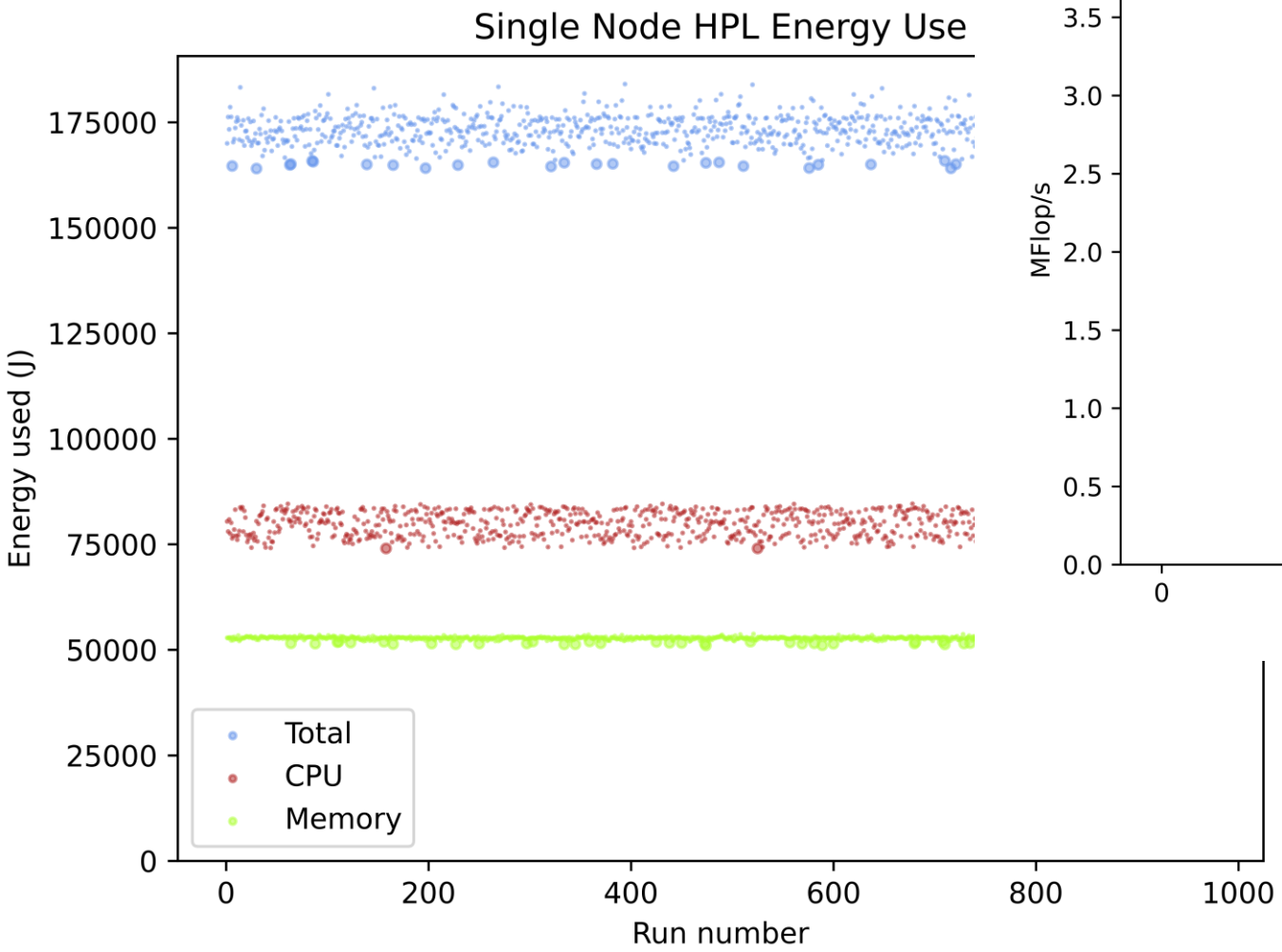
20 nodes
80 processes
80 files

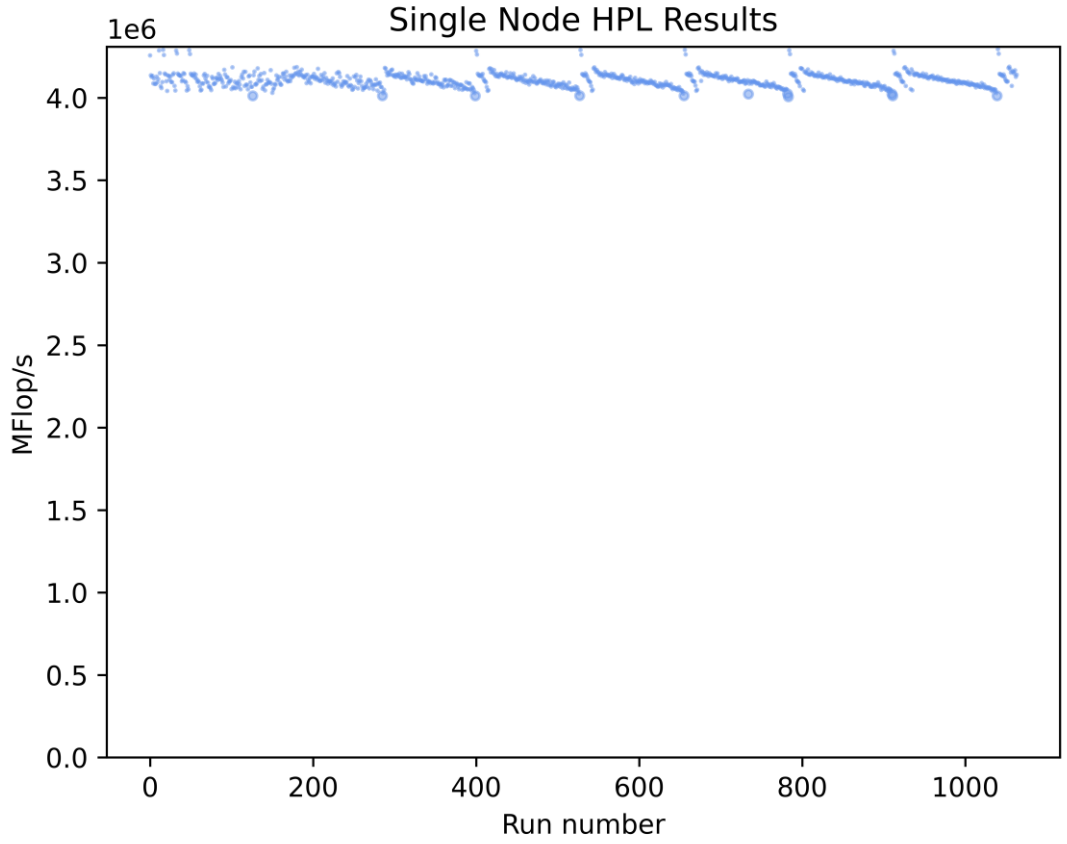
1 node
4 processes
80 files



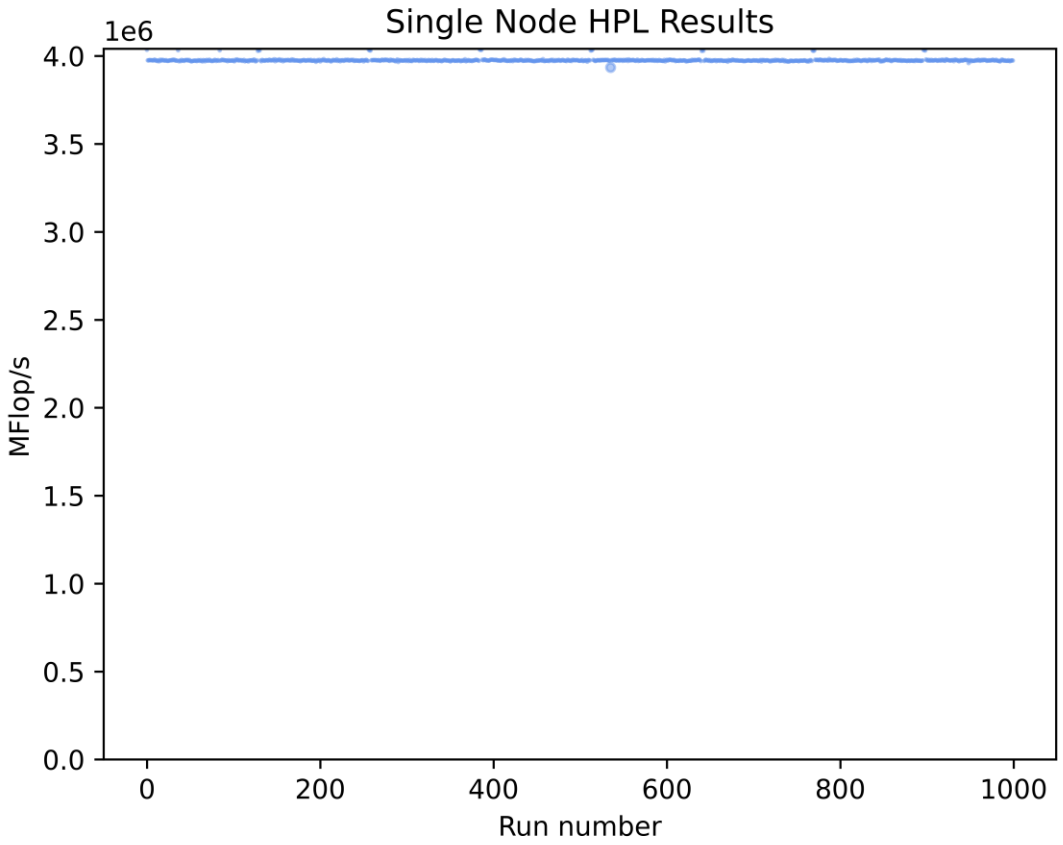


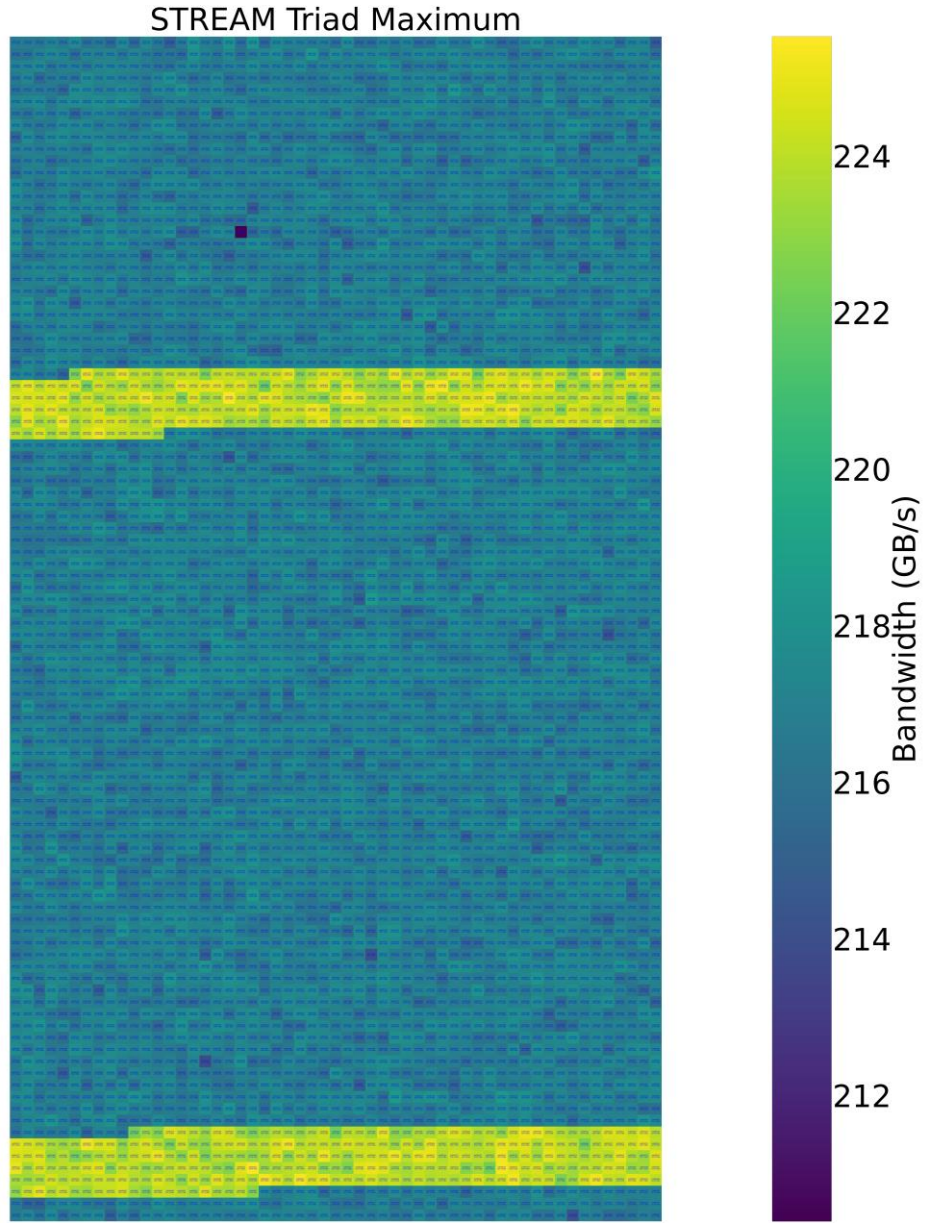
System trends





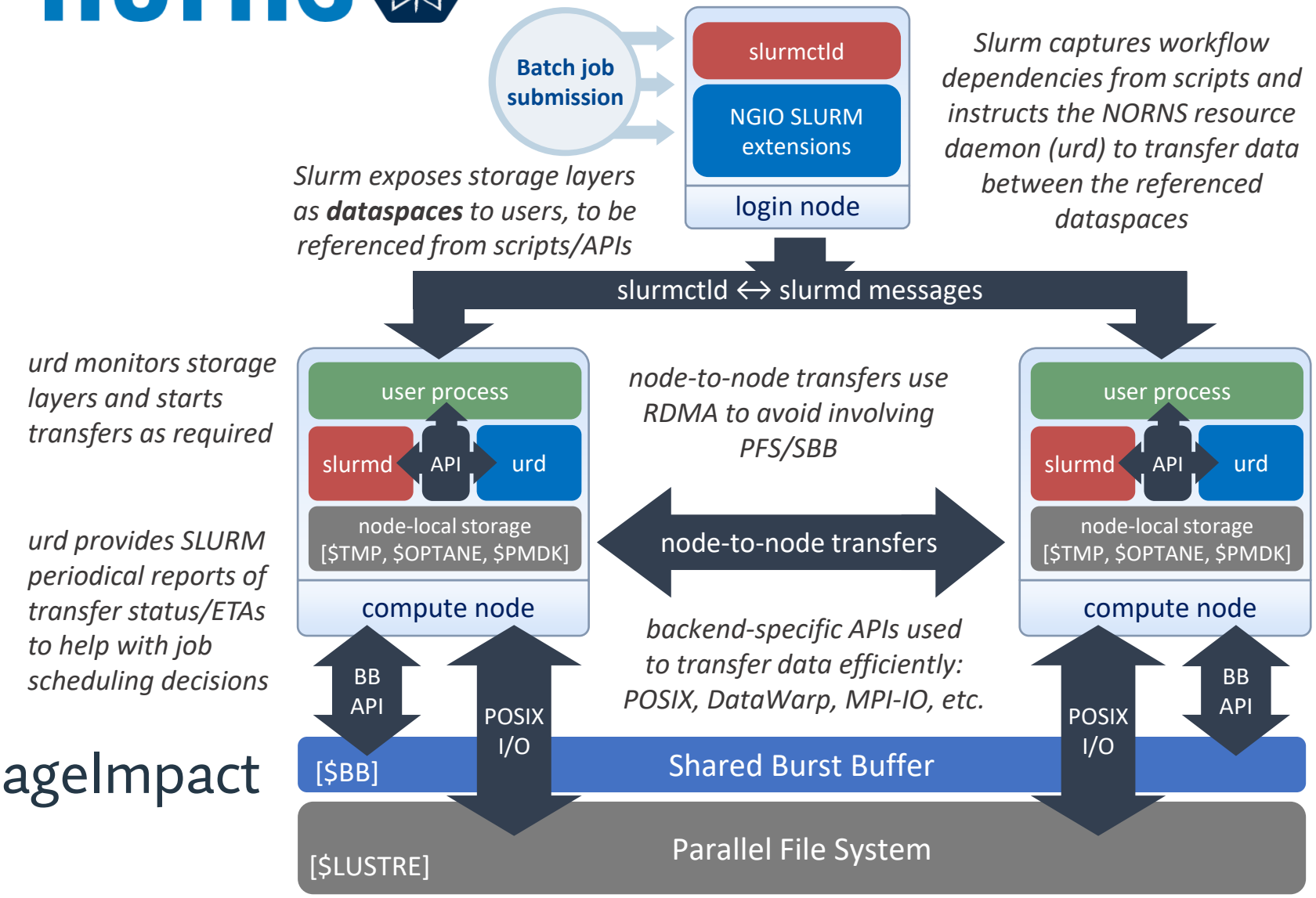
System trends





Active systemware

- How much on node resources can we steal
- What will applications tolerate
- NodeResourceUsageImpact evaluation

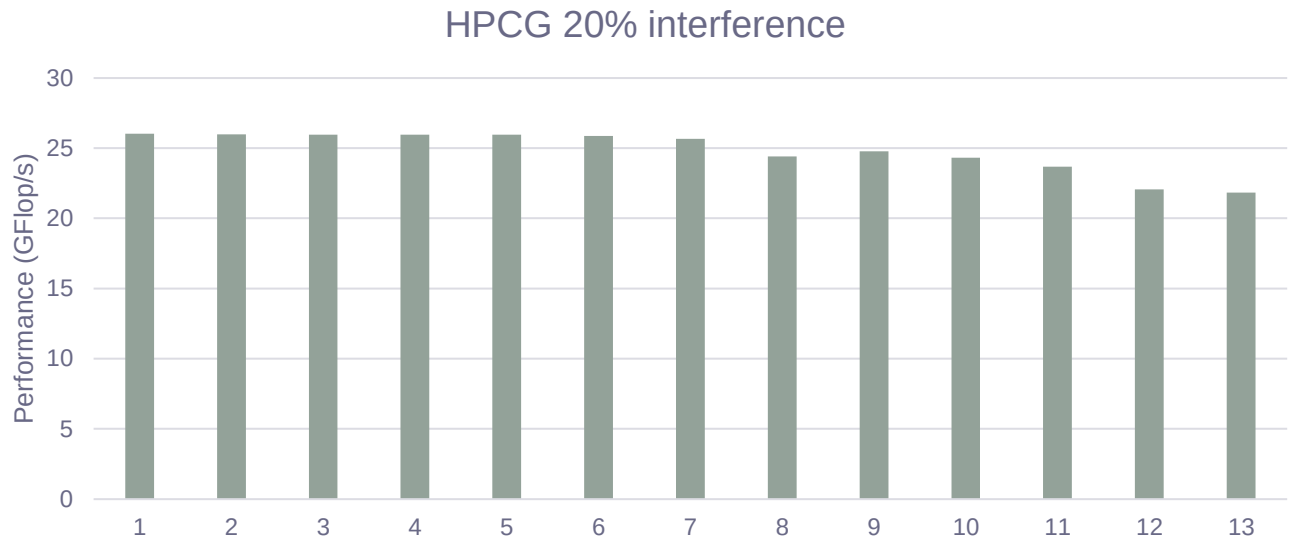


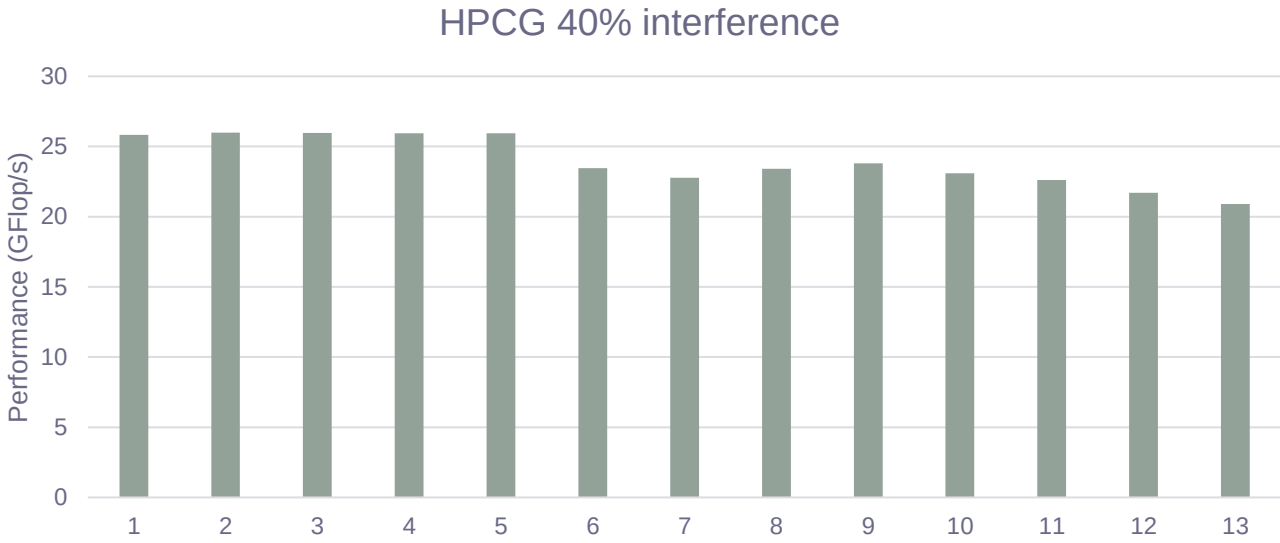
NRUI

- Tunable framework to inject noise into nodes applications are running on
- Exploits the MPI profiling interface
  - Spawns separate worker
  - Run configurable task
    - Floating point
    - Integer
    - Memory traffic
    - I/O
    - Network traffic
- Transparent to the application

Application Performance

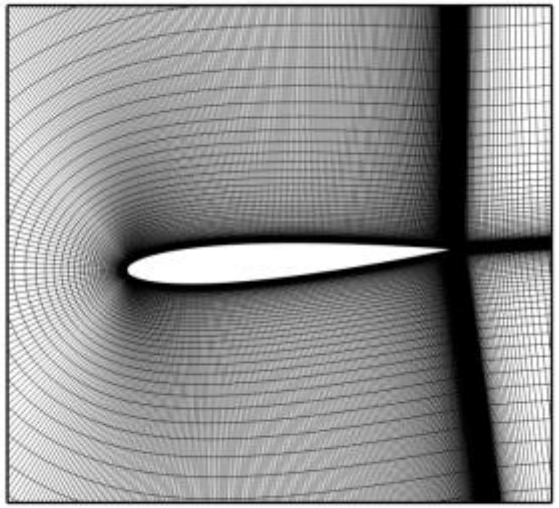
- HPCG
  - Typically memory and cpu bound
  - Significantly more impacted by memory than floating point operations in practice

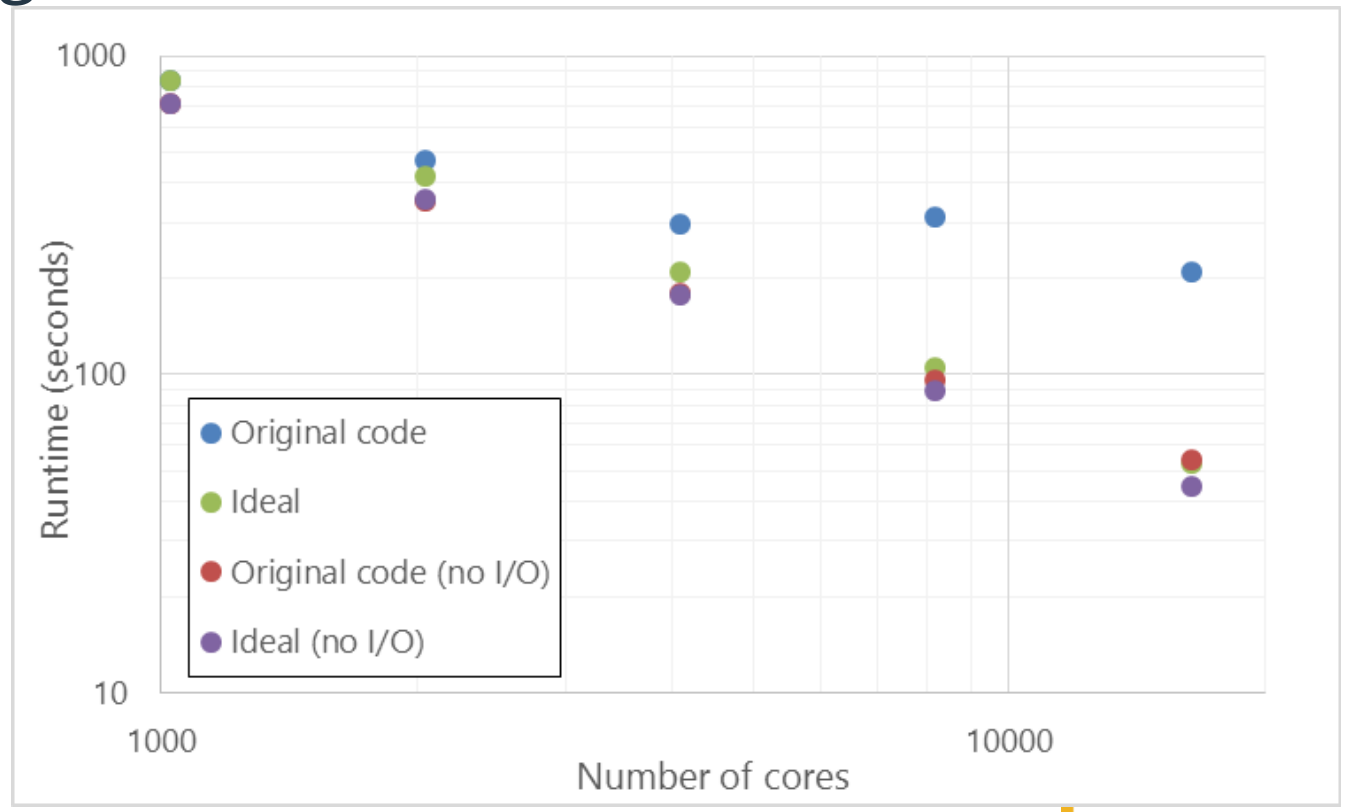




Application performance

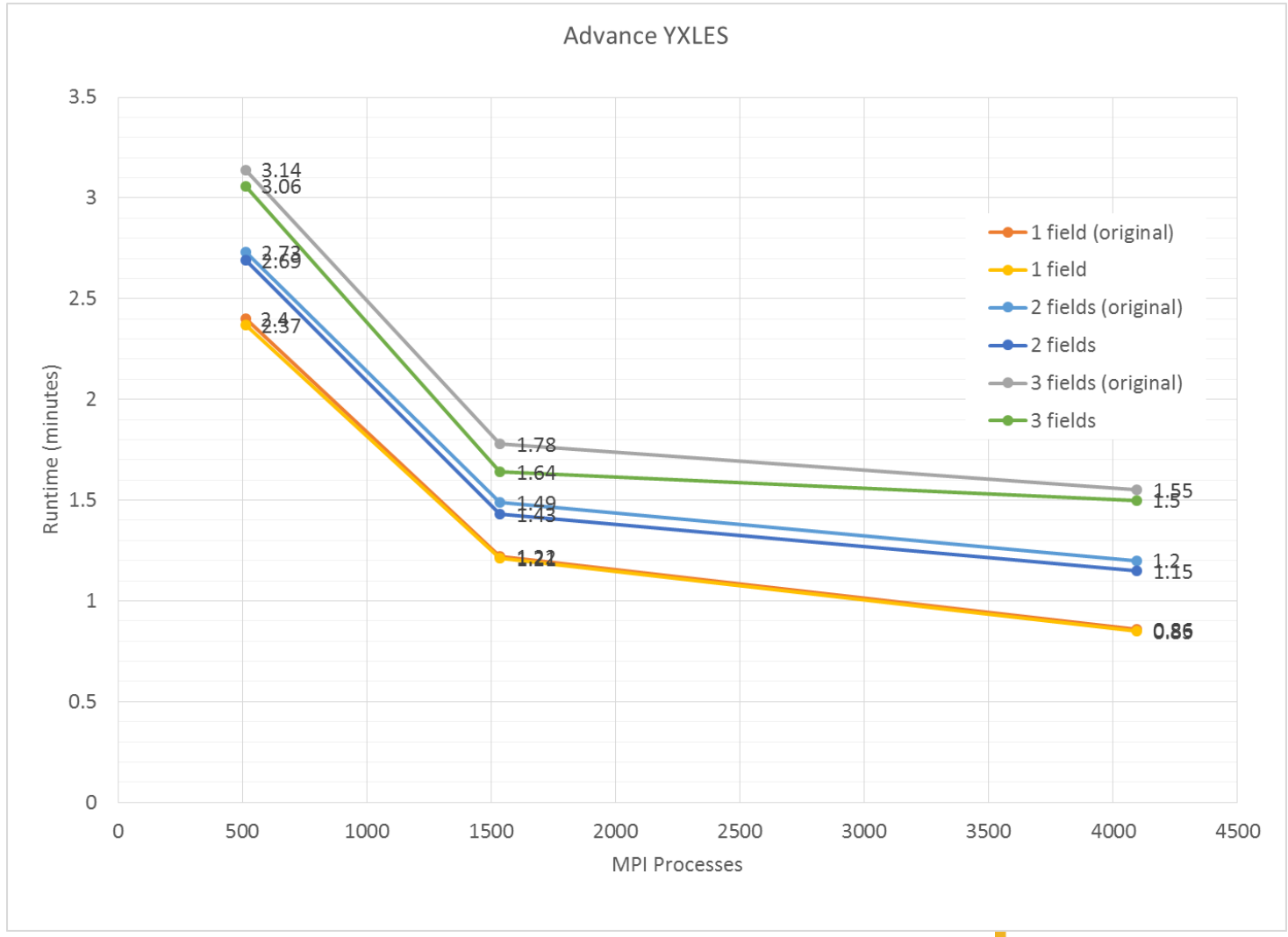
- CFD applications
- Large scale, local nearest neighbour comms





Application Performance

- GS2
  - Plasma physics application
  - 2x performance slowdown
    - Competing for nodes
  - Minimal performance slowdown
    - Running on hyperthreads

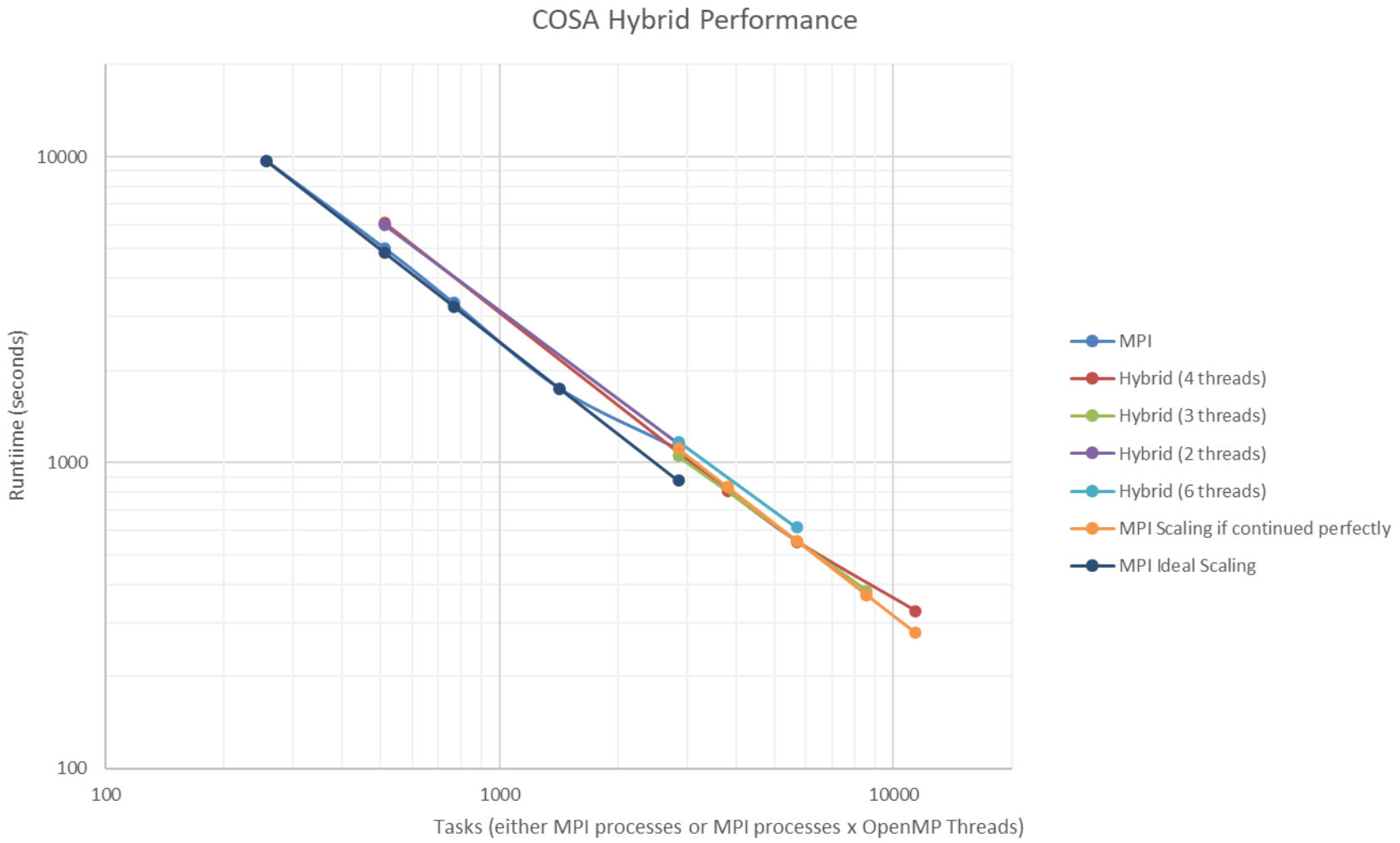


NRUI for performance fuzzing

- Scope to evaluate performance impact for systemware/middleware
  - Particularly at scale

- Investigate code sensitivity to hardware features
  - Fuzz particular “hardware resources” to evaluate the impact
  - Estimate future hardware performance

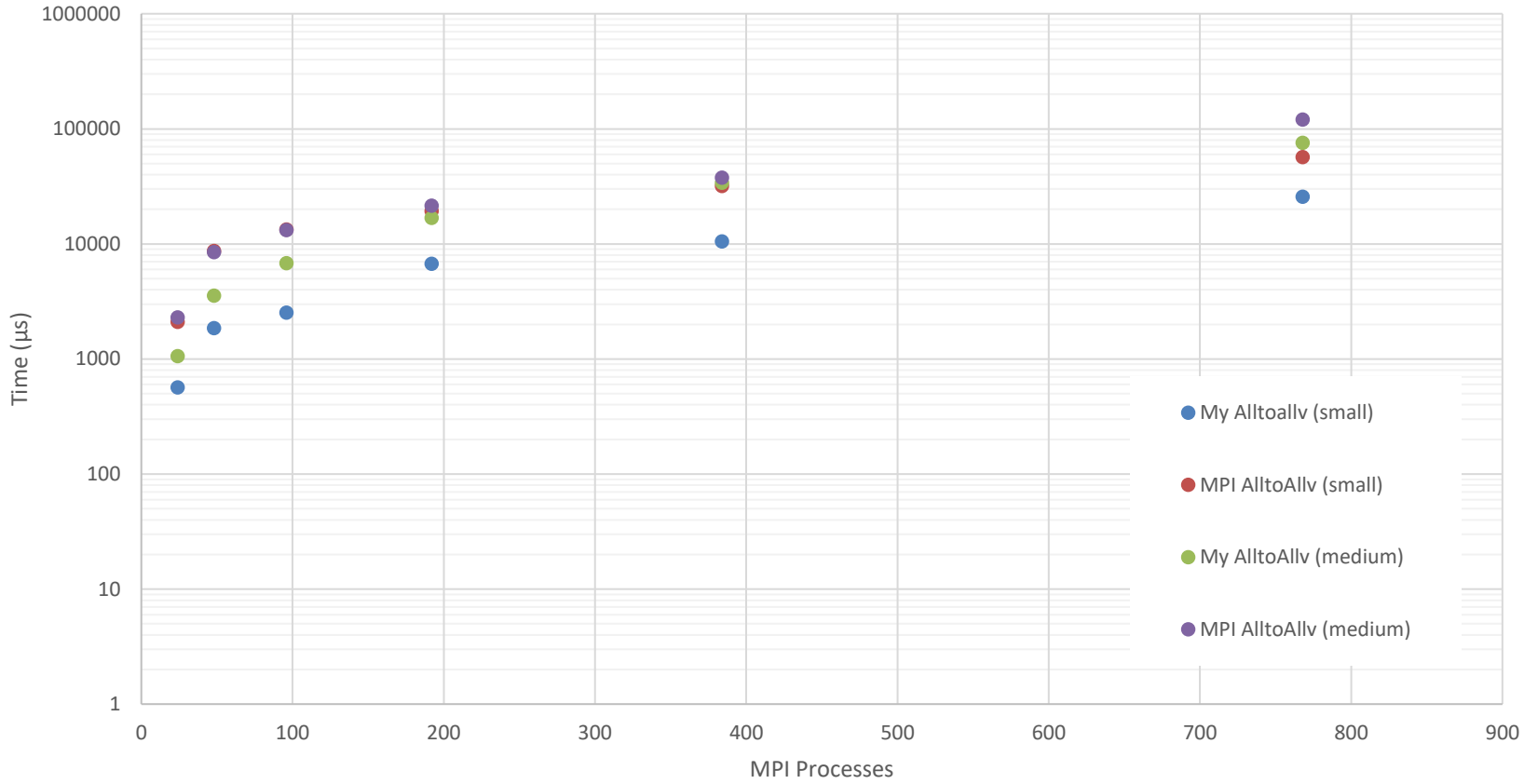
Reducing MPI process counts



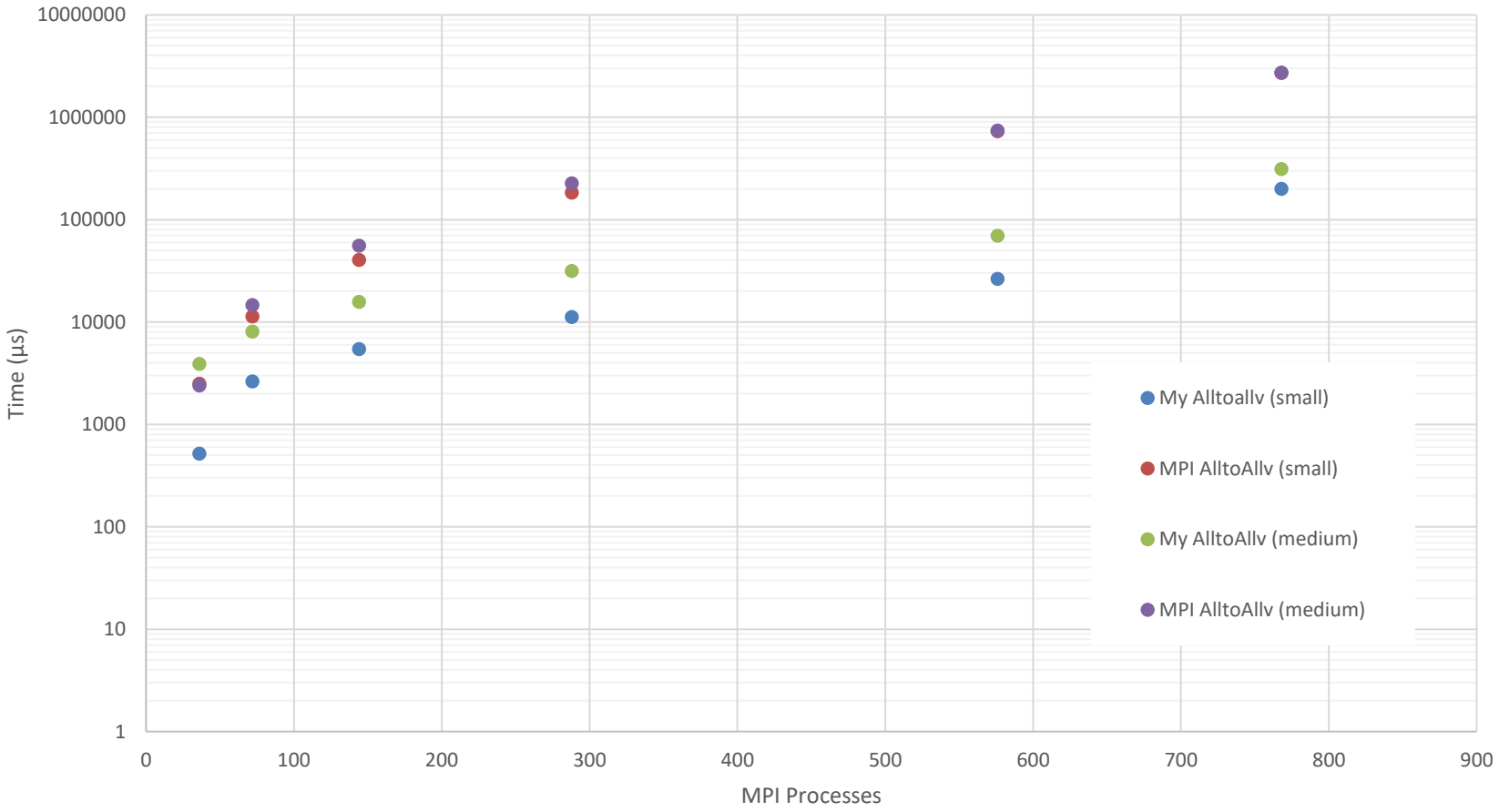
MPI + MPI

- Reduce MPI process count on node
- MPI runtime per node or NUMA region/network end point
- On-node collective optimisation
  - Shared-memory segment + planned collectives

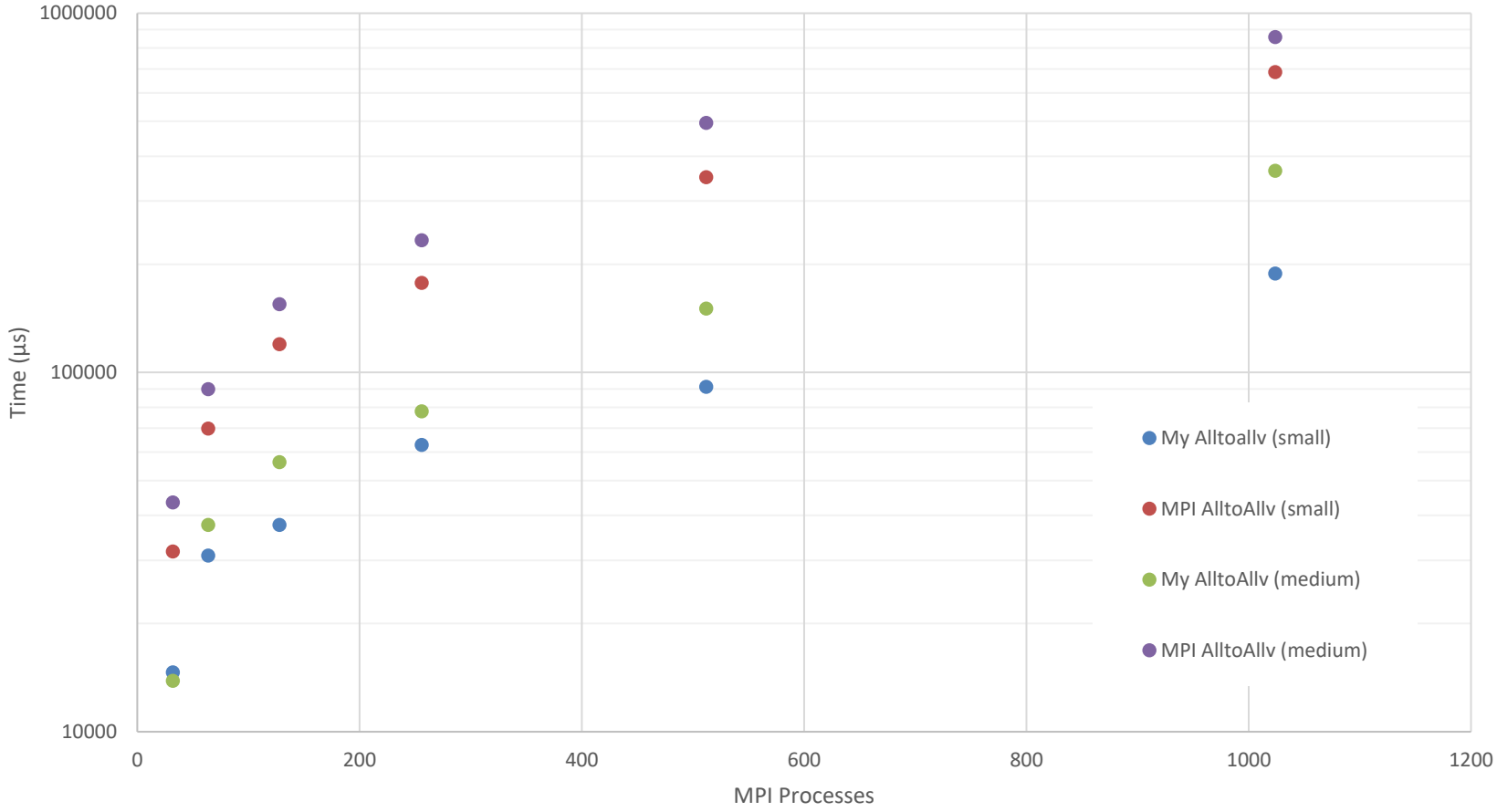
Planned Alltoallv performance



Planned Alltoallv performance

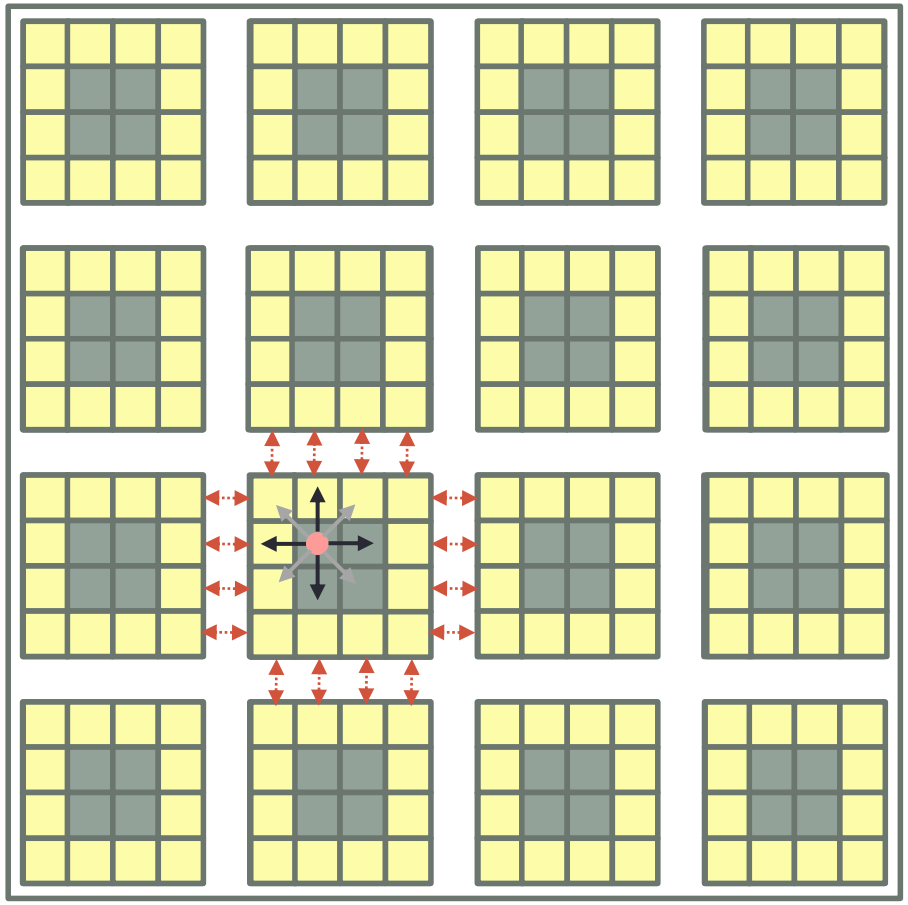


Planned Alltoallv Performance



Common simulation methodologies

- Standard computational and communication pattern
- Calculate over array or matrix
- Communicate halo data at the end of calculation



Overlap for Optimisation

- Standard pattern forces computation and communication separation
  - Network unused for large portion of simulation
- Interleaving possible with code modification
  - Code “mutilation” required
  - Architecture specific optimisations

Communication Code

```
for (iter=1;iter<=maxiter; iter++){
     MPI_Irecv(&old[0][1], NP, MPI_FLOAT, prev, 1,
               MPI_COMM_WORLD, &requests[0]);
     MPI_Irecv(&old[MP+1][1], NP, MPI_FLOAT, next, 2,
               MPI_COMM_WORLD, &requests[1]);
     MPI_Isend(&old[MP][1], NP, MPI_FLOAT, next, 1,
               MPI_COMM_WORLD, &requests[2]);
     MPI_Isend(&old[1][1], NP, MPI_FLOAT, prev, 2,
               MPI_COMM_WORLD, &requests[3]);
     MPI_Waitall(4, requests, statuses);
     for (i=1;i<MP+1;i++){
         for (j=1;j<NP+1;j++){
             new[i][j]=0.25*(old[i-1][j]+old[i+1][j]+
                 old[i][j-1]+old[i][j+1] - edge[i][j]);
     }
```

Communication code

```
for (iter=1;iter<=maxiter; iter++){
       requestnum = 0;
       for (j=0;j<NP;j++){
           MPI_Irecv(&tempprev[j], 1, MPI_FLOAT, prev, 1,
                     MPI_COMM_WORLD, &requests[requestnum]);
           requestnum++;
           MPI_Irecv(&tempnext[j], 1, MPI_FLOAT, next, 2,
                     MPI_COMM_WORLD, &requests[requestnum]);
           requestnum++;
       }
       for (i=1;i<MP+1;i++){
           for (j=1;j<NP+1;j++){
               new[i][j]=0.25*(old[i-1][j]+old[i+1][j]+old[i][j-1]+old[i][j+1]
                               - edge[i][j]);
               if(i == MP){
                   MPI_Isend(&new[i][j], 1, MPI_FLOAT, next, 1,
                             MPI_COMM_WORLD, &requests[requestnum]);
                   requestnum++;
               }else if(i == 1){
                   MPI_Isend(&new[i][j], 1, MPI_FLOAT, prev, 2,
                             MPI_COMM_WORLD, &requests[requestnum]);
                   requestnum++;
               }
           }
       }
       for (i=1;i<MP+1;i++){
           for (j=1;j<NP+1;j++){
               old[i][j]=new[i][j];
           }
       }
       MPI_Waitall(requestnum, requests, statuses);
       if(prev != MPI_PROC_NULL){
           for (j=1;j<NP+1;j++){
               old[0][j] = tempprev[j-1];
               old[MP+1][j] = tempnext[j-1];
           }
       }
       if(next != MPI_PROC_NULL){
           for (j=1;j<NP+1;j++){
               old[MP+1][j] = tempnext[j-1];
           }
       }
   }
```

MDMP Fundamentals

```
#pragma send(…)

#pragma recv(…)
```

```
for (iter=1;iter<=maxiter; iter++){
#pragma recv(old[0][0], NP, prev)
#pragma recv(old[MP+1][1], NP, next)
#pragma send(old[MP][1], NP, next)
#pragma send(old[1][1], NP, prev)
  for (i=1;i<MP+1;i++){
            for (j=1;j<NP+1;j++){
                new[i][j]=0.25*(old[i-1][j]+old[i+1][j]+
               old[i][j-1]+old[i][j+1] - edge[i][j]);
            }
        }
        for (i=1;i<MP+1;i++){
            for (j=1;j<NP+1;j++){
                old[i][j]=new[i][j];
            }
        }
  }
```

MDMP Fundamentals

- Communicating Regions

```
#pragma commregion
#pragma commregionfinished
```

- Defines the scope to consider for the data evaluation and communication optimisation
  - Count read and writes of data to be communicated
  - Communicate once; last write occurs (sends), last read/write occurs (receives)

MDMP Fundamentals

```
#pragma commregion
for (iter=1;iter<=maxiter; iter++){
#pragma recv(old[0][0], NP, prev)
#pragma recv(old[MP+1][1], NP, next)
#pragma send(old[MP][1], NP, next)
#pragma send(old[1][1], NP, prev)
  for (i=1;i<MP+1;i++){
            for (j=1;j<NP+1;j++){
                new[i][j]=0.25*(old[i-1][j]+old[i+1][j]+
               old[i][j-1]+old[i][j+1] - edge[i][j]);
            }
        }
        for (i=1;i<MP+1;i++){
            for (j=1;j<NP+1;j++){
                old[i][j]=new[i][j];
            }
        }
  }
#pragma commregionfinished
```

MDMP Fundamentals

- Combined these enable runtime scheduling of communications to optimise network usage
  - Auto-tuning possible
  - Potential for errors
  - Adds computational overheads
  - May damage memory performance
  - Relies on users placing communicating regions correctly

MDMP Performance

- STREAMS benchmarks
  - No communicating region

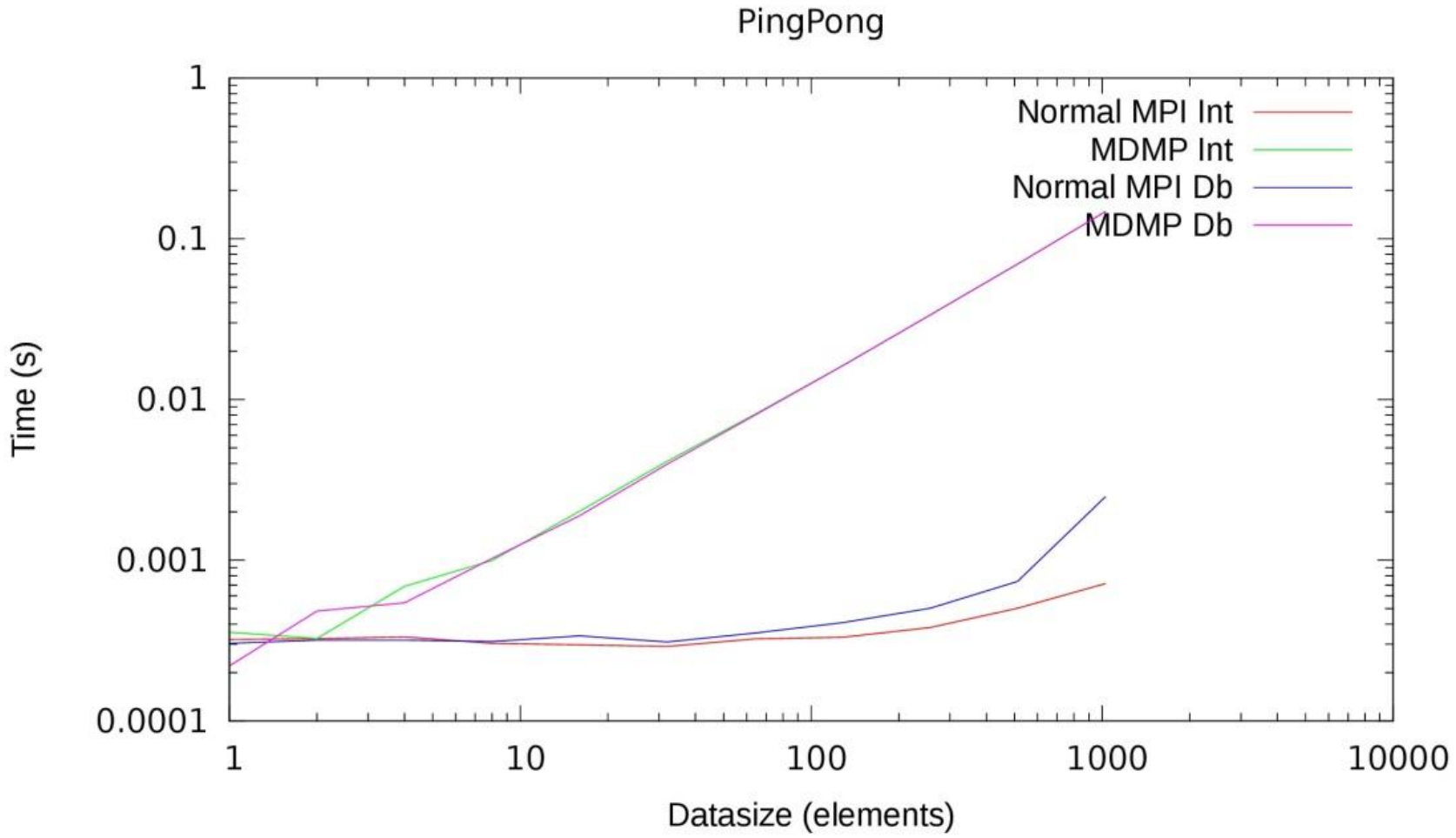
Operation	Original Time	MDMP Time
Int Assign	0.000007	0.000020
Db Assign	0.000009	0.000024
Db Copy	0.000009	0.000027
Db Scale	0.000017	0.000025
Db Add	0.000035	0.000033
Db Triad	0.000035	0.000037

- STREAMS benchmarks
  - Communicating region

Operation	Original Time	MDMP Time
Int Assign	0.000007	0.000048
Db Assign	0.000009	0.000057
Db Copy	0.000009	0.000066
Db Scale	0.000017	0.000063
Db Add	0.000035	0.000071
Db Triad	0.000035	0.000076

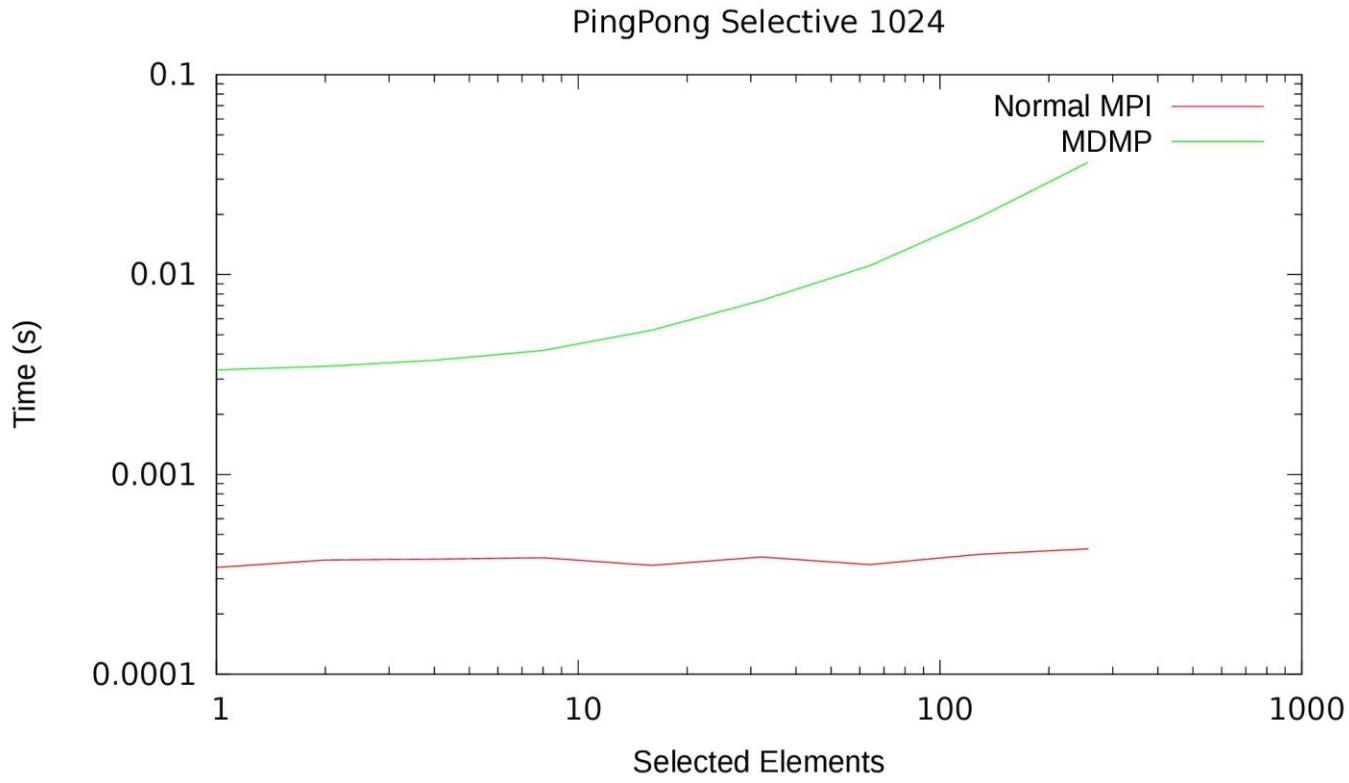
MDMP Performance

- PingPong



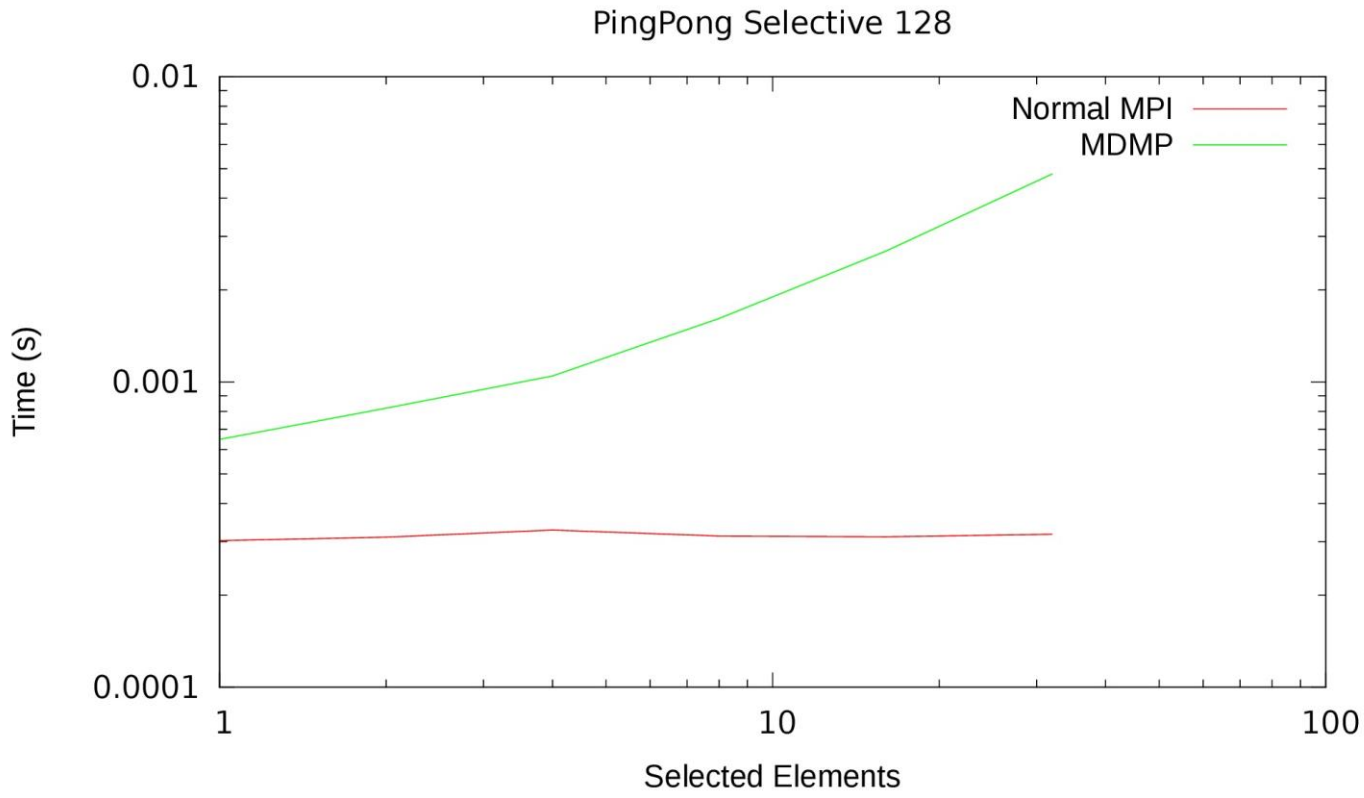
MDMP Performance

- Selective PingPong benchmark



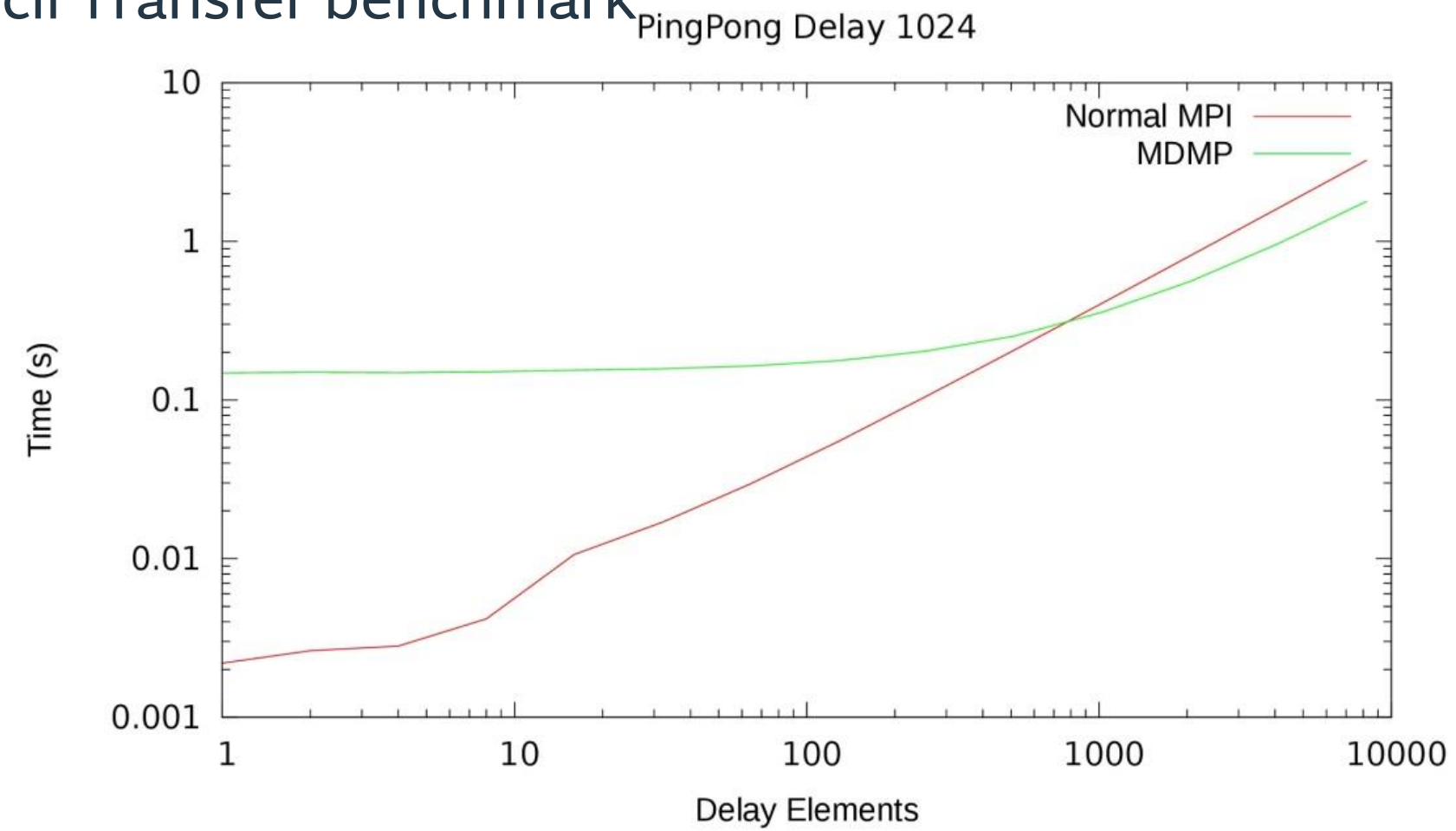
MDMP Performance

- Selective Stencil Transfer benchmark



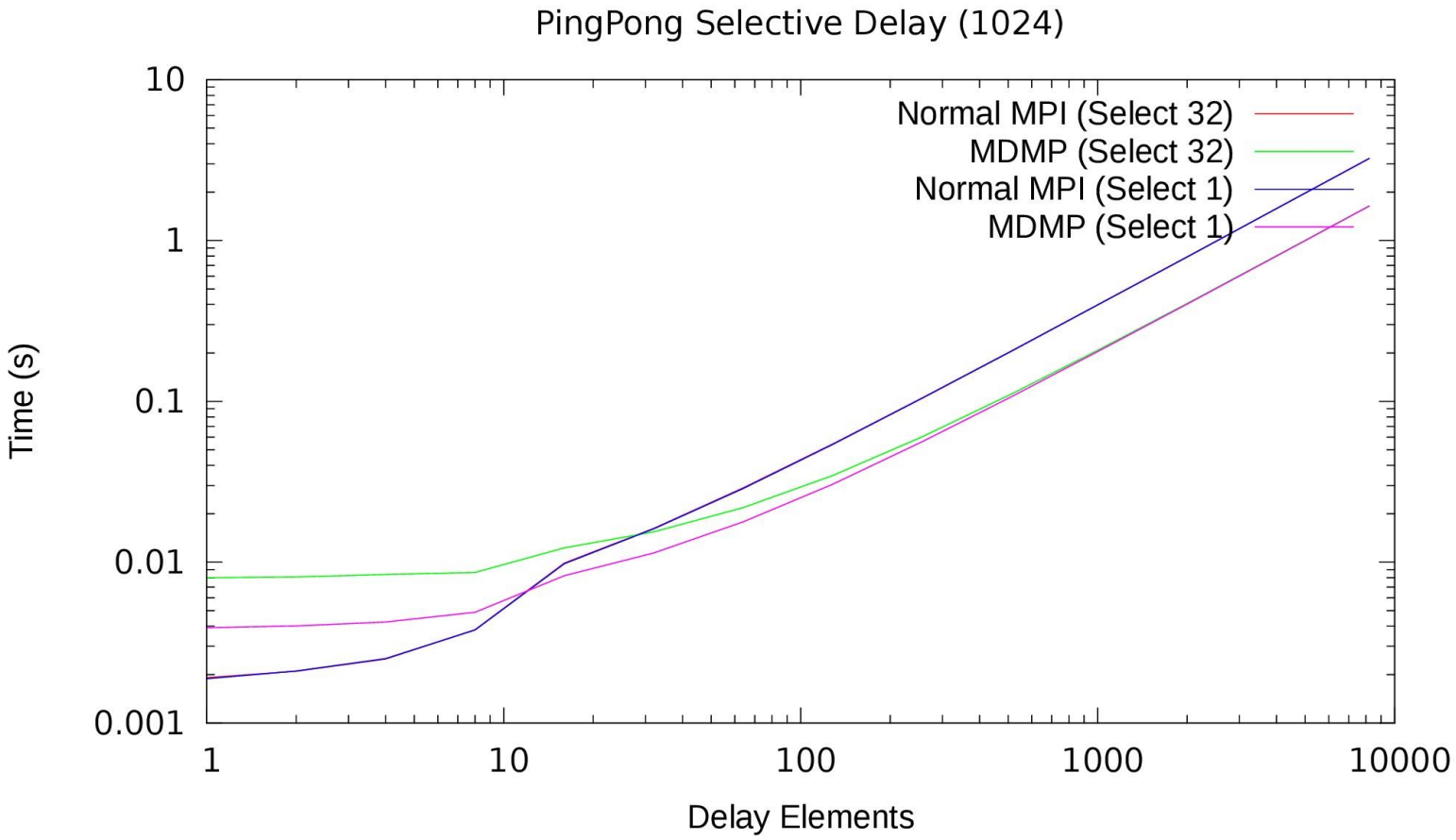
MDMP Performance

- Delay Stencil Transfer benchmark



MDMP Performance

- Delay selective Stencil Transfer benchmark



MDMP

- Simplified MPI programming
- Aims to optimise communication
  - Automating evaluation and scheduling of communications
- Won't work for everything
- Won't necessarily give best performance
- Requires more memory
- Can default to mapping to MPI only
- Can just use MPI
  - Selective optimisations after development

Summary

- A range of smaller scale options for optimisation/improving efficiency for applications and systems
  - Potential for “incremental” improvements
- Looking at what can be considered “free” for optimisation
  - Compute can be abused to improve overall performance?
- Things that provide optimised “experience” might trump absolute performance
  - Directives
  - Middleware
  - Etc....