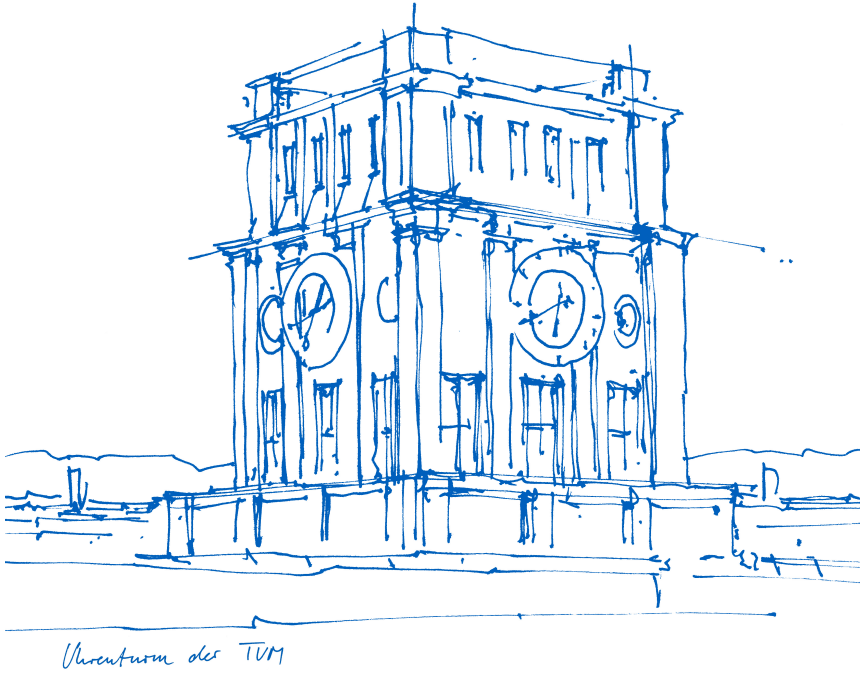
Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik

PLEASE FILL OUT
THE LECTURE
SURVEY
(Dedicated time until 16:30)

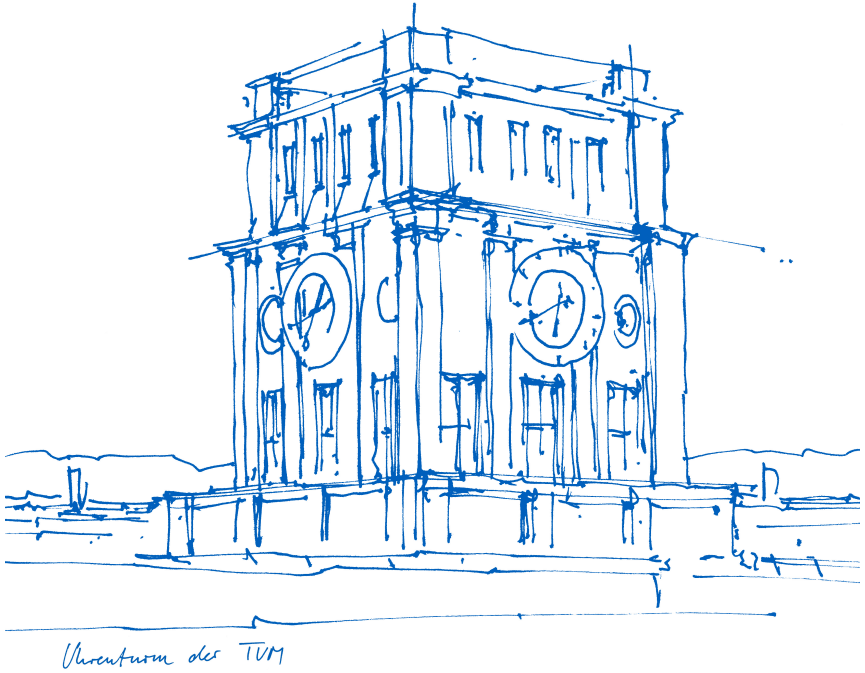


Lecture IN-2147
Parallel Programming

SoSe 2018

Martin Schulz
Exercises: Amir Raoofy
Technische Universität München
Fakultät für Informatik

Lecture 11:
Scaling



Summary from Last Time(s)

Range of choices in MPI API
- Point to point calls, blocking & non-blocking
- Collectives, blocking & non-blocking
- Communicator management
- One sided communication
- Datatypes

Hybrid Programming

Performance Analysis and Optimization
- Key metrics, such as speed-up
- Importance of load balance
- Categories of performance analysis tools: profiling vs. tracing

Workflow of performance analysis
- Instrumentation
- Execution
- Analysis
- Refinement

Scaling and its Impact

The level of concurrency (number of HW threads) is rising in HPC
- Single thread performance no longer growing
- Trend towards multicores
- Trend towards highly parallel accelerators
- Still rising number of nodes

Concurrency needs to be exploited in software
- Problems need to be able to be split into smaller pieces
- No dependencies
- Fine-grained work distribution with load balance

New application requirements
- Heterogeneous workloads
- More complex workflows

New programming approaches (next 2 lectures)
- Accelerators require special treatment
- Task-based models emerging as one option

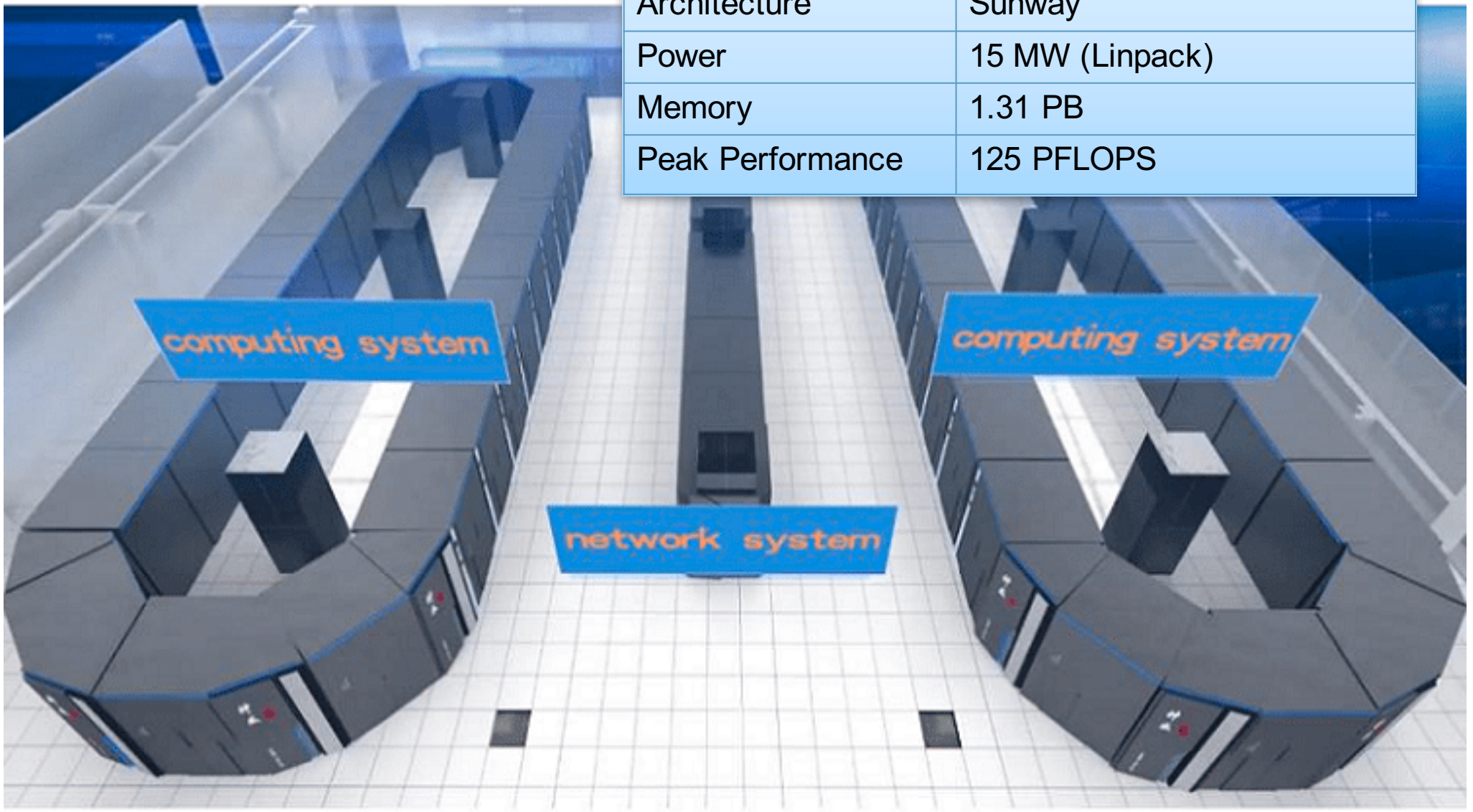
Top500 List as of November 2017

#	Site	Manufacturer	Computer	Country	Cores	Rmax [Pflops]	Power [MW]
1	National Supercomputing Center in Wuxi	NRCPC	Sunway TaihuLight NRCPC Sunway SW26010, 260C 1.45GHz	China	10,649,600	93.0	15.4
2	National University of Defense Technology	NUDT	Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi	China	3,120,000	33.9	17.8
3	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC50, Xeon E5 12C 2.6GHz, Aries, NVIDIA Tesla P100	Switzerland	361,760	19.6	2.27
4	Japan Agency for Marine-Earth Science and Technology	ExaScaler	Gyoukou ZettaScaler-2.2 HPC System, Xeon 16C 1.3GHz, IB-EDR, PEZY-SC2 700Mhz	Japan	19,860,000	19.1	1.35
5	Oak Ridge National Laboratory	Cray	Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x	USA	560,640	17.6	8.21
6	Lawrence Livermore National Laboratory	IBM	Sequoia BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	1,572,864	17.2	7.89
7	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	979,968	14.1	3.84
8	Lawrence Berkeley National Laboratory	Cray	Cori Cray XC40, Intel Xeons Phi 7250 68C 1.4 GHz, Aries	USA	622,336	14.0	3.94
9	JCAHPC Joint Center for Advanced HPC	Fujitsu	Oakforest-PACS PRIMERGY CX1640 M1, Intel Xeons Phi 7250 68C 1.4 GHz, OmniPath	Japan	556,104	13.6	2.72
10	RIKEN Advanced Institute for Computational Science	Fujitsu	K Computer SPARC64 VIIIfx 2.0GHz, Tofu Interconnect	Japan	795,024	10.5	12.7

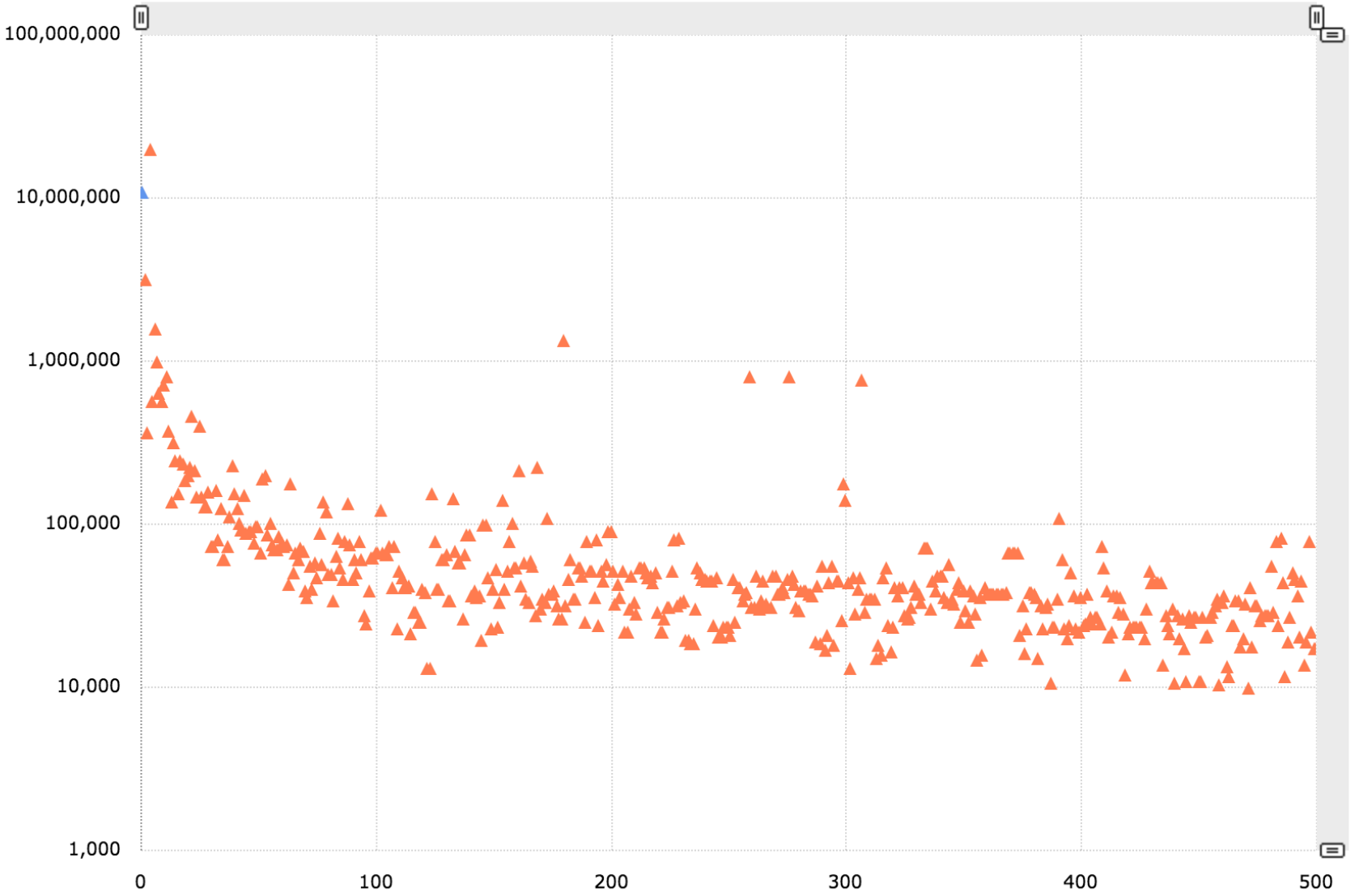
Source: Top500

SunwayTaihu Light

Location	National Supercomputer Center, Wuxi, Jiangsu, China
Architecture	Sunway
Power	15 MW (Linpack)
Memory	1.31 PB
Peak Performance	125 PFLOPS



Number of Cores Across the Top500 Machines



Source: Top500

Consequences of Scaling

Types of Scaling

Impact on Programmability

Impact on Algorithms and Data Structures

Impact on Mapping Code to Machines

Impact on I/O

Types of Scaling

Scaling in this case means changes in the number HW threads used

Need to adjust application and runtime parameters as we do this
- Problem size
- Problem decomposition

Useful to keep one characteristics constant

Weak Scaling

Keeping the problem size per HW thread/core/node constant

- Larger machine -> larger problem
  - Expanding the problem domain
  - Increasing refinement
- Traditionally most common way to deal with scaling in HPC

Assumptions

- Machines are never big enough
- Fully loading a node/core/HW thread is the right thing to do
  - Exploiting the resources on one unit and keeping that constant

Advantages

- Execution properties (like cache behavior) often stay fixed
- Easier to scale, as overheads stay roughly constant as well

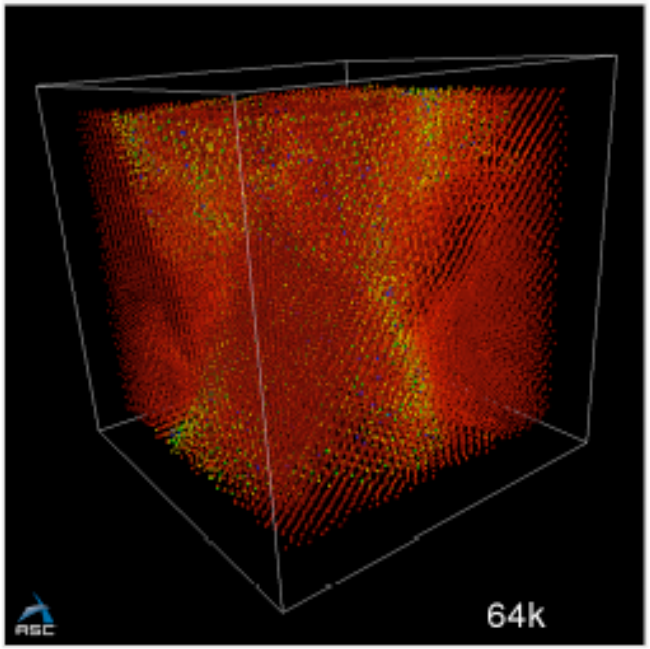
Challenges

- Keeping load constant can be tricky for complex applications (2D/3D problems)
- Multiple ways to repartition a workload

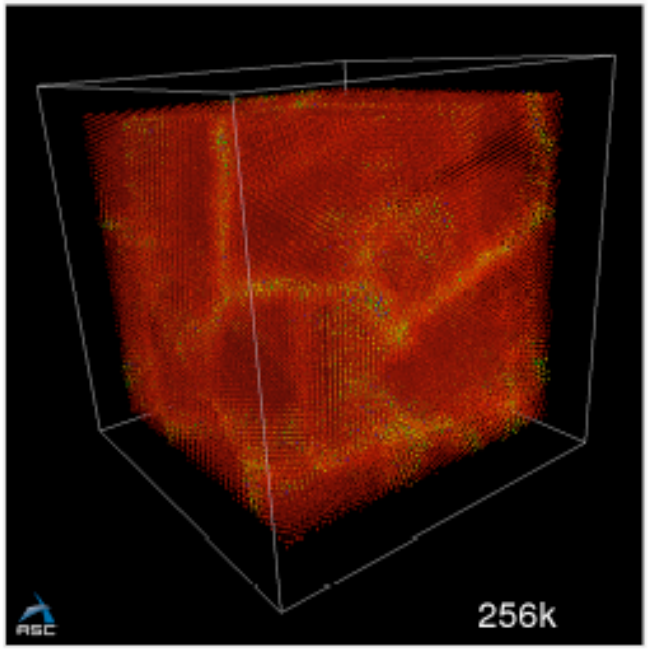
Example: Material Solidification Process

Molecular dynamics at LLNL with ddcMD: 2 Million atom run (2005)

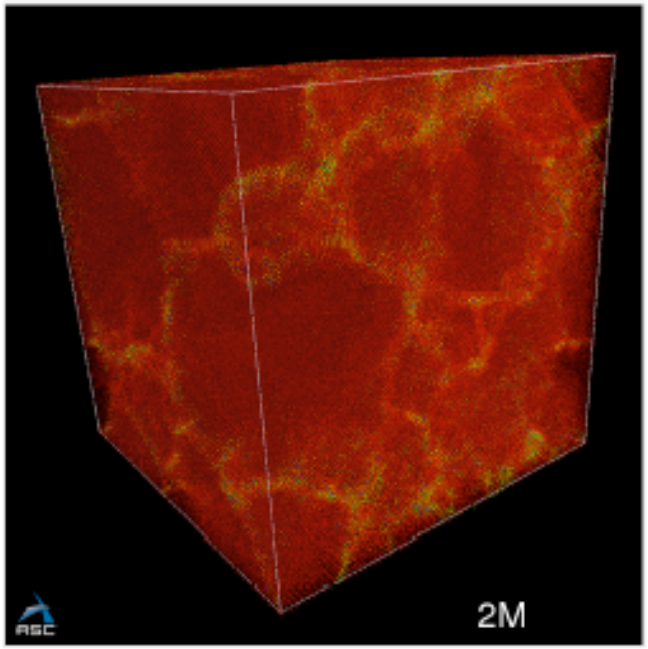
- Used all of Blue Gene/L (128K cores)
- Close to perfect scaling
- New scientific observations



64,000 atoms



256,000 atoms



2,048,000 atoms

https://e-reports-ext.llnl.gov/pdf/336469.pdf

Strong Scaling

Keeping the total problem constant
- Larger machine -> (hopefully) faster execution
- Need to adjust problem distribution
- Traditionally not the most common type of scaling
  - But becoming rapidly more and more relevant

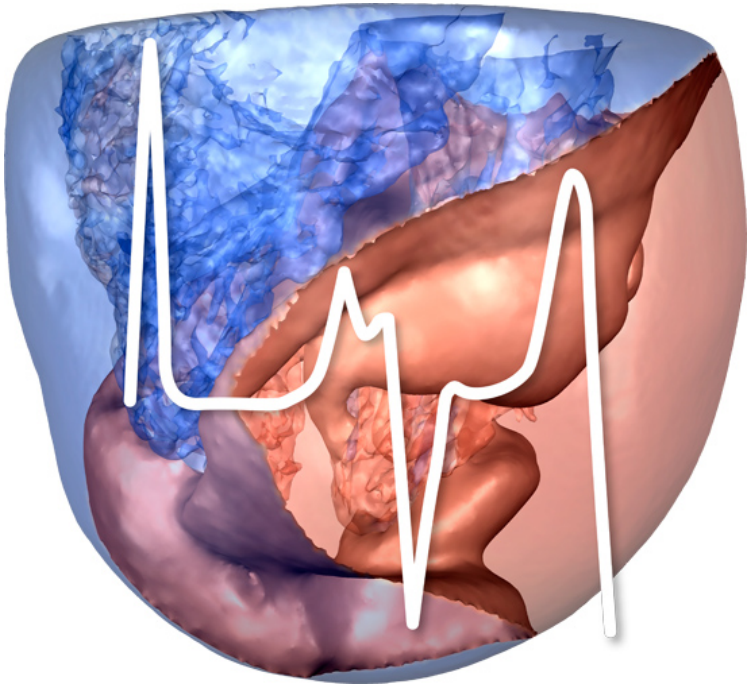
Assumptions:
- The machine is big enough for the problem
- Goal: reducing time to solution and increasing throughput

Needed for time critical tasks
- Emergency response to natural disasters
- Real-time simulations
- Large-scale ensemble calculations

Challenges
- Harder to scale, as overheads grow with smaller per HW thread workloads
- Changing executing characteristics (cache miss rates, etc.)

Example: Cardiac Simulation (BG/Q at LLNL)

https://education.llnl.gov/programs/science-on-saturday/lecture/541 and https://str.llnl.gov/Sep12/streitz.html



Electrophysiology of the human heart

- Close to real time heart simulation
- Near cellular resolution

Parallel programming aspects

- Strong scaling problem
- Simulation on 1.600.000 cores
- ”Bare metal” programming
- Achieved close to 12 Pflop/s

Types of Scaling (cont.)

Scaling in this case means changes in the number HW threads used

Need to adjust application and runtime parameters as we do this
- Problem size
- Problem decomposition

Useful to keep one characteristics constant

Also other tuning options may be necessary
- Algorithmic knobs (due to changed conditions)
- Runtime knobs (e.g., due to changed message sizes)
- Thread vs. process balance
- Job mapping to machine

Basically running a different set of programs
- Can expose new bugs/correctness issues
- Can expose new performance issues

Impact on Programmability

Things get harder at scale
- Access to batch queues
- Startup and finalization times
- I/O access to files and startup information

This includes debugging
- Printf becomes problematic
  - 1 printf on 1,000,000 HW threads = 13,600 pages of text at 8pt font
- Interactive debugging no longer feasible

Aim at reproducing bugs at small scale
- Isolating kernels
- Similar conditions by applying the right scaling

But: in same cases we need to debug at scale
- Bugs sometimes don’t manifest themselves at small scale
- Size reduction and/or debugging additions change conditions

Debugging at Scale

Traditional debugging paradigm does not scale well.

- Huge quantities of symbol table and process state information
- Too much information to present to the user

- First need to reduce search space to manageable subset
- Apply traditional debugging techniques on subset

How do we debug applications at O(1M)?!

Observation 1:

parallel applications have a lot of processes executing the same code.

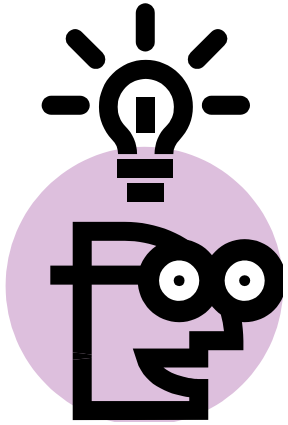
Lightweight tools to quickly identify ***process equivalence classes***

Feed a representative of each class into a full-featured debugger

Observation 2:

Stack traces are a good indicator of process behavior

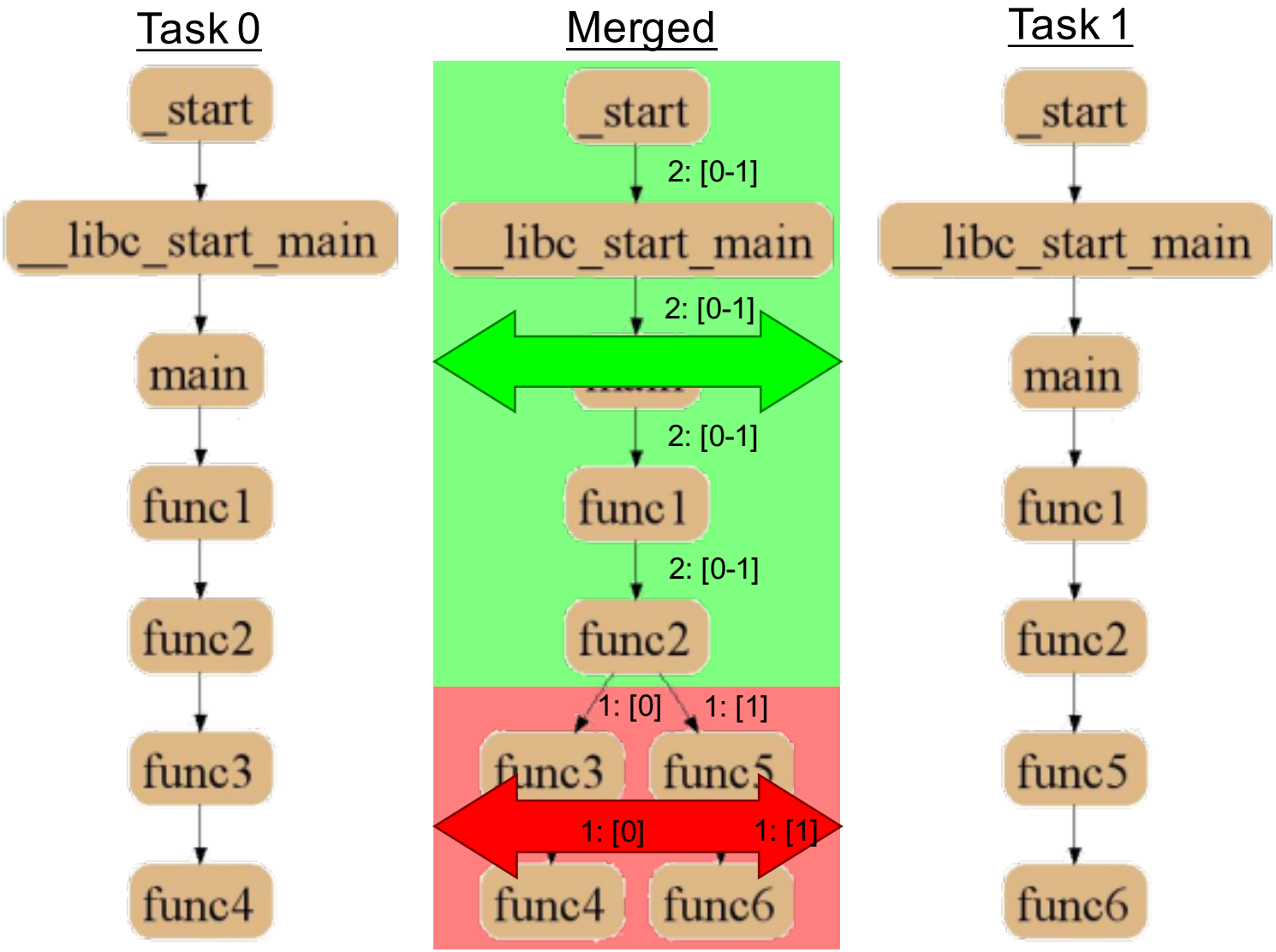
Use stack traces to identify equivalence classes



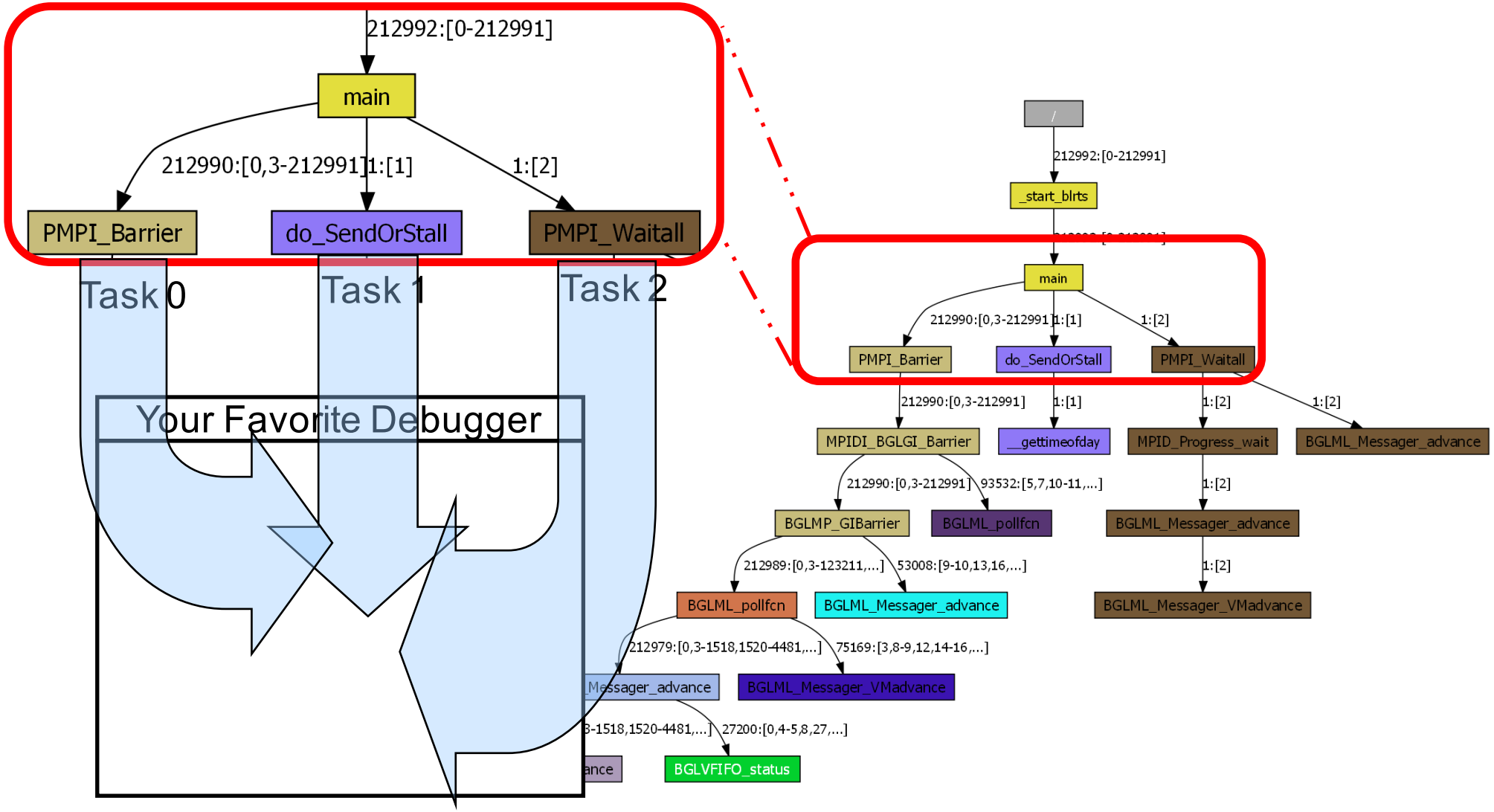
Observation 3:

Time varying behavior provides additional insight

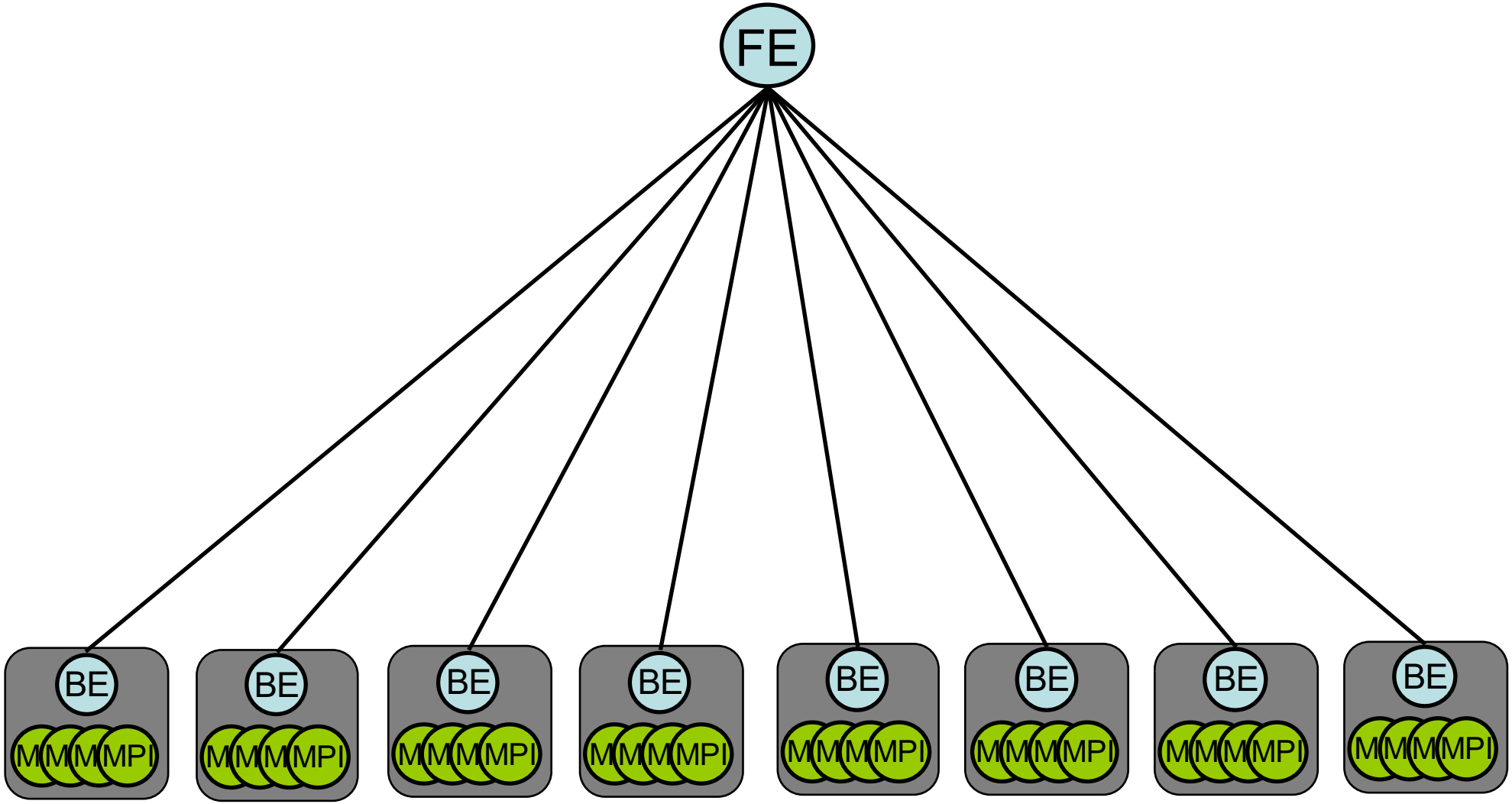
Stack Traces: the basis for STAT



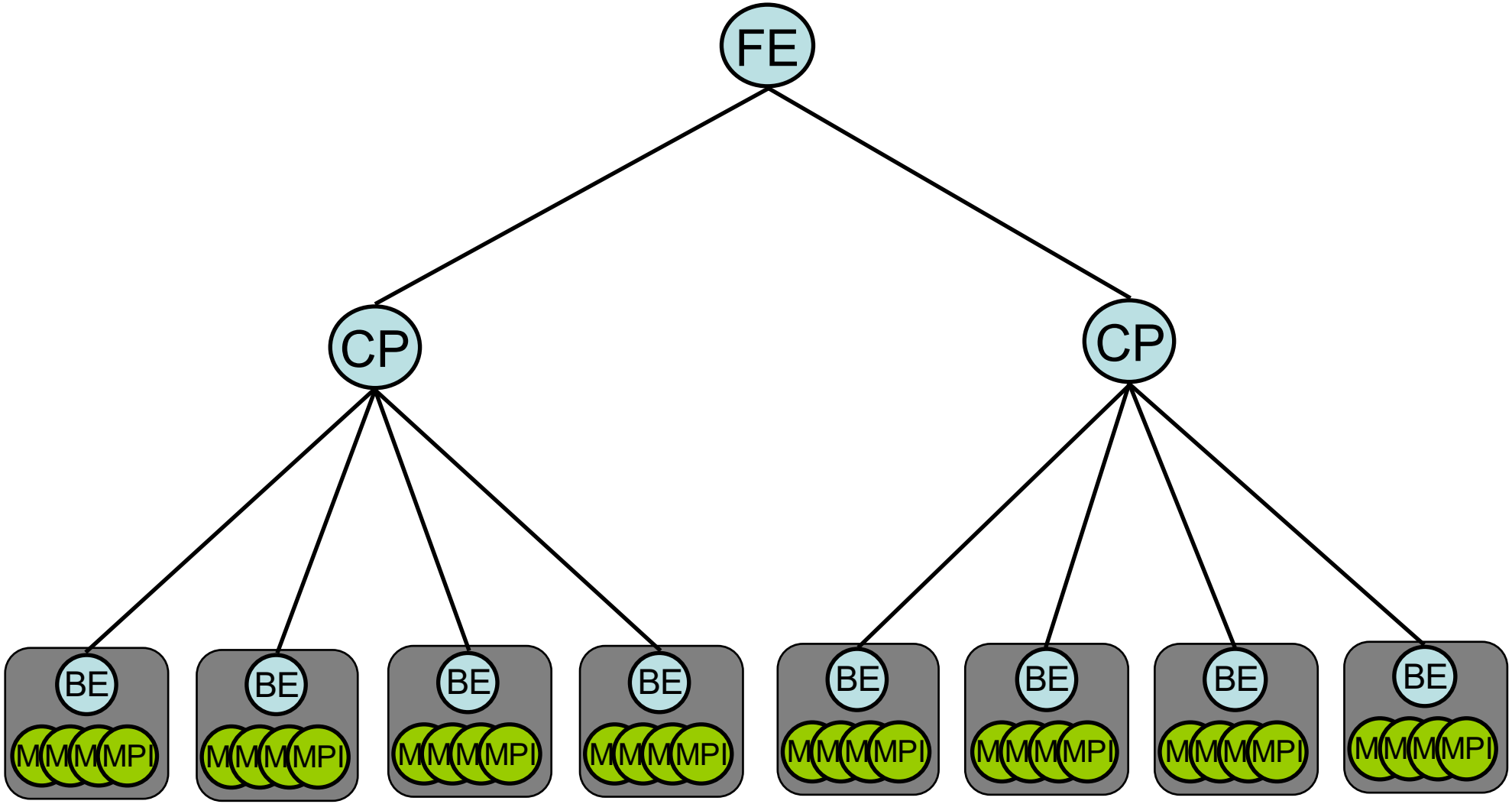
STAT compliments traditional parallel debuggers



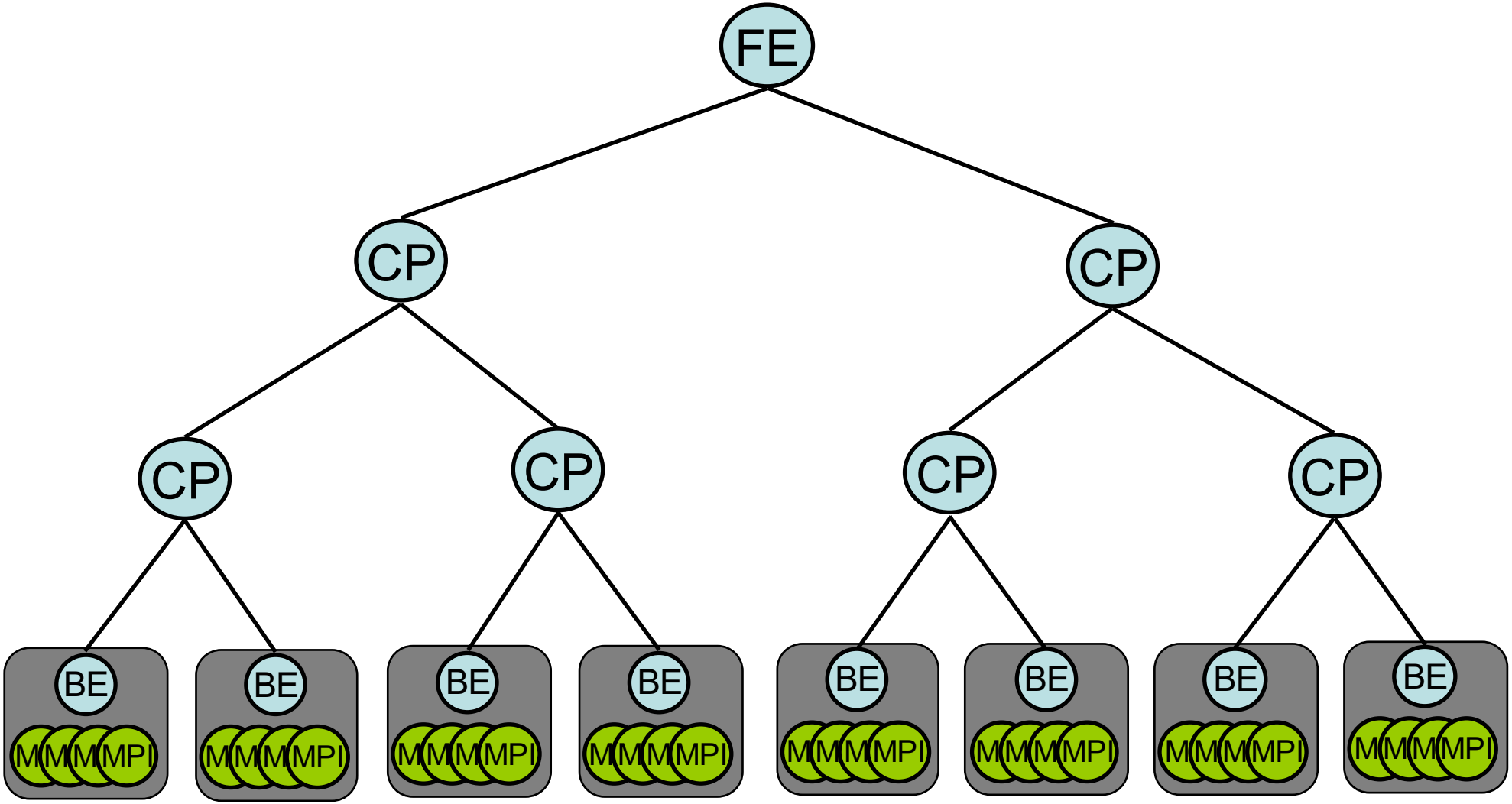
Need to Gather Data From All MPI Processes



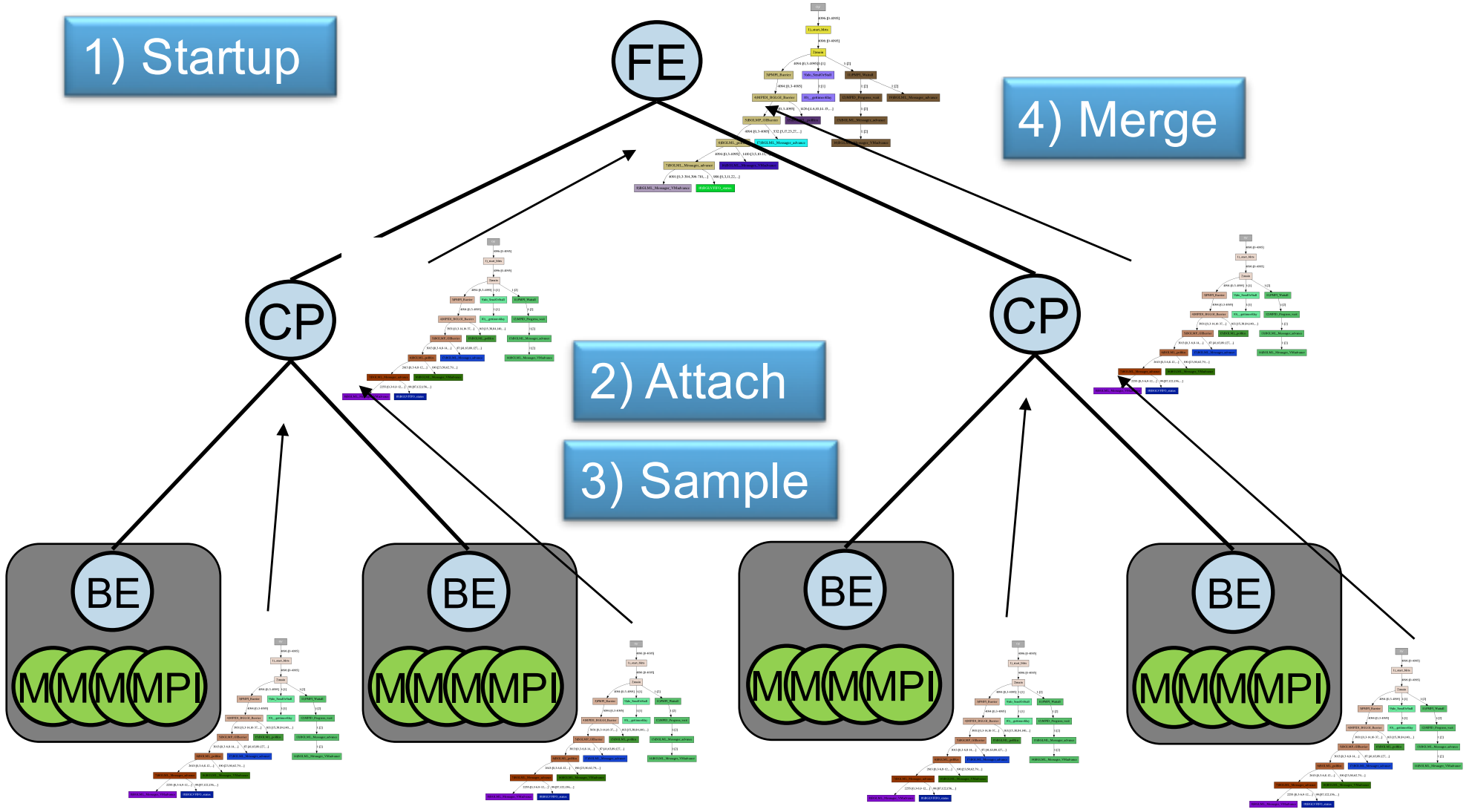
Scalable Communication is Necessary



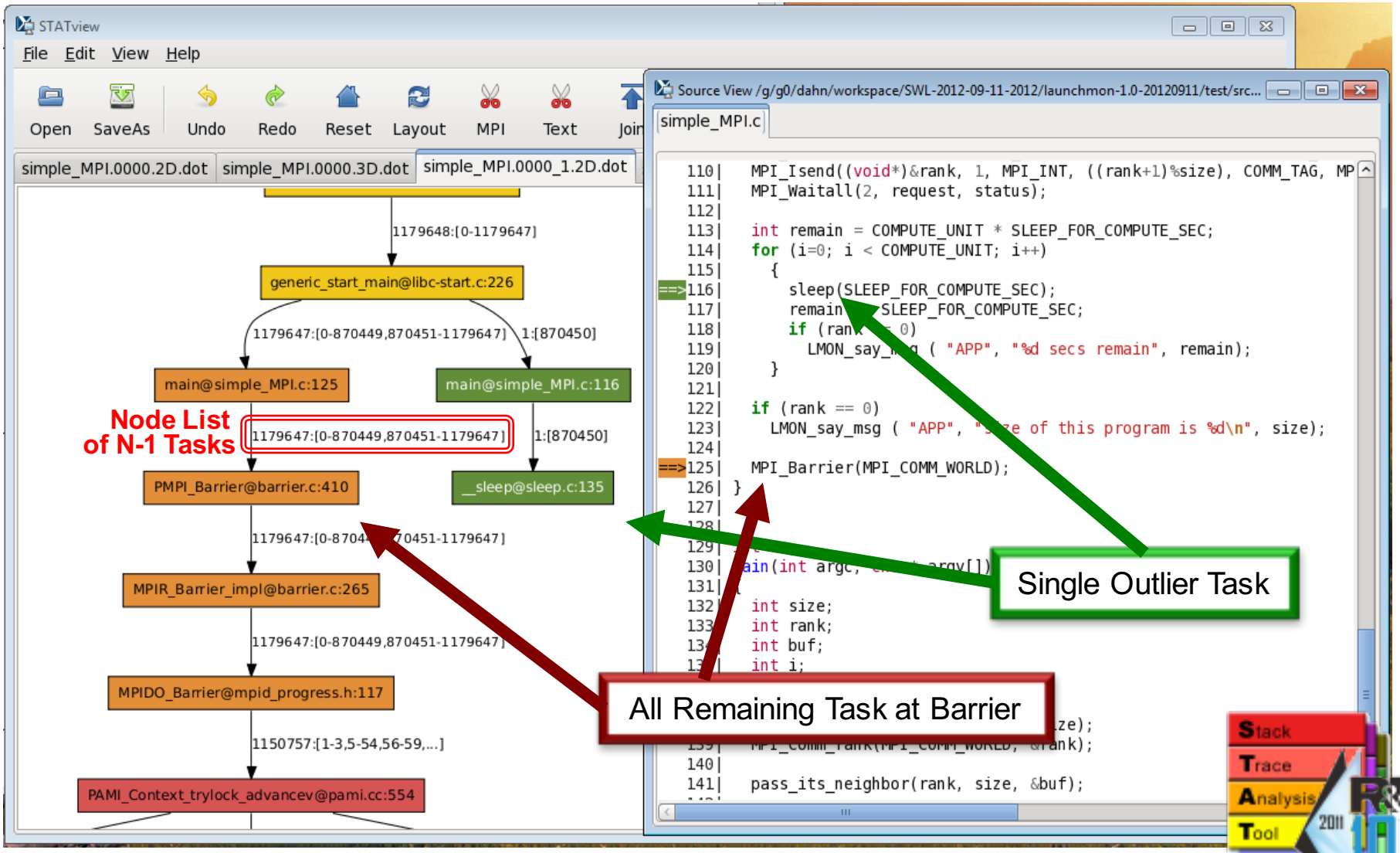
Scalable Communication is Necessary



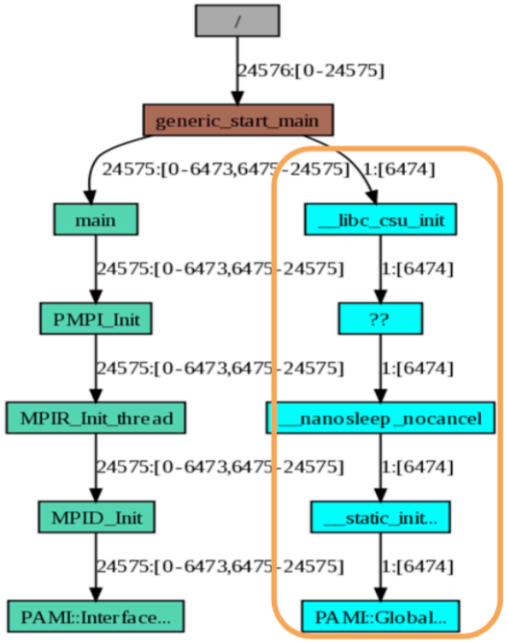
Scalable Aggregation is Necessary



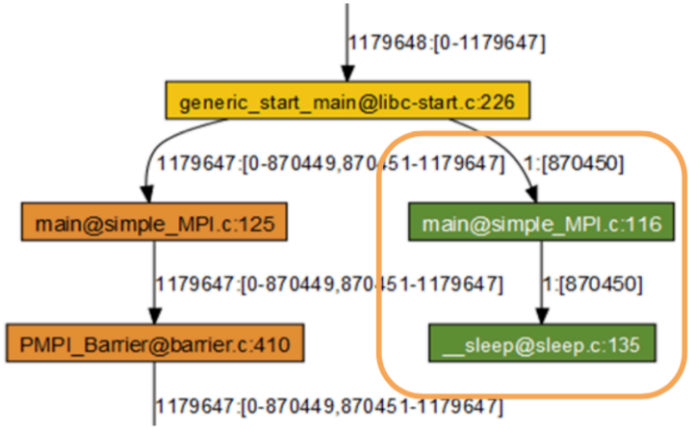
Real Use Case at > 1,000,000 Processes



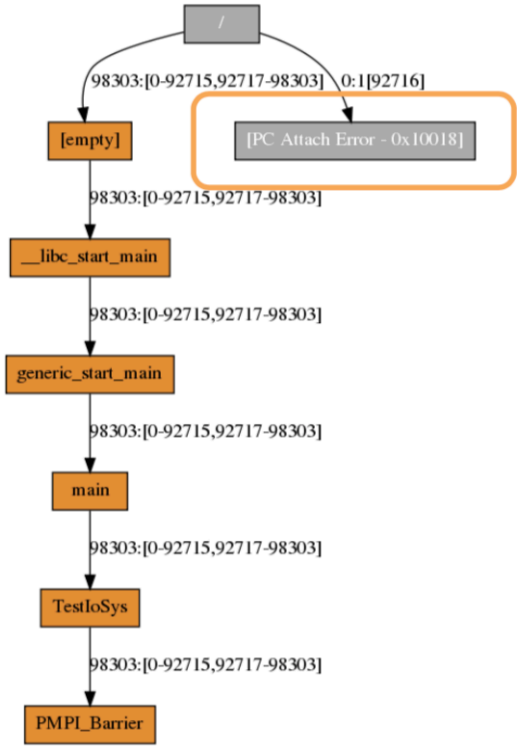
Other Real “War” Stories



A crash during init. due to a bad CN.



A hang due to a decrementor on a core running super slow at *over one million processes.*

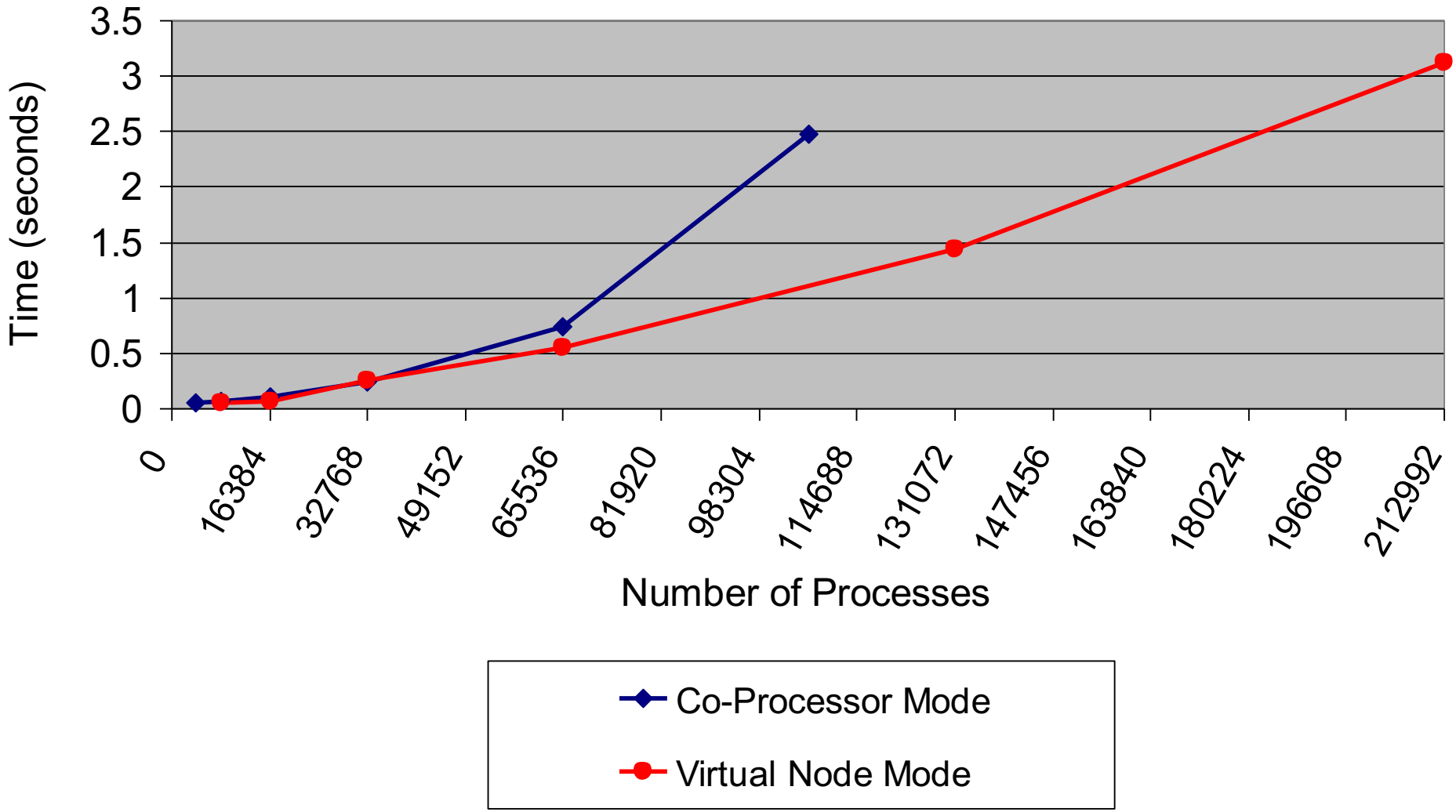


92716: **bgqio199**

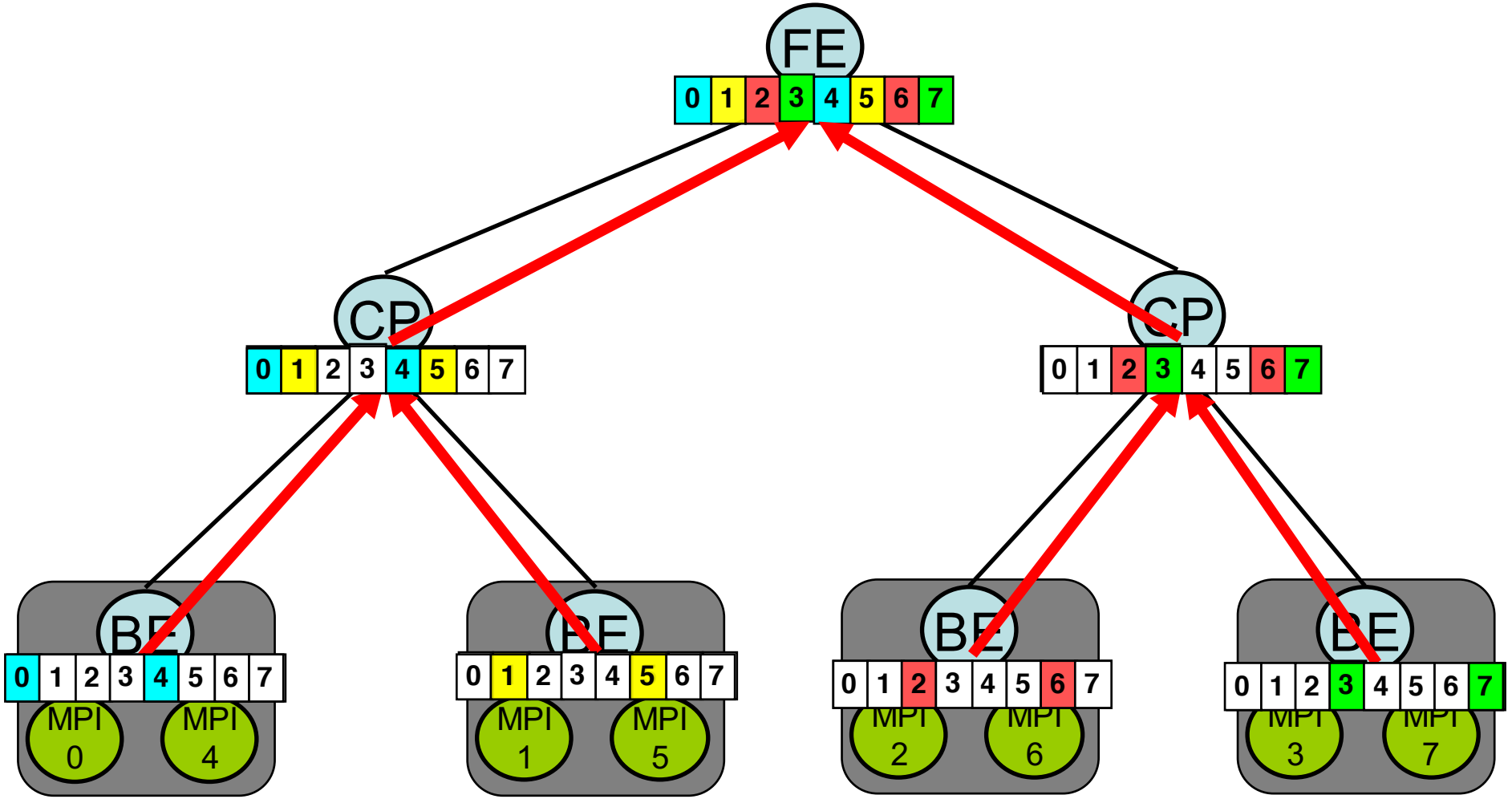
Hang due to a bug in LUSTRE using the full 96 BGQ racks.

From: Dong Ahn, LLNL

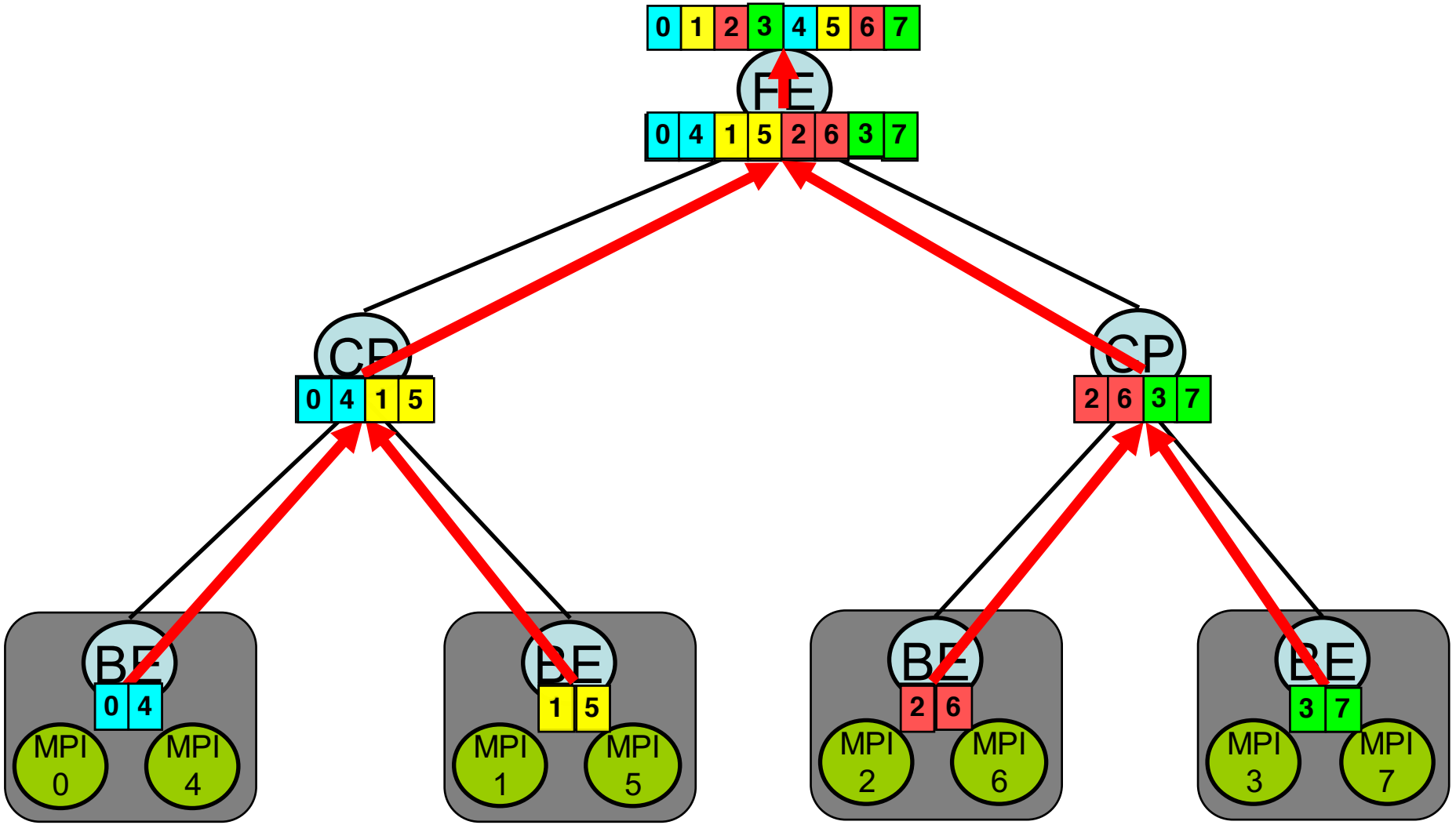
STAT Merge Times on BG/L



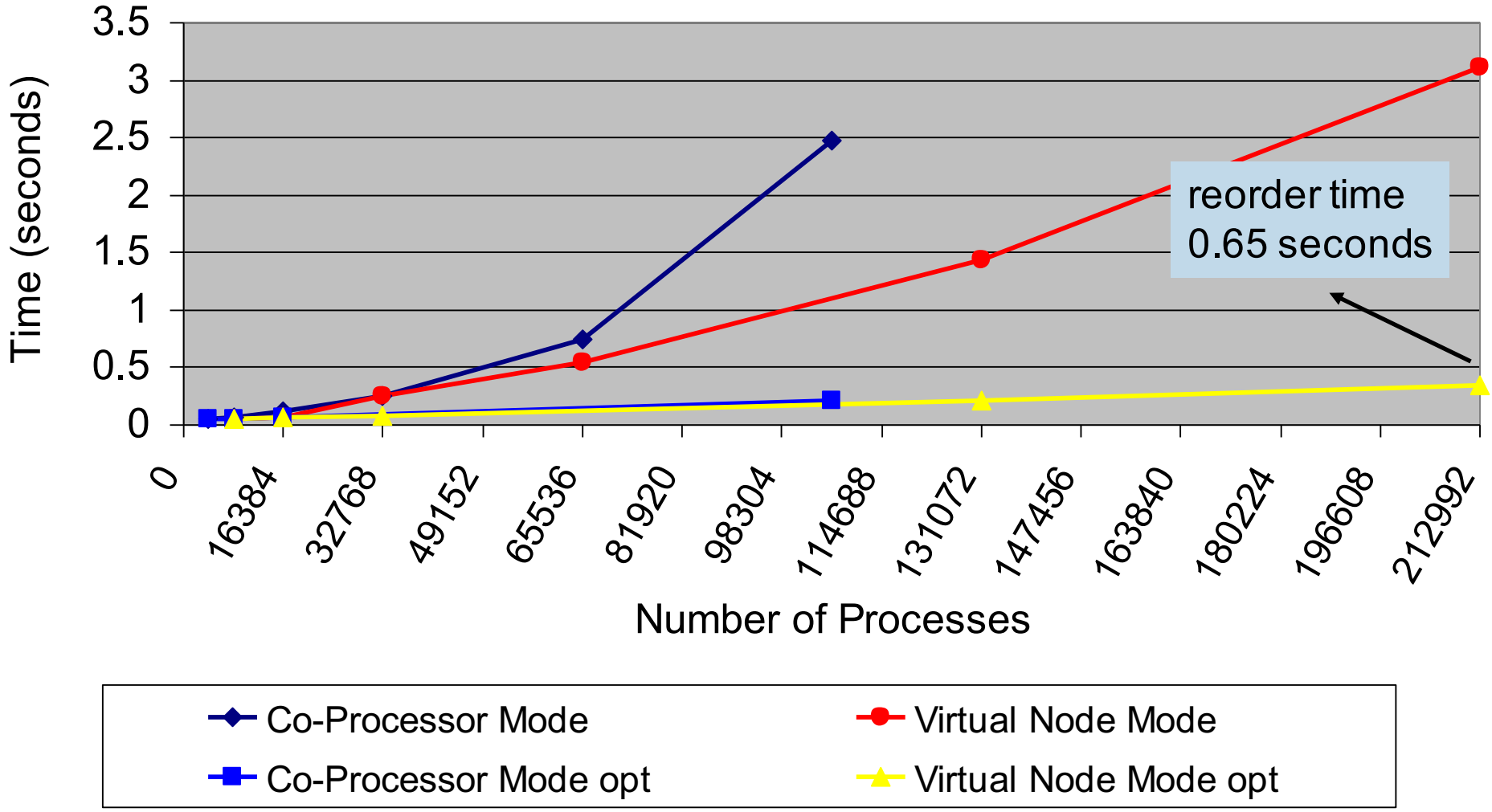
STAT's Original Bit Vector Implementation



STAT's Optimized Bit Vector Implementation



STAT Merge Time with Bit Vector Optimization



Impact on Algorithms and Data Structures

Scaling can put new pressures/requirements on algorithms & data structures

Avoid anything O(N) if at all possible
- Data structures
  - Keep O(N) data distributed, never replicated
- Loops
- Data transfers (`MPI_Alltoall` is not scalable)
- MPI routines with O(N) sized arguments

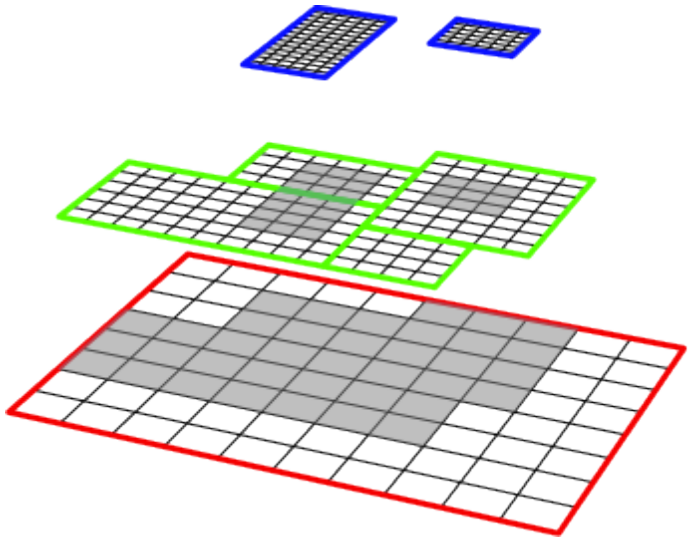
Create smaller subcommunicators
- Avoid using all of MPI_COMM_WORLD
- Localized communication

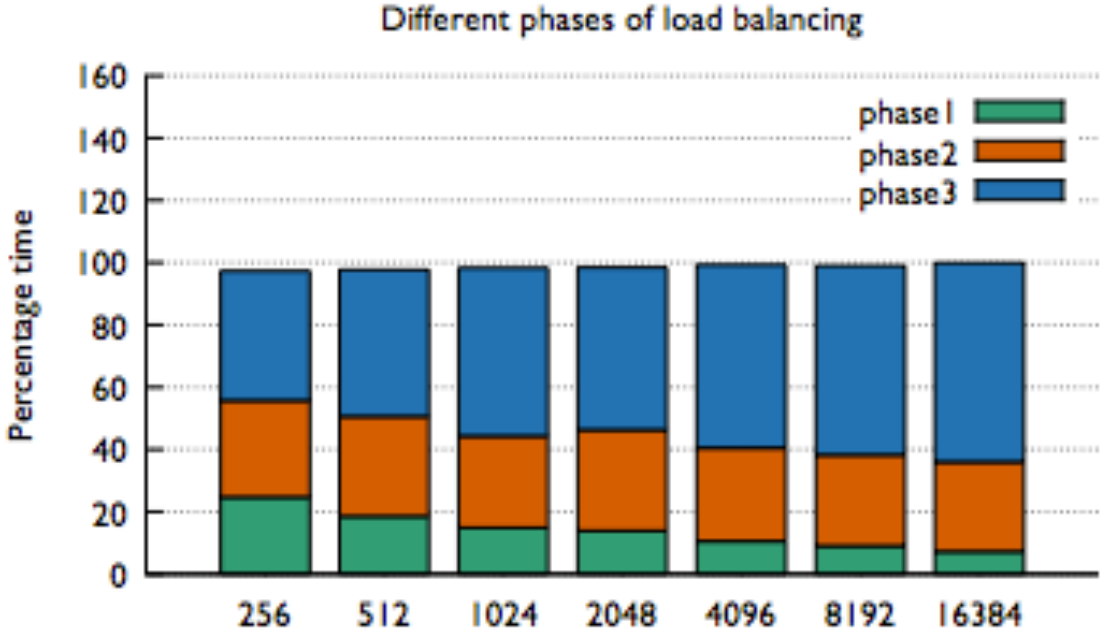
Sometimes new algorithms are needed
- Some algorithms are fine/ideal at small scale, but get worse at large scale
  - Larger setup and higher (constant) overhead amortized at scale
- Dynamic justments/switching may help

Example: Optimizing Load Balancing in AMR

Adaptive Mesh Refinement (SAMRAI library)

- Different levels of patches to refine in areas of interest
- Requires active load balancing
- Load balancing shows bad scaling behavior
- Dominates at large scale

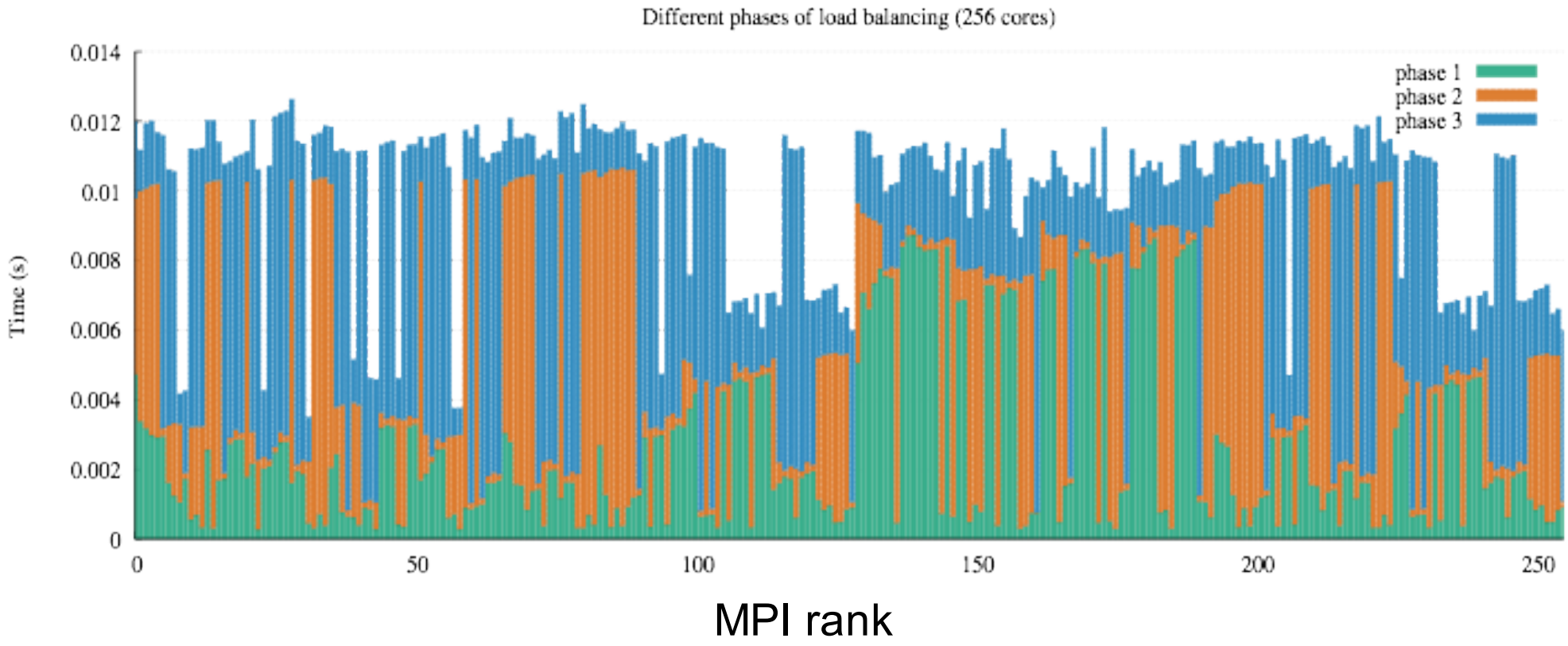




Attempt 2: Timings in MPI rank space

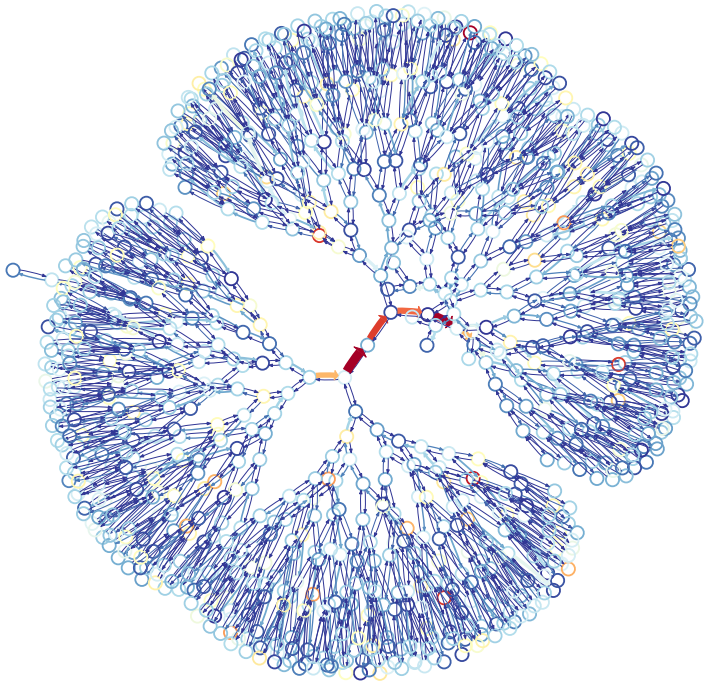
Per node timings for each phase

- Bottleneck is in phase 1 and not phase 3
- Limited correlation based on rank space

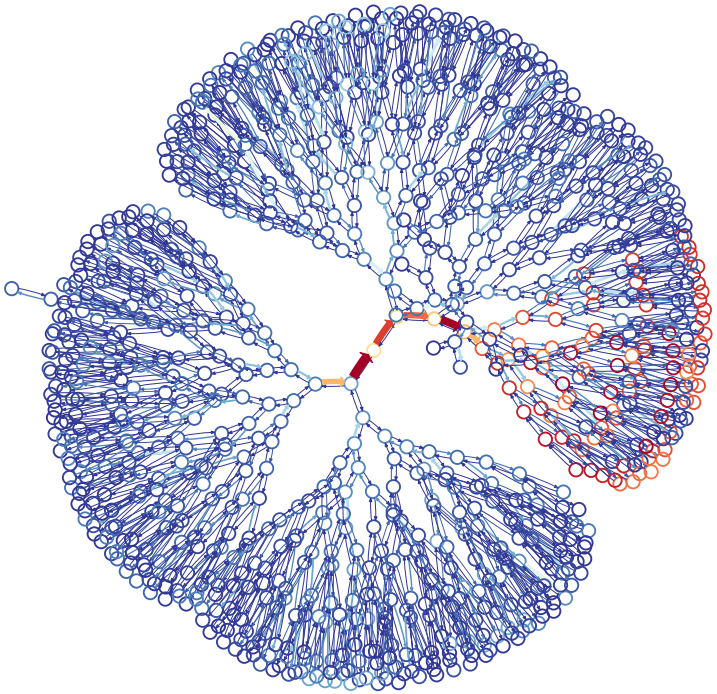


Alternative: Map Performance Metrics onto Underlying Communication Graph (1024 processes)

Load (Cells per process)
Before balancing



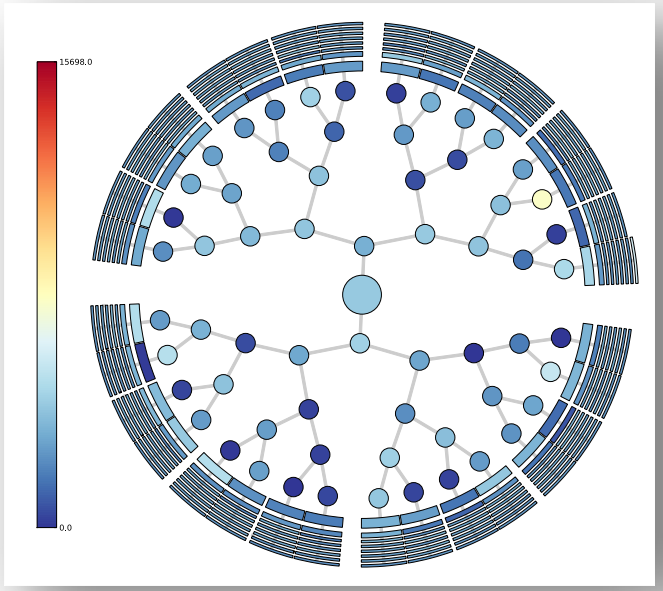
Time spent
redistributing boxes



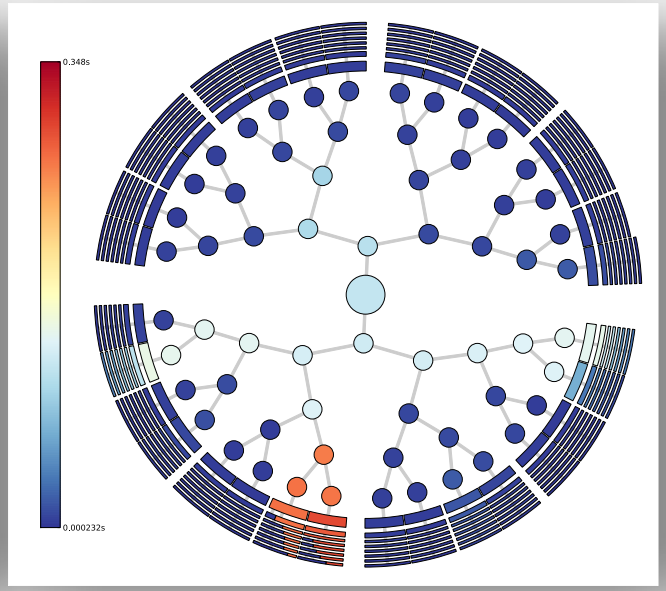
Visualizing Large Communication Graphs

Display of individual nodes is not scalable

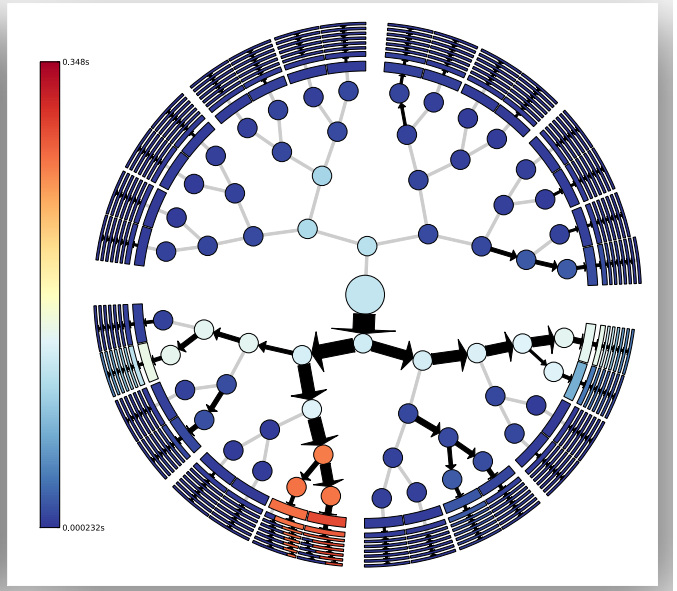
- Need to group nodes
- Outer layers are best targets for this
- Keep metric information / coloring



Load on 16k cores



Wait time for box distribution

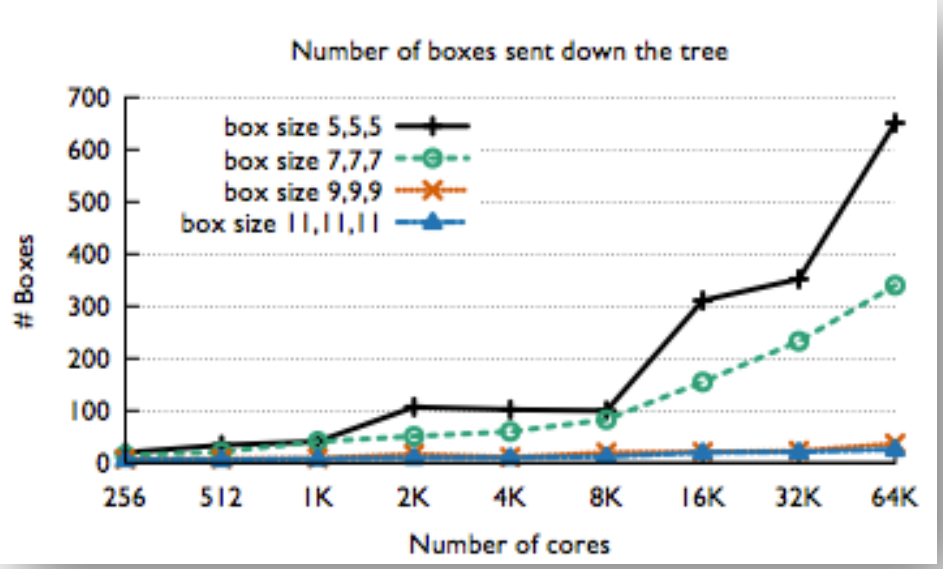


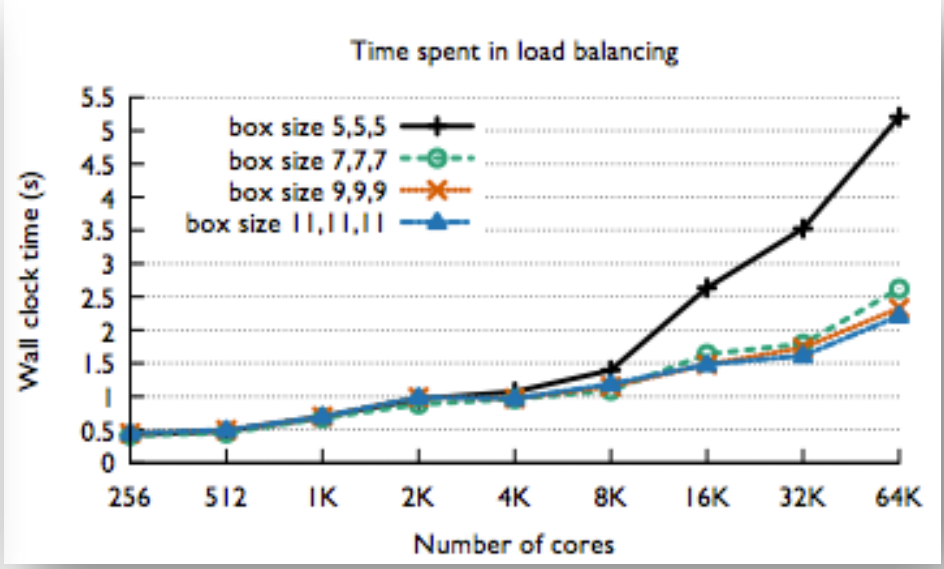
Wait time with flow information

Performance Improvements

Need to address flow problem

- Reduce traffic through root
- Box size / granularity is one of the knobs
- Ultimately need new communication/balancing algorithm



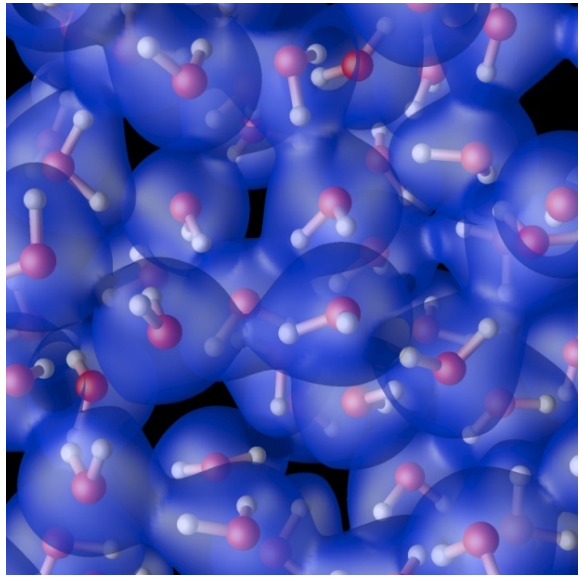


Impact on Mapping Codes to Machines

Example: FPMD Code on Blue Gene/L

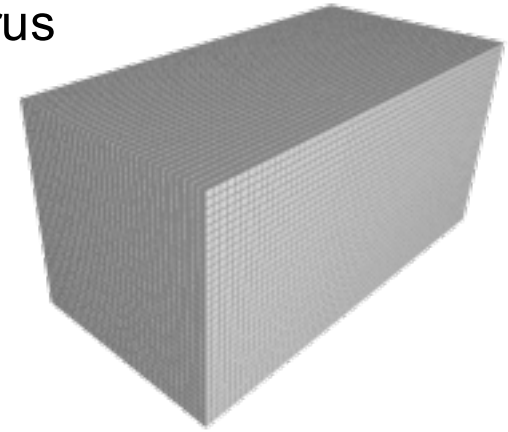
Material simulation using first principles
- No empirical parameters
- Iterative process

Communication Structure
- Dense matrix of wave function coefficients
  - Regular domain decomposition
  - Row/Column communication
- Dominant communication: MPI_Bcast

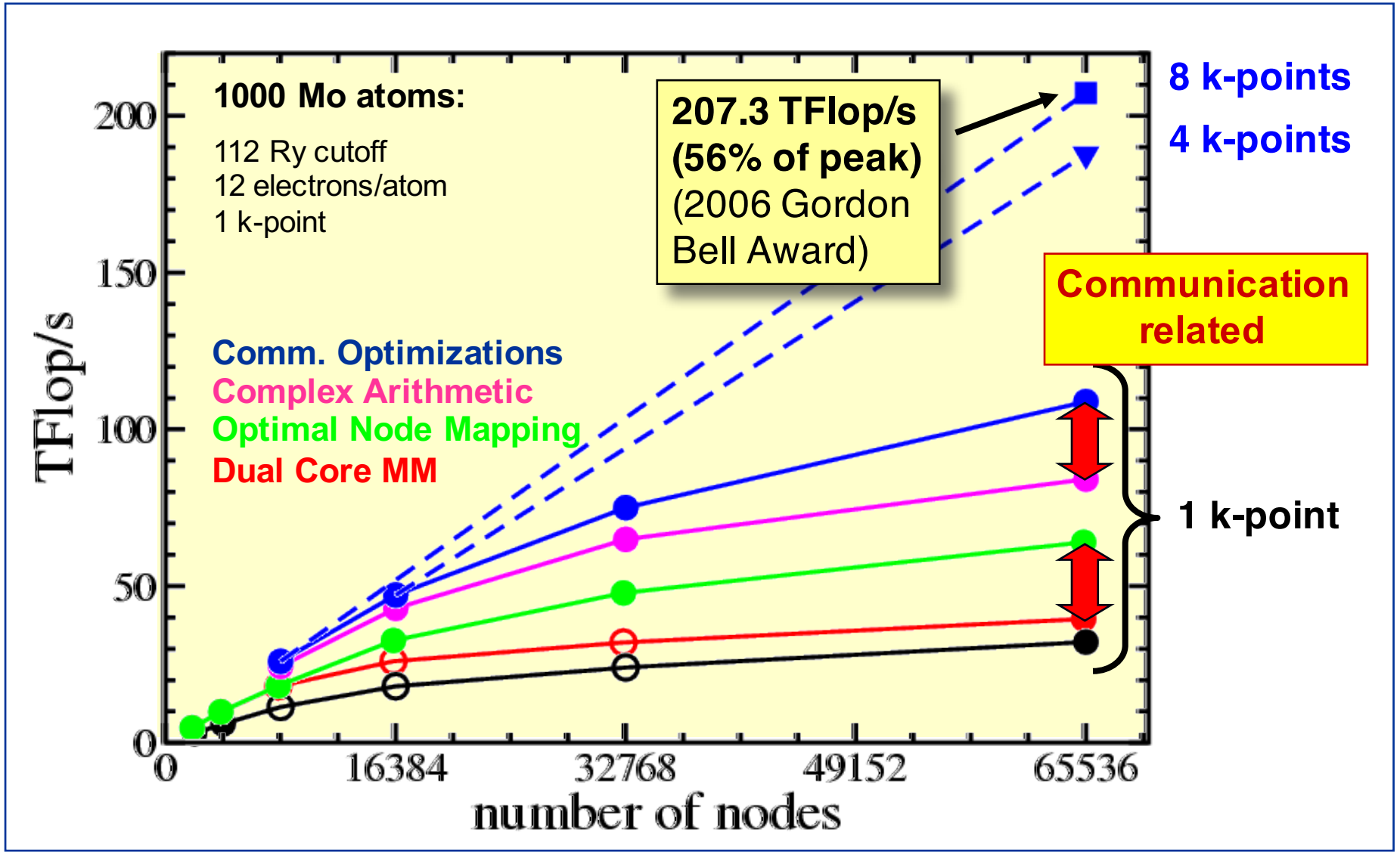


Electron density surrounding water molecules, calculated from first-principles

Mapping matrix decomposition onto BG/L's 3D torus
- 65,536 nodes in a 64x32x32 torus
- Problem split into 512 rows and 128 columns
- Mapping of rows onto target machine

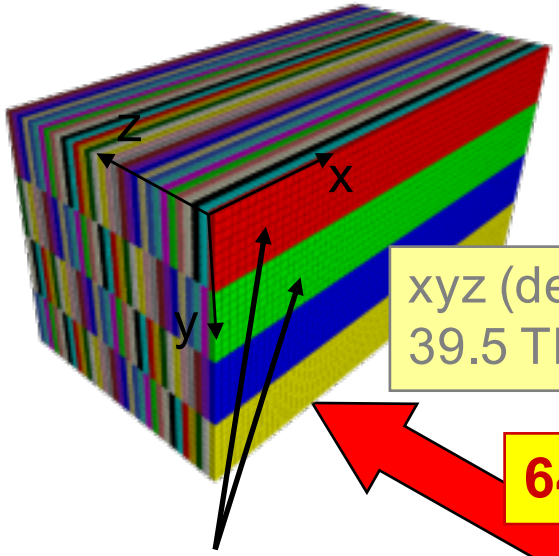


QBox Performance (Gordon Bell Code 2006)



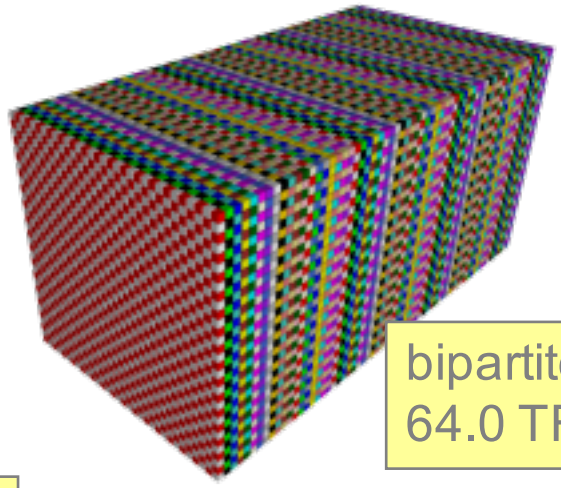
Impact of Node Mappings on Performance

65536 nodes, in a 64x32x32 torus

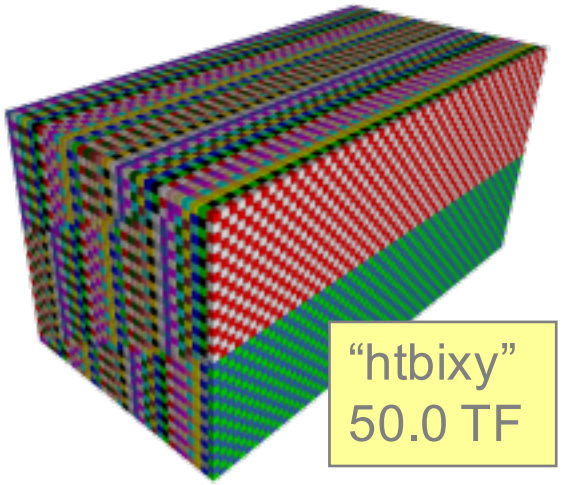


xyz (default)
39.5 TF

512 tasks per MPI subcommunicator

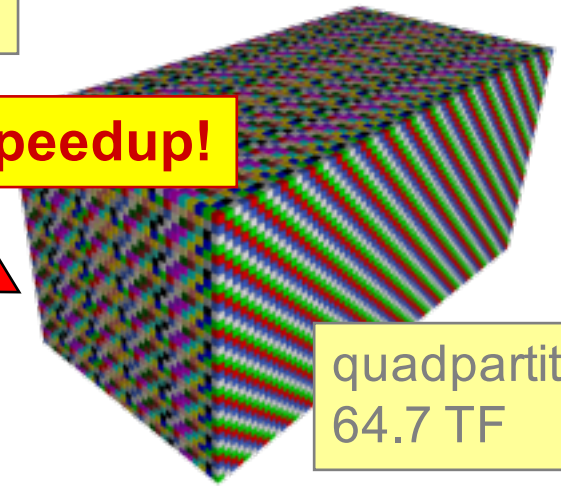


bipartite
64.0 TF

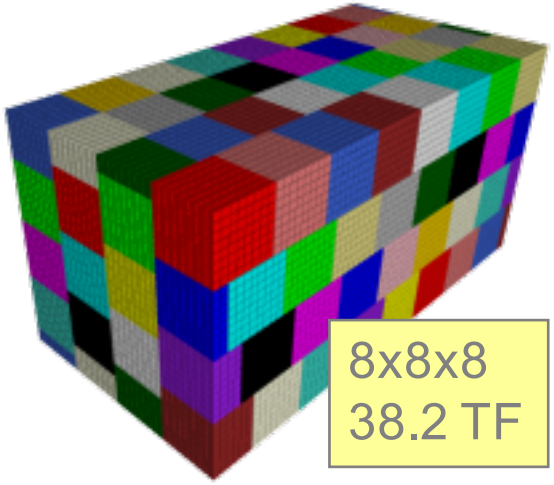


“htbixy”
50.0 TF

64% speedup!



quadpartite
64.7 TF



8x8x8
38.2 TF

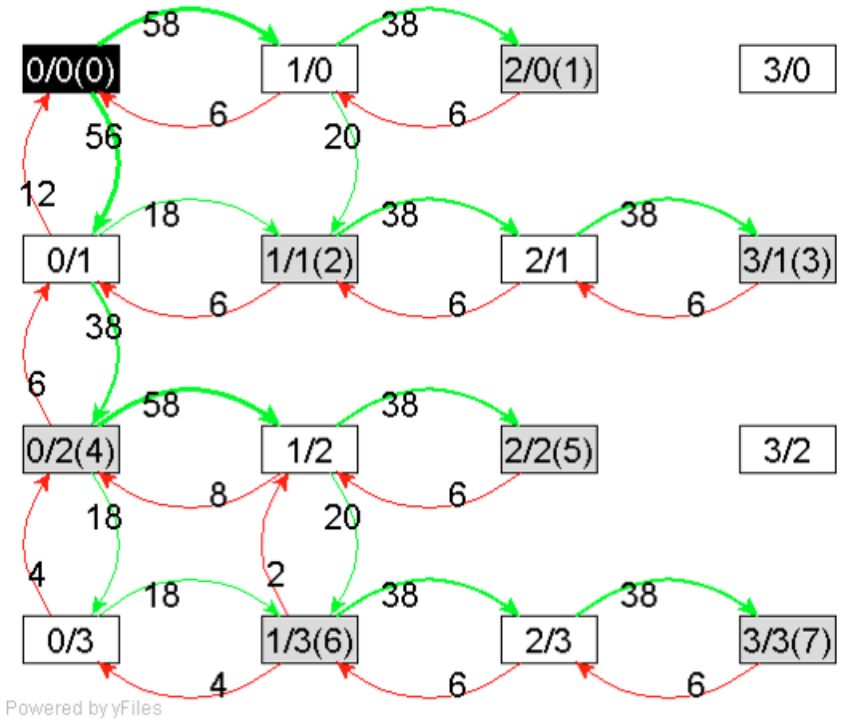
Why Are We Seeing this Performance Behavior?

Observation 1:
- Need to optimize for both row and columns
- Take interference into account

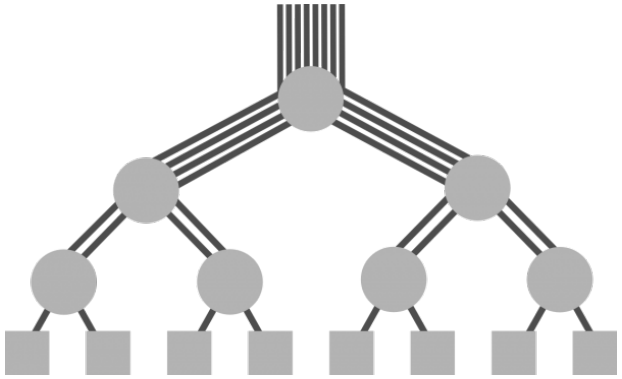
Observation 2:
- Need to optimize for bandwidth
- Drive as many links as possible

Optimization process was manual
- Detecting communication patterns
- Trial and error mappings
- Explain performance post mortem
- Iterative refinement

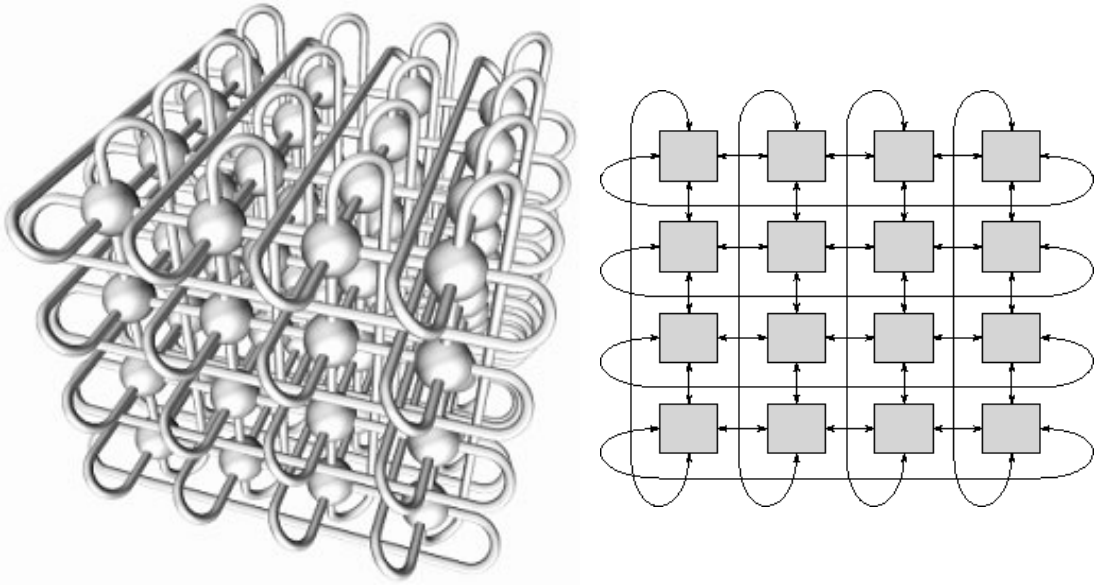
All runs had to be at scale



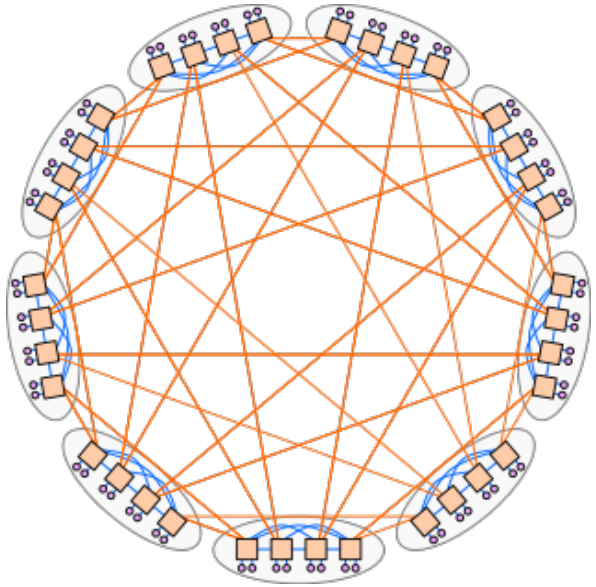
Other Network Topologies



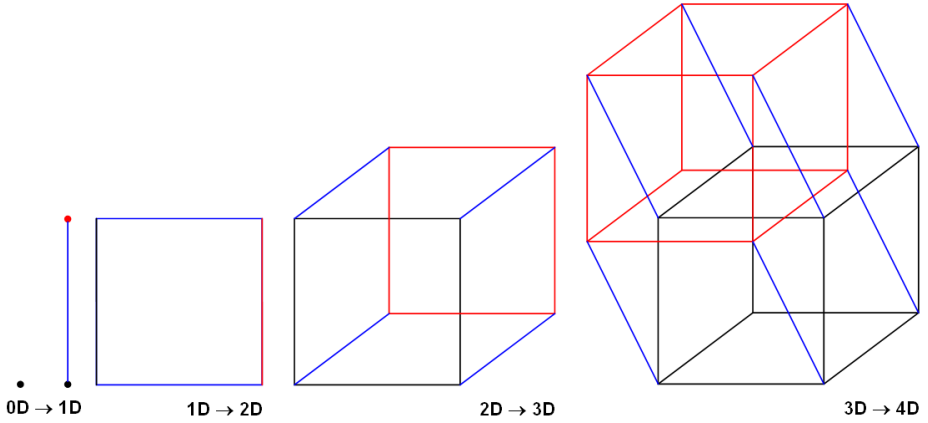
Fat-Tree: most cluster interconnects



Torus: BG/L (3D), BG/Q (4.5D), K (6D), Cray XT3 (3D)



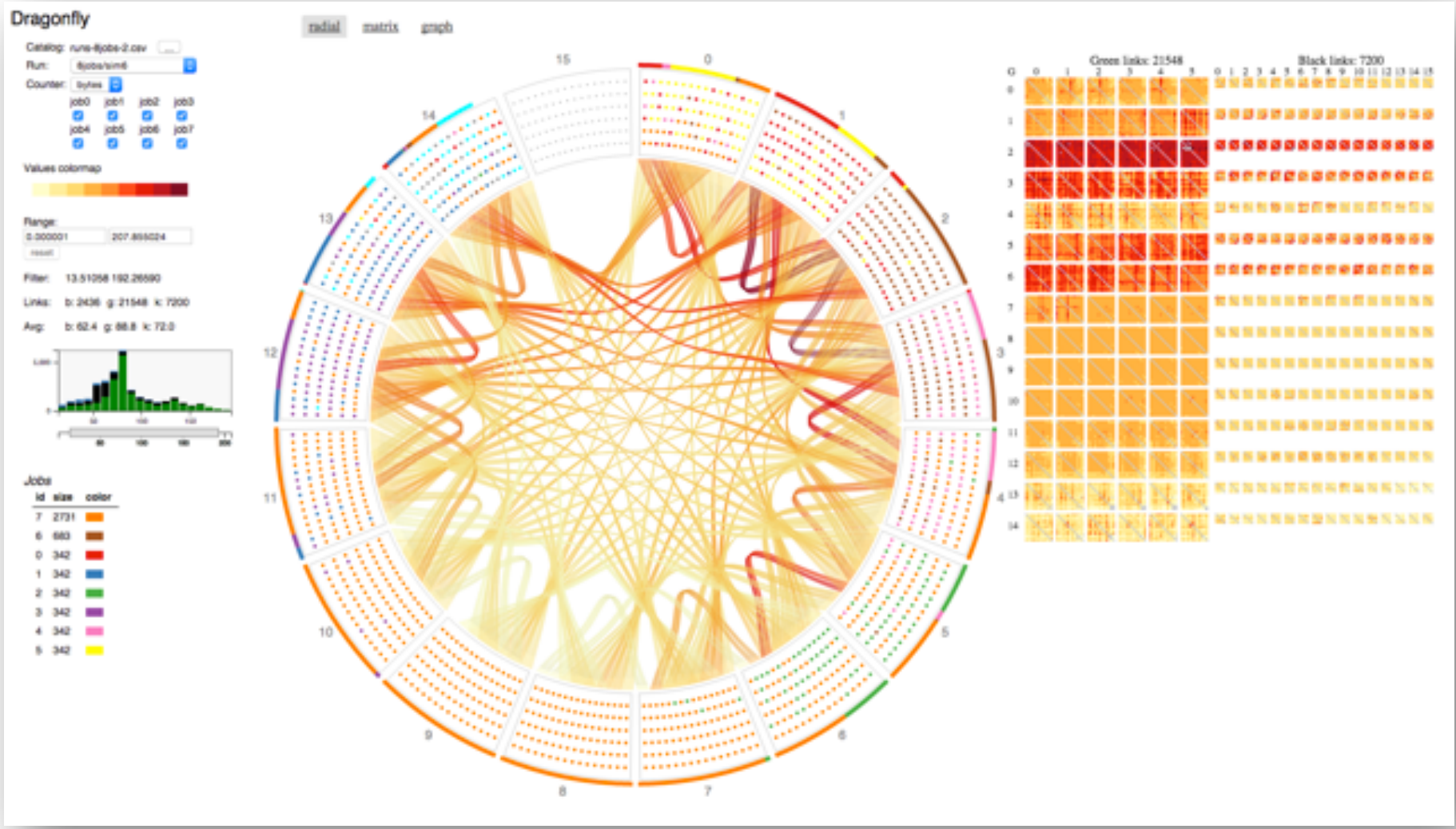
Dragonfly: Cray XE



Hypercube: Intel Paragon, SGI Altix (modified)

Sources: Clusterdesign.org, ANL, Fujitsu, Wikipedia

Dragonfly and its Properties



From: Bhatele and Bremer, LLNL

MPI Process Topologies

Mapping MPI processes to the machine is a hard problem
- Depends on algorithm and system
- Often hard to do for programming manually

Topologies define an intuitive name space
- Simplifies algorithm design
- Provides MPI with topology information
- Enables more efficient mappings within MPI

MPI supports
- multidimensional grids and tori
- arbitrary graphs

Information attached to a communicator
- Special creation functions
- Ability to query topologies later on

Note/Warning: these are often not the most optimized routines, but getting better

Cartesian Grid Topologies

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
                    int dims[], int periods[], int reorder,
                    MPI_Comm *comm_cart)
```

IN comm_old : Starting communicator
IN ndims: number of dimensions
IN dims: array with lengths in each dimension
IN periods: logical array specifying whether the grid is periodic
IN reorder: Reorder ranks allowed?
OUT comm_cart: resulting communicator

Creates a new communicator with Cartesian topology attached

- Still allows communication as usual
- But optimized for Cartesian communication

Cartesian Query Functions

Get a rank from a tuple of coordinates

```
int MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank)
```

Get a tuple of coordinates from a rank

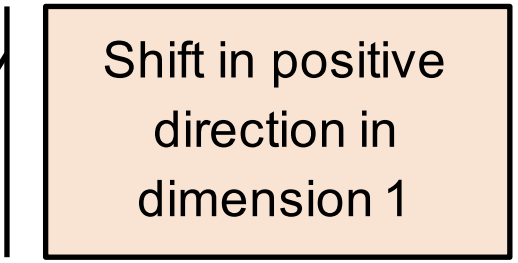
```
int MPI_Cart_coords(MPI_Comm comm, int rank,
      int maxdims, int coords[])
```

Get a send/receive pair for use with MPI_Sendrecv

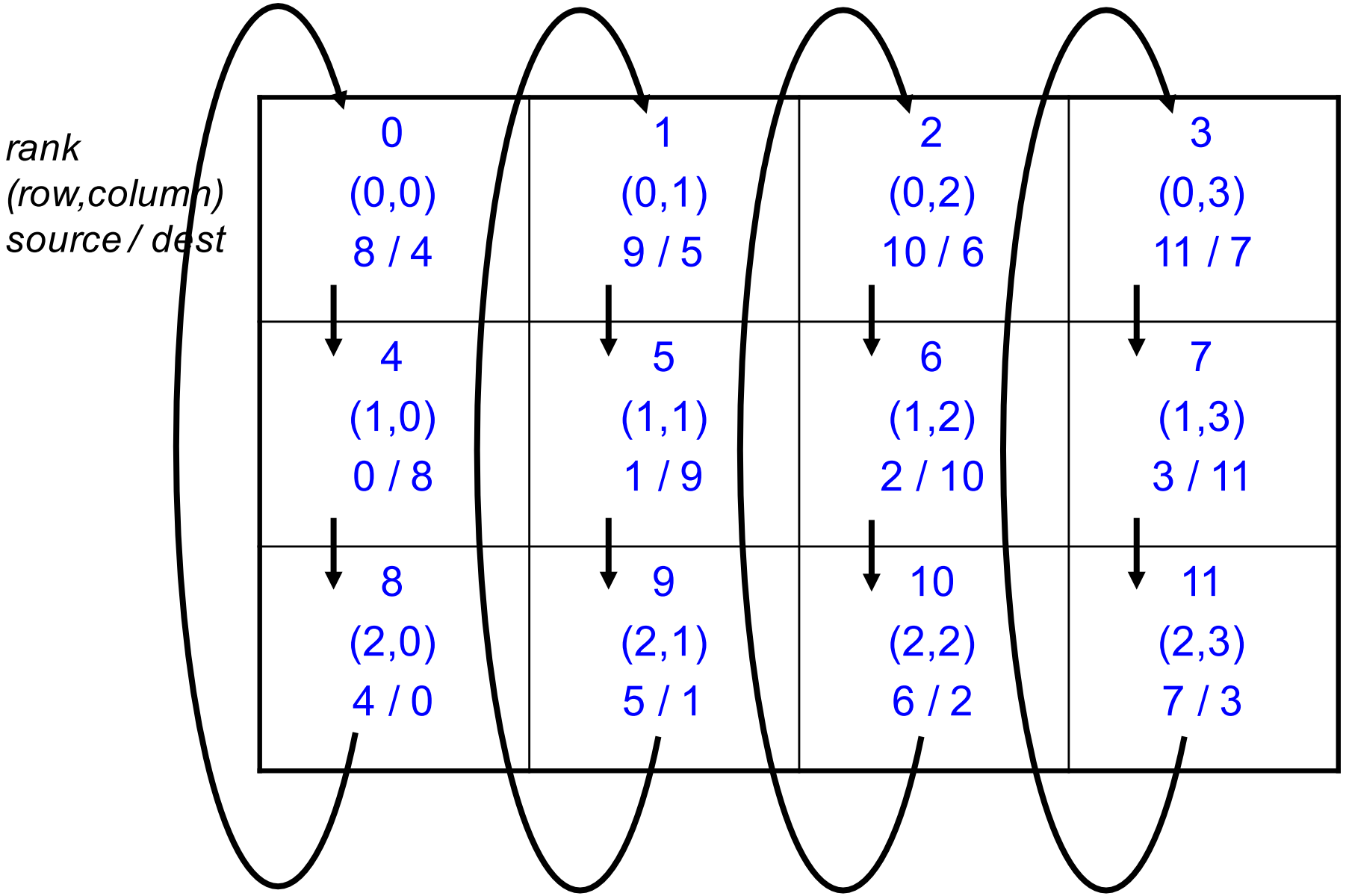
```
int MPI_Cart_shift(MPI_Comm comm, int direction,
      int disp, int *rank_source, int *rank_dest)
```

Example: Cartesian Topologies

```
double buf1, buf2;
MPI_Comm comm_2d;
MPI_Status status;

dims[0]=3; dims[1]=4;
periods[0]=true; periods[1]=true;    /* Torus, not Grid */
reorder=false;

MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder
                &comm_2d);
MPI_Cart_coords(comm_2d, rank, 2, &coords);
MPI_Cart_shift(comm_2d, 1, 1, &source, &dest);

MPI_Sendrecv(buf1, 1, MPI_DOUBLE, dest,   42,
             buf2, 1, MPI_DOUBLE, source, 42, comm_2d, &status);
```



Example: Cartesian Topologies



MPI Communicator From Graphs

Ability to specify localized communication using arbitrary graphs

```
int MPI_Graph_create(MPI_Comm comm_old,
                     int nnodes, int index[], int edges[],
                     int reorder, MPI_Comm *comm_graph)
```

IN: comm_old	Starting communicator
IN: nnodes	Number of Graph nodes
IN: index	Indicies for starting edges in edge list
IN: edges	Edges of graph (destination nodes)
IN: reorder	Reorder ranks allowed?
OUT: comm_dist_graph	Resulting communicator

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

$$\begin{aligned} \text{nnodes} &= 4 \\ \text{index} &= 2, 3, 4, 6 \\ \text{edges} &= 1, 3, 0, 3, 0, 2 \end{aligned}$$

MPI Communicator From Distributed Graphs

Ability to specify localized communication using arbitrary ***distributed*** graphs

```
int MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
      int indegree,  int sources[],       int sourceweights[],
      int outdegree, int destinations[], int destweights[],
      MPI_Info info, int reorder, MPI_Comm *comm_dist_graph)
```

IN: comm_old	Starting communicator
IN: indegree, sources, sourceweights	List of incoming edges
IN: outdegree, destinations, destweights	List of outgoing edges
IN: info	Optimization hints
IN: reorder	Reorder ranks allowed?
OUT: comm_dist_graph	resulting communicator

Note: graphs across all processes must be consistent

process	neighbors
0	1, 3
1	0
2	3
3	0, 2

process	indegree	sources	sourceweights	outdegree	destinations	destweights
0	2	1,3	1,1	2	1,3	1,1
1	1	0	1	1	0	1
2	1	3	1	1	3	1
3	2	0,2	1,1	2	0,2	1,1

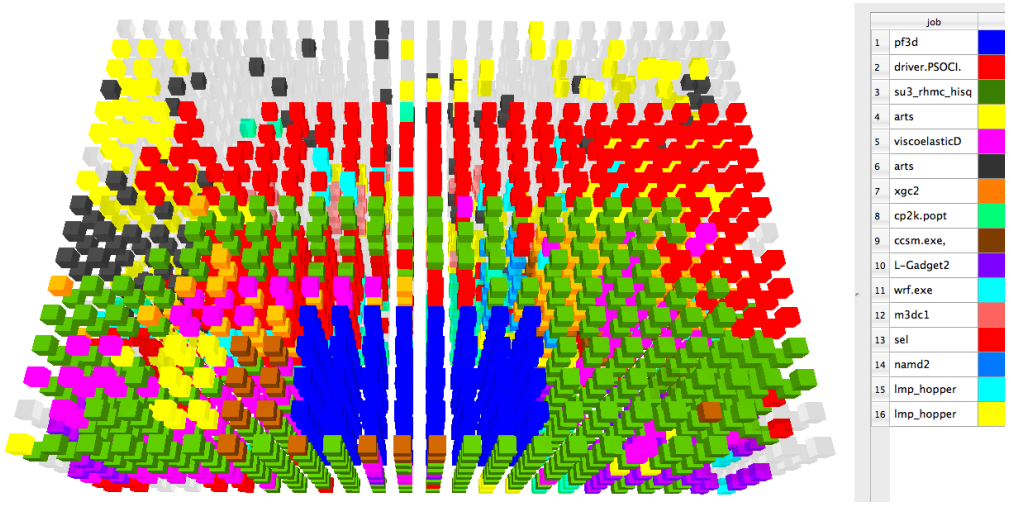
MPI Neighborhood Collectives

Enables communication localized to topology neighbors
- Cartesian communicators: shift by one in all directions
- Graphs: only locally attached edges

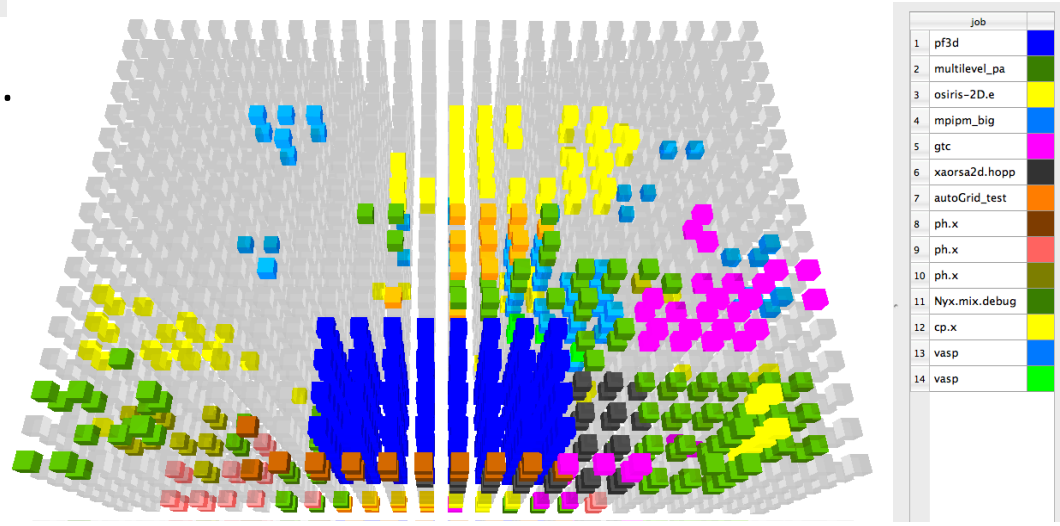
Available functions
- `MPI_Neighbor_allgather`
- `MPI_Neighbor_allgatherv`
- `MPI_Neighbor_alltoall`
- `MPI_Neighbor_alltoallv`
- `MPI_Neighbor_alltoallw`

Note:
- Functions are collective wrt. entire communicator
- Functions only communicate locally

Additional Issue: Variability due to Contention



Slow run of pf3d on Cray XE6 system.



25% faster messaging rate without congestion.

Consequence of Variability

Performance analysis is becoming statistics
- Single runs are no longer sufficient to understand performance
- Need to combine data from several runs
- Need to understand variability in the system and the results
- Record and document as much metadata as possible (static and dynamic)

Reading:
- Scientific Benchmarking of Parallel Computing Systems
- Torsten Hoefler and Roberto Belli, SC15

Some lessons
- Avoid summarizing ratios; summarize the costs or rates the ratios are based on
- Report if the measurement values are deterministic
  - For nondeterministic data, report confidence intervals of the measurement.
- Do not assume normality of collected data without diagnostic checking.
- If possible, show upper performance bounds to facilitate interpretability

Impact on I/O

I/O can take a majority of the execution share at scale

- Reading of configurations files
- Reading of input decks
- Visualization dumps
- Checkpointing

Options

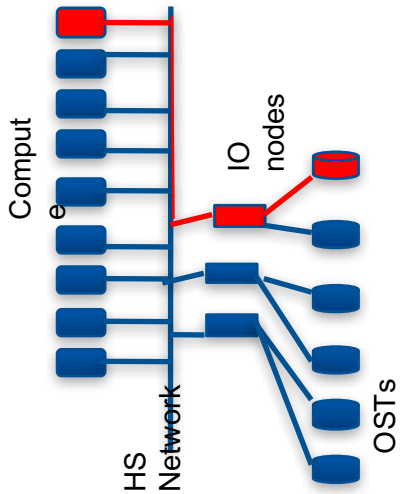
- Posix I/O directly to the file system
- MPI I/O routines
- Specialized libraries with special data formats like HDF5

Things to look for

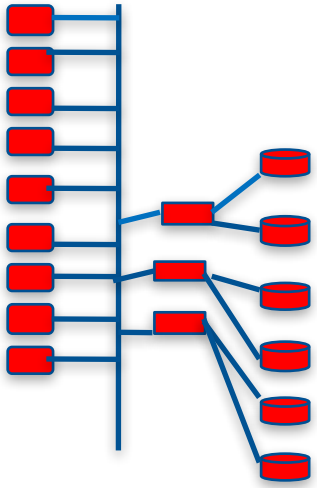
- Using the right file system
- Exploiting parallelism
  - Writing from multiple processes to one file
  - Writing from multiple processes to many files
- Metadata management performance

Exploiting Parallelism in I/O

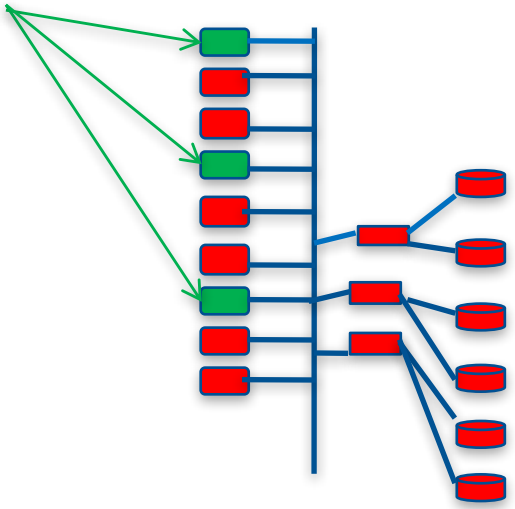
1 PE writes; BW limited



1 file per process; BW enhanced



Subset of PEs do I/O; Could be most optimal



Best scenario depends on

- Data size
- Internal data distribution
- Type of parallel filesystem
- System installation (sizing of metadata servers)

Requires benchmarking and experience, but be aware of variability (esp. for I/O)

Summary

HPC Systems are increasingly concurrent, which we need to exploit
- Weak scaling: constant problem size per HW thread/core/node
- Strong scaling: constant total problem size

Impact on Programmability
- Work at scale is getting harder -> try to keep small reproducers
- Tool chain (e.g., debuggers) need to be adjusted as we scale

Impact on Algorithms and Data Structures
- Avoiding any algorithm or data structure that is O(N) or worse
- May have to rethink algorithms and/or data structures to be used

Impact on Mapping Code to Machines
- Mapping of applications to the topology of the machine can be crucial at scale
- MPI topologies (grids or graphs) can help, as they enable MPI to optimize

Impact on I/O
- I/O can be a large fraction of the execution time
- Need to exploit parallelism and tweak parameters (potentially on every system)