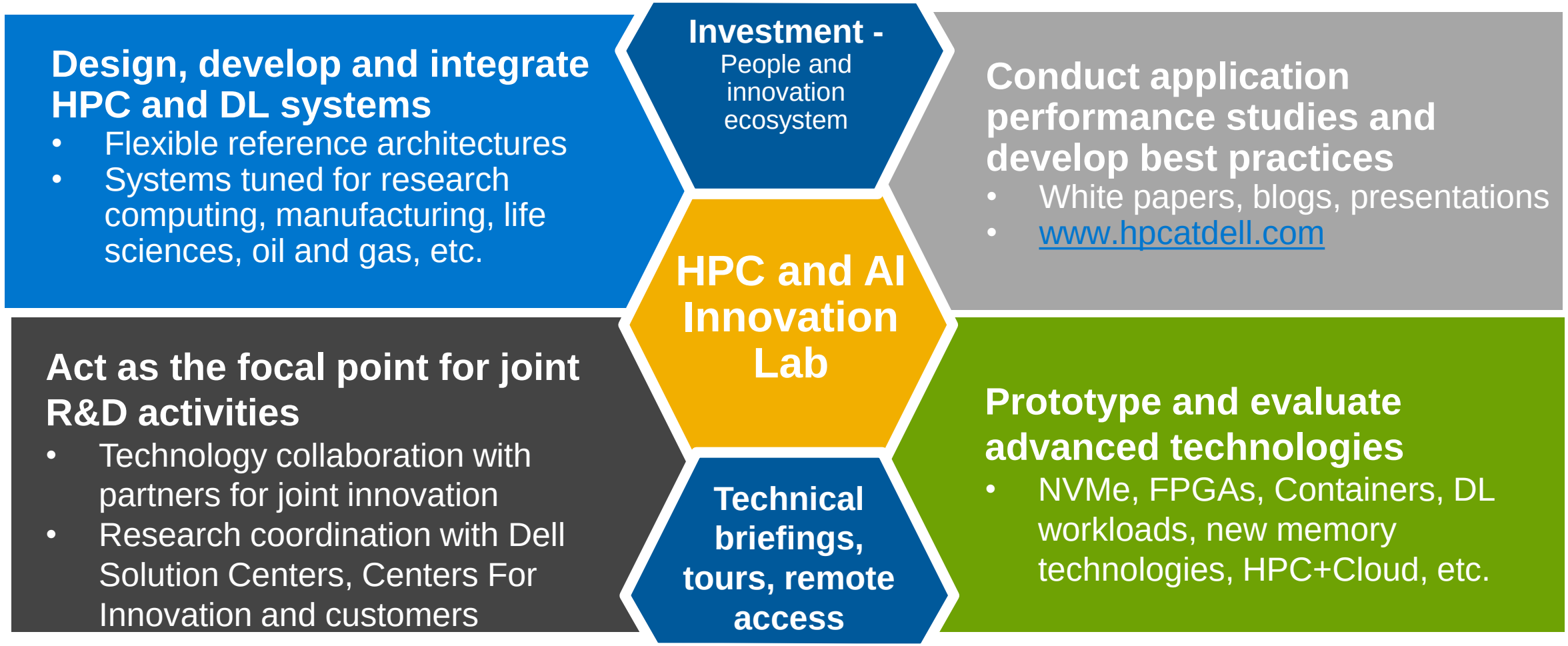
Experiences with MVAPICH on large-scale workloads

HPC Platform Efficiency and Power Savings

Martin Hilgeman
Distinguished Member of Technical Staff
HPC Performance Lead
Martin.Hilgeman@dell.com

DELL Technologies

Dell Technologies HPC and AI Team Charter



World-class infrastructure in the Innovation Lab

13K ft.^{2} lab, 1,300+ servers, ~10PB storage dedicated to HPC in collaboration with the community

Zenith

- Dell PowerEdge C6620 based on Intel Scalable processors
- Liquid cooled and air cooled

Rattler

- Dell PowerEdge XE8545 with NVIDIA GPUs

Minerva

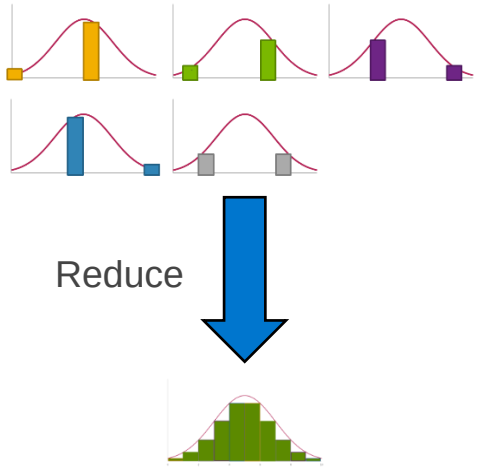
- Dell PowerEdge R6625 based on AMD EPYC processors
- Liquid cooled and air cooled

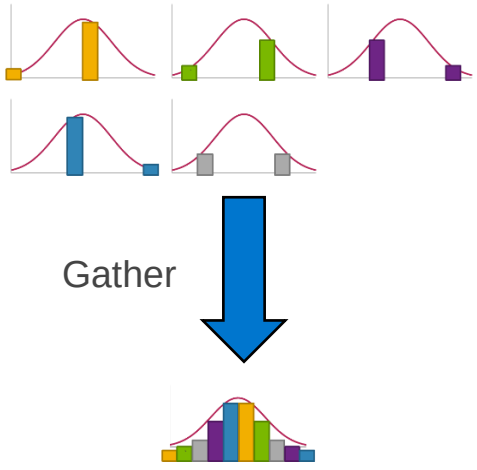
Blue Bonnet

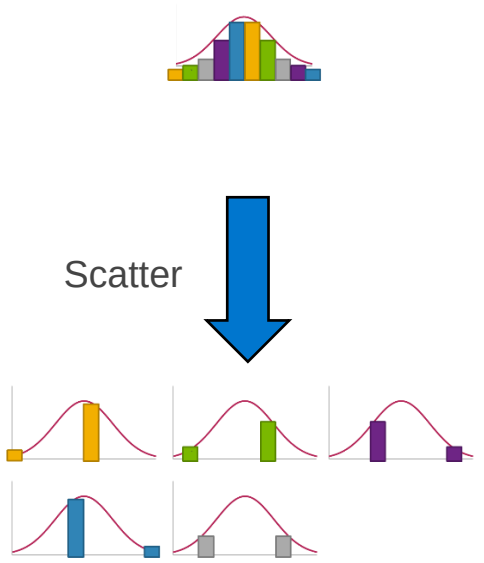
- Dell PowerEdge R6625 based on AMD EPYC processors
- Broadcom RoCE v2

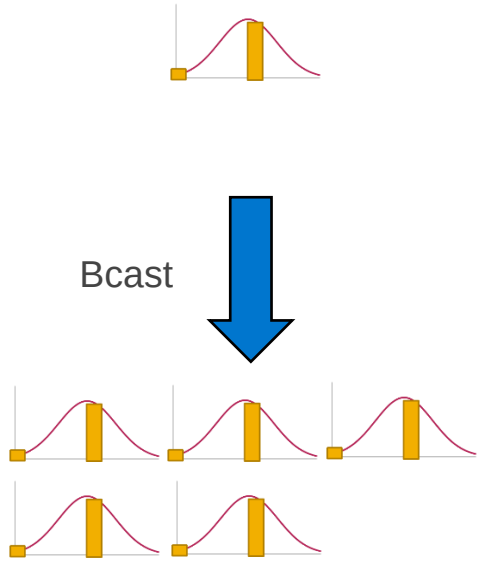
Load imbalance and MPI collectives

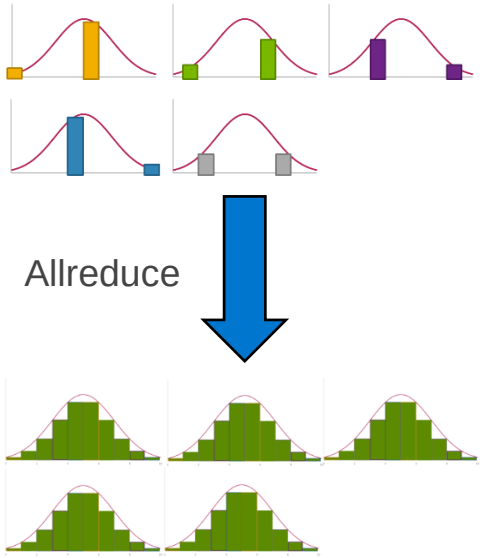
Collective MPI functions

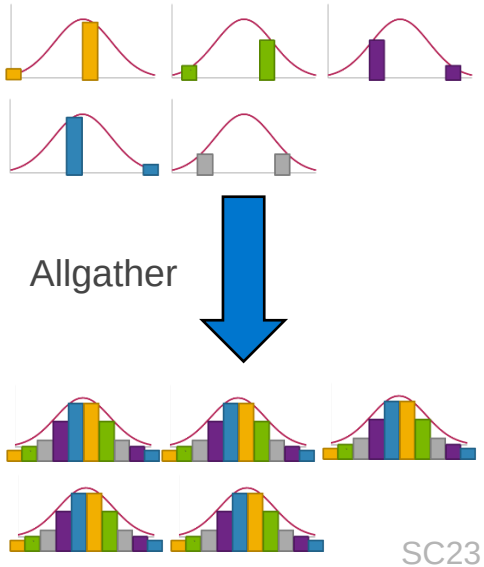


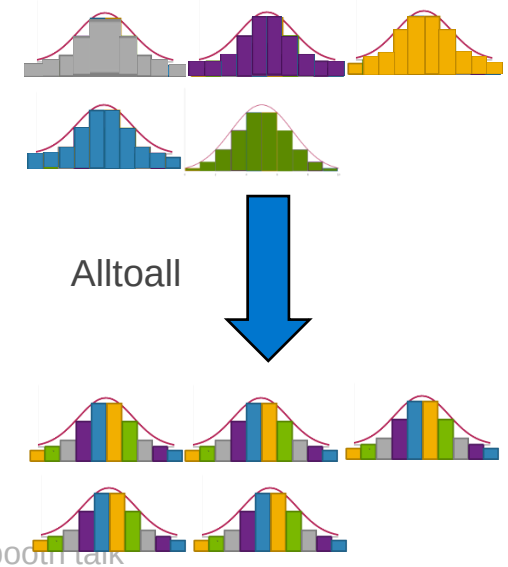




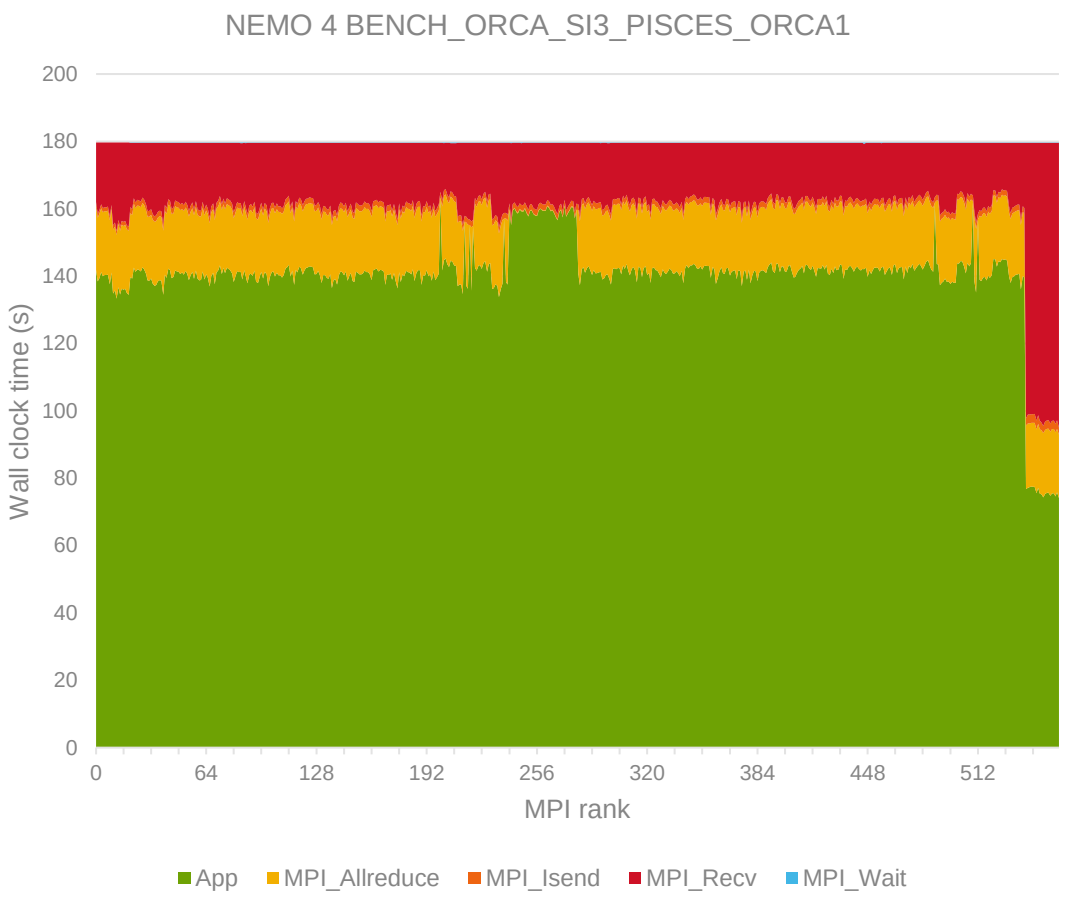








Collective MPI function rationale



- Collective functions spend a lot of time “fixing“ load imbalance
- MPI ranks spend different amounts of time in MPI_Allreduce

MPI-3 feature: non-blocking collectives

- MPI-3 has added support for non-blocking collectives
  - They combine the potential benefits of nonblocking point-to-point operations, to exploit overlap and to avoid synchronization, with the optimized implementation and message scheduling provided by collective operations
  - The call initiates the operation, which indicates that the system may start to copy data out of the send buffer and into the receive buffer
  - Workings are similar to non-blocking point to point calls (Isend/Irecv)

- Blocking:

```
int MPI_Allreduce(  const void *sendbuf, void *recvbuf,
                    int count, MPI_Datatype datatype, MPI_Op op,
                    MPI_Comm comm)
```

- Non-blocking:

```
int MPI_Iallreduce(const void *sendbuf, void *recvbuf,
                    int count, MPI_Datatype datatype, MPI_Op op,
                    MPI_Comm comm, MPI_Request *request)
```

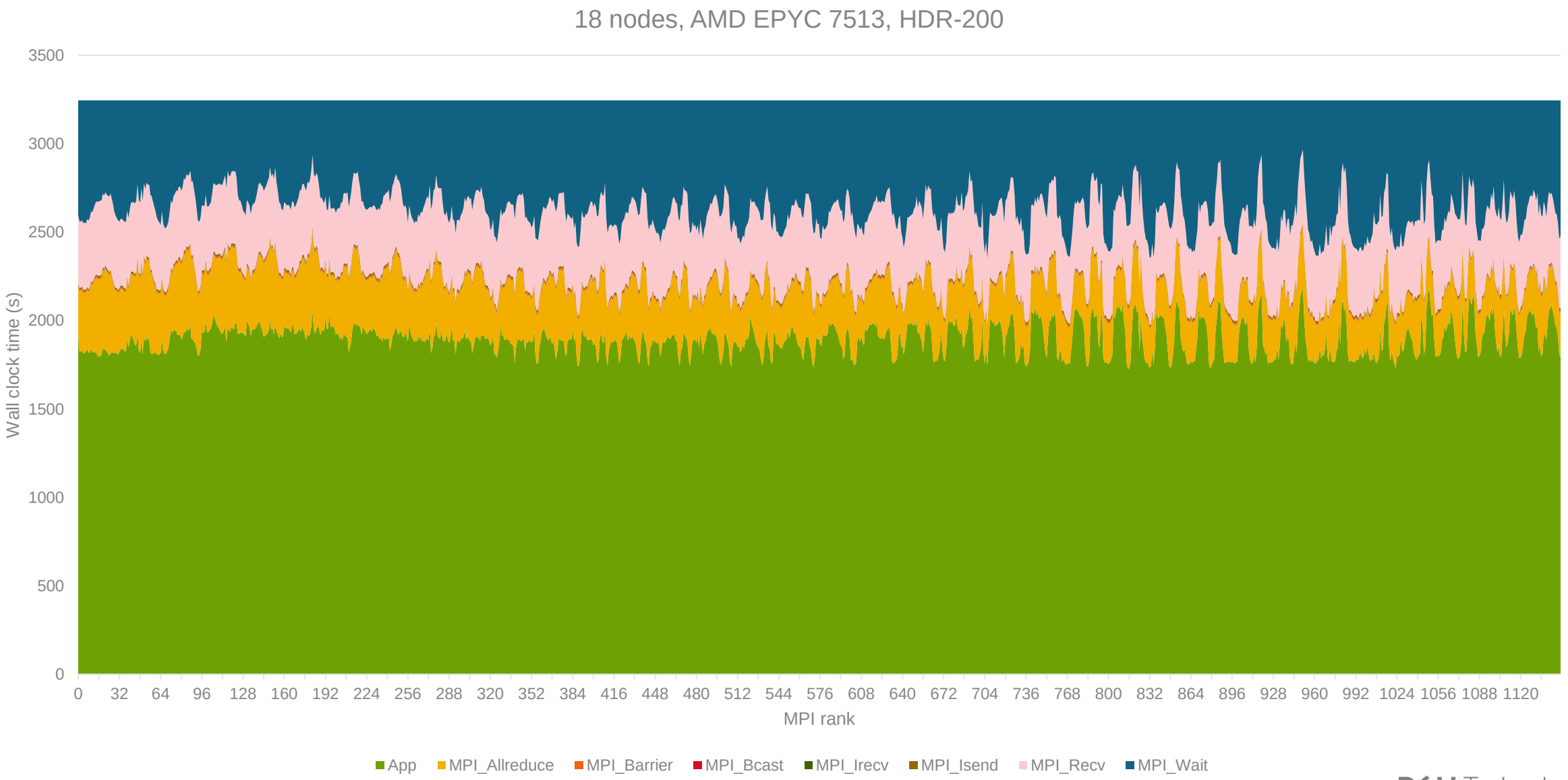
The simple collective example – non blocking

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int size, my_rank, result = 0;
    MPI_Request request;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    MPI_Iallreduce(&my_rank, &result, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD, &request);

    do_something();
    printf("[MPI Process %d] Doing something else here.\n", my_rank);

    MPI_Wait(&request, &status);

    printf("[MPI Process %d] The sum of all ranks is %d.\n", my_rank,
result);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

- Similar calling sequence as MPI_Allreduce
- MPI_Request parameter is to track completion (like in Isend/Irecv)

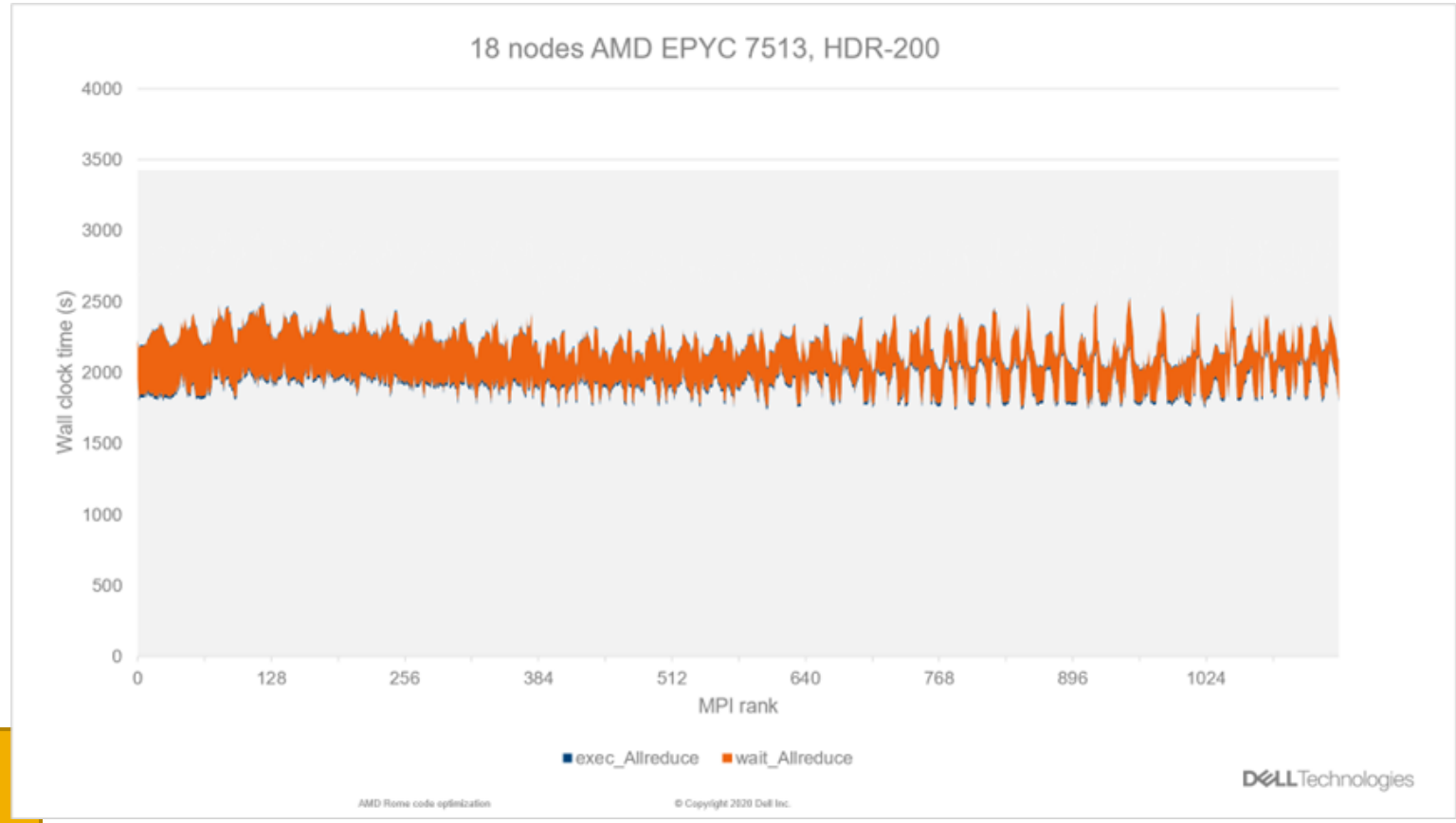
```
% mpirun –np 4 iallreduce.exe
[MPI Process 3] Doing something else here.
[MPI Process 0] Doing something else here.
[MPI Process 1] Doing something else here.
[MPI Process 1] The sum of all ranks is 6.
[MPI Process 2] Doing something else here.
[MPI Process 2] The sum of all ranks is 6.
[MPI Process 3] The sum of all ranks is 6.
[MPI Process 0] The sum of all ranks is 6.
```

MOM5: load imbalance inhibits parallel scaling



Result of perfect interconnect simulation

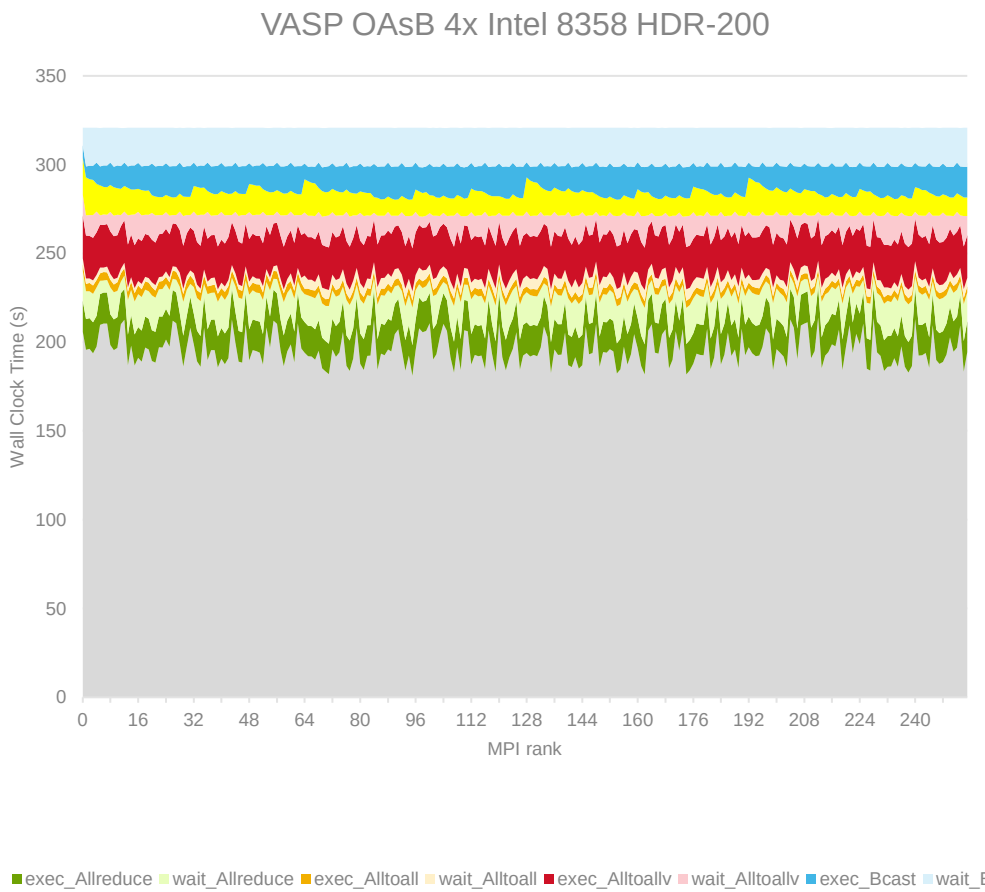
- Wait_*: load imbalance
- Exec_*: actual communication
- Communication overhead is negligible
- Load imbalance is the cause of parallel overhead and potential scaling issue

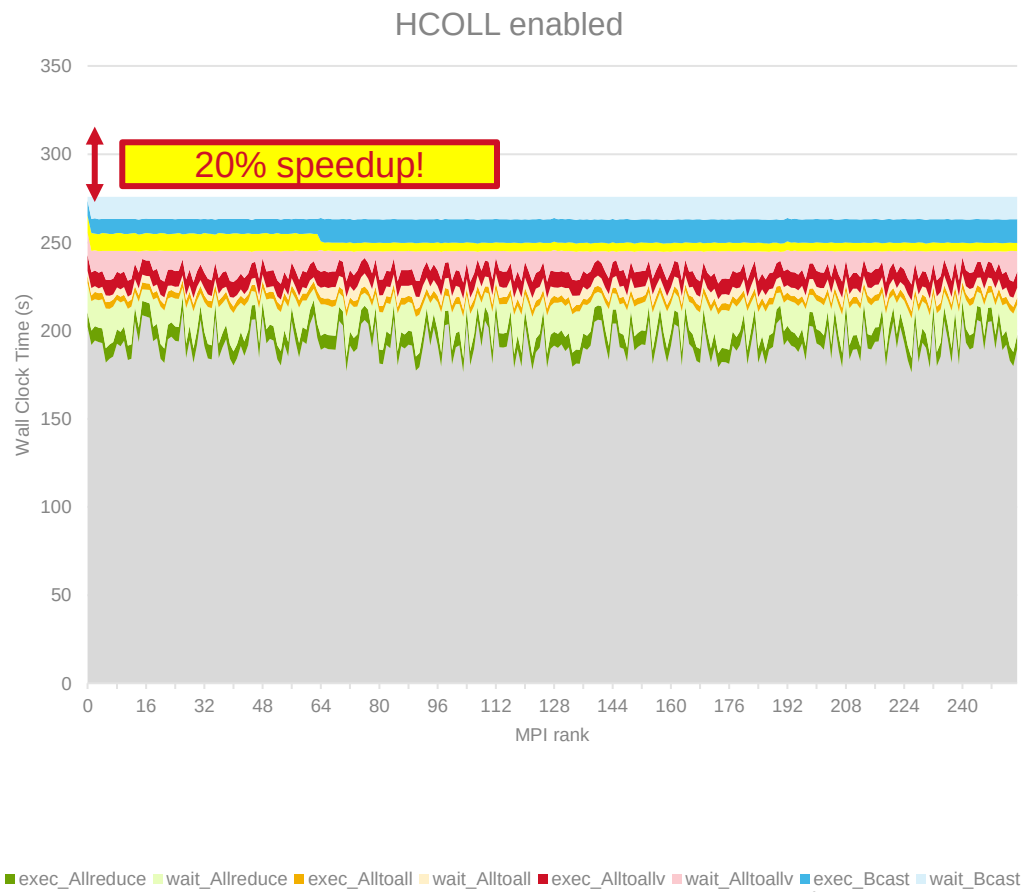


123k calls, 119k < 64 bytes
Minimum time: 191 seconds
Maximum time: 463 seconds
Transfer time: 18 seconds

Use hardware offload of the IB card for collectives

```
% mpirun -genv HCOLL_ENABLE=1              -np 512 <app>   # MVPAPICH2
% mpirun --mca coll_hcoll_enable 1         -np 512 <app>   # Open MPI
% mpirun -genv I_MPI_COLL_EXTERNAL hcoll -np 512 <app>   # Intel MPI
```





Case Study: GROMACS with MVAPICH

Why we like MVAPICH

- Rich set of diagnostics
  - `MVP_SHOW_ENV_INFO=2` combined with an MPI profiler gives you good insight where to tune for eager/rendezvous switch points and choices for collective algorithm and zero copy

- **FAST** job startup on > 4096 cores
  - `MVP_HOMOGENEOUS_CLUSTER=1`

- MPI-IO works as should be and allows us to give MPI file info hints via `ROMIO_HINTS=my_hints_file.txt`

- We set `MVP_USE_ALIGNED_ALLOC=1` by default to work around the occasional job failure

Application characteristics

Interaction	Parallelism	Communication Pattern	Notes
Particle-particle	MPI_Sendrecv	Point-to-point	Latency sensitive
PME	MPI_Alltoall	All-to-all	Lot of data movement and very bandwidth intensive

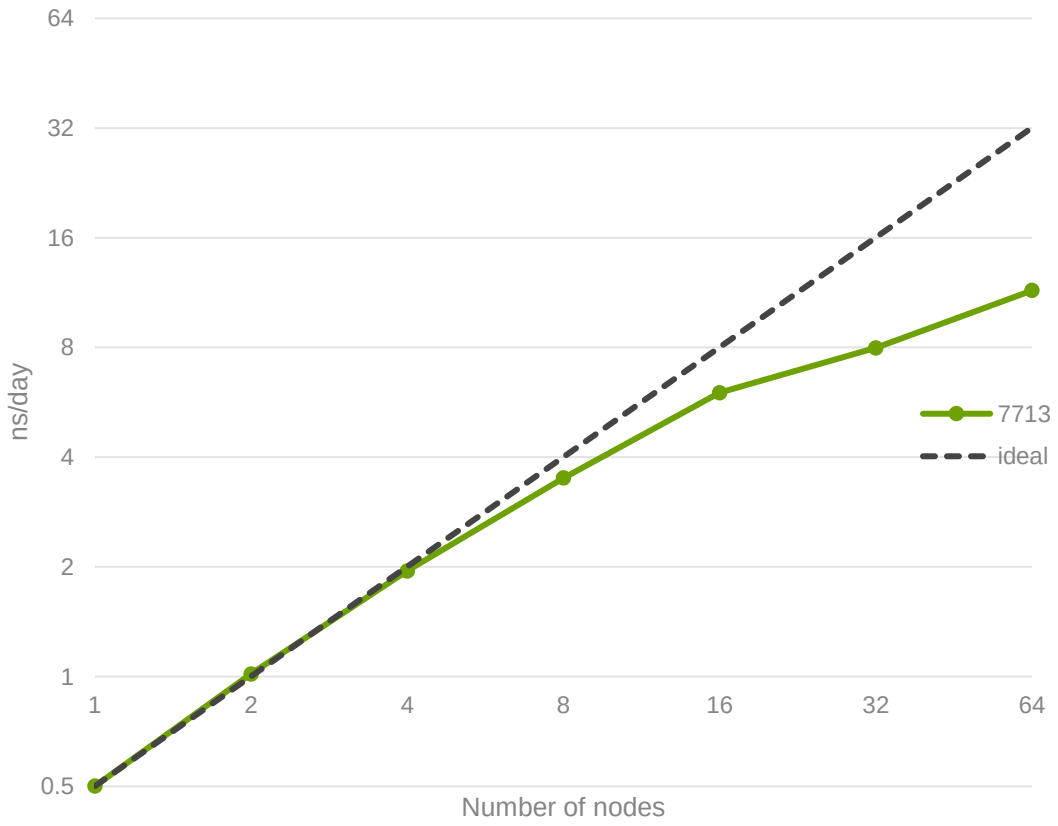
Application and data set

- Gromacs 2023.2
  - Compiled with AMD AOCC 4.0.0, AMD AOCL 4.0 for BLAS3, FFTs and math
  - MVAPICH 2.3.7
- Protein in water
  - 24.4 M atoms, 15000 steps
  - Combined PP/PME run, no tuning of PP/PME rank ratio

Compute environment

Dell	PowerEdge C6625
Processor	AMD EPYC 7713 64C 2.00 GHz
Memory	256 GB dual rank DDR4 @ 3200 MT/s
Interconnect	NVIDIA ConnectX-6 HDR-200
OS	RHEL 8.6

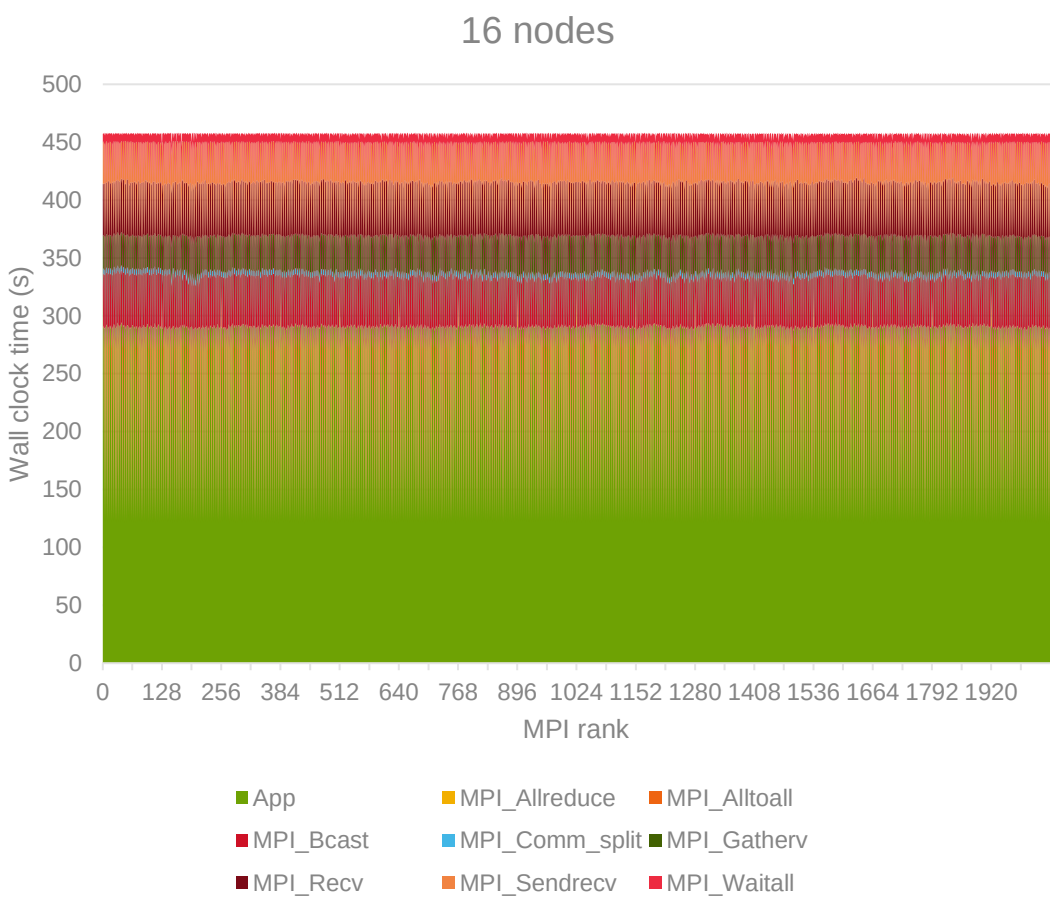
Initial performance results



Environment	
Compiler	AMD AOCC 4.0.0
Math library	AMD AOCL 4.0
MPI library	MVAPICH 2.3.7

- Scaling tails off quickly
- Theoretically data data set should scale well

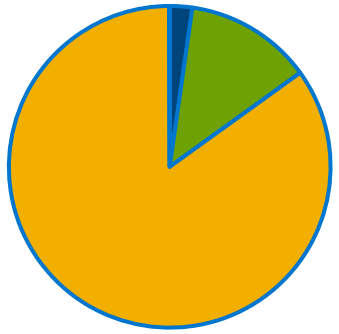
MPI communication breakdown



- Much time spent in
  - Bcast
  - Gatherv
  - Sendrecv
  - Alltotall (PME ranks only)
  - Recv/Waitall (load imbalance between PP and PME ranks?)

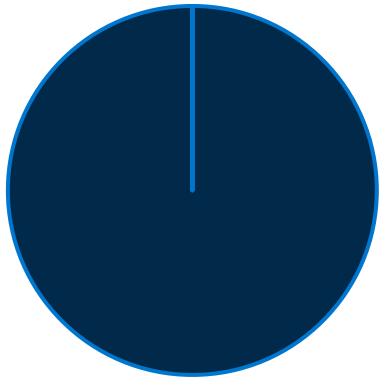
MPI message sizes

Bcast



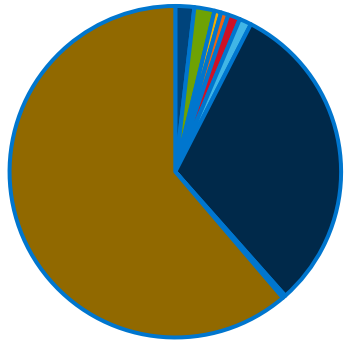
0-64 bytes
64-512 bytes
512-2048 bytes
> 4194304 bytes

Gatherv



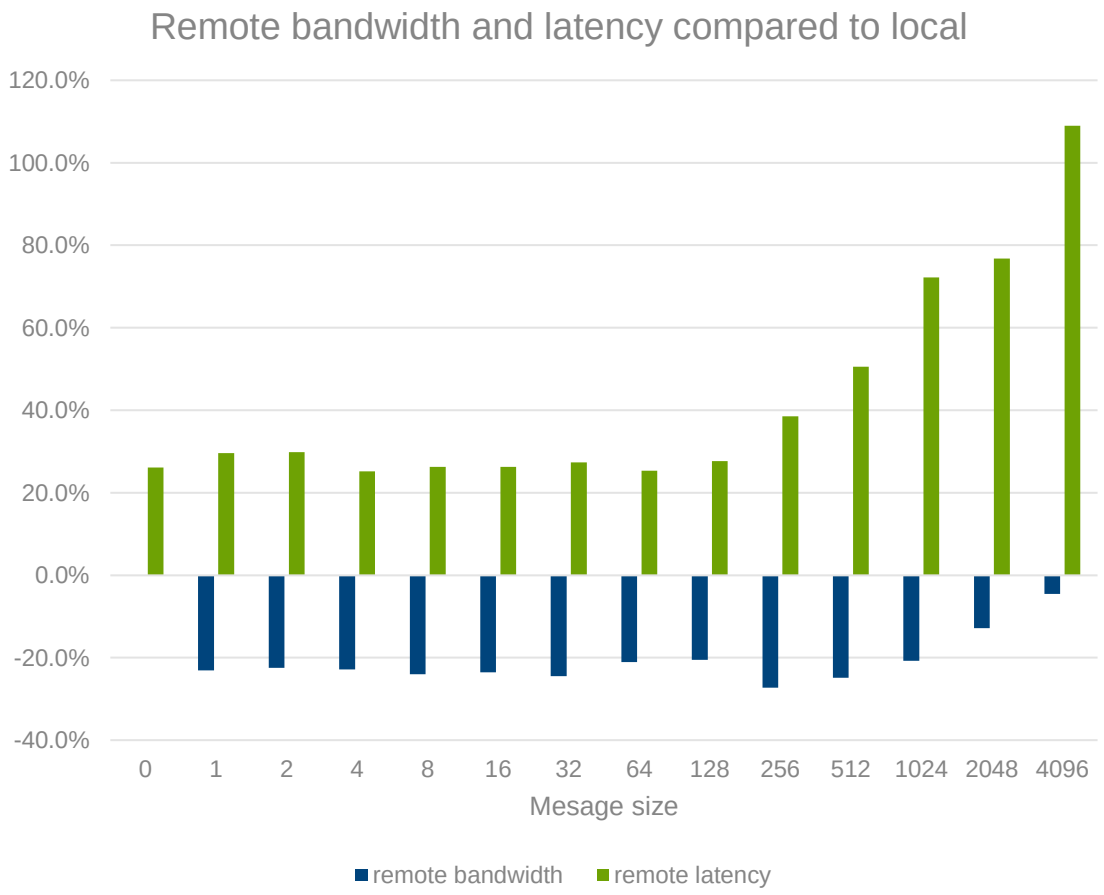
65536-262144 bytes

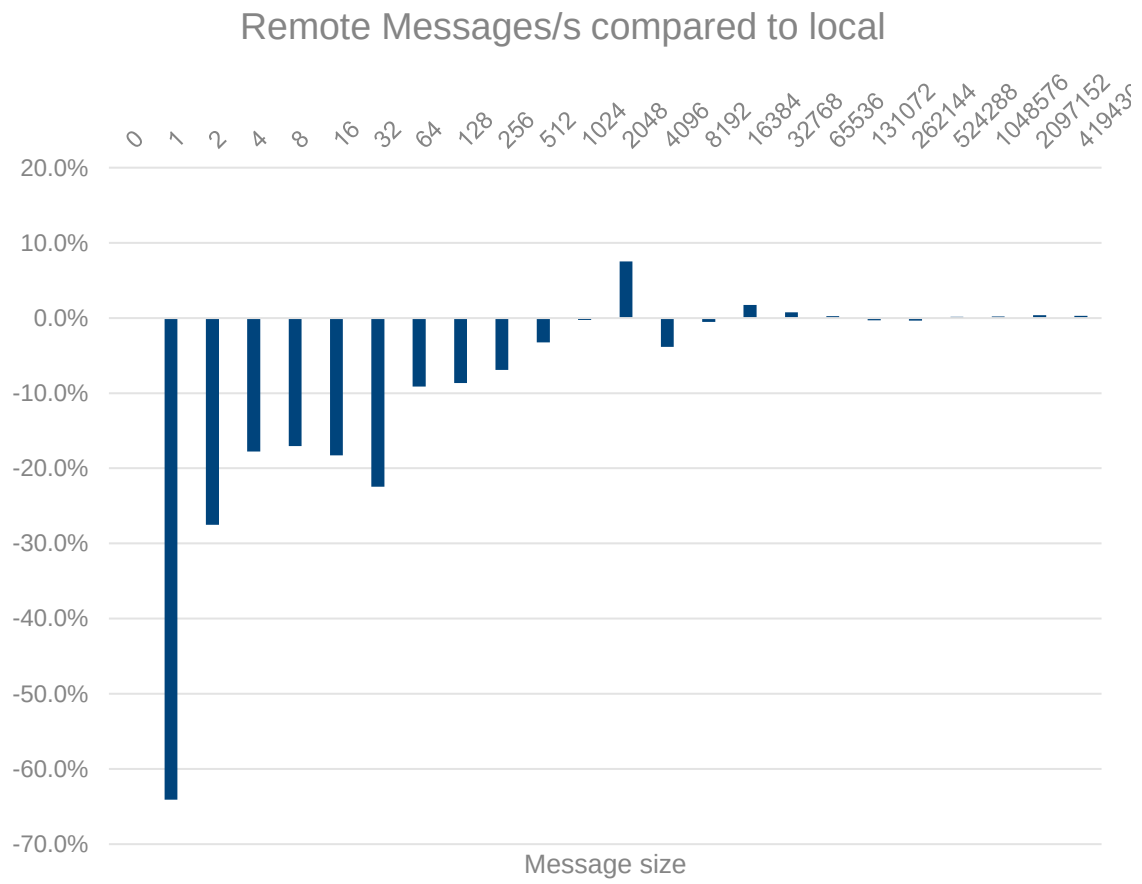
Sendrecv



0-64 bytes
64-512 bytes
512-2048 bytes
2048-4096 bytes
4096-16384 bytes
16384-65536 bytes
65536-262144 bytes
262144-524288 bytes
524288-1048576 bytes

Step 7: Look at MPI itself





ApbDis and DLWM BIOS Options

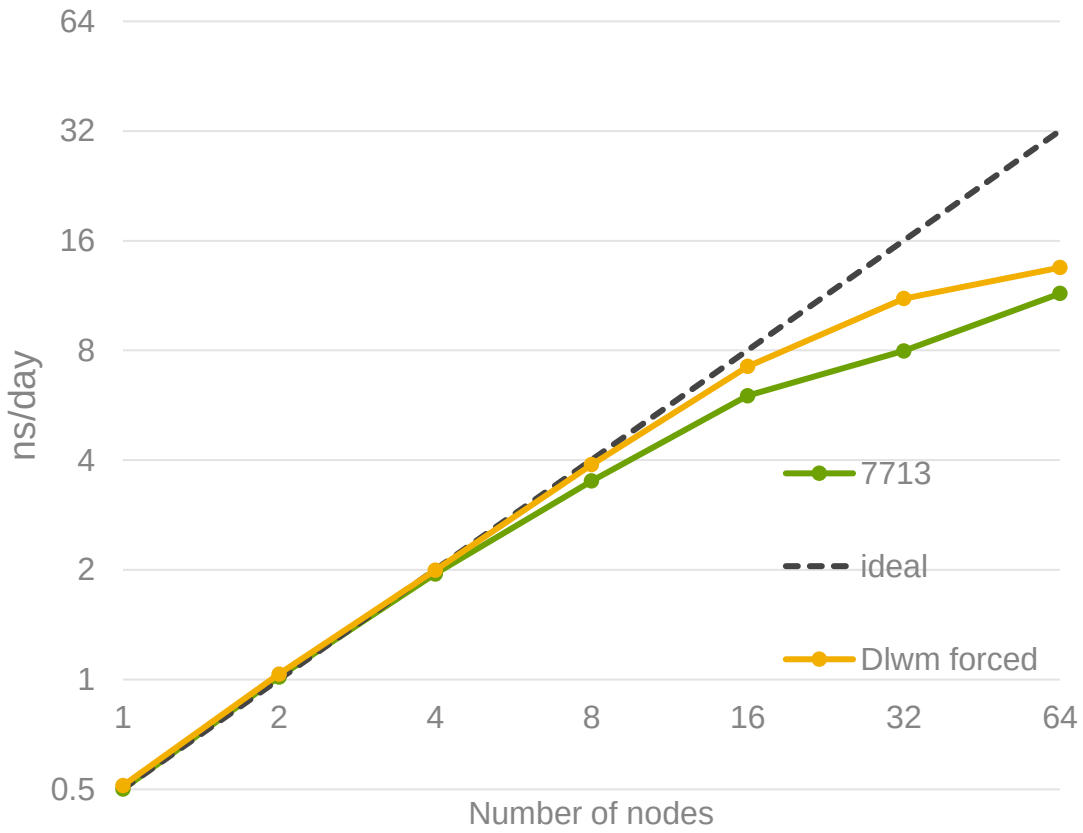
- ApbDis (Algorithmic Performance Boost Disable)
  - Governs the boost behavior of the Infinity Fabric.
  - AMD recommend setting this to 1/Enabled for HPC workloads along with Fixed SOC P-state set to P0 (max performance).
- DLWM (Dynamic Link Width Management)
  - xGMI Dynamic Link Width Management saves power during periods of low socket-to socket data traffic by reducing the number of active xGMI lanes per link from 16 to 8.
  - AMD recommends setting this to Forced x16 for HPC workloads.
- With ApbDis=Disabled and DLWM=Unforced, Infinity Fabric state transitions and xGMI link width transitions cause system blackout periods of 100 to 200 µs.
- Both of these settings impact Infinity Fabric latency, bandwidth and power consumption.

Performance and Power Impacts of ApbDis

BIOS Configuration	ApbDis	DLWM	Local Latency	Remote Latency	System Idle Power	HPL (GFLOPS)
Performance Optimized	Disabled	Unforced	1.06 µs	1.45 µs	270 w	3600
ApbDis=1	Enabled	Unforced			+45 w	3393 (-6%)
DLWM=x16	Disabled	Forced x16			+25 w	3490 (-3%)
ApbDis=1 + DLWM=x16	Enabled	Forced x16	1.05 µs	1.23 µs	+80 w	3277 (-9%)

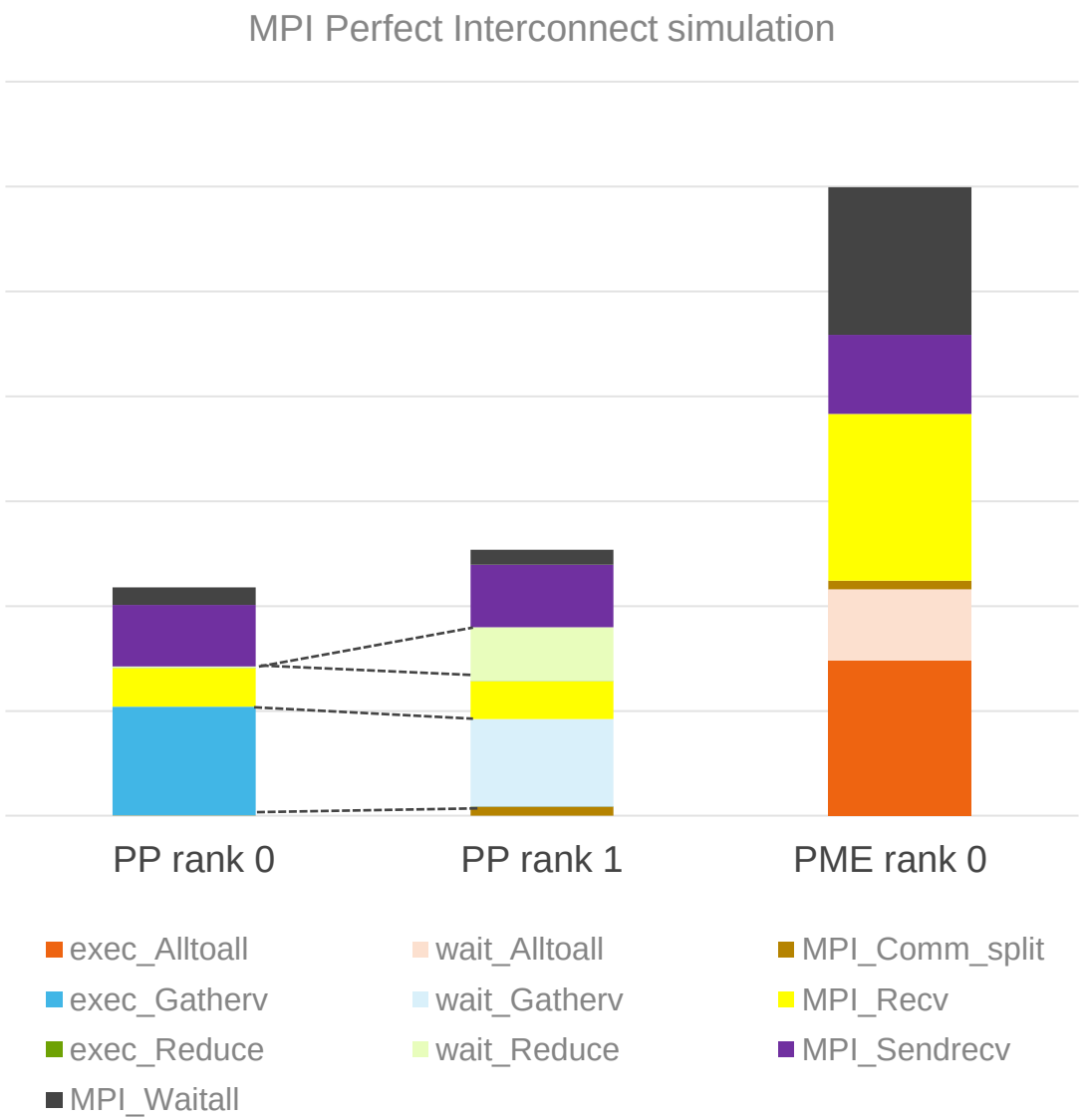
- All testing performed using R6525 with AMD Milan 7713 and BIOS v2.0.1.
- Server-to-server latency measured using OSU benchmarks with HDR InfiniBand and sockets local and remote to the InfiniBand HCA.
  - Remote latency test transits the xGMI link on both servers.

Disabling DLWM improves performance



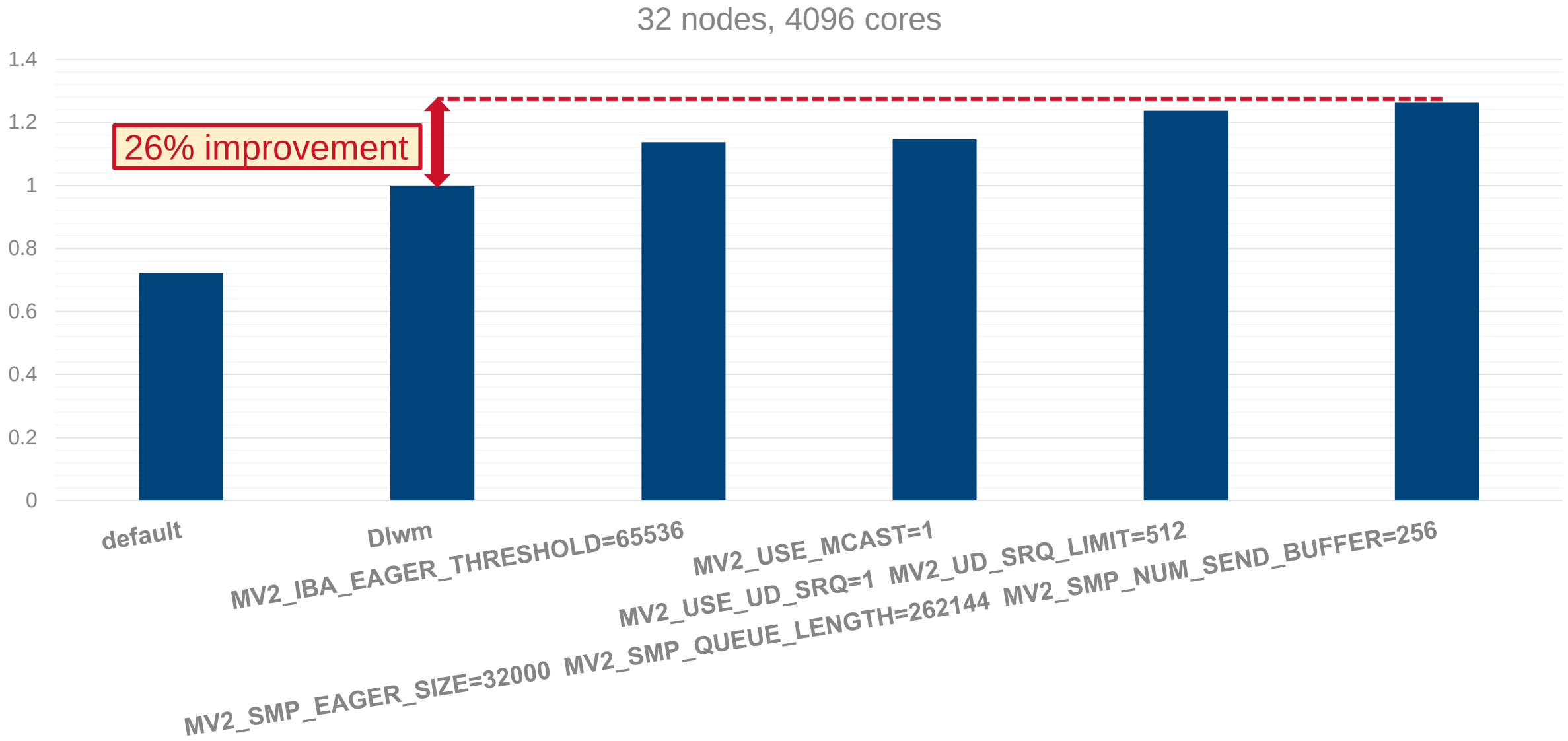
- Better performance at 8 and 16 nodes
- Still bad at larger node counts

Deeper investigation

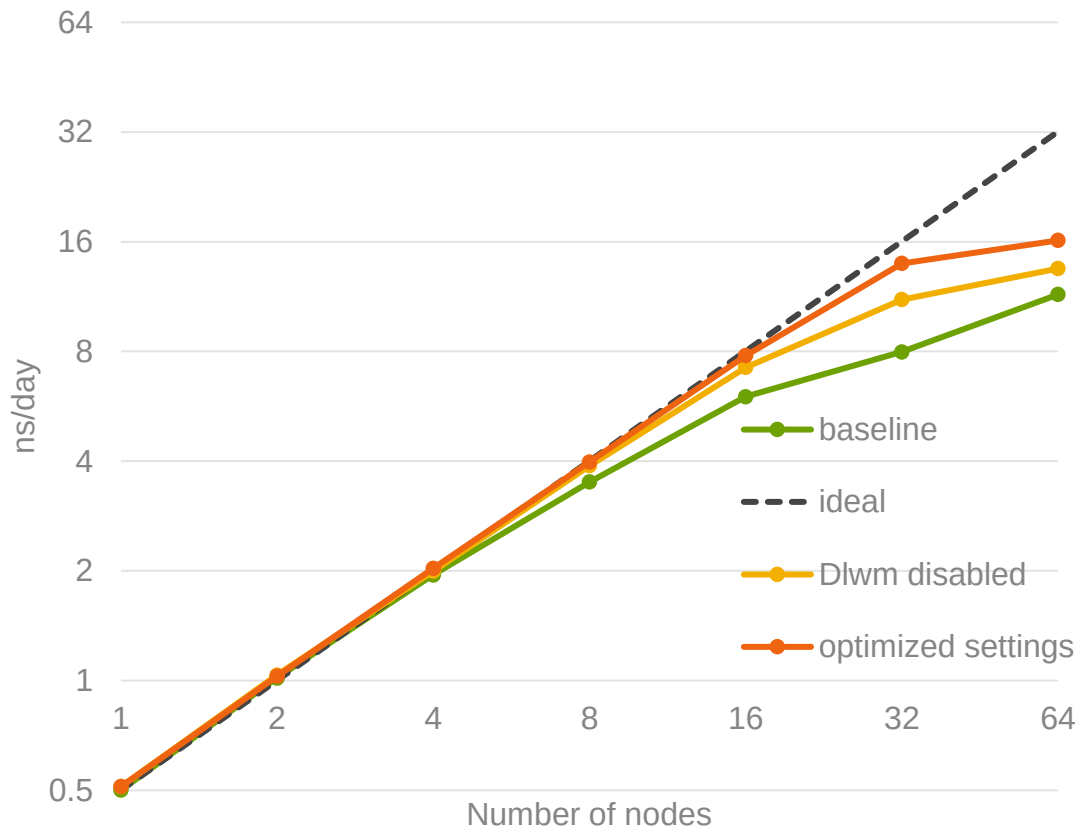


- Much time spent in MPI_Gatherv for only few calls
- Large MPI_Wait in PME, load imbalance?
- MPI_Alltoall execution dominates, bandwidth limited?

Improving MVAPICH performance

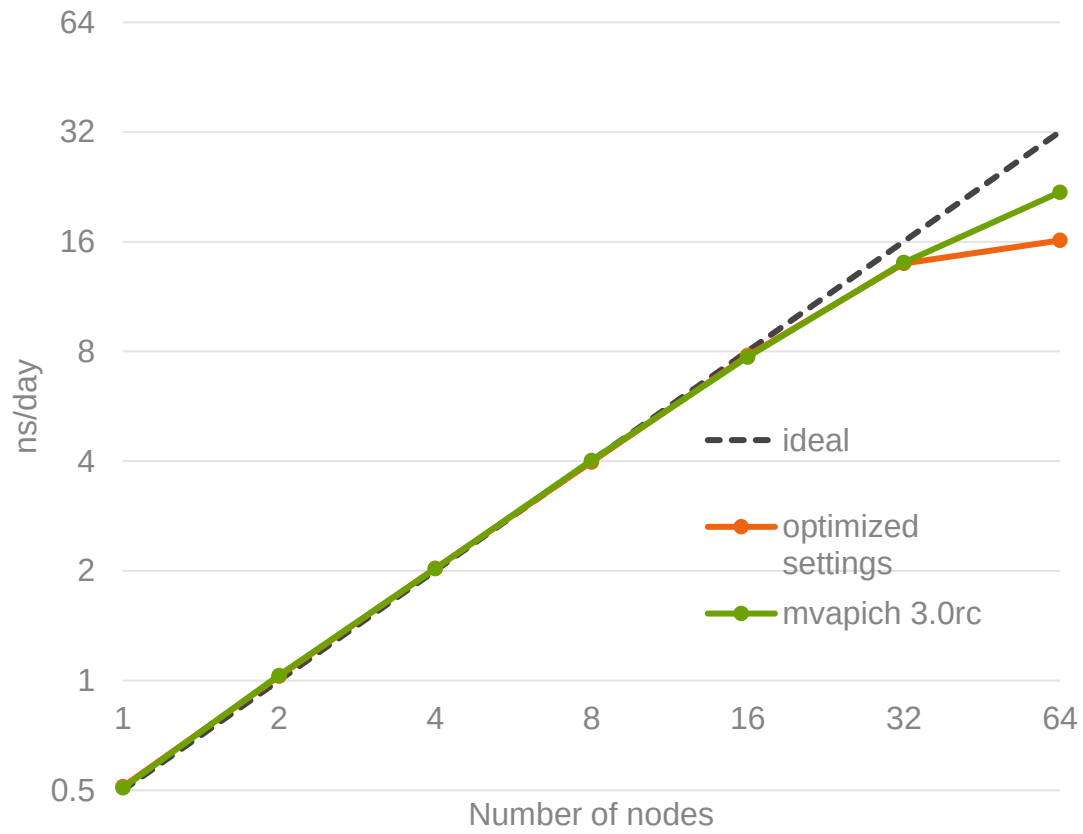


Final result



- 26% performance improvement on 32 nodes
- 64 node result does not scale due to Gatherv

A peek at MVAPICH 3.0



- UCX allows better scalability on 64 nodes

Conclusion

- MVAPICH is a robust MPI implementation allowing performance at scale for popular HPC applications
- Performance can be further improved through a rich set of tunables, which can be set with environment variables


DELL Technologies