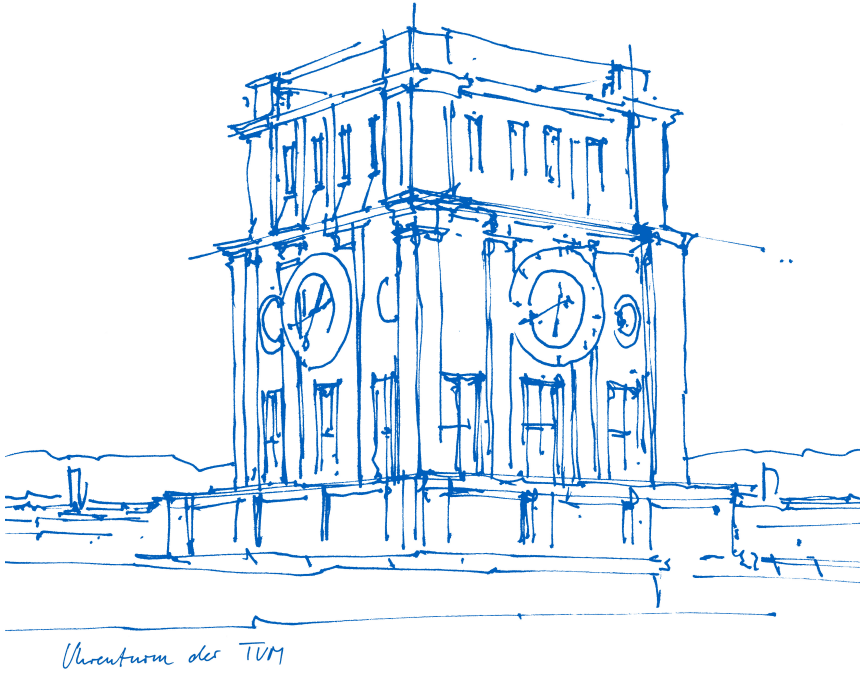
Lecture IN-2147
Parallel Programming

SoSe 2018

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

Lecture 6:
Remaining OpenMP Issues
Distributed Memory and HPC



Summary: Aspects of "Good" Shared Memory Programming

Writing parallel code
- Aim at parallelizing the outer loop first
- Understand and resolve data dependencies
- Code transformations can help in removing loop carried dependencies
- Code transformations can also impact performance (good and bad!)

Correctness is mostly up to the programmer
- Little support from languages, runtimes or compilers
- Avoid race behaviors
- Tools can help in finding races

Performance is often tied directly to the underlying architecture
- Reduce or avoid synchronization
- Pay attention to the memory hierarchy
- Find good thread/data mappings in NUMA systems

Last time: SIMD programming
- Relationship: Architecture vs. ISA vs. Intrinsics vs. „High-level" OpenMP

Remaining OpenMP Issues

OpenMP Tasking

- Basics here
- More on tasking in the last lecture

OpenMP Device Constructs

- Skipping for now
- Will be covered as part of accelerator programming

Memory Models and Memory Consistency

Drawbacks of Work Sharing

Main concept for parallelism discussed so far: Work Sharing

- Distribute work in for loops to threads
- In many cases: static distribution (needed for NUMA)
- In many cases: equal work distributed

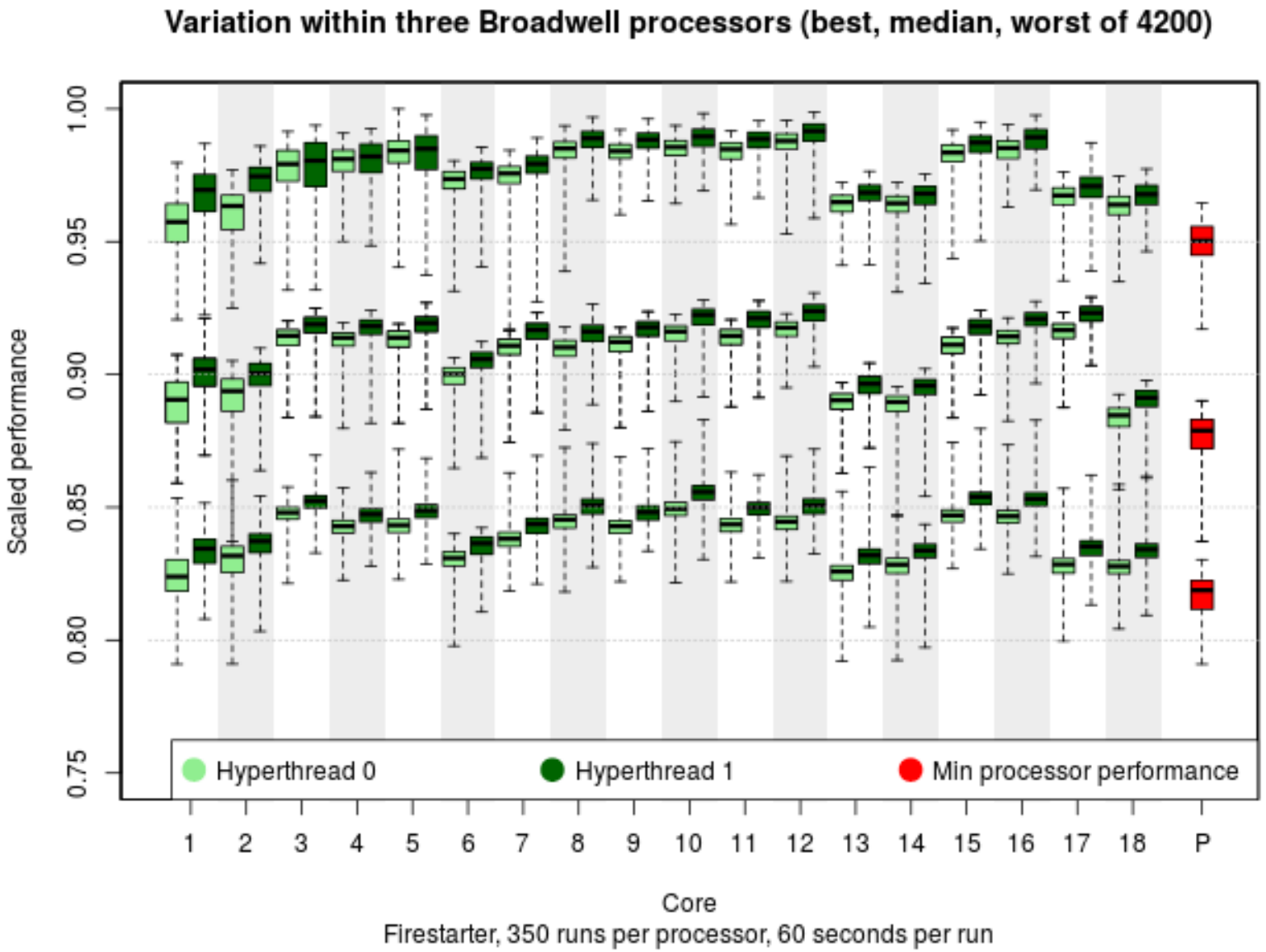
Drawback 1: possible imbalance caused by workload

- Static distribution of equal chunks cannot tolerate load imbalance
- Dynamic distribution can help, but
  - Problems with NUMA first touch
  - Overhead due to fine grained scheduling

Drawback 2: possible imbalance caused by machine

- Machines are no longer homogeneous
- Differences between nodes, threads, runs, …

Processor Variability



Drawbacks of Work Sharing

Main concept for parallelism discussed so far: Work Sharing
- Distribute work in for loops to threads
- In many cases: static distribution (needed for NUMA)
- In many cases: equal work distributed

Drawback 1: possible imbalance caused by workload
- Static distribution of equal chunks cannot tolerate load imbalance
- Dynamic distribution can help, but
  - Problems with NUMA first touch
  - Overhead due to fine grained scheduling

Drawback 2: possible imbalance caused by machine
- Machines are no longer homogeneous
- Differences between nodes, threads, runs, …

Drawback 3: limited programming flexibility
- Limited (in OpenMP) to for loops or coarse grained sections
- Hierarchical parallelism not easy to follow and optimize

Explicit Tasking (since OpenMP 3.0)

Main concept: users create tasks
- Independent pieces of work
- Guaranteed to be executable in any order

OpenMP runtime system schedules the tasks as needed
- Typically maintains one or more task queues
- Tasks distributed to threads
  - Dispatched to open thread as they became available
  - Thread becomes free when task finishes

Tied vs. Untied tasks
- Tied: once a task starts it will remain on the hardware threads
  - Default behavior
  - Easy to reason about
- Untied: tasks can move to different hardware threads
  - Execution can be interrupted and the task moved
  - Advantage: more flexibility and better resource utilization

The OpenMP Task Construct

Explicit creation of tasks

```
#pragma omp parallel
{
  #pragma omp single {
  for ( elem = l->first; elem; elem = elem->next)
     #pragma omp task
          process(elem)
  }
// all tasks are complete by this point
}
```

Task scheduling
- Tasks can be executed by any thread in the team

Barrier
- All tasks created in the parallel region have to be finished.

Tasking Syntax in OpenMP

```
#pragma omp task [clause list]
{ ... }
```

Select clauses

`if (scalar-expression)`

- FALSE: Execution starts immediately by the creating thread
- The suspended task may not be resumed until the new task is finished.

untied

- Task is not tied to the thread starting its execution.
- It might be rescheduled to another thread.

`Default (shared|none), private, firstprivate, shared`

- Default is firstprivate.

priority(value)

- Hint to influence order of execution
- Must not be used to rely on task ordering

Example: Tree Traversal

```
struct node {
  struct node *left;
  struct node *right;
};

void traverse( struct node *p ) {
  if (p->left)
    #pragma omp task // p is firstprivate by default
    traverse(p->left);
  if (p->right)
    #pragma omp task // p is firstprivate by default
    traverse(p->right);
  process(p);
}

#pragma omp parallel
#pragma omp single
   traverse(root);
```

Task Wait and Task Yield

```
#pragma omp taskwait
{ ... }
```

Waits for completion of immediate child tasks

- Child tasks: Tasks generated since the beginning of the current task

```
#pragma omp taskyield
{ ... }
```

The taskyield construct specifies that the current task can be suspended

- Explicit task scheduling point

Implicit task scheduling points

- Task creation
- End of a task
- Taskwait
- Barrier synchronization

Task Dependencies (since OpenMP 4.0)

Defines in/out dependencies between tasks

- **`Out`**: variables produced by this task
- **`In`**: variables consumed by this task
- **`Inout`**: variables is both in and out
- Influences scheduling order

Implemented as clause for task construct

```
#pragma omp task depend(dependency-type: list)
{ ... }
```

Example:

```
#pragma omp task shared(x, ...) depend(out: x) // T1
      preprocess_some_data(...);
#pragma omp task shared(x, ...) depend(in: x)  // T2
      do_something_with_data(...);
#pragma omp task shared(x, ...) depend(in: x)  // T3
      do_something_independent_with_data(...);
```

Performance Considerations

Advantages
- Implicit load balancing
- Simple programming model
- Many complications and bookkeeping pushed to runtime

Consideration 1: Task granularity
- Fine grained allow for more resource utilization
  - But also cause more overhead
- Coarse grained tasks reduce overhead
  - But also cause schedule fragmentation
- Choosing the right granularity is up to the programmer!

Consideration 2: NUMA optimization
- Each tasks can run anywhere
  - Programmer cannot influence this
- Modern runtimes provide optimizations
  - NUMA aware scheduling

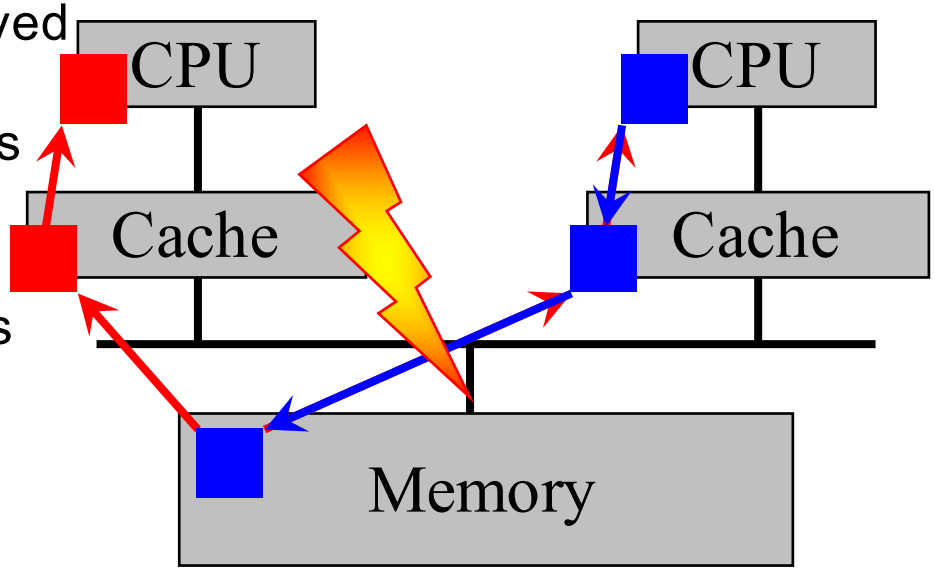
Memory Models

Hardware shared memory means concurrent accesses to shared state
- Multiple hardware threads work independently
- Question: which updates are seen when by which thread?

Two fundamental concepts
- Memory/Cache Coherency: reasoning about updates to one memory location
- Memory Consistency: reasoning about updates to several memory locations

A system is coherent if
- Program order for loads/stores are preserved
- All stores eventually become visible
- All processors see the same order of writes

Establishing coherency
- One cores: implemented within processors
- Cross cores: two main protocol families for cache coherency
  - Snoop-based protocols
  - Directory-based protocols



Memory Consistency

What can we expect when we read from and write to memory

- What is the relative ordering?
- When will updates be available?

Think of memory consistency as a contract between the programmer and the system

Most intuitive for the programmer: **Sequential Consistency**

> *[Lamport] "A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program"*

Reference: Adve, S. V. and K. Gharachorloo, " Shared memory consistency models: A tutorial," IEEE Computer, December 1996.

Consequences of Sequential Consistency

SC is natural from a programmer's perspective

Why is the program order requirement hard?
- Network can reorder
- Use of write-buffers
- Compiler optimizations

Why is the atomicity requirement hard?
- A write has to take place instantly with respect to all processors
- Can lead to a system-wide serialization of writes

We need to relax consistency requirements
- Necessary to achieve well performing systems
  - Increase the level of concurrency
- Compensate with synchronization primitives
  - Need them anyway for proper synchronization of contents
  - Can implicitly execute consistency operations

Create the "illusion" of sequentially consistency

Relaxed Consistency Models

Processor Consistency (should probably be HW Threads Consistency by now)

- Writes by any thread are seen by all threads in the order they were issued
- For any variable, all threads see writes in the same order
- But: different threads may see a different order

Weak Consistency

- Data operations vs. synchronization operations
- Synchronization operations are sequentially consistent
- When a synchronization operation is issued, the memory pipeline is flushed

Release Consistency

- Further subdivide synchronization operations into “acquire” and “release”
  - Matches the idea of ”acquiring a lock” and “releasing a lock”
- Before accessing a variable, all acquire operations have to be complete
- Before completing a release, all read and write operations have to be complete
- All acquires and releases have to be sequentially consistent

The OpenMP Memory Model

OpenMP uses a relaxed consistency similar to weak consistency

- Temporarily, the view on memory from different threads can be inconsistent
- Memory state is made consistent at particular constructs
- Programmer need to be aware of this and synchronize additionally as needed

Memory synchronization points (or flush points)

- Entry and exit of parallel regions
- Barriers (implicit and explicit)
- Entry and exit of critical regions
- Use of OpenMP runtime locks
- Every task scheduling point

However, note the following are NOT synchronization points:

- Entry and exit of work sharing regions
- Entry and exit of master regions

Implementing Manual Synchronization

Thread 1

```
a = foo();
flag = 1;
```

Thread 2

```
while (flag);
b = a;
```

What can go wrong?

- Compiler can reorder memory accesses
- Compiler can keep variables in registers

OpenMP's Flush Directive

```
#pragma omp flush [(list)]
```

Synchronizes data of the executing thread with main memory
- E.g., copies in register or cache
- It does not update implicit copies at other threads

Operates on variables in the given list
- If no list is specified, all shared variables accessible in the region

Semantics:
- Load/stores executed before the flush have to be finished
- Load/stores following the flush are not allowed to be executed early

Implementing Manual Synchronization

Thread 1

```
a = foo();
#pragma omp flush
flag = 1;
#pragma omp flush
```

Thread 2

```
while (flag)
{
  #pragma omp flush
}
#pragma omp flush
b = a;
```

Implementing Manual Synchronization

Thread 1

```
a = foo();
#pragma omp flush(a,flag)
flag = 1;
#pragma omp flush(flag)
```

Thread 2

```
while (flag)
{
  #pragma omp flush(flag)
}
#pragma omp flush(a,flag)
b = a;
```

Practical implications

- Necessary for any “manual” synchronization
- Mainly to avoid problems from optimizations
  - Reorderings, register allocations
- Hardware memory model itself in most cases stronger than this relaxed consistency
  - Codes may run, but could be incorrect and hence non-portable!
  - Flushes are rarely variable specific, but can be performance critical in some cases

Scaling Cache Coherent Shared Memory Machines

UMA architectures

- Central memory system becomes the main bottleneck
- Memory consistency rules are necessary, but expensive
- Must sustain bandwidth for ALL CPUs

NUMA architectures

- Memory system becomes distributed
  - More bandwidth, reduced bottleneck
- CC protocol becomes huge overhead
  - Each memory access potentially invalidates data in any cache
  - Cannot scale

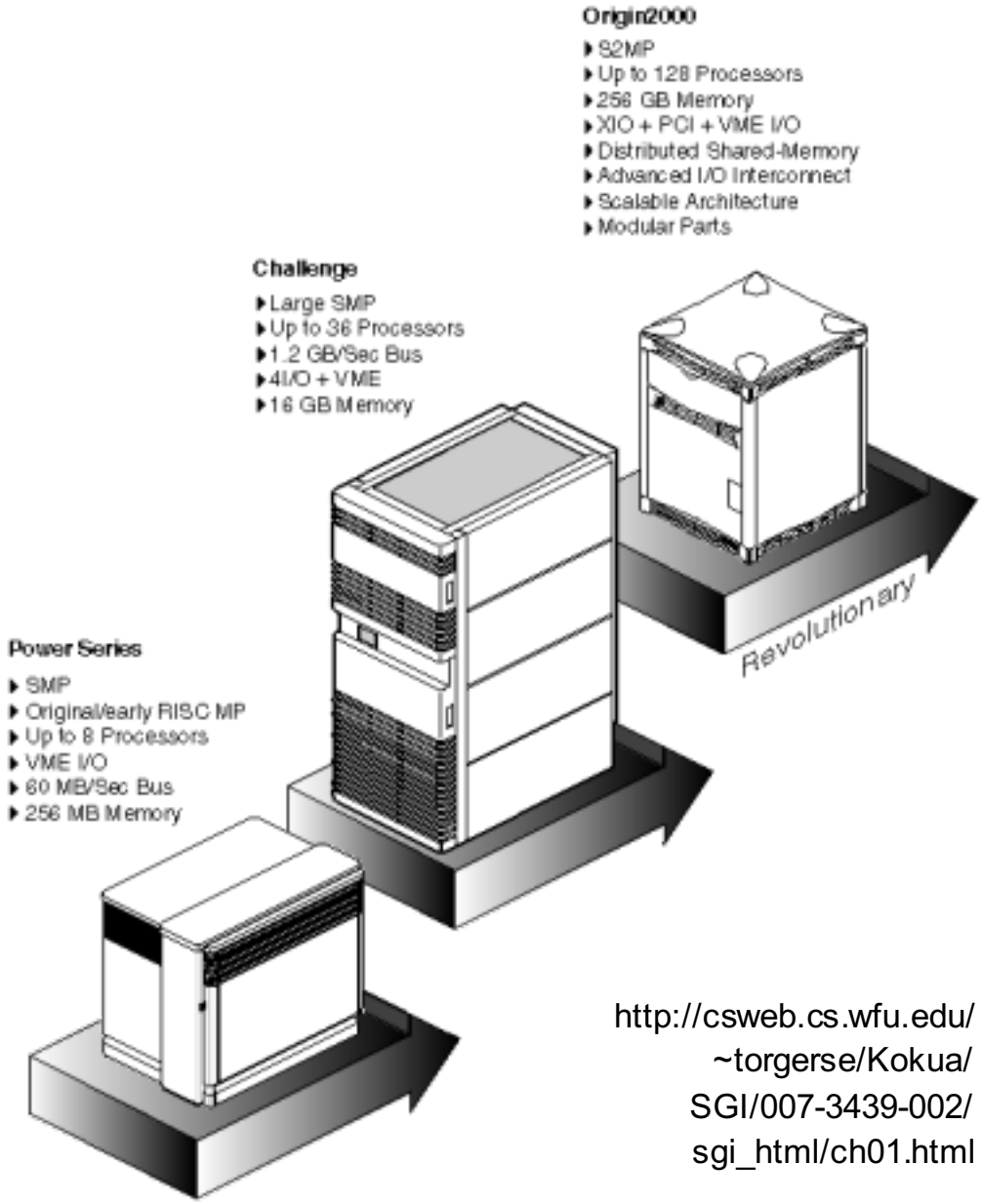
Examples of Large Shared Memory Machines

UMA systems are easy to use

- Predictable performance
- Standard thread model works well
- Easy to scale codes

But: UMA approach is limited

- Not many systems beyond 8 CPUs
- One of the largest:
  SGI's Challenge Series
  with up to 36 CPUs



http://csweb.cs.wfu.edu/~torgerse/Kokua/SGI/007-3439-002/sgi_html/ch01.html

SGI Altix: Example of a Large Scale CC-NUMA

Altix line started with Itanium processors, later switched to Xeon processors
Connected with SGI's NUMA link

- Global CC address space
- Requires single system image / one global OS instance
- Drove many NUMA extensions in Linux

LRZ used to have an Altix 4000

- First HPC system in Garching location

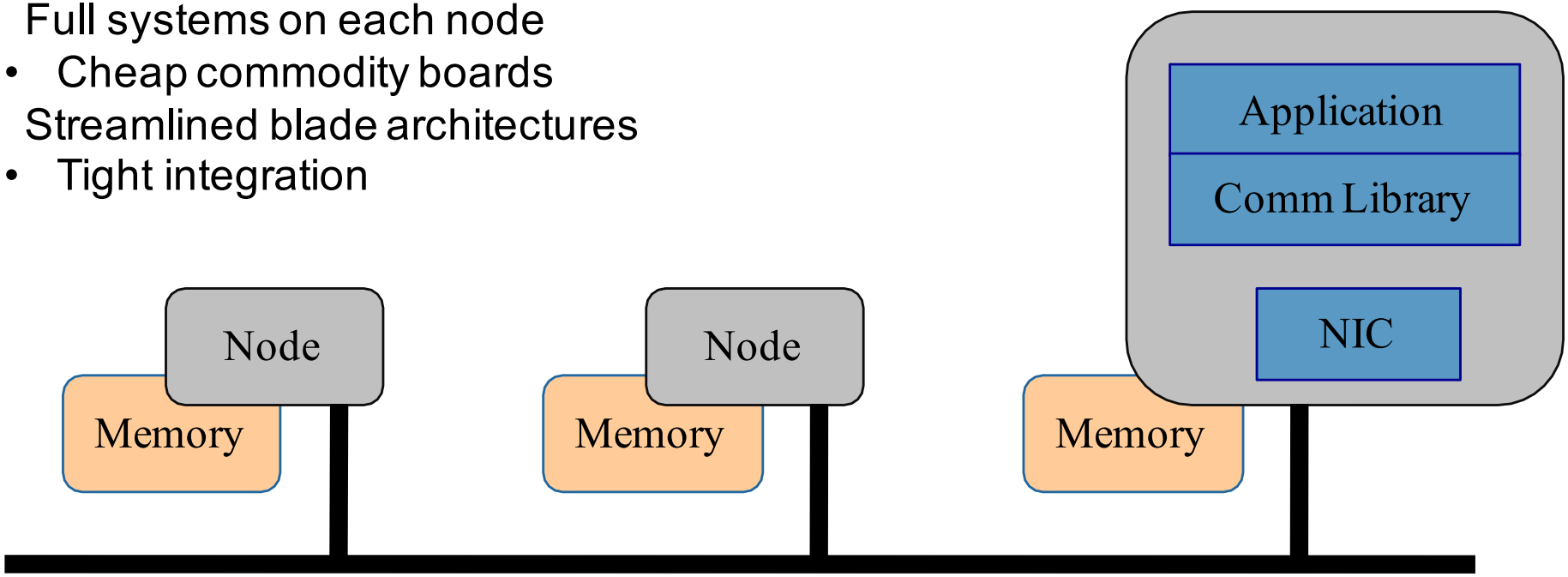
Largest system configuration

- Up to 8192 nodes
- CC protocol becomes a bottleneck
- NUMA issues degrade performance

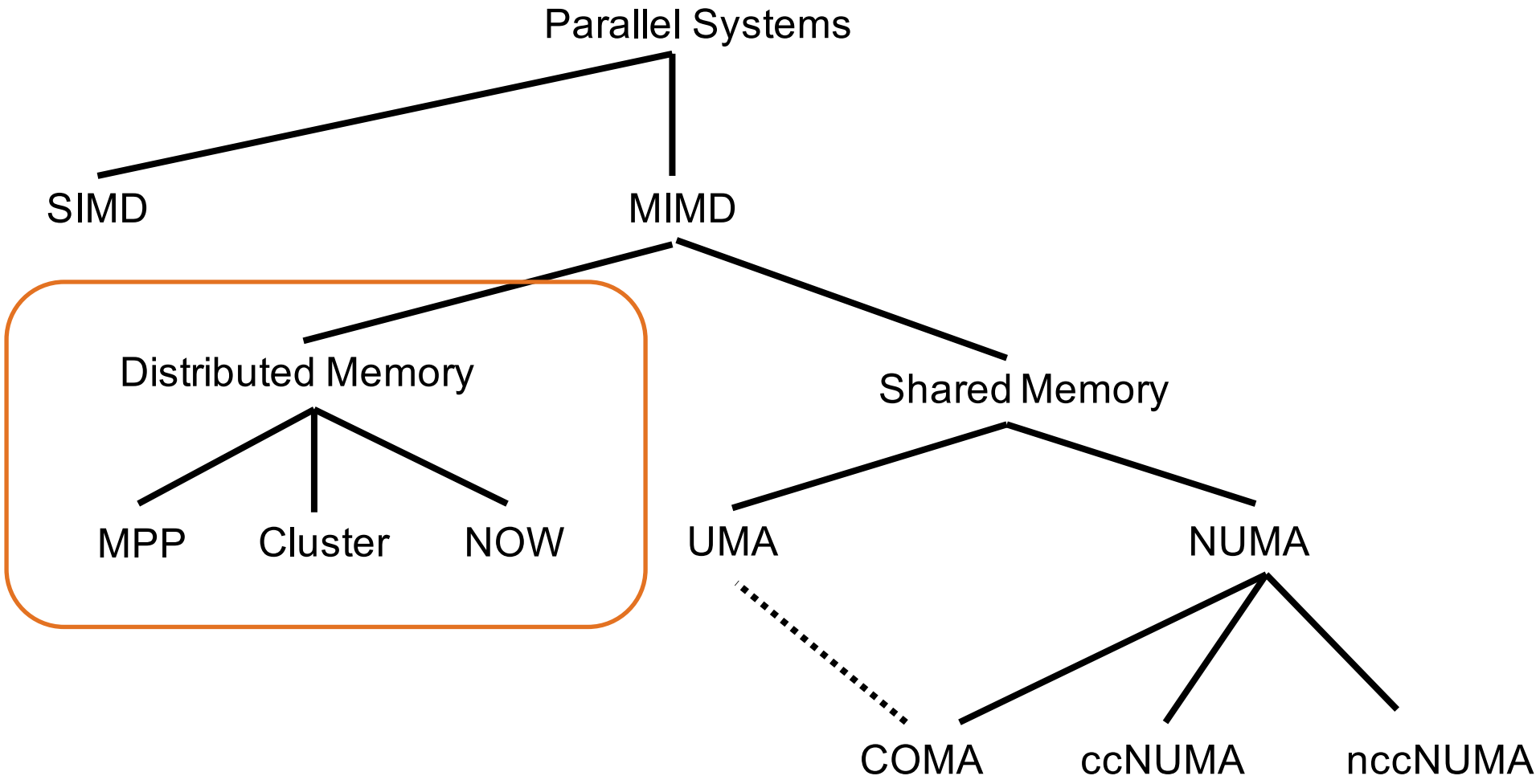
Distributed Memory Machines (Hardware)

Separate compute nodes
- Independent “computers” / “Shared Nothing Architecture”
- Separate processors
- Separate memories
- Connected through an explicit network

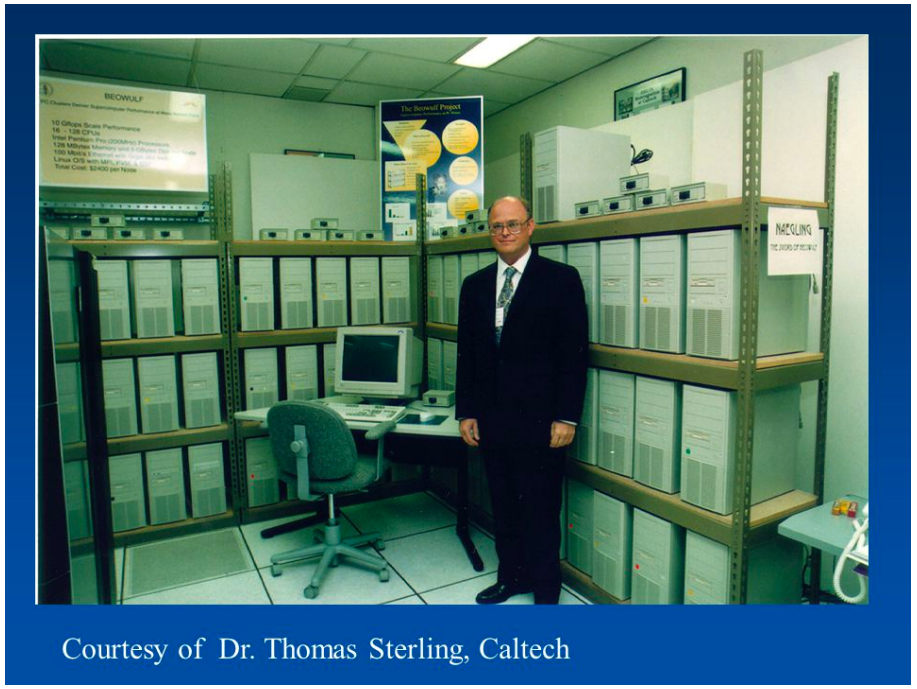
Difference lies in the level of integration
- Full systems on each node
  - Cheap commodity boards
- Streamlined blade architectures
  - Tight integration



Distributed Memory Machines



One Side: Compute Clusters



Courtesy of Dr. Thomas Sterling, Caltech

Started as “Cheap Parallel Processing”

- Several connected PCs / NOW
- Simple communication HW/SW
- “Beowulf” Systems

Use of standard SW stacks

- Linux played a big role

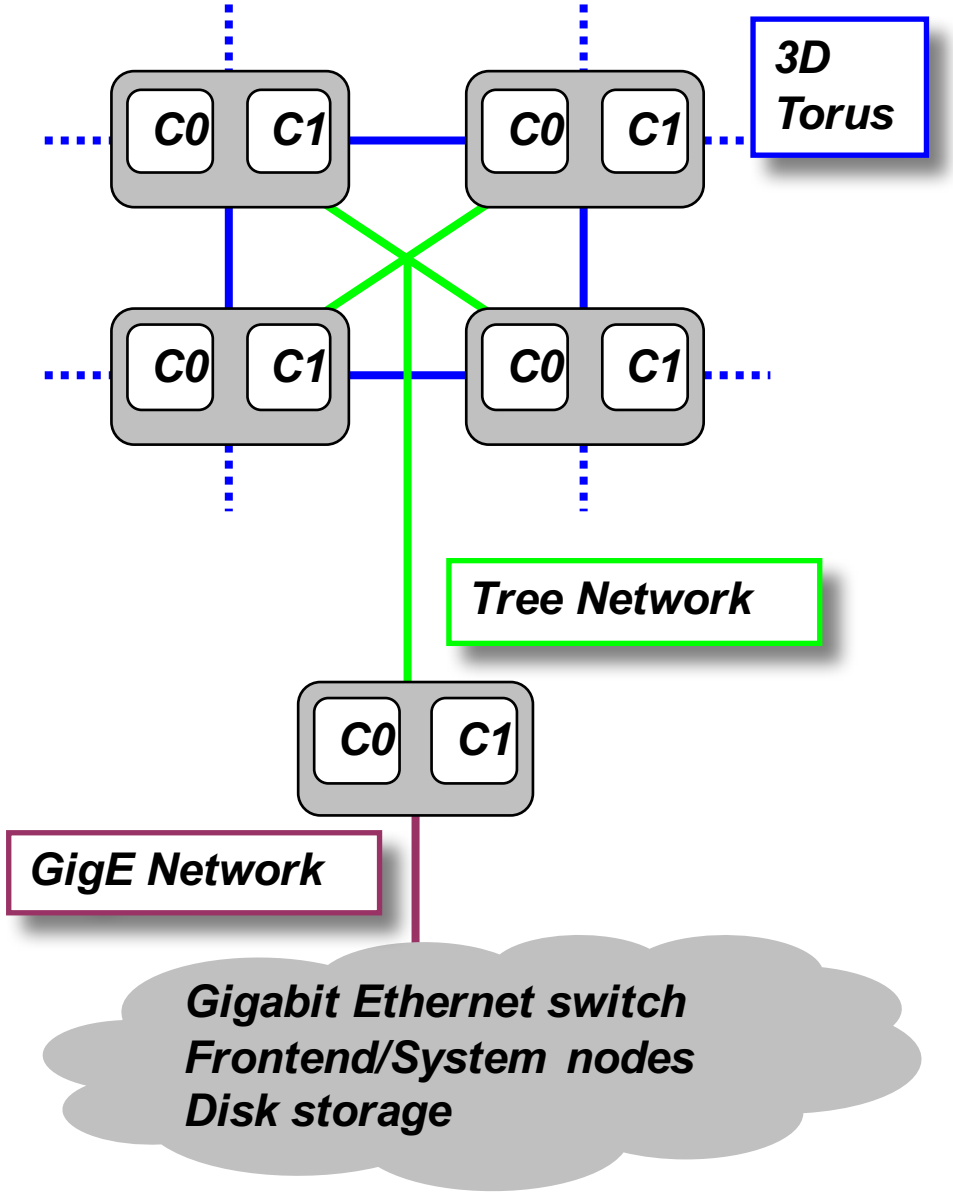
Today’s system

- More professional
- 1U nodes
- Blade designs
- Targeted cooling systems

Difference to HPC systems is getting less and less

The Other Side: Tightly Integrated Systems Example: Blue Gene/L

Blue Gene/L Architecture



Compute Nodes
- Dual core PowerPC 440
- 700 MHz clock
- 512 MB memory

Networks
- 3D torus
- Tree network

I/O nodes
- Same node architecture
- Serving up to 64 nodes
- Connected to tree network
- GigEthernet to outside

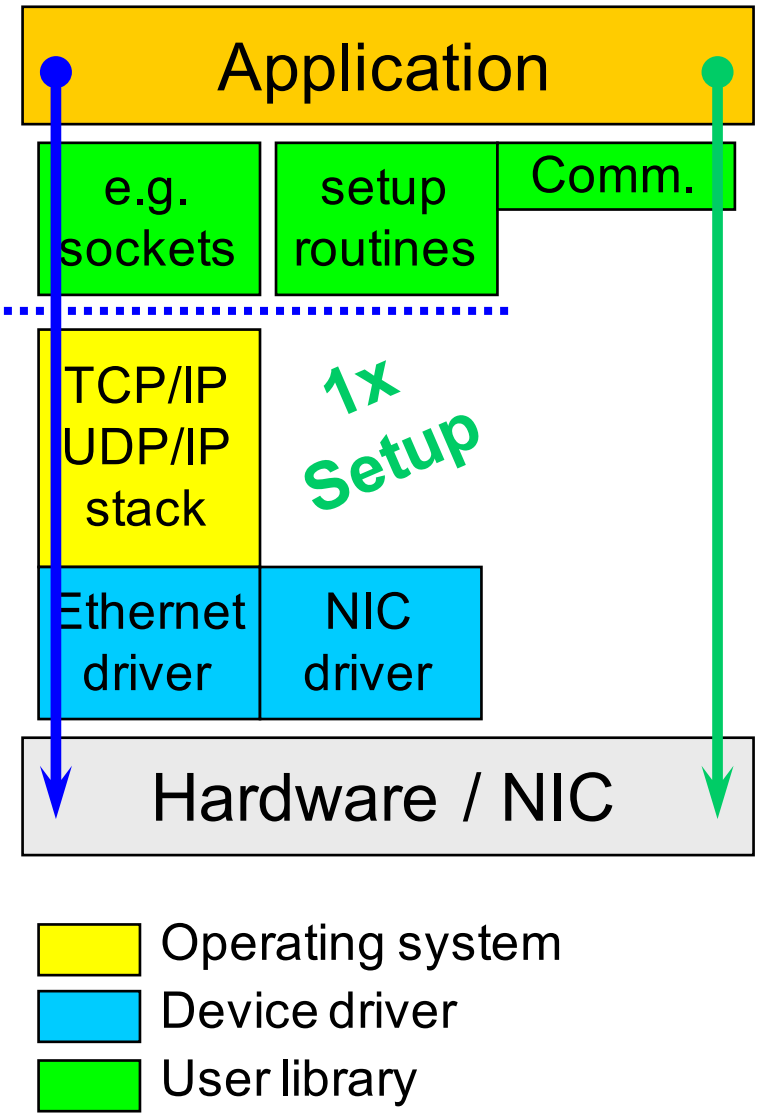
LLNL's Sequoia

Location	Lawrence Livermore National Laboratory, CA, USA
Architecture	IBM Power
Power	7.9 MW (Linpack)
Memory	1.5 PB
Peak Performance	20.13 PFLOPS
HW threads	> 6 million

Networks

Standard: Ethernet
- Cheap and ubiquitous
- The standard protocol is TCP/IP
- Large software overheads
- Networks of Workstations (NOW)

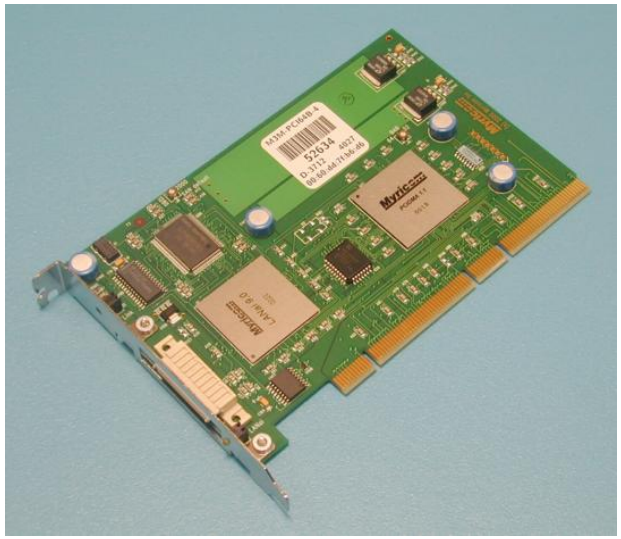
Alternative: User-level Communication
- Only setup via OS kernel
  - Protected routines
  - Establish security/protection
- Direct communication from user-level
  - Removes OS overhead
  - Enables high bandwidth/low latencies



The Beginnings: SCI & Myrinet

Scalable Coherent Interface

- Based on IEEE Standard
- Remote access to memory
- Implemented as PCI-SCI bus bridge
- Bandwidth: 80- 320 MB/s



Myrinet

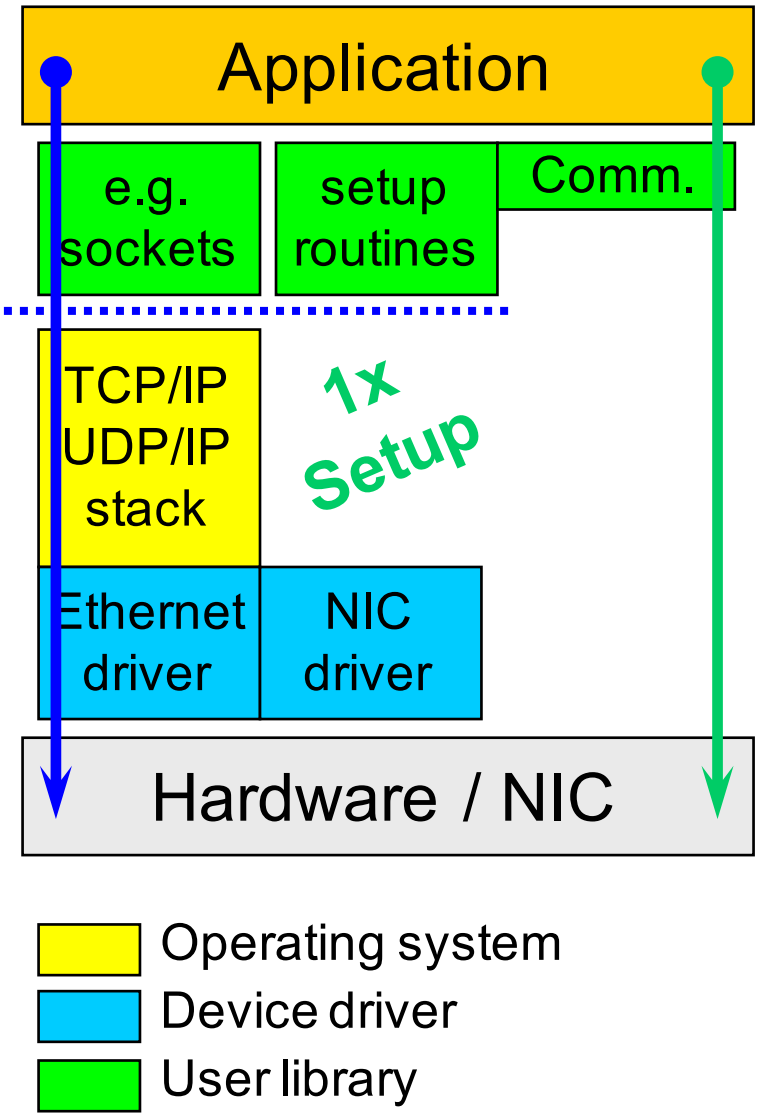
- Processor on each card
- Can copy pieces of memory to remote nodes
- Bandwidth: 100-250 MB/s

Networks

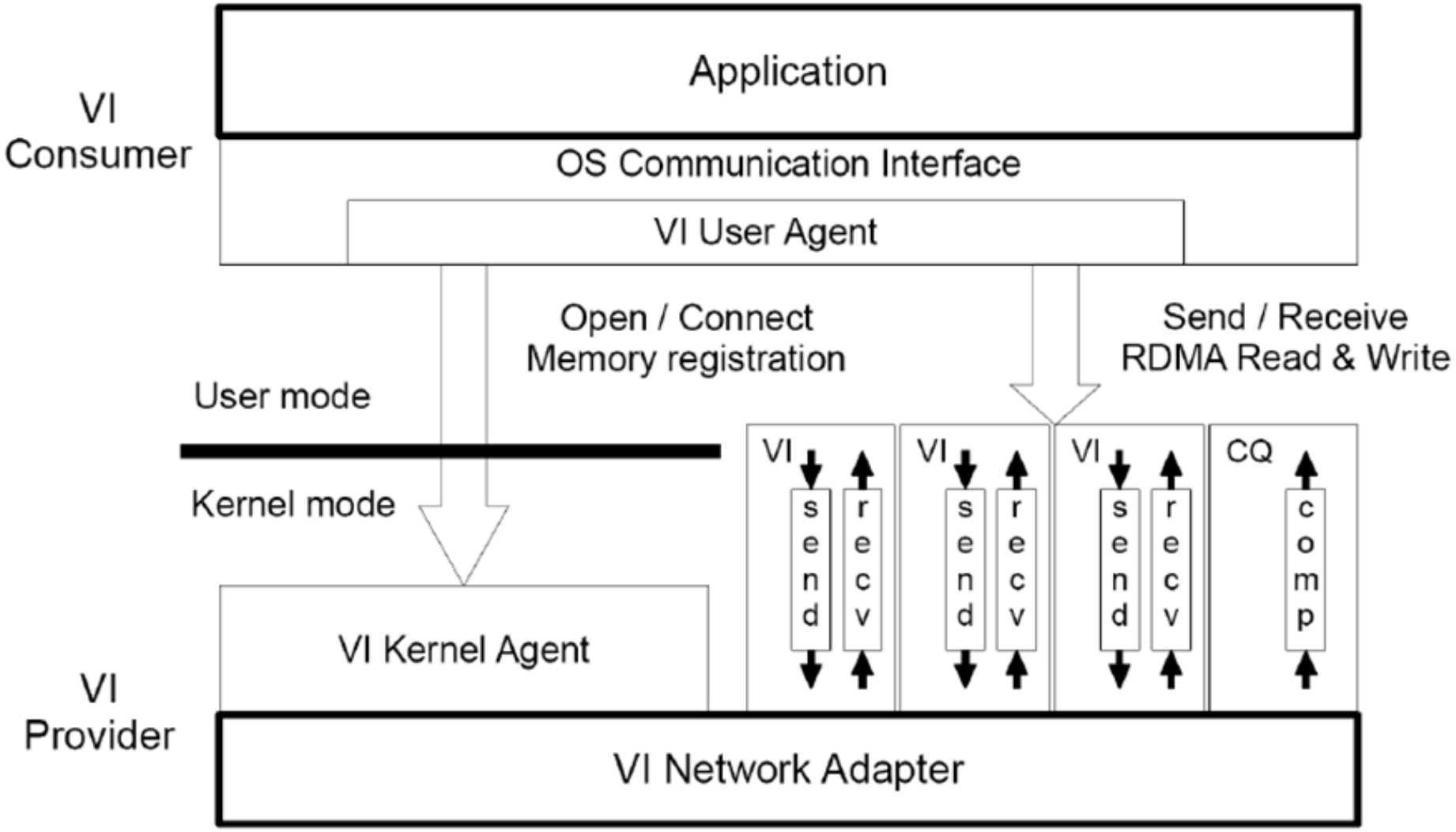
Standard: Ethernet
- Cheap and ubiquitous
- The standard protocol is TCP/IP
- Large software overheads
- Networks of Workstations (NOW)

Alternative: User-level Communication
- Only setup via OS kernel
  - Protected routines
  - Establish security/protection
- Direct communication from user-level
  - Removes OS overhead
  - Enables high bandwidth/low latencies

Today: standard for cluster communication
- Hidden from the user
- VIA Architecture
- Most establish example: Infiniband

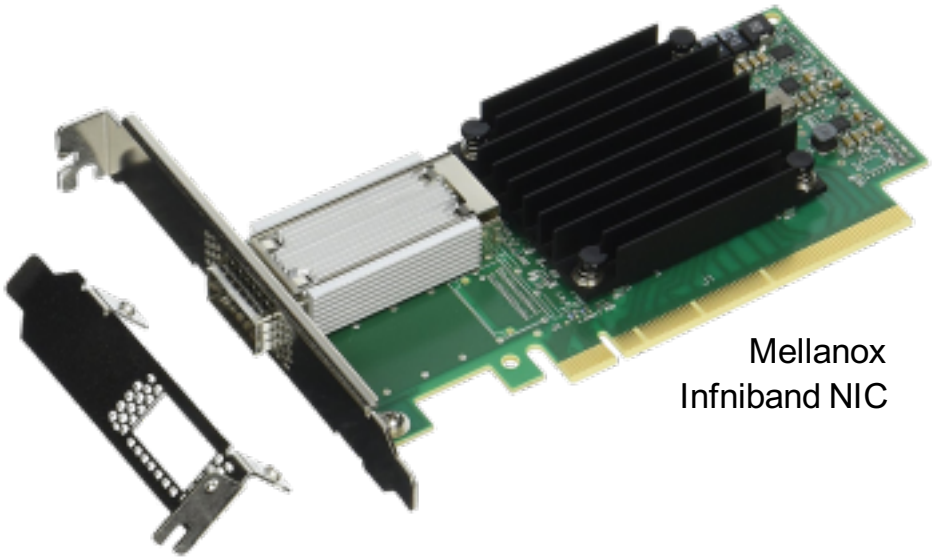


Virtual Interface Architecture (VIA)



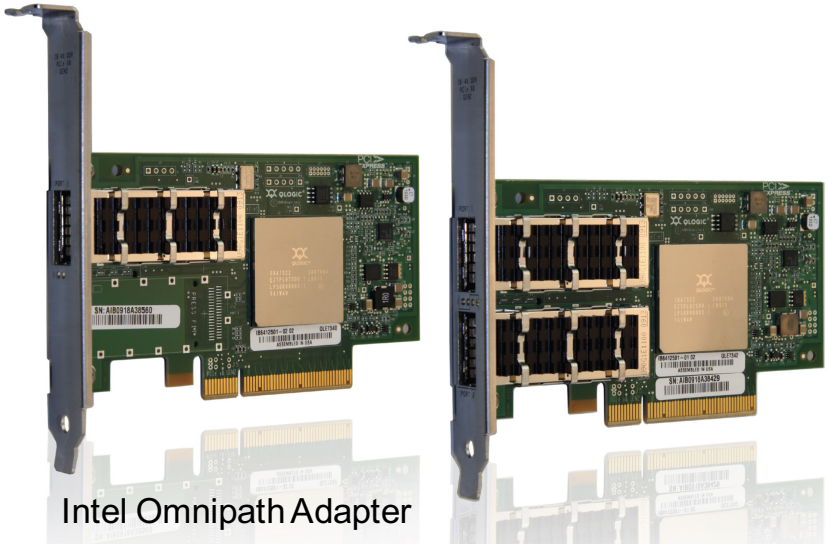
Currently Most Used Cluster Networks

Infiniband (max. HDR @ 50 Gbit/s)
- Based on the VIA architecture
- Intended as compute *and* storage network
- Switched network, typically fat tree



Mellanox
Infniband NIC

Implementations focus on “offloading”
- Active NIC performs communication
- More complex NICs, but host system has a reduced workload

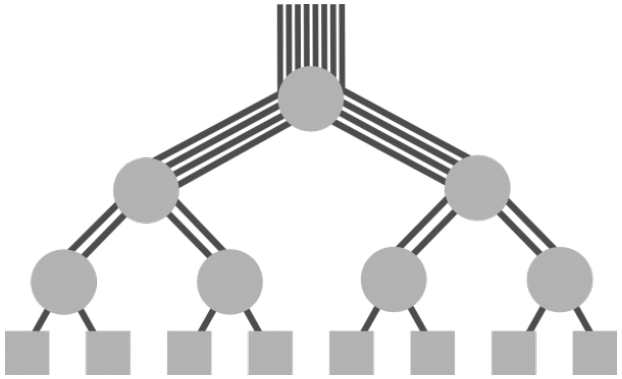


Intel Omnipath Adapter

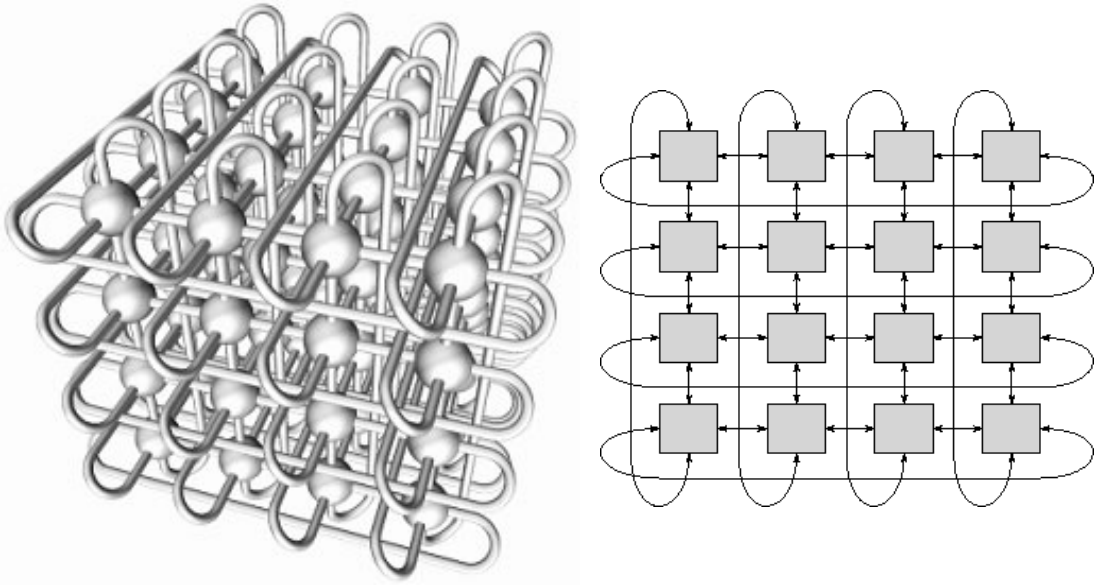
Intel’s Omnipath (up to 100 Gbit/s)
- Intended as Infiniband successor
- Focus on scalability
- Common fat tree, but other topologies possible

Key feature “onloading”
- Simpler NIC
- Processor takes on more responsibilities
- Higher CPU load, but close to data

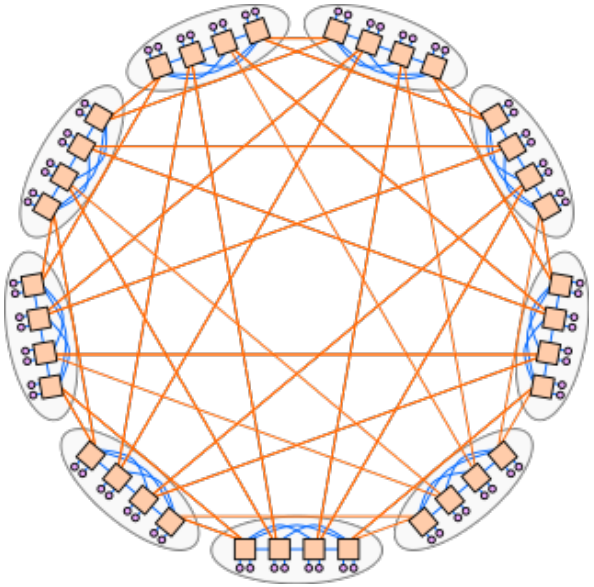
Network Topologies



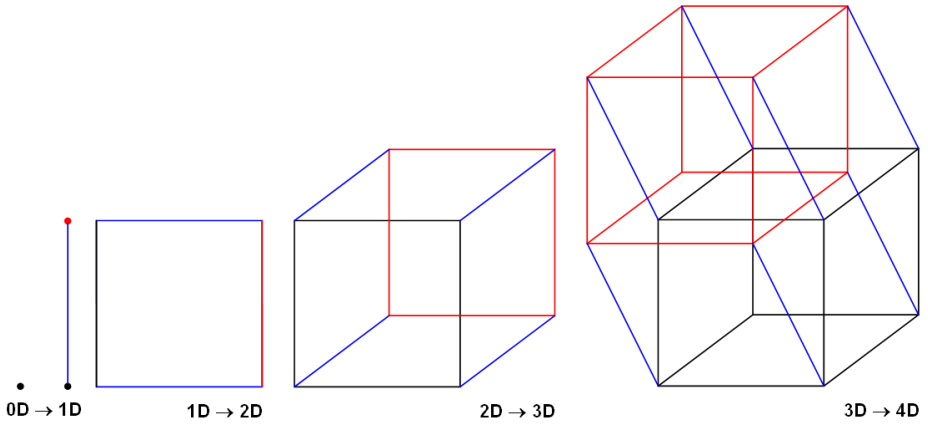
Fat-Tree: most cluster interconnects



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



Dragonfly: Cray XE



Hypercube: Intel Paragon, SGI Altix (modified)

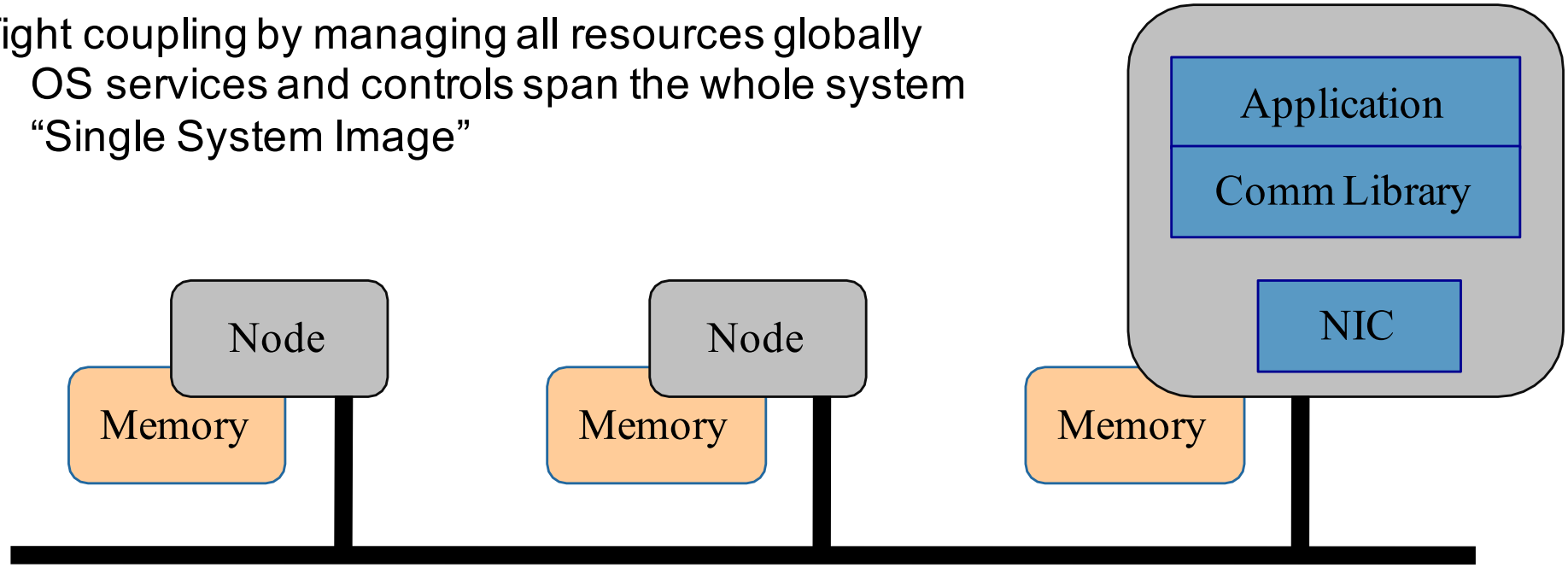
Sources: Clusterdesign.org, ANL, Fujitsu, Wikipedia

Distributed Memory Machines (Software)

Independent nodes
- At first separated software stacks
- Running independently from each other
- Need to be coordinated or coupled through resource management

Loose coupling by only managing node allocation
- Individual nodes maintain separate OS instances
- Communication through standard OS services

Tight coupling by managing all resources globally
- OS services and controls span the whole system
- “Single System Image”



HPC Ecosystem

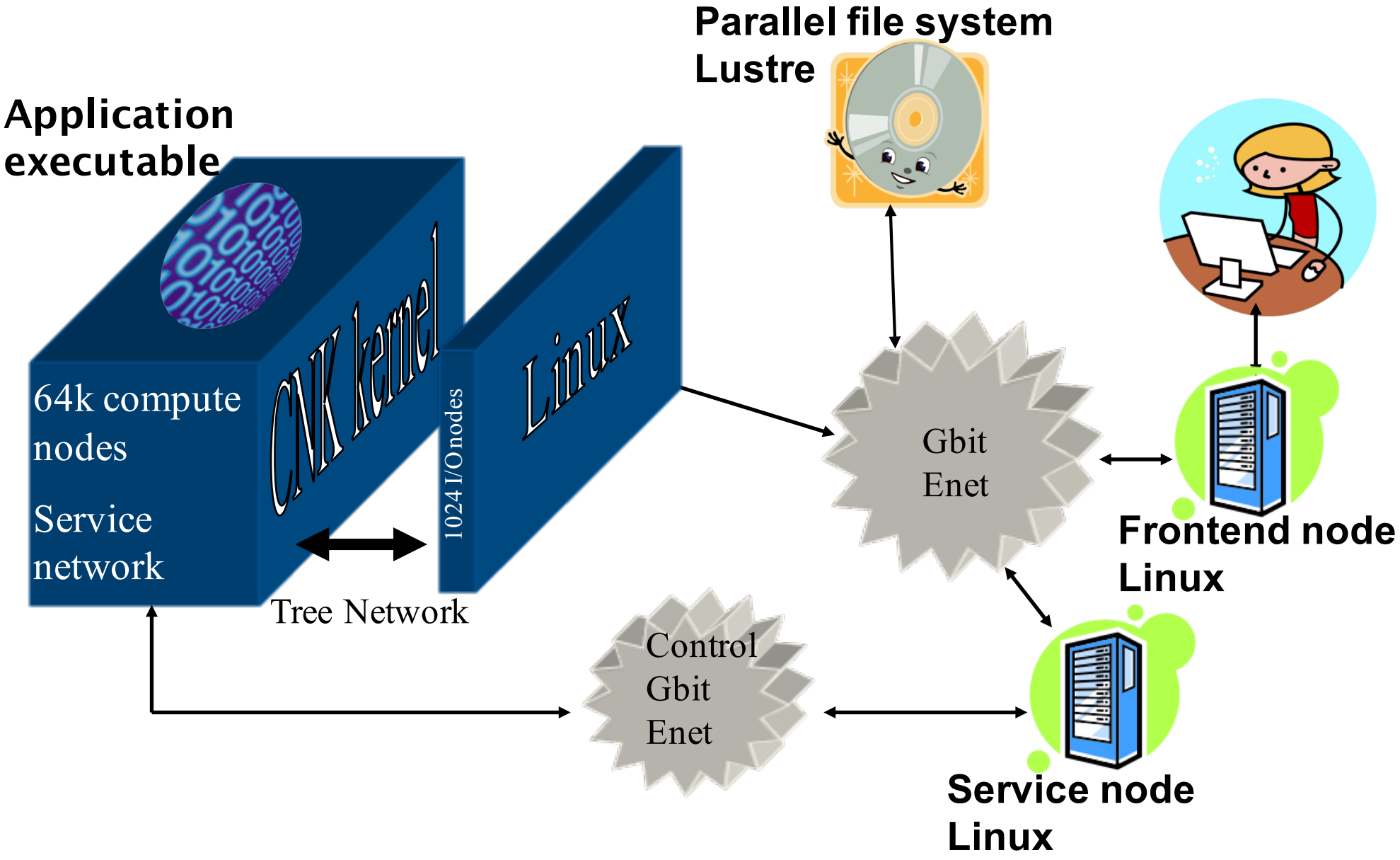
Compute nodes often simplified

- Headless with no graphics support
- No local disk

HPC is more than just compute nodes

- Head/Login/Compilation nodes
- System nodes
  - RAS (Reliability, Availability, Serviceability) components
  - Resource manager
- Storage system
  - Parallel file systems
  - Often driven by dedicated I/O nodes
- Tape archive
- Visualization systems

Blue Gene/L
Major System Software Components



Consequences for Programmability

Problems need to be manually partitioned and distributed

- Data has to be managed separately in the different memories

Communication must be programmed explicitly

- Use of dedicated communication libraries (e.g., MPI)

Fault tolerance, fault handling and debugging is more complex

- Have to reason about distributed state

Large scale MPP system often have slightly different OS environments

- Reduced services to minimize noise

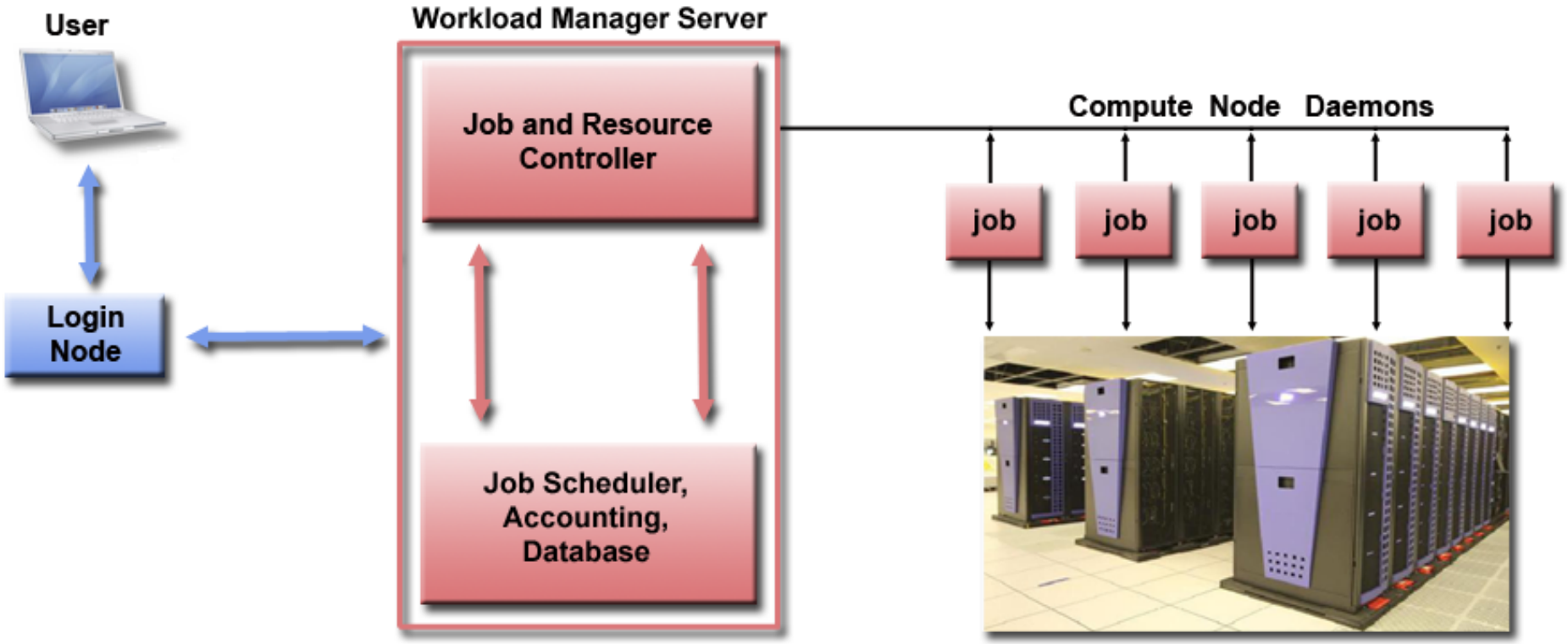
Login/Compile/Compute nodes may be different

- Requires “cross-compilation”

Access to HPC Systems

HPC Systems have multiple users

- Requires access through a job/resource management system
- Users create job files/descriptions and submit them
- Once resources are free, job gets scheduled



Source: LLNL

Batch Systems

Very common: SLURM Workload Manager
Simple Linux Utility for Resource Management

- Open source resource management
- Most commonly used commands
  - `salloc`: allocate a set of nodes
  - `srun`: run a given job on the current or a new allocation
  - `sinfo`: view partition and node status
  - `squeue`: view information about queues
  - `scancel`: remove a submitted job from the queue

Other resource managers (incomplete list)

- Load Leveler (IBM)
- Torque
- Moab
- Cobalt (ANL)

Often wrapped for better user experience

- Can also be hierachical

A “Close By” Exampe: LRZ’s SuperMUC

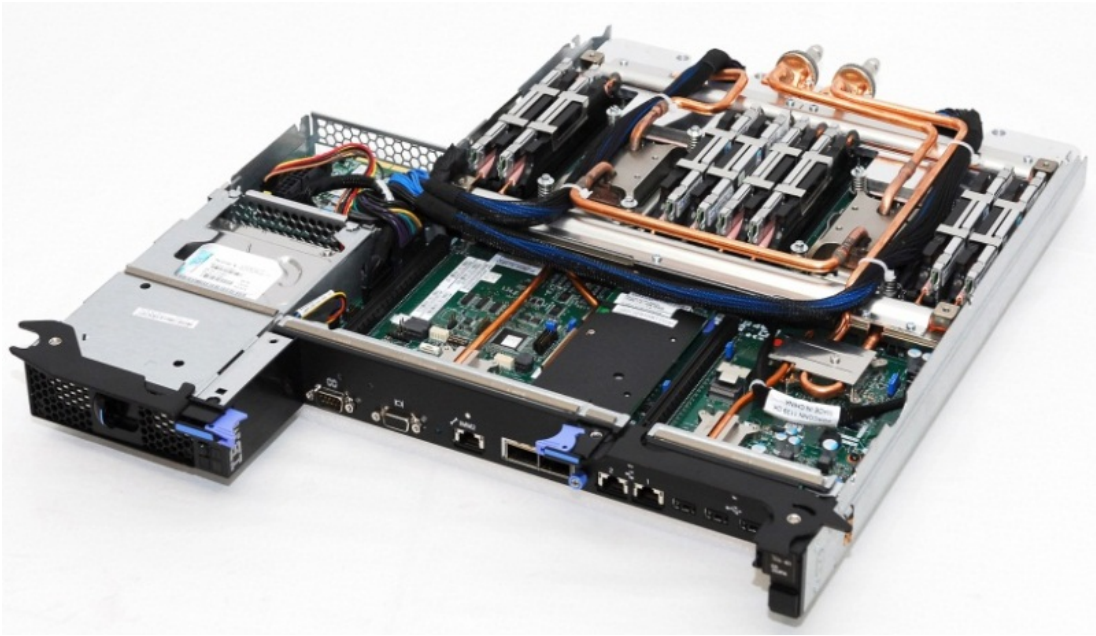
44	Leibniz Rechenzentrum Germany	**SuperMUC** - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR IBM/Lenovo	147,456	2,897.0	3,185.1	3,423
45	Leibniz Rechenzentrum Germany	**SuperMUC Phase 2** - NeXtScale nx360M5, Xeon E5-2697v3 14C 2.6GHz, Infiniband FDR14 Lenovo/IBM	86,016	2,813.6	3,578.3	1,481

SuperMUC is a Distributed Memory Architecture

18 partitions called islands with 512 nodes

Node is a shared memory system with 2 processors

- Sandy Bridge-EP
  Intel Xeon E5-2680 8C
  – 2.7 GHz (Turbo 3.5 GHz)
- 32 GByte memory
- Inifiniband network interface

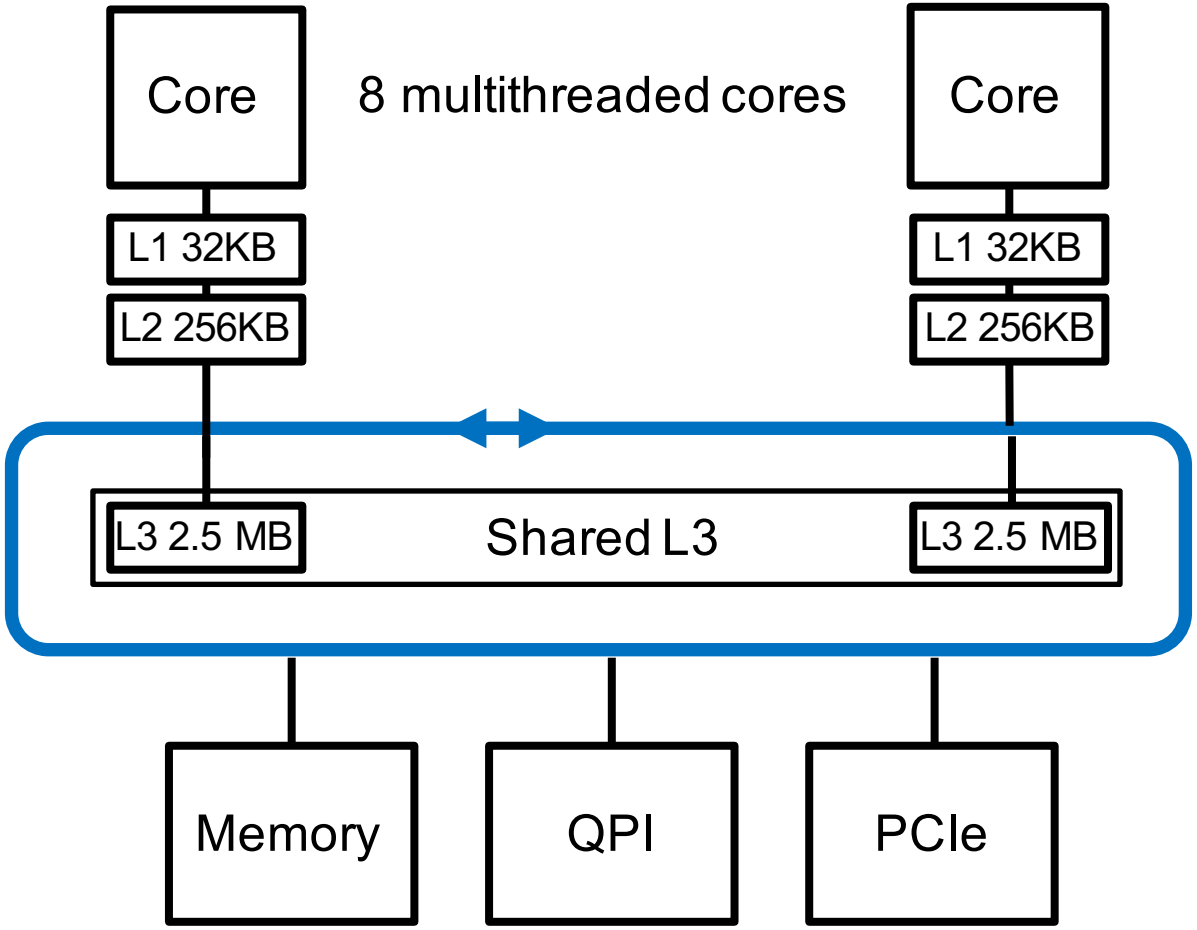


Processor/Socket has 8 cores

- 2-way hyperthreading
- 21.6 GFlops @ 2.7 GHz per core
- 172.8 GFlops per processor

Sandy Bridge Processor

Latency:
- 4 cycles
- 12 cycles
- 31 cycles

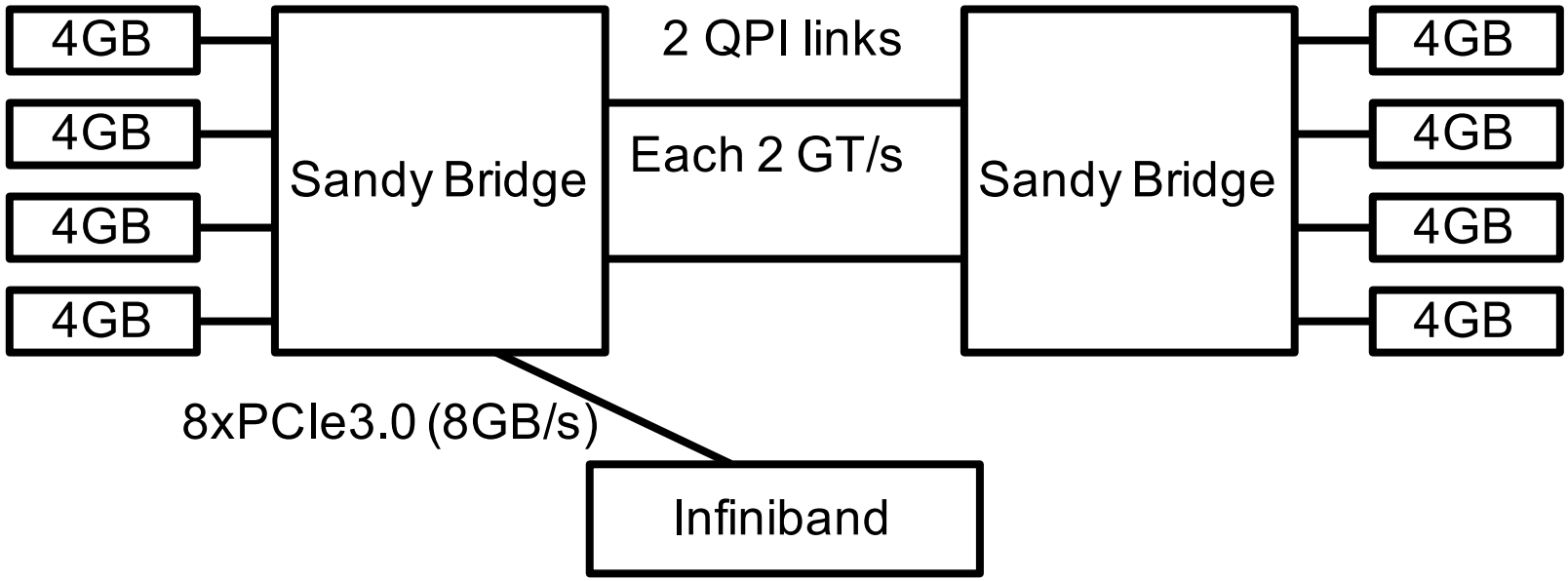


Bandwidth:
- 2*16/cycle
- 32 / cycle
- 32 / cycle

Network frequency equal to core frequency

L3 cache
- Partitioned with cache coherence based on core valid bits
- Physical addresses distributed by a hash function

NUMA Node



2 processors with 32 GB of memory

Aggregate memory bandwidth per node 102.4 GB/s

Latency

- local ~50ns (~135 cycles @2.7 GHz)
- remote ~90ns (~240 cycles)

9288 Compute Nodes

Picture:
Cold Corridoor

Infiniband (red)
and
Ethernet (green)
cabling

Credit: Matthias Brehm, Herbert Huber, LRZ High Performance Systems Division

Interconnection Network

Infiniband FDR-10

- FDR means Fourteen Data Rate
- FDR-10 has an effective data rate of 38.79 Gbit/s
- Latency: 100 nsec per switch, 1usec MPI
- Vendor: Mellanox

Intra-Island Topology: non-blocking tree

- 256 communication pairs can talk in parallel.

Inter-Island Topology: Pruned Tree 4:1

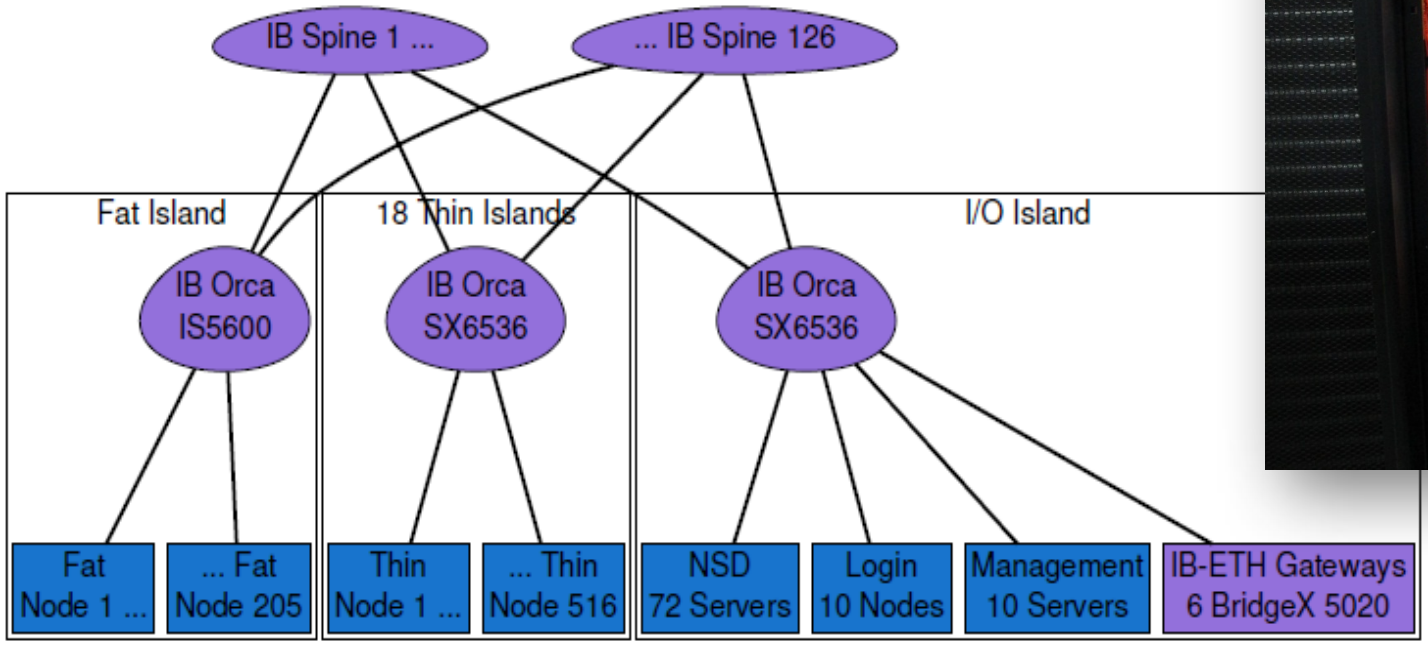
- 128 links per island to next level

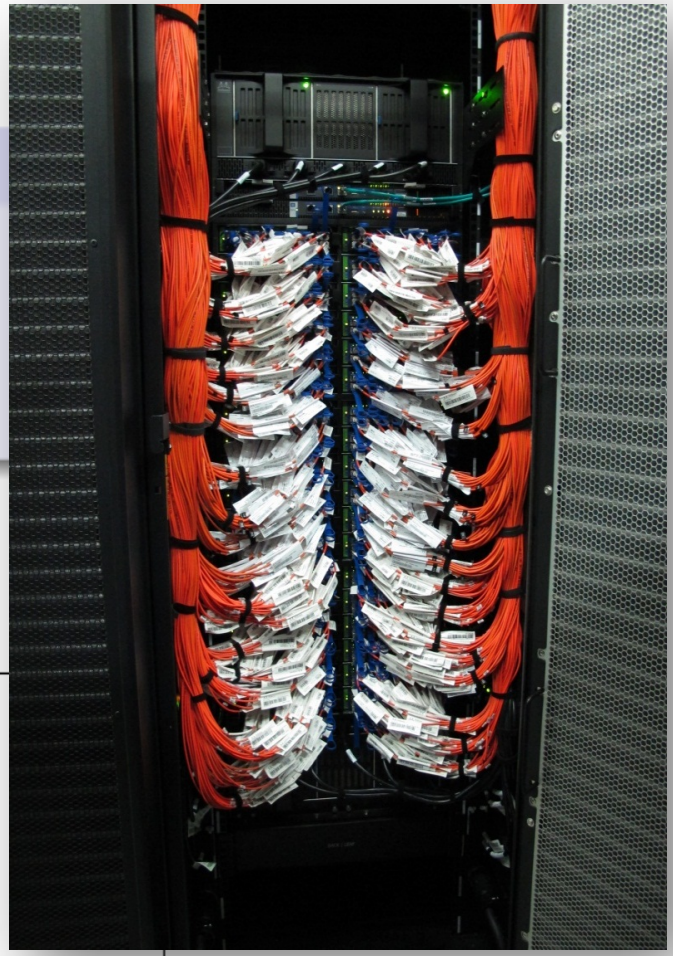
Infiniband Interconnect

Prunned Fat Tree

Simplified Illustration

- 126 Spine Switches (Mellanox SX6036, FDR10)
- 1 Island with 205 Fat Nodes (QDR)
- 18 Islands with 516 Thin Nodes each (FDR10)

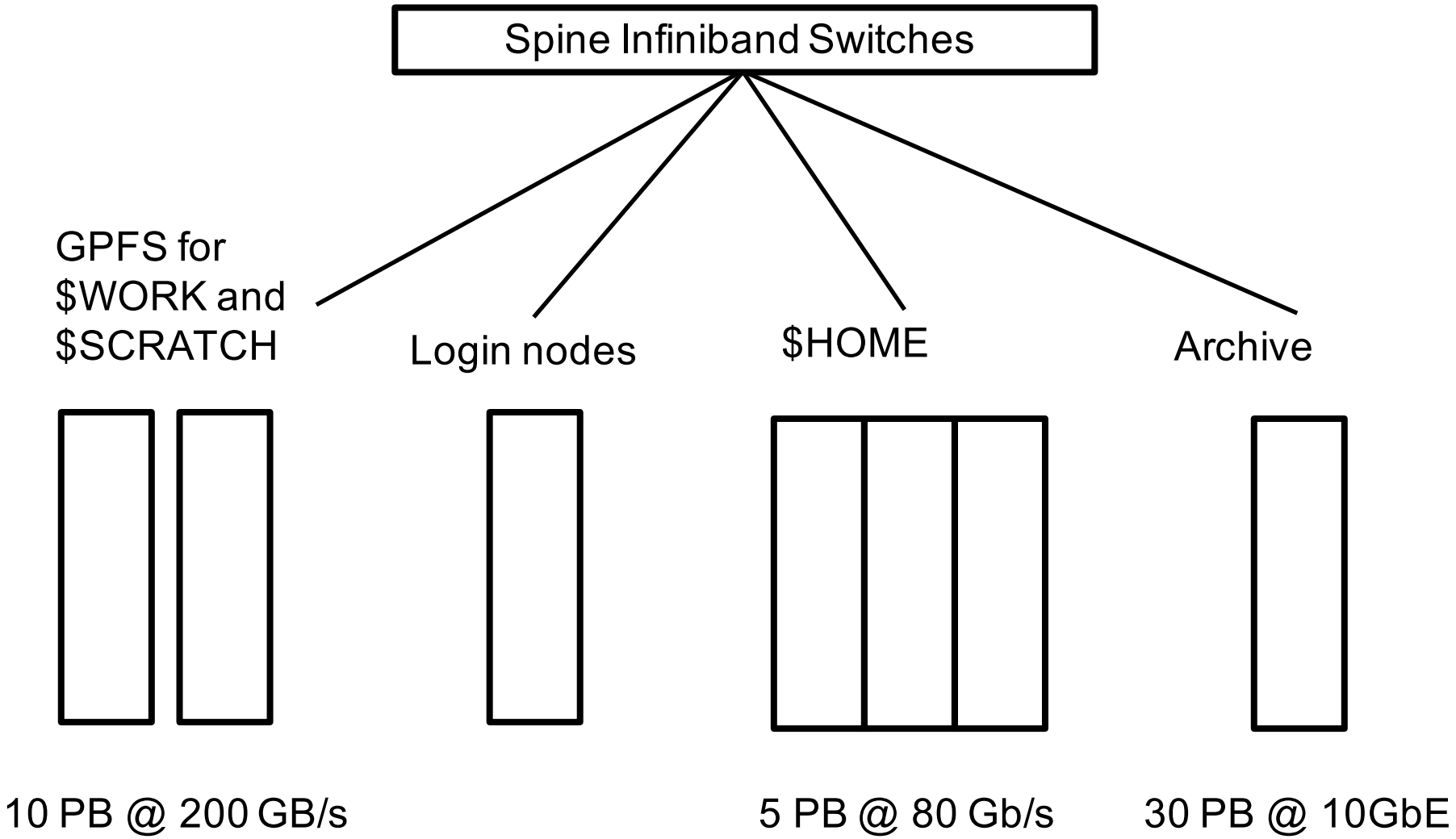




In total 11900 Infiniband Cables

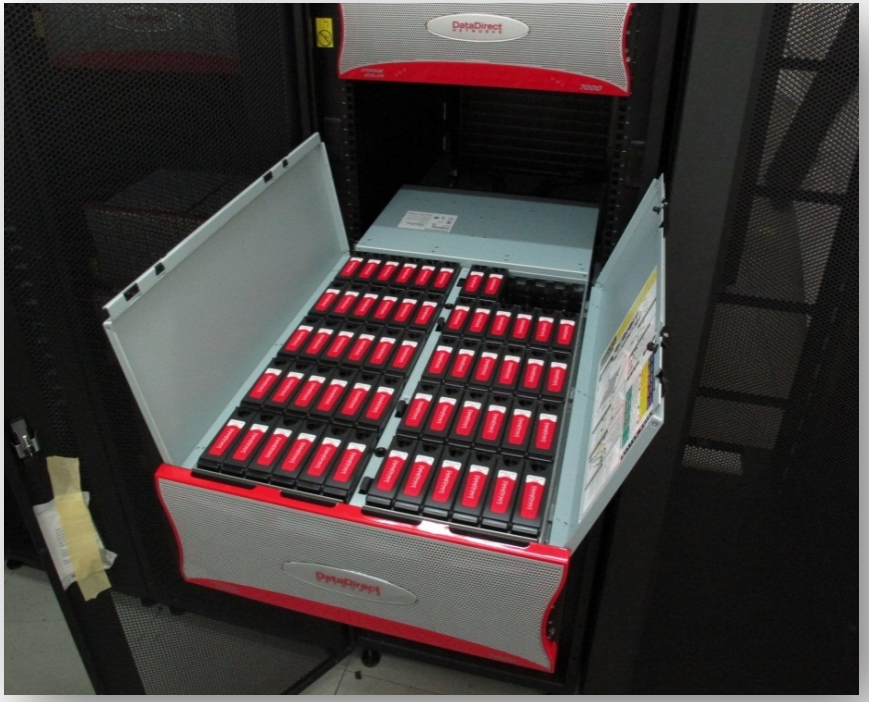
Credit: Matthias Brehm, Herbert Huber, LRZ High Performance Systems Division

IO System


Spine Infiniband Switches
GPFS for
$WORK and
$SCRATCH
Login nodes
$HOME
Archive
10 PB @ 200 GB/s
5 PB @ 80 Gb/s
30 PB @ 10GbE

Parallel File System GPFS

10 Pbyte, 200 GigaByte/s I/O Bandwidth



9 DDN SFA 12k Controller
5040 3 TByte SATA Disks

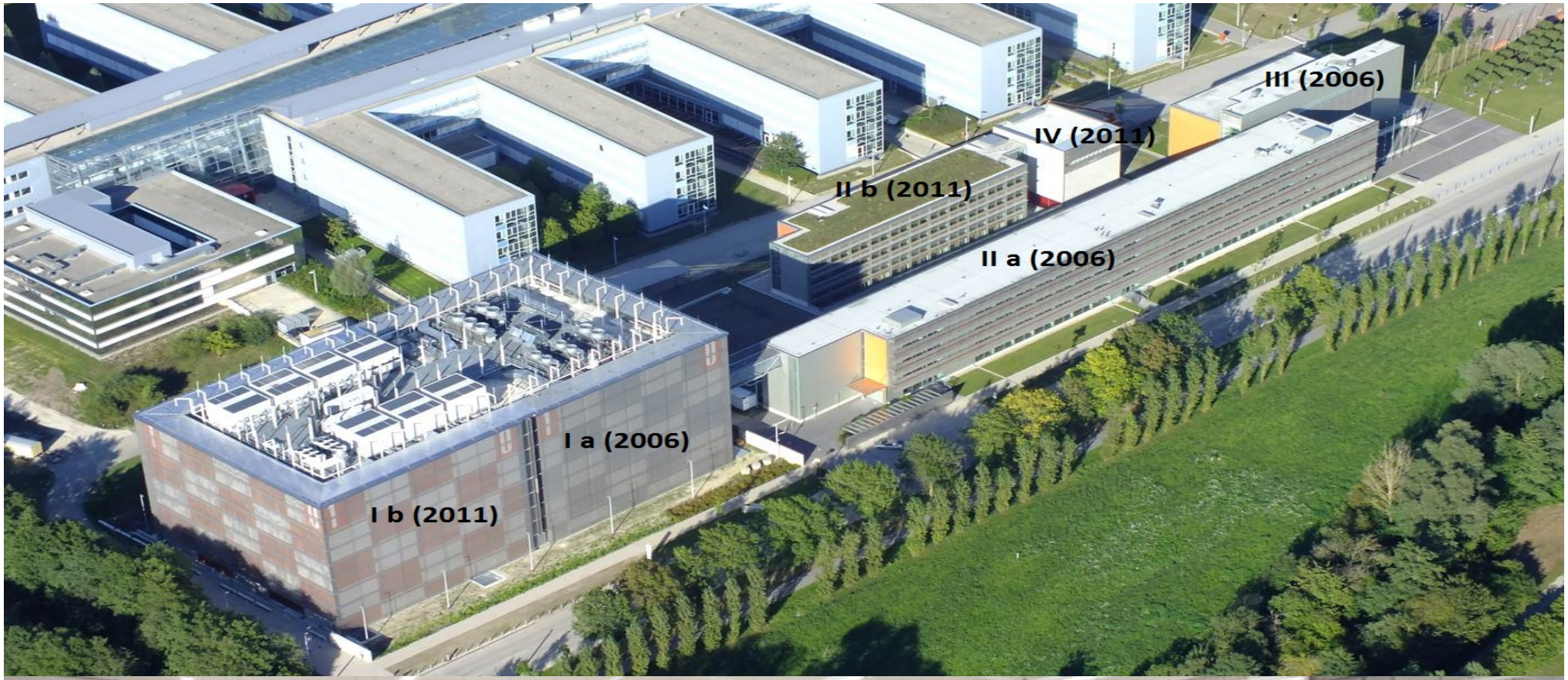
Credit: Matthias Brehm, Herbert Huber, LRZ High Performance Systems Division

SuperMUC Costs

	2010-2014 Phase 1	2014-2016 Phase 2
High End System		
Investment Costs (Hardware and Software)	**53 Mio €**	**~ 19 Mio €**
Operating Costs (Electricity costs and maintenance for hardware und software, some additional personnel)	**32 Mio €**	**~ 29 Mio €**
SUM	**85 Mio €**	**~48 Mio €**
Extension Buildings (construction and Infrastructure)	**49 Mio €**	

Source: LRZ

The Leibniz Supercomputing Centre (LRZ)



LRZ Data Center Facts

Some more Facts

- **3160.5 m²** (34 019 ft²) IT Equipment Floor Space (6 rooms on 3 floors)
- **6393.5 m²** (68 819 ft²) Infrastructure Floor Space
- **2 x 10 MW** 20kV Power Supply
- **Powered Entirely by Renewable Energy**
- **> 6M €** Annual Power Bill

Towers

uting

rvers, Network

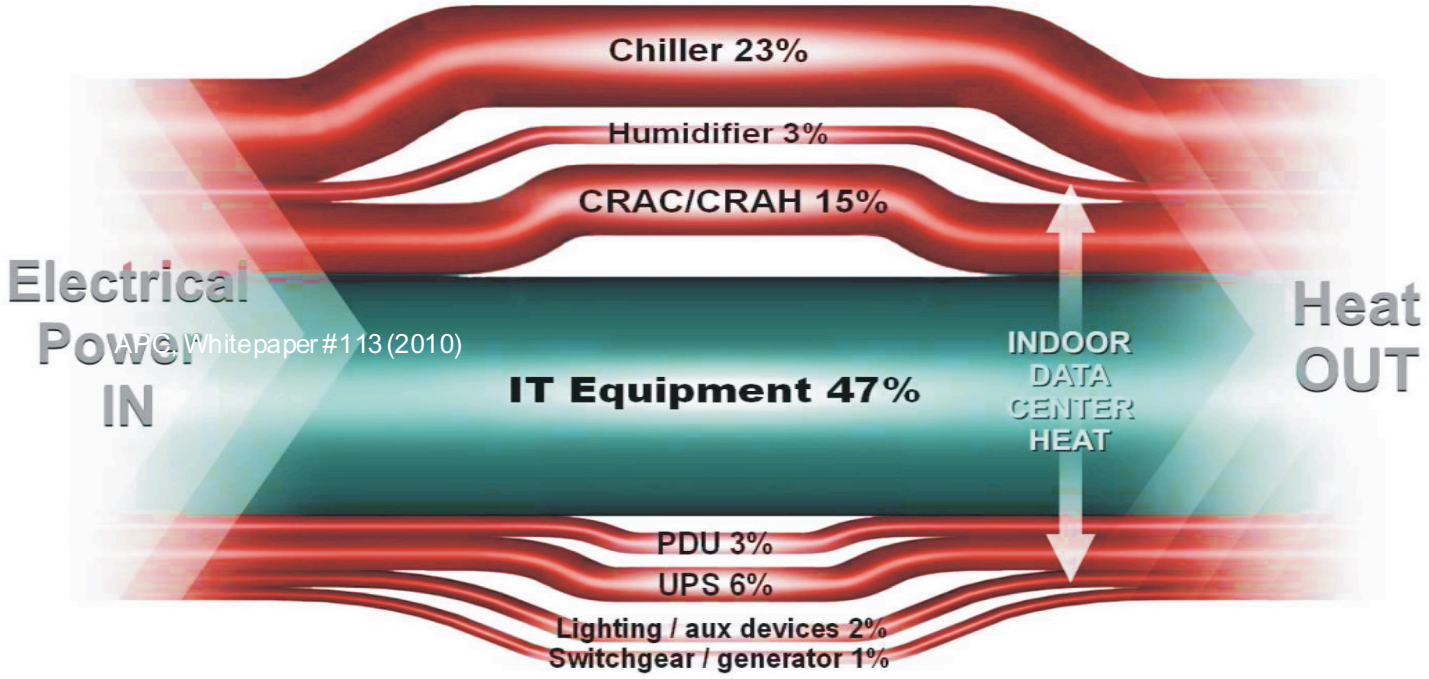
Archive/Backup

ir Processing

Transformers,

Energy Consumption in Data Centers

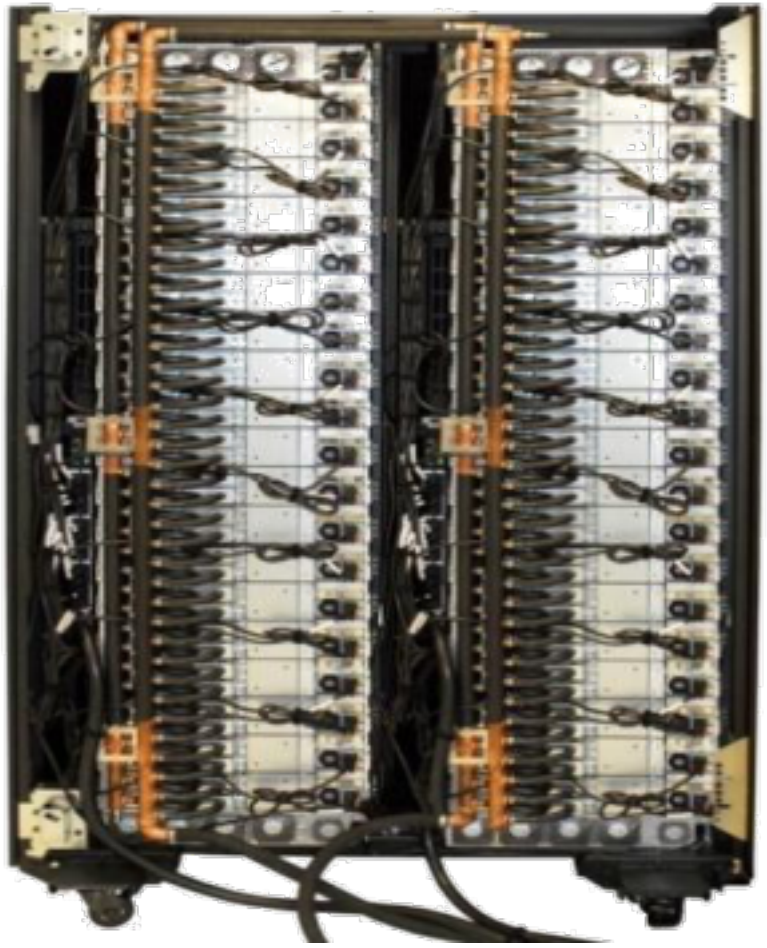
Data Centers are “Heaters with integrated logic”



Cooling Setup for SuperMUC at LRZ

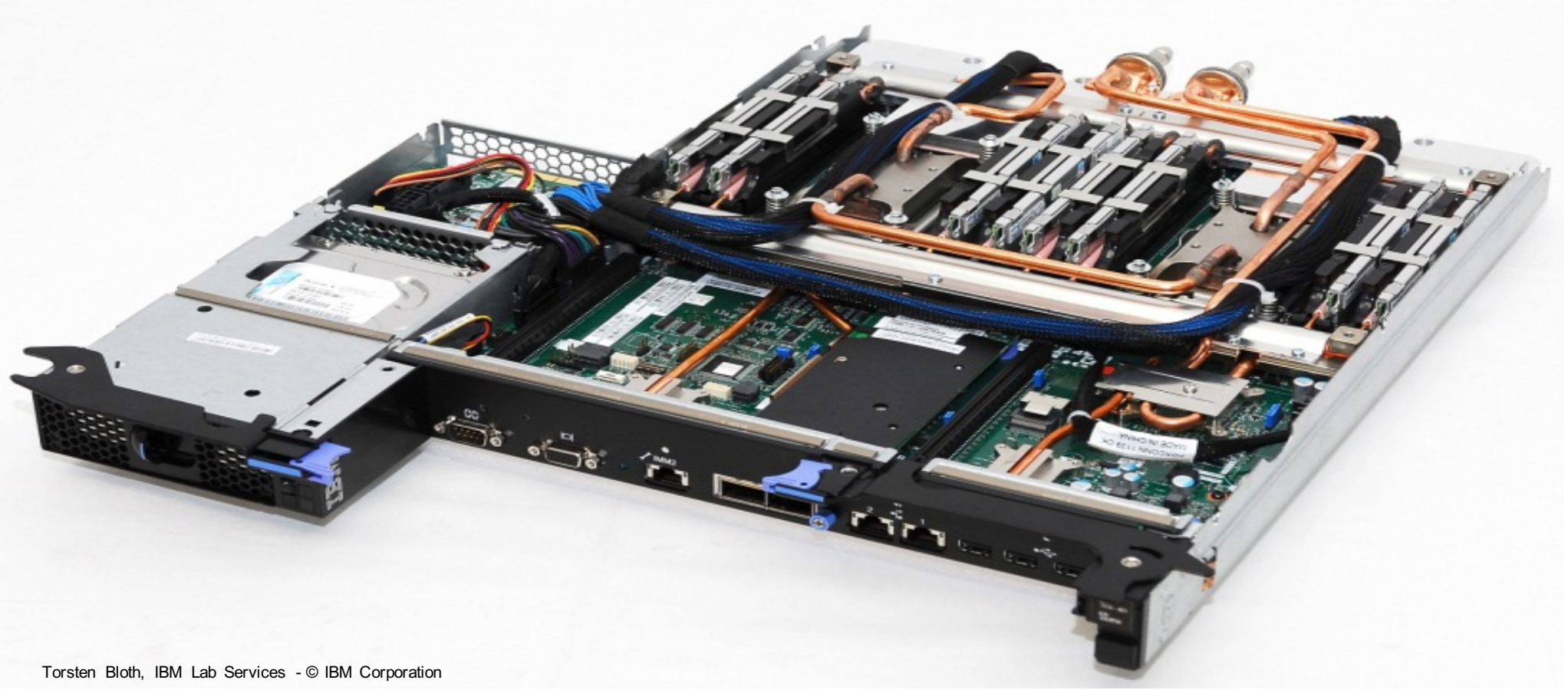
IBM System x iDataPlex Direct Water Cooled Rack

iDataplex DWC Rack
w/ water cooled nodes
(front view)



iDataplex DWC Rack
w/ water cooled nodes
(rear view of water manifolds)

IBM iDataplex dx360 M4



Torsten Bloth, IBM Lab Services - © IBM Corporation

Cooling Infrastructure

Photos: StBAM2 (staatl. Hochbauamt München 2)

Why Is the Infrastructure Important?

- Determines data center overheads
  - Matching average operating power consumption with cooling infrastructure will reduce overheads

- Infrastructure limits the possible cooling technologies for the HPC systems
  - Being set up only for air cooling will not allow an easy switch to water cooled systems

- Trade-offs to reduced overhead can be a source for additional costs later on
  - Switching of power conditioning to reduce overheads can allow brown-outs to shutdown or damage system parts

- Mistakes made here can be costly in the long run
  - HPC system replaced every 3-5 years
  - Infrastructure replaced every 10-20 years

All The Way to Earthquake Safety
RIKEN's K Computer


WHY?

Wide Range of HPC Application Spaces

Climate modeling

Weather forecasting

Nuclear Physics

Oil and Gas
- Reservoir modeling

Bioscience, Medicine
- Genomic research

Material Science

Automobile/Aeronautics Industry
- CFD
- Virtual Crashtests

City planning

Graph analysis
- Security application

Finance

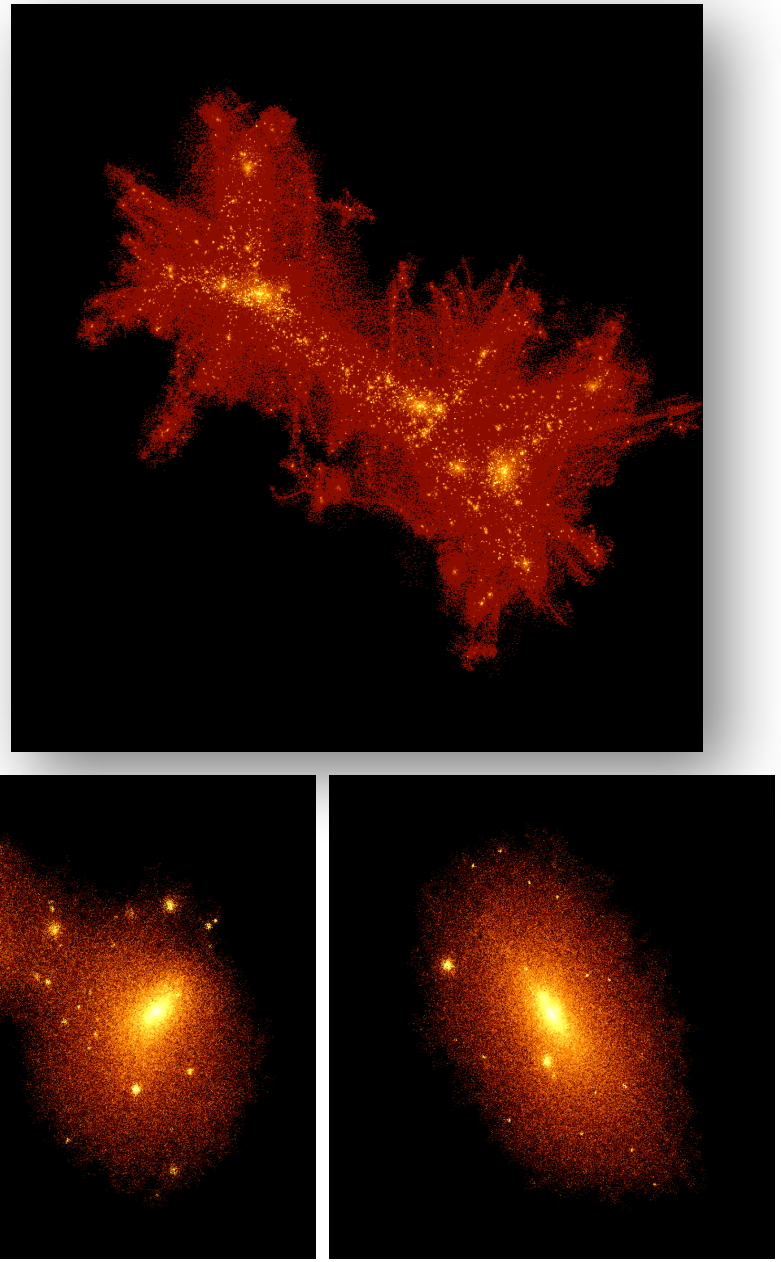
Astrophysics: Simulation of Galaxies

Example / Astrophysikalisches Institut Potsdam

Goal: Study the formation and movement of the non-linear dynamics of galaxies and galaxy clusters

Methods: Higly efficient parallel adaptive refinement Tree (ART) code, which allows to study the Influence of small changes in the early universe until today.

High resolution is for precise computations necessary. Only the largest supercomputers can be used for such simulations. Most important is the amount of main memory.

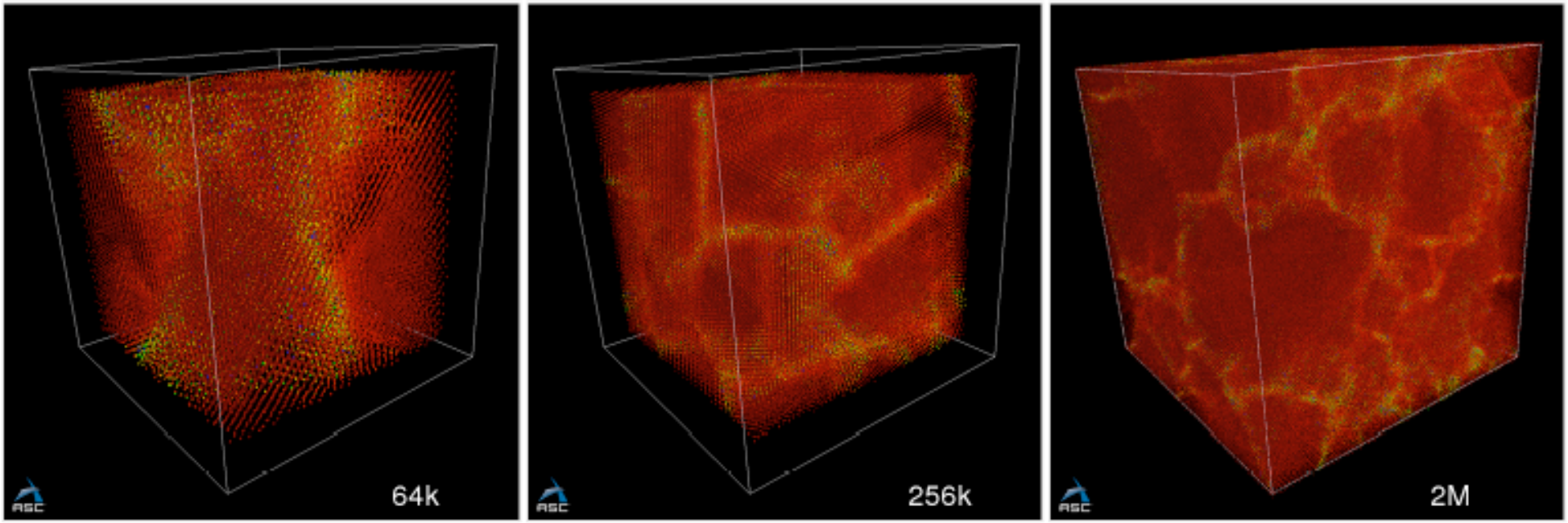


Detail of a cosmological simulation of galaxy formation, credit: S. Gottlöber, A. Klypin, A.Kravtsov

Material Solidification Process

Molecular dynamics: 2 Million atom run (2005)

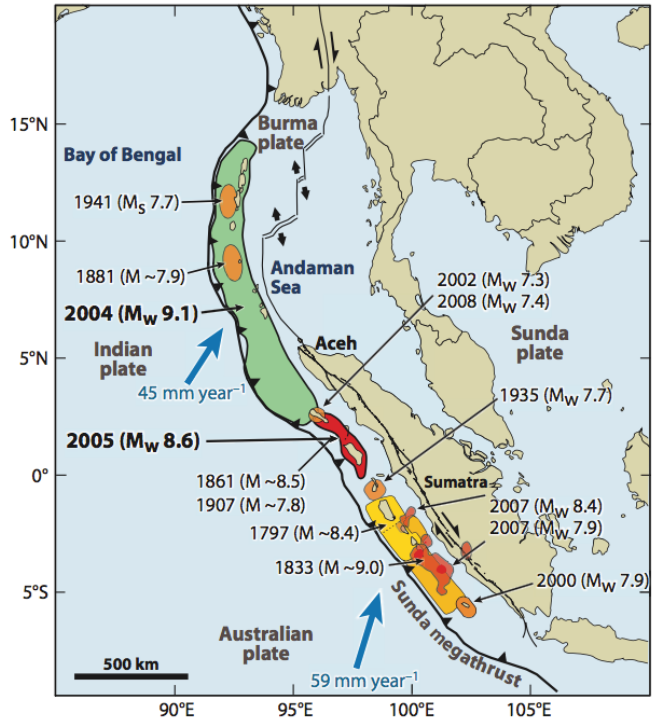
- Used all of Blue Gene/L (128K cores)
- Close to perfect scaling
- New scientific observations

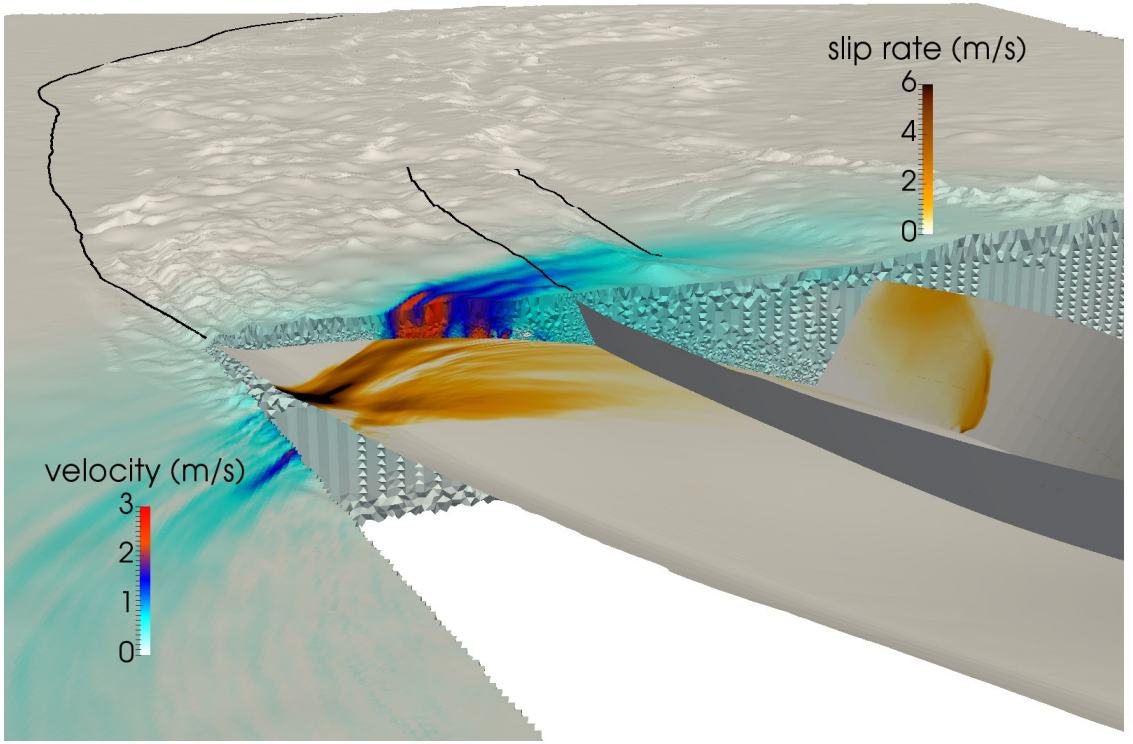


64,000 atoms 256,000 atoms 2,048,000 atoms

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

Example – Simulation of the Sumatra Earthquake





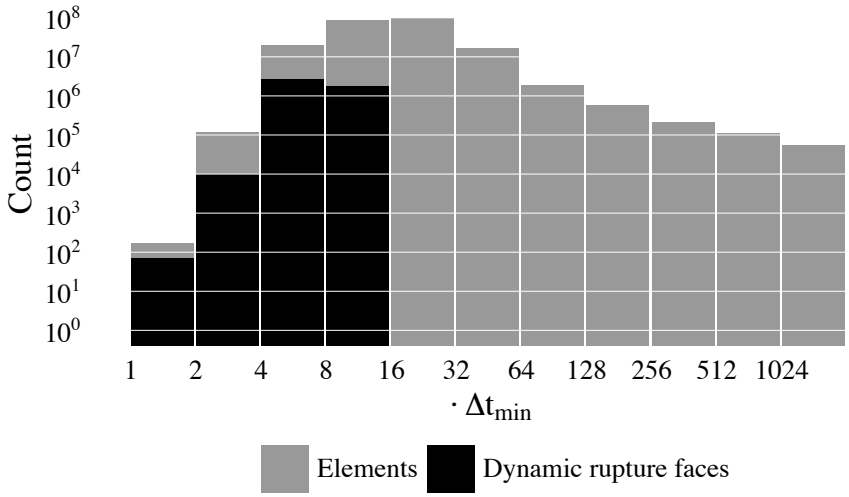
Uphoff et al.: Extreme Scale Multi-Physics Simulations of the Tsunamigenic 2004 Sumatra Megathrust Earthquake, SC17

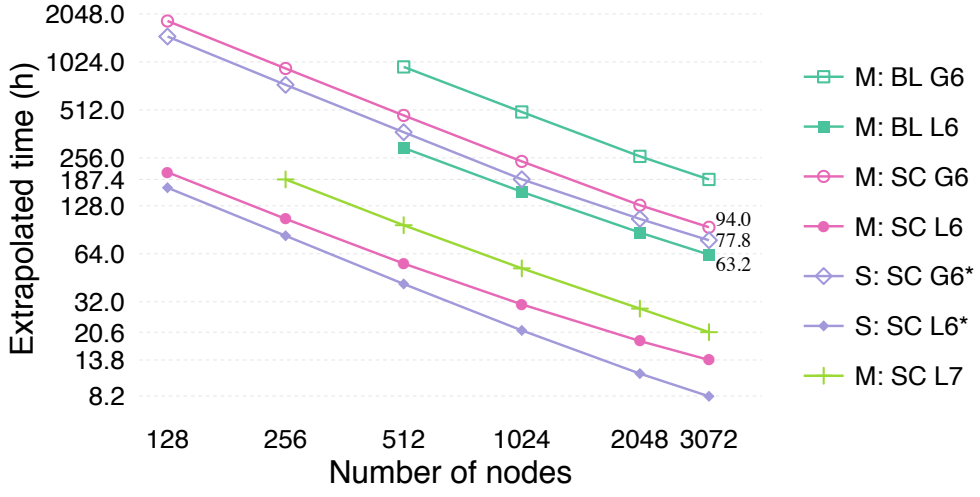
- simulates rupture process and seismic wave propagation
- high-order discontinuous Galerkin method on an adaptive unstructured tetrahedral mesh

Example – Simulation of the Sumatra Earthquake

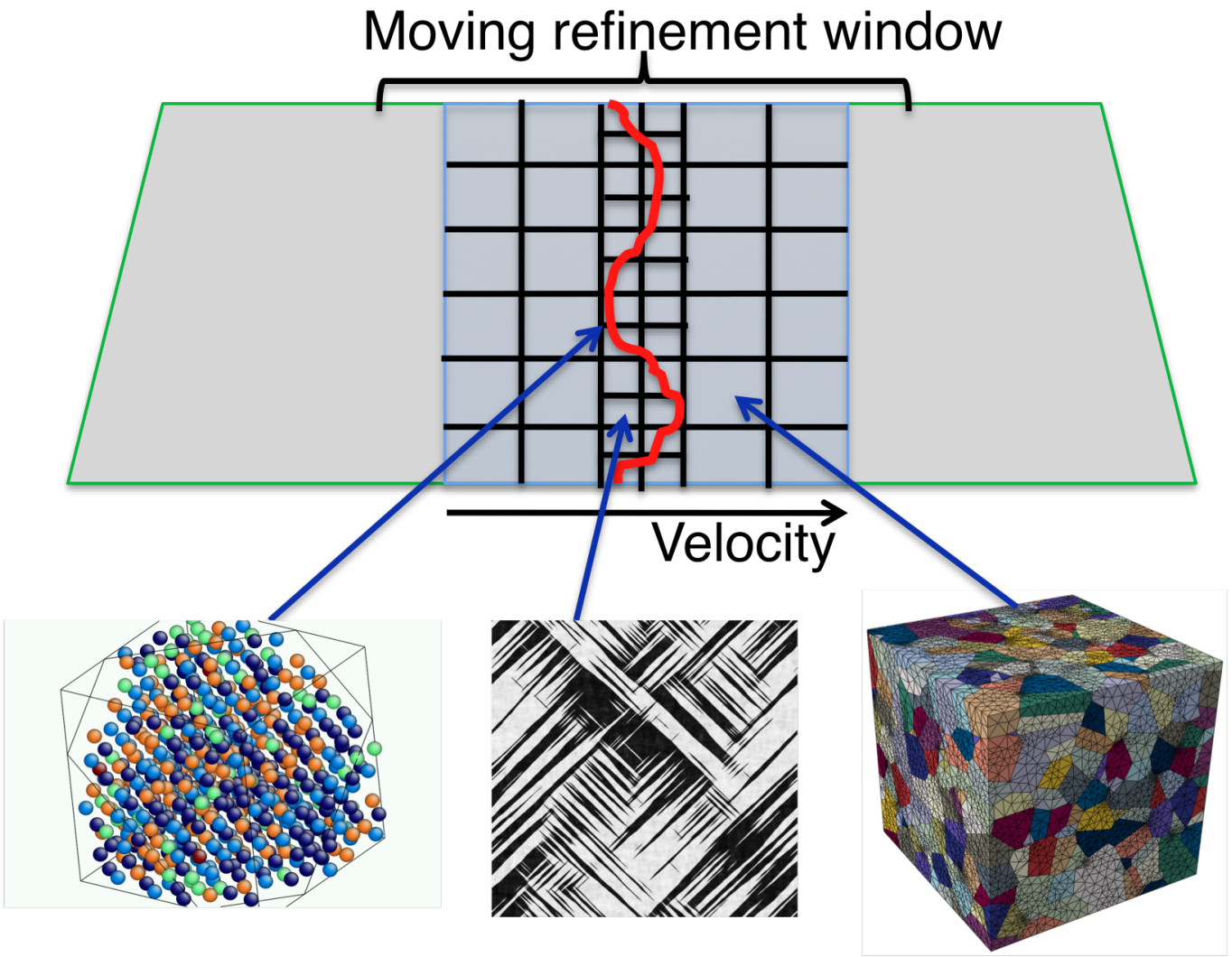
HPC-Related Data:

- 221 Mio grid cells, more than 10^{11} unknowns
- more than 3 Mio time steps on the smallest grid cells (11 clusters with $\times$2-increasing timestep size
- production run: 13.9h on SuperMUC phase 2(86,016 cores)
- 13 TB checkpoint data, 2.8 TB for post-processing (asynchronous IO; costs entirely overlapped by computation)





Scale Bridging Example: Material Science



ExMatEx

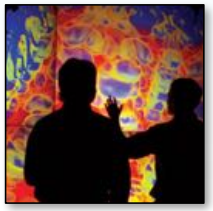
Source: Jim Belak

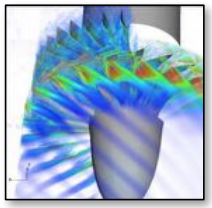
US Exascale / ECP Application Targets

Capable exascale system applications will deliver broad coverage of 6 strategic pillars

National security

Stockpile stewardship





Energy security

Turbine wind plant efficiency

Design and commercialization of SMRs

Nuclear fission and fusion reactor materials design

Subsurface use for carbon capture, petro extraction, waste disposal

High-efficiency, low-emission combustion engine and gas turbine design

Carbon capture and sequestration scaleup

Biofuel catalyst design

Economic security

Additive manufacturing of qualifiable metal parts

Urban planning

Reliable and efficient planning of the power grid

Seismic hazard risk assessment

Scientific discovery

Cosmological probe of the standard model of particle physics

Validate fundamental laws of nature

Plasma wakefield accelerator design

Light source-enabled analysis of protein and molecular structure and design

Find, predict, and control materials and properties

Predict and control stable ITER operational performance

Demystify origin of chemical elements

Earth system

Accurate regional impact assessments in Earth system models

Stress-resistant crop analysis and catalytic conversion of biomass-derived alcohols

Metagenomics for analysis of biogeochemical cycles, climate change, environmental remediation

Health care

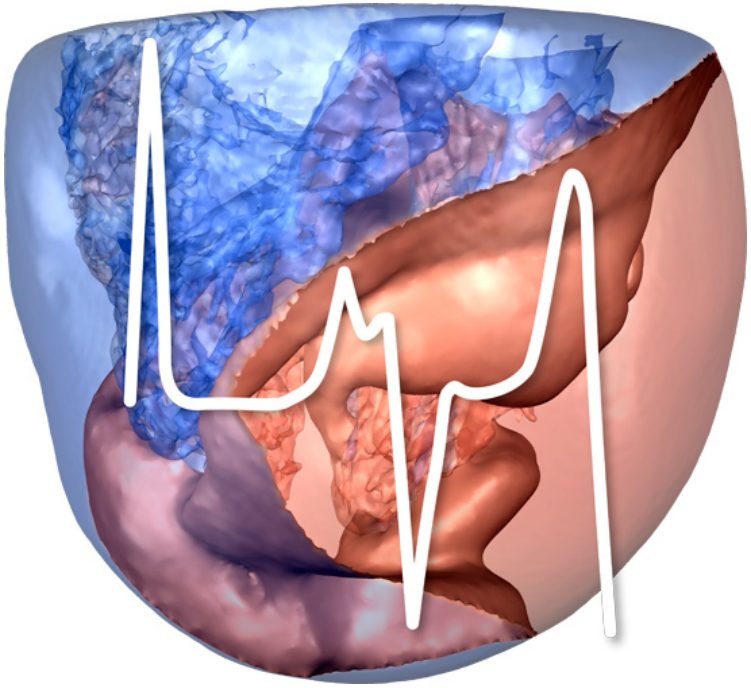
Accelerate and translate cancer research

EXASCALE COMPUTING PROJECT

Source: P. Messina, ECP

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

Summary

Covered remaining topics for OpenMP
- Tasking to allow for more programmability
- OpenMP memory model and flush directive
- Device construct covered later

Distributed Memory Architectures
- Highly scalable to thousands (or more) nodes
- Network architecture and topology play an important role
- Facilities are becoming more important as we scale

Architectural choice has significant impact on programmability and usability
- Need a different programming model with explicit communication
- Need system software to tie nodes together
- Access to such a shared environment is more controlled

HPC has a wide range of application areas

Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

Seminar Announcement
David Montoya, Los Alamos National Laboratory

JUNE 1, 2018 | 14:30-15:30 | LRZ, SEMINARRAUM 2

„UNITED STATES EXASCALE COMPUTING PROJECT (ECP) OVERVIEW AND STATUS WITH A FOCUS ON FACILITY INTEGRATION“

THIS TALK WILL BE PRESENTED IN ENGLISH

Joint special lecture hosted by LRZ and TUM Chair I10 (Computer Architecture and Parallel Systems)