HPC - HIGH PERFORMANCE COMPUTING
(SUPERCOMPUTING)

DISTRIBUTED OPERATING SYSTEMS, SCALABILITY, SS 2014

Hermann Härtig
Understand

- Systems Software for “High Performance Computing” (HPC), today & expected
- MPI as a common programming model
- What is “noise”?
- How to use incomplete information for informed decisions
- Advanced Load Balancing techniques (heuristics)
Characteristics of MPP Systems:

- Highly optimised interconnect networks
- Distributed memory
- Size today: few 100000 CPUs (cores) + XXL GPU

Successful Applications:

- CPU intensive computation, massively parallel Applications, small execution/communication ratios, weak and strong scaling
- Cloud ?

Not used for:

- Transaction-management systems
- Unix-Workstation + Servers
Characteristics of Cluster Systems:

- Use COTS (common off the shelf) PCs/Servers and COTS networks
- Size: No principle limits

Successful Applications:

- CPU intensive computation, massively parallel Applications, larger execution/communication ratios, weak scaling
- Data Centers, google apps
- Cloud, Virtual Machines

Not used for:

- Transaction-management system
- Michael Flynn (1966):
  SISD, SIMD, MIMD, (MISD)SIMD

- SPMD: Single Program Multiple Data
  Same program runs on “all” nodes
  works on split-up data
  asynchronously but with explicit synch points
  implementations: message passing/shared memory/...paradigms: “map/reduce” (google) / GCD (apple) / task queues / ...

- often: while (true) {  work;  exchange data (barrier)}
DIVIDE AND CONQUER

node 1

CPU #1
CPU #2

node 2

CPU #1
CPU #2

Hermann Härtig, TU Dresden, Distributed OS, Parallel Systems
DIVIDE AND CONQUER

node 1

node 2

CPU #1

CPU #2

part 1

part 2

part 3

part 4

CPU #1

CPU #2
DIVIDE AND CONQUER

Node 1

CPU #1

CPU #2

Part 1

Part 2

Node 2

CPU #1

CPU #2

Part 3

Part 4
DIVIDE AND CONQUER

node 1

result 1

CPU #1

CPU #2

result 2

node 2

result 3

CPU #1

result 4

CPU #2
DIVIDE AND CONQUER

node 1

CPU #1

result 1

result 2

result 3

result 4

node 2

CPU #1

CPU #2
DIVIDE AND CONQUER

node 1

CPU #1

CPU #2

result

node 2

CPU #1

CPU #2
IMBALANCES & FAILURES

Communication

Computation

Communication
IMBALANCES & FAILURES

Communication

Computation

Communication
AMDAHL’S LAW

Compute; communicate; compute; …

- Examples (idealized, take with grain of salt !!!):
  - Compute: 10 micro, 100 micro, 1 ms
  - Communicate: 5 micro, 10 micro, 100 micro, 1 ms
    assuming here: communication cannot be sped up

Amdahl's law: $1 / (1-P+P/N)$

- P: section that can be parallelized
- 1-P: serial section
- N: number of CPUs
AMDAHL’S LAW

Compute( = parallel section),
communicate( = serial section)

→

possible speedup for \( N = \infty \)

- \( 1\text{ms}, 100\ \mu\text{s}: \quad 1/0.1 \quad \rightarrow \quad 10 \)
- \( 1\text{ms}, 1\ \mu\text{s}: \quad 1/0.001 \quad \rightarrow \quad 1000 \)
- \( 10\ \mu\text{s}, 1\ \mu\text{s}: \quad 0.01/0.001 \quad \rightarrow \quad 10 \)
- ...
WEAK VS. STRONG SCALING

Strong:
- accelerate same problem size

Weak:
- extend to larger problem size
AMDAHL’S LAW

Jitter, “Noise”, “micro scrabblers”:

- Occasional addition to computation/communication time in one or more processes
- Holds up all other processes

Compute( = parallel section), jitter (→ add to serial section), communicate( = serial section):
possible speedup for $N=\infty$

- $1\text{ms, }100\mu\text{s, }100\ \mu\text{s}$: $1/0.2 \rightarrow 5 \ (10)$
- $1\text{ms, }100\mu\text{s, }1\ \mu\text{s}$: $1/0.101 \rightarrow 10 \ (1000)$
- $10\ \mu\text{s, }10\mu\text{s, }1\ \mu\text{s}$: $0.01/0.011 \rightarrow 1 \ (10)$
STATE OF THE ART IN HPC
STATE OF THE ART IN HPC

- dedicate full partition to application (variant: “gang scheduling”)
- load balancing done (tried) by applications or user-level runtime (Charm++)
- avoid OS calls
- “scheduler”: manages queue of application processes assigns partitions to applications supervises run-time
- applications run from checkpoint to checkpoint
- nodes access remote memory via load/store operations
- busy waiting across nodes (within partition)
- barrier ops supported by network
- compare&exchange on remote memory operation
- no OS calls for message ops (busy waiting)
Library for message-oriented parallel programming

Programming model:
- Multiple instances of same program
- Independent calculation
- Communication, synchronization
DIVIDE AND CONQUER

node 1

CPU #1

CPU #2

problem

node 2

CPU #1

CPU #2
DIVIDE AND CONQUER

node 1

CPU #1

CPU #2

part 1

part 2

part 3

part 4

node 2

CPU #1

CPU #2
DIVIDE AND CONQUER

node 1

CPU #1

part 1

CPU #2

part 2

node 2

CPU #1

part 3

CPU #2

part 4
DIVIDE AND CONQUER

node 1

CPU #1

result 1

CPU #2

result 2

node 2

CPU #1

result 3

CPU #2

result 4
DIVIDE AND CONQUER

node 1

CPU #1

CPU #2

result 1

result 2

result 3

result 4

node 2

CPU #1

CPU #2
DIVIDE AND CONQUER

node 1

CPU #1

CPU #2

result

node 2

CPU #1

CPU #2

Hermann Härtig, TU Dresden, Distributed OS, Parallel Systems
MPI program is started on all processors

- `MPI_Init()`,
- `MPI_Finalize()`

Communicators (e.g., `MPI_COMM_WORLD`)

- `MPI_Comm_size()`
- `MPI_Comm_rank()`: “Rank” of process within this set

Typed messages

Dynamically create and spread processes using `MPI_Spawn()` (since MPI-2)
Communication
MPI EXECUTION

- Communication
- Point-to-point

```c
MPI_Send(
    void* buf,
    int count,
    MPI_Datatype,
    int dest,
    int tag,
    MPI_Comm comm
)
```
MPI EXECUTION

- Communication
- Point-to-point

```c
MPI_Recv(
    void* buf,
    int count,
    MPI_Datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status *status)
```
MPI EXECUTION

- Communication
  - Point-to-point
  - Collectives

```c
MPI_Bcast(
    void* buffer,
    int count,
    MPI_Datatype,
    int root,
    MPI_Comm comm
)
```
MPI EXECUTION

- Communication
- Point-to-point
- Collectives

```c
MPI_Reduce(
    void* sendbuf,
    void *recvbuf,
    int count
    MPI_Datatype,
    MPI_Op op,
    int root,
    MPI_Comm comm
)
```
MPI EXECUTION

- Communication
  - Point-to-point
  - Collectives
- Synchronization
MPI EXECUTION

- Communication
  - Point-to-point
  - Collectives
- Synchronization
- Test

MPI_Test(
  MPI_Request* request,
  int *flag,
  MPI_Status *status
)
MPI EXECUTION

- Communication
  - Point-to-point
  - Collectives
- Synchronization
  - Test
  - Wait

MPI_Wait(
  MPI_Request* request,
  MPI_Status *status
)
MPI EXECUTION

- Communication
  - Point-to-point
  - Collectives

- Synchronization
  - Test
  - Wait
  - Barrier

```c
MPI_Barrier(
  MPI_Comm comm
)
```
BLOCK AND SYNC

- blocking call
- non-blocking call

- synchronous communication
- asynchronous communication
BLOCK AND SYNC

<table>
<thead>
<tr>
<th>synchronous communication</th>
<th>blocking call</th>
<th>returns when message has been delivered</th>
</tr>
</thead>
<tbody>
<tr>
<td>asynchronous communication</td>
<td>non-blocking call</td>
<td></td>
</tr>
<tr>
<td>synchronous communication</td>
<td>non-blocking call</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>blocking call</td>
<td>returns when message has been delivered</td>
<td></td>
</tr>
<tr>
<td>asynchronous communication</td>
<td>returns when send buffer can be reused</td>
<td></td>
</tr>
<tr>
<td>Synchronous Communication</td>
<td>Blocking Call</td>
<td>Non-Blocking Call</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------</td>
<td>-------------------</td>
</tr>
<tr>
<td></td>
<td>returns when message has been delivered</td>
<td>returns immediately, following test/wait checks for delivery</td>
</tr>
<tr>
<td>Asynchronous Communication</td>
<td>returns when send buffer can be reused</td>
<td></td>
</tr>
</tbody>
</table>
## BLOCK AND SYNC

<table>
<thead>
<tr>
<th></th>
<th>blocking call</th>
<th>non-blocking call</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>synchronous communication</strong></td>
<td>returns when message has been delivered</td>
<td>returns immediately, following test/wait checks for delivery</td>
</tr>
<tr>
<td><strong>asynchronous communication</strong></td>
<td>returns when send buffer can be reused</td>
<td>returns immediately, following test/wait checks for send buffer</td>
</tr>
</tbody>
</table>
int rank, total;
MPI_Init();
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &total);

MPI_Bcast(...);
/* work on own part, determined by rank */

if (id == 0) {
    for (int rr = 1; rr < total; ++rr)
        MPI_Recv(...);
    /* Generate final result */
} else {
    MPI_Send(...);
}
MPI_Finalize();
- Interposition layer between library and application
- Originally designed for profiling
- Interposition layer between library and application
- Originally designed for profiling
EXA-SCALE: HW+SW ASSUMPTIONS

- Large number of nodes:
  - Many compute cores
  - 1 or 2 service cores
- Failure rate exceeds checkpoint rate
- Fast local persistent storage on each node
- Not all cores available all the time (dark silicon due to heat/energy issues)
- Compute + communication heavy applications, may not be balanced
- Short term changes of frequency?
for applications with extreme (bad) computation/communications ratio:
NOT MUCH, but
-> avoid “noise”, use common sense

all others:
handle faults
use dark silicon
balance load
gossip
over decomposition & over subscription
predict execution times
use scheduling tricks
optimise for network/memory topology
Use common sense to avoid:

- OS usually not directly on the critical path, BUT OS controls: interference via interrupts, caches, network, memory bus, (RTS techniques)
- avoid or encapsulate side activities
- small critical sections (if any)
- partition networks to isolate traffic of different applications (HW: Blue Gene)
- do not run Python scripts or printer daemons in parallel
+ Hebrew Uni (Mosix team) + ZIB (FS team)
Fast and Fault-Tolerant Microkernel-based OS

- get rid of partitions
- use a micro-kernel (L4)
- OS supported load balancing
- use RAID for fast checkpoints

DFG-supported
Microkernels, virtualization, split architectures

MOSIX-style online system management (gossip)

Distributed in-memory (on-node) checkpointing

MPI + applications
GOAL FOR EXASCALE HPC

Many-core Node

Hermann Härtig, TU Dresden, Distributed OS, Parallel Systems
GOAL FOR EXASCALE HPC

Many-core Node
Many-core Node
GOAL FOR EXASCALE HPC

Many-core Node
SMALL? PREDICTABLE?
MOSIX: LOAD BALANCING

Hermann Härtig, TU Dresden, Distributed OS, Parallel Systems
REDUNDANT CHECKPOINT
REDUNDANT CHECKPOINT
REDUNDANT CHECKPOINT
REDUNDANT CHECKPOINT

Hermann Härtig, TU Dresden, Distributed OS, Parallel Systems
REDUNDANT CHECKPOINT
REDUNDANT CHECKPOINT
REDUNDANT CHECKPOINT
REDUNDANT CHECKPOINT
REDUNDANT CHECKPOINT
EXPERIMENTS: IMBALANCES, OVERDECOMPOSITION AND OVERSUBSCRIPTION
TOWARDS BALANCING

MPI ranks

time

Barrier
Imbalance in application workload
Reassign work to react to node failure
SPLITTING BIG JOBS

overdecomposition & “oversubscription”
Execute small jobs in parallel (if possible)
Dealing with Load Imbalances

**Unbalanced** compute times of ranks per time step

Application: COSMO-SPECS+FD4

**Balanced** compute times of ranks per time step

Application: COSMO-SPECS+FD4
Application: COSMO-SPECS+FD4

**Unbalanced** compute times of ranks per time step
Application: COSMO-SPECS+FD4
### Oversubscription

<table>
<thead>
<tr>
<th>Time</th>
<th>Non-blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.000 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.500 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.000 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.500 s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Application:** COSMO-SPECS+FD4 (no load balancing)

- Taurus 16 nodes w/ 16 Xeon E5-2690 (Sandy Bridge) @ 2.90GHz
- 1x - 8x oversubscription (256 - 2048 MPI ranks, same problem size)
Application: COSMO-SPECS+FD4 (no load balancing)

- Taurus 16 nodes w/ 16 Xeon E5-2690 (Sandy Bridge) @ 2.90GHz
- 1x - 8x oversubscription (256 - 2048 MPI ranks, same problem size)
Application: COSMO-SPECS+FD4 (no load balancing)

- Taurus 16 nodes w/ 16 Xeon E5-2690 (Sandy Bridge) @ 2.90GHz
- 1x - 8x oversubscription (256 - 2048 MPI ranks, same problem size)
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)
- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
### Oversubscription

<table>
<thead>
<tr>
<th>Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000 s</td>
<td>256 Ranks, 1-4 nodes, orig</td>
</tr>
<tr>
<td>800 s</td>
<td>256 Ranks, 1-4 nodes, patched</td>
</tr>
<tr>
<td>600 s</td>
<td>Approximate linear scale</td>
</tr>
<tr>
<td>400 s</td>
<td></td>
</tr>
<tr>
<td>200 s</td>
<td></td>
</tr>
<tr>
<td>0 s</td>
<td></td>
</tr>
</tbody>
</table>

Oversubscription factor (fewer cores)

Application: COSMO-SPECS+FD4 (no load balancing)
- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)
- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
Application: COSMO-SPECS+FD4 (no load balancing)

- ATLAS nodes w/ 64 AMD Opteron 6274 cores @ 2.2 GHz
- Number of ranks remained constant, but number of cores was reduced
EXPERIMENTS:
GOSSIP SCALABILITY
Distributed Bulletin Board

- Each node keeps vector with per-node info (own + info received from others)
- Once per time step, each node sends to 1 other randomly selected node a subset of its own vector entries (called “window”)
- Node merges received window entries into local vector (if newer)
MOSIX: GOSSIP ALGORITHM

Each time unit:
- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node
Each time unit:

• Update local info

• Find all vector entries up to age T (called a window)

• Send window to 1 randomly selected node
**Each time unit:**
- Update local info
- Find all vector entries up to age $T$ (called a window)
- Send window to 1 randomly selected node

**Upon receiving a window:**
- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received
**Each time unit:**
- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node

**Upon receiving a window:**
- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received
Each time unit:

- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node

Upon receiving a window:

- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received
Each time unit:
- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node

Each window:
- A:0, B:12, C:2, D:4, E:11...

Upon receiving a window:
- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received

Each window:
- A:1, B:2, C:3, D:3, E:0...

Each window:
- A:1, C:3, D:5...

Hermann Härtig, TU Dresden, Distributed OS, Parallel Systems
Table 1: Average vector age (relative to the unit of time) for window sizes ranging from 10% to 100% of the number of nodes.

<table>
<thead>
<tr>
<th>Window size (rel. to node count)</th>
<th>1024 Nodes</th>
<th>2048 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>7.01</td>
<td>7.71</td>
</tr>
<tr>
<td>20%</td>
<td>7.18</td>
<td>7.87</td>
</tr>
<tr>
<td>30%</td>
<td>7.09</td>
<td>7.78</td>
</tr>
<tr>
<td>40%</td>
<td>7.03</td>
<td>7.73</td>
</tr>
<tr>
<td>50%</td>
<td>7.09</td>
<td>7.78</td>
</tr>
<tr>
<td>60%</td>
<td>7.09</td>
<td>7.78</td>
</tr>
<tr>
<td>70%</td>
<td>7.09</td>
<td>7.78</td>
</tr>
<tr>
<td>80%</td>
<td>7.09</td>
<td>7.78</td>
</tr>
<tr>
<td>90%</td>
<td>7.09</td>
<td>7.78</td>
</tr>
<tr>
<td>100%</td>
<td>7.09</td>
<td>7.78</td>
</tr>
</tbody>
</table>
### Circulating among colony nodes

<table>
<thead>
<tr>
<th>Colony nodes</th>
<th>Method</th>
<th>Windows not exceeding age</th>
<th>Whole vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>Approx.</td>
<td>19.15</td>
<td>6.00</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>18.87</td>
<td>6.04</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>18.75</td>
<td>5.99</td>
</tr>
<tr>
<td>256</td>
<td>Approx.</td>
<td>36.49</td>
<td>8.49</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>36.33</td>
<td>8.57</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>36.06</td>
<td>8.55</td>
</tr>
<tr>
<td>512</td>
<td>Approx.</td>
<td>71.15</td>
<td>13.27</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>71.01</td>
<td>13.34</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>70.85</td>
<td>13.37</td>
</tr>
<tr>
<td>1K</td>
<td>Approx.</td>
<td>140.44</td>
<td>22.69</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>139.76</td>
<td>22.73</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>140.14</td>
<td>22.83</td>
</tr>
<tr>
<td>2K</td>
<td>Approx.</td>
<td>279.03</td>
<td>41.47</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>267.82</td>
<td>41.58</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>278.94</td>
<td>41.66</td>
</tr>
<tr>
<td>4K</td>
<td>Approx.</td>
<td>556.20</td>
<td>78.99</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>479.96</td>
<td>79.10</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>556.20</td>
<td>79.39</td>
</tr>
<tr>
<td>8K</td>
<td>Approx.</td>
<td>1,110.53</td>
<td>154.02</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>798.97</td>
<td>153.80</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>1,102.99</td>
<td>155.16</td>
</tr>
</tbody>
</table>

Circulating among colony nodes and windows not exceeding age. The average age of the whole vector is shown in the last column.
To approximate the average age of the vectors when colonies c 

3.2 Average vector age 

runs, each lasted 100 units of time after reaching a steady st 

no other processes were running in the nodes.

For each colony size, the third row in Table 1 shows the averag 

\[ W(n) = \frac{1}{n} \sum_{i=1}^{n} T(n) \]

where 

\[ T(n) = \ln(n) \]

Nodes: Vector Age

<table>
<thead>
<tr>
<th>Colony nodes</th>
<th>Method</th>
<th>Circulating among colony nodes</th>
<th>whole vector</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>windows not exceeding age</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>Approx.</td>
<td>19.15</td>
<td>6.00</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>18.87</td>
<td>6.04</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>18.75</td>
<td>5.99</td>
</tr>
<tr>
<td>256</td>
<td>Approx.</td>
<td>36.49</td>
<td>8.49</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>36.33</td>
<td>8.57</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>36.06</td>
<td>8.55</td>
</tr>
<tr>
<td>512</td>
<td>Approx.</td>
<td>71.15</td>
<td>13.27</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>71.01</td>
<td>13.34</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>70.85</td>
<td>13.37</td>
</tr>
<tr>
<td>1K</td>
<td>Approx.</td>
<td>140.44</td>
<td>22.69</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td>139.76</td>
<td>22.73</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>140.14</td>
<td>22.83</td>
</tr>
<tr>
<td>2K</td>
<td>Approx.</td>
<td>279.03</td>
<td>41.47</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td><strong>267.82</strong></td>
<td>41.58</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>278.94</td>
<td>41.66</td>
</tr>
<tr>
<td>4K</td>
<td>Approx.</td>
<td>556.20</td>
<td>78.99</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td><strong>479.96</strong></td>
<td>79.10</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>556.20</td>
<td>79.39</td>
</tr>
<tr>
<td>8K</td>
<td>Approx.</td>
<td>1,110.53</td>
<td>154.02</td>
</tr>
<tr>
<td></td>
<td>Simulation</td>
<td><strong>798.97</strong></td>
<td>153.80</td>
</tr>
<tr>
<td></td>
<td>Measured</td>
<td>1,102.99</td>
<td>155.16</td>
</tr>
<tr>
<td>1M</td>
<td>Approx.</td>
<td>141,911</td>
<td>19,209</td>
</tr>
<tr>
<td>1G</td>
<td>Approx.</td>
<td>145M</td>
<td>19M</td>
</tr>
</tbody>
</table>
Problem: average age or window sizes too big for extreme numbers of nodes
MASTER: GLOBAL VIEW
MASTER: GLOBAL VIEW
SYSTEM ARCHITECTURE
L4 MICRO KERNELS

- apps
- commodity OS

- critical application

- Auth
- displa
- IO

- L4/Re
- L4
“Merkel Phone“
MOSIX MIGRATION

Home Node

OS Virtualization Layer

Linux Kernel

App (local)

Remote Node

OS Virtualization Layer

Linux Kernel

App (Guest)

MOSIX system call rerouting
Randomized Gossip

Distributed Bulletin Board

- Each node keeps a vector with per-node info (own + info received from others)
- Once per time step, each node sends to 1 other randomly selected node a subset of its own vector entries (called “window”)
- Node merges received window entries into local vector (if newer)
Distributed Bulletin Board

- Each node keeps vector with per-node info (own + info received from others)
- Once per time step, each node sends to 1 other randomly selected node a subset of its own vector entries (called “window”)
- Node merges received window entries into local vector (if newer)
Each time unit:
- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node
Each time unit:

- Update local info
- Find all vector entries up to age $T$ (called a window)
- Send window to 1 randomly selected node
Each time unit:

- Update local info
- Find all vector entries up to age $T$ (called a window)
- Send window to 1 randomly selected node

Upon receiving a window:

- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received
Each time unit:
- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node

Upon receiving a window:
- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received
Each time unit:

- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node

Upon receiving a window:

- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received
Each time unit:

- Update local info
- Find all vector entries up to age T (called a window)
- Send window to 1 randomly selected node

Upon receiving a window:

- Update the received entries’ age (+1 for transfer)
- Update entries in local vector where newer information has been received
3.2 Security

Security is of paramount importance for storage systems, as it protects the privacy of individual users and keeps data safe from unauthorized manipulation in the face of shared resources and inherently insecure environments. Relevant aspects of the security architecture include the authentication of users, the authorization of accesses and the encryption of messages and data.

3.2.1 Authentication

XtreemFS clients and servers are not required to run in a trusted environment. Clients running on any machine may access any XtreemFS installation that is reachable over the network. Consequently, servers cannot assume that clients are inherently trustworthy, nor can clients assume that servers are trustworthy.

To solve the problem, XtreemFS supports SSL connections between all clients and servers. When establishing a new server connection, e.g., in the course of mounting a volume or initially writing a file, clients and servers exchange X.509 certificates to ensure a mutual authentication. The distinguished name of a client certificate reflects the identity of the user on behalf of whom subsequent operations are executed. User and group IDs are thus unforgeable and allow for a secure authentication of individual users.

3.2.2 Authorization

A complementary issue is the assignment and evaluation of access rights. XtreemFS offers a common POSIX authorization model with different access flags for the owning user, the owning group and all other users. An optional extension are POSIX access control lists (ACLs), which allow the definition of access rights at the granularity of individual users and groups.

File system calls with path names are directed to the MRC, where they can be authorized locally, as the MRC stores all relevant metadata to perform access control.
XTREEMFS: FAST PATH

Client Node

- XtreemFS Client
- MPI App
- Linux
- L4 XtreemFS

Checkpoint Node

- XtreemFS OSD
- L4 XtreemFS
- Linux
- Checkpoint Store

High-Performance Interconnect

Establish fast connection

Open
Write

Hermann Härtig, TU Dresden, Distributed OS, Parallel Systems
Compute Node

- MPI Process Mgr
- MPI Rank (Compute Part)
- MPI Library
- Local MPI SHM Buffer
- L4 Microkernel

PMI
DESIGN CHALLENGES
Fine-grained work splitting for system-supported load balancing?

How to synchronize? RDMA + polling ./ Block?

Gossip + Heuristics for EXASCALE?

Application / system interface? “Yell” for help?

Compute processes, how and where to migrate / reroute communication?

Replication instead of / in addition to checkpoint/restart?

Reuse Linux (device drivers)?
HARDWARE-WISHES

- Perf counters for Network
- fast redirection of messages
- flash on node circumventing FTL
- quick activation of threads without polling