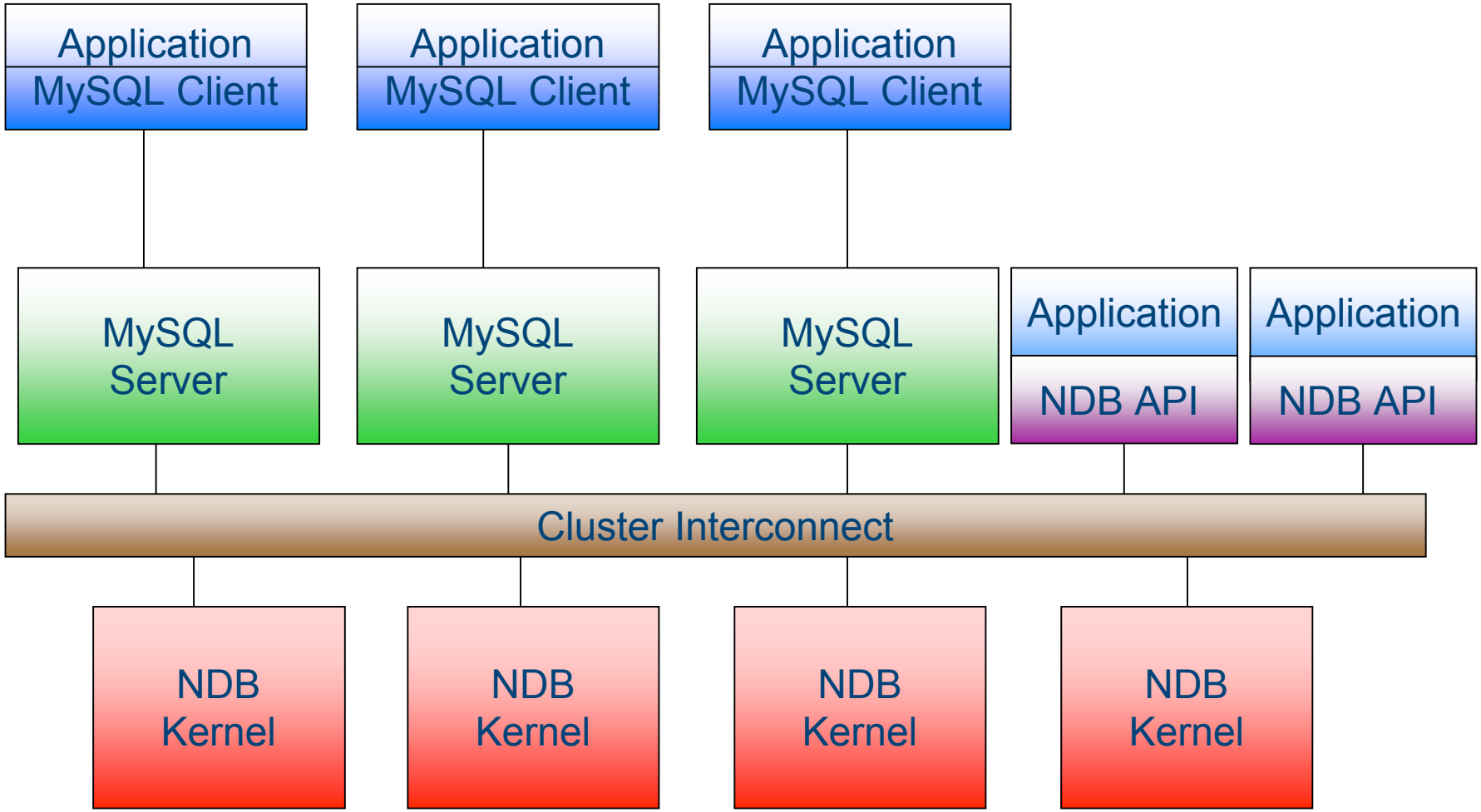
Performance Guide for MySQL Cluster

Mikael Ronström, Ph.D
Senior MySQL Architect
Sun Microsystems

MySQL Cluster



Aspects of Performance

- Response times
- Throughput
- Low variation of response times

Experience Base

- DBT2 (similar to TPC-C) using SQL
- DBT2 using NDB API
- TPC-W
- Prototyping efforts with customers in area of real-time systems
- Loads of benchmarks executed using NDB API

Possible Areas how to Improve Performance

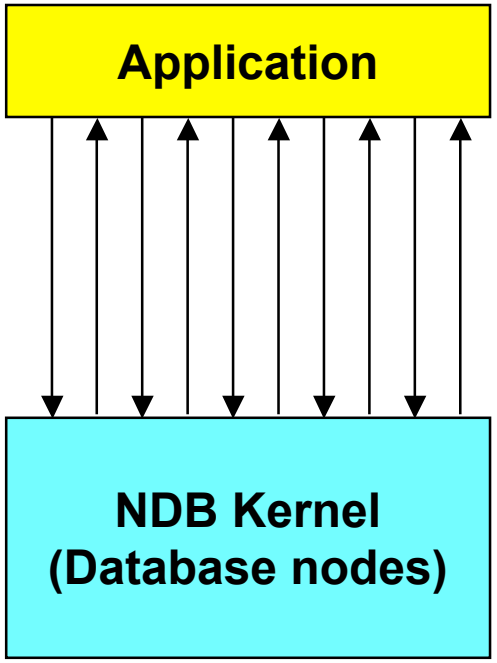
- Use of low level API (NDB API)
- Use of new features in MySQL Cluster Carrier Grade Edition version 6.3 (currently at version 6.3.13)
- Ensure proper partitioning of your Tables
- Use of HW
- Use of features in MySQL Cluster 5.0

Use of low-level NDB API for Application Programming

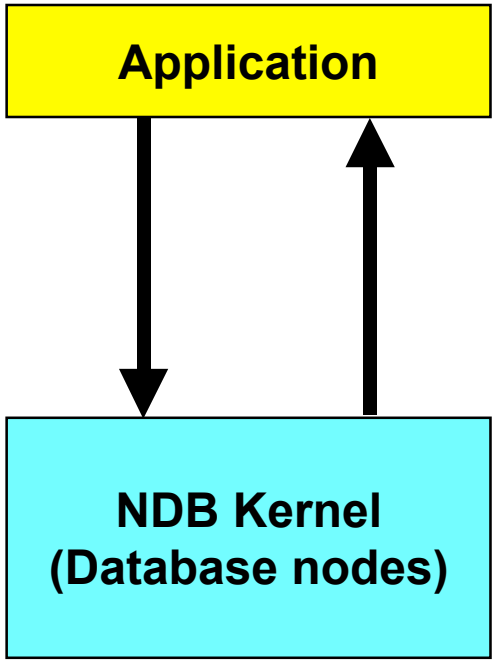
- NDB API is a C++ record access API
- Supports sending parallel record operations within same transaction or in different transactions
- Two modes, synchronous/asynchronous
- Hints to select transaction coordinator
- Simple interpreter for filters and simple additions/subtractions

Looking at performance

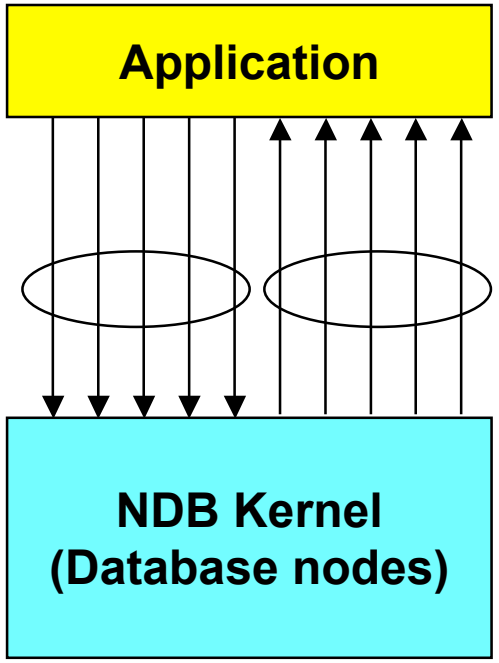
Five synchronous insert transactions (10 x TCP/IP time)



Five inserts in one synchronous transaction (2 x TCP/IP time)



Five asynchronous insert transactions (2 x TCP/IP time)



Example of prototyping using NDB API

- Step 1: Develop prototype using MySQL C API

=> Performance: X, Response time: Y

- Step 2: Develop same functionality using synchronous NDB API

=> Performance: 3X, Response time: ~0.5Y

- Step 3: Develop same functionality using asynchronous NDB API

=> Performance: 6X, Response time: ~0.25Y

Conclusion on when to use NDB API

- When performance is critical
- When real-time response time is critical
- When scalability of application is important (in terms of threads, application nodes, data nodes)

Conclusion on when NOT to use NDB API

- When design time is critical
- When use of standard API's is critical
- For complex queries where it makes sense to let the MySQL optimiser handle writing the query plan

Use of new features in MySQL Cluster Carrier Grade Edition version 6.3.13

- Polling based communication
- Epoll replacing select system call (Linux)
- Send buffer gathering
- Real-time scheduler for threads
- Lock threads to CPU
- Distribution Awareness
- Avoid read before Update/Delete with PK

Polling-based communication

- Avoids wake-up delay in conjunction with new messages
- Avoids interrupt delay for new messages
- Drawback: CPU used heavily also at lower throughput
- Significant response time improvement
- If used in connection with Real-time Scheduling also very reliable response time (e.g. 100% within 3 millisecond response time at fairly high load)

Interrupt Handling in Dolphin SuperSockets

- Dolphin HW has checksums integrated

⇒No interrupt processing required to process Network Protocol

- Interrupt Processing only required to wake sleeping process waiting for events on the Dolphin SuperSockets Socket

Socket Interface to Interrupts

- Interrupts enabled when no data available in select/poll call where timeout is > 0
- Interrupts enabled after blocking receive call with no data available
- Otherwise Interrupts Disabled

=> No interrupts happening when using Polling-based Communication

Polling-based communication Benchmark Results

- Improving performance when CPU isn't limited
- Decrease performance when CPU is limiting factor (e.g. 1 data node per Core)
- 10% performance improvement on 2, 4 and 8 data node clusters using DBT2
- 20% improvement using Dolphin Express

all dump 506 200

(spin for 200 microseconds before going to sleep, will call select(0)/epoll_wait(0) while spinning)

Epoll replacing select system call

- Decreases overhead of select system call in large clusters
- Increases interrupt overhead of Intel e1000 Ethernet driver
- Improved performance 20% on 32-node clusters
- Improved performance of up 10-15% also on smaller clusters where CPU wasn't a bottleneck (together with Polling mode 20% improvement)
- Slight decrease of performance on CPU-limited configurations

  (=1 data node per CPU)

Extra Round of Execution before Sending Messages

- Adapting NDB Scheduler to receive another round of messages and execute them before Sending Messages
- Larger size of Messages Sent

⇒Increases Throughput

⇒Increases Response Time

all dump 502 50

(set all nodes to continue executing until 50 microseconds have passed)

Setting Threads to Real-time

- Use Real-time Scheduling in NDB Kernel
- Maintenance Threads at Higher Priority
- Main Thread lower priority

⇒Avoids decreased priority at high loads

⇒Decreases response time

3 dump 503 1

(set node 3 process in real-time priority)

Locking Threads to CPU's

- Lock Maintenance Threads (Connection Threads, Watch Dog, File System Threads) to a CPU
- Lock Main Thread to a CPU

⇒ No cache thrashing due to moving threads

- Interacting with real-time priority + new scheduler in NDB

⇒ Main Thread owning CPU

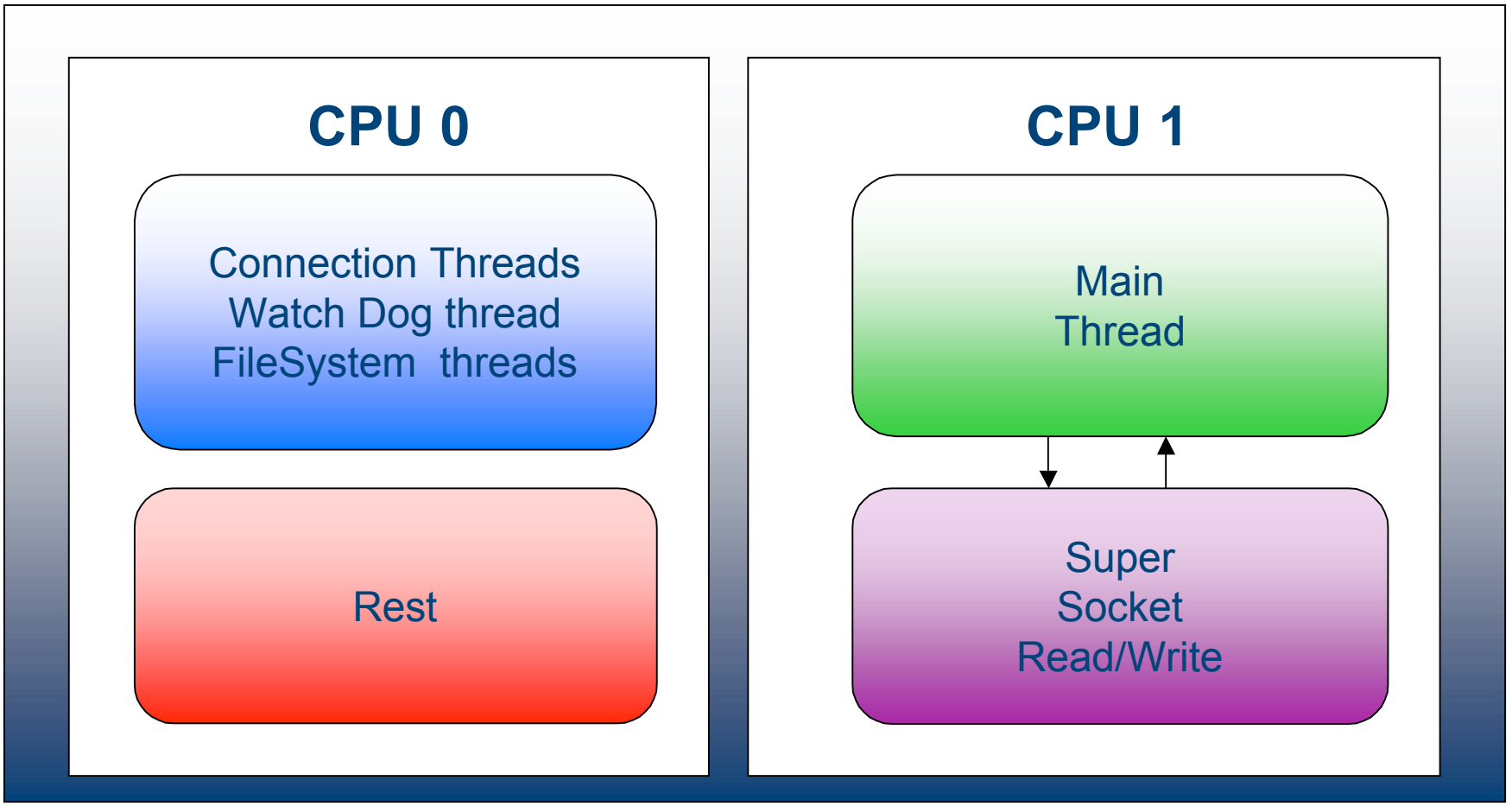
2 dump 505 1

(locks maintenance threads on node 2 to CPU 1)

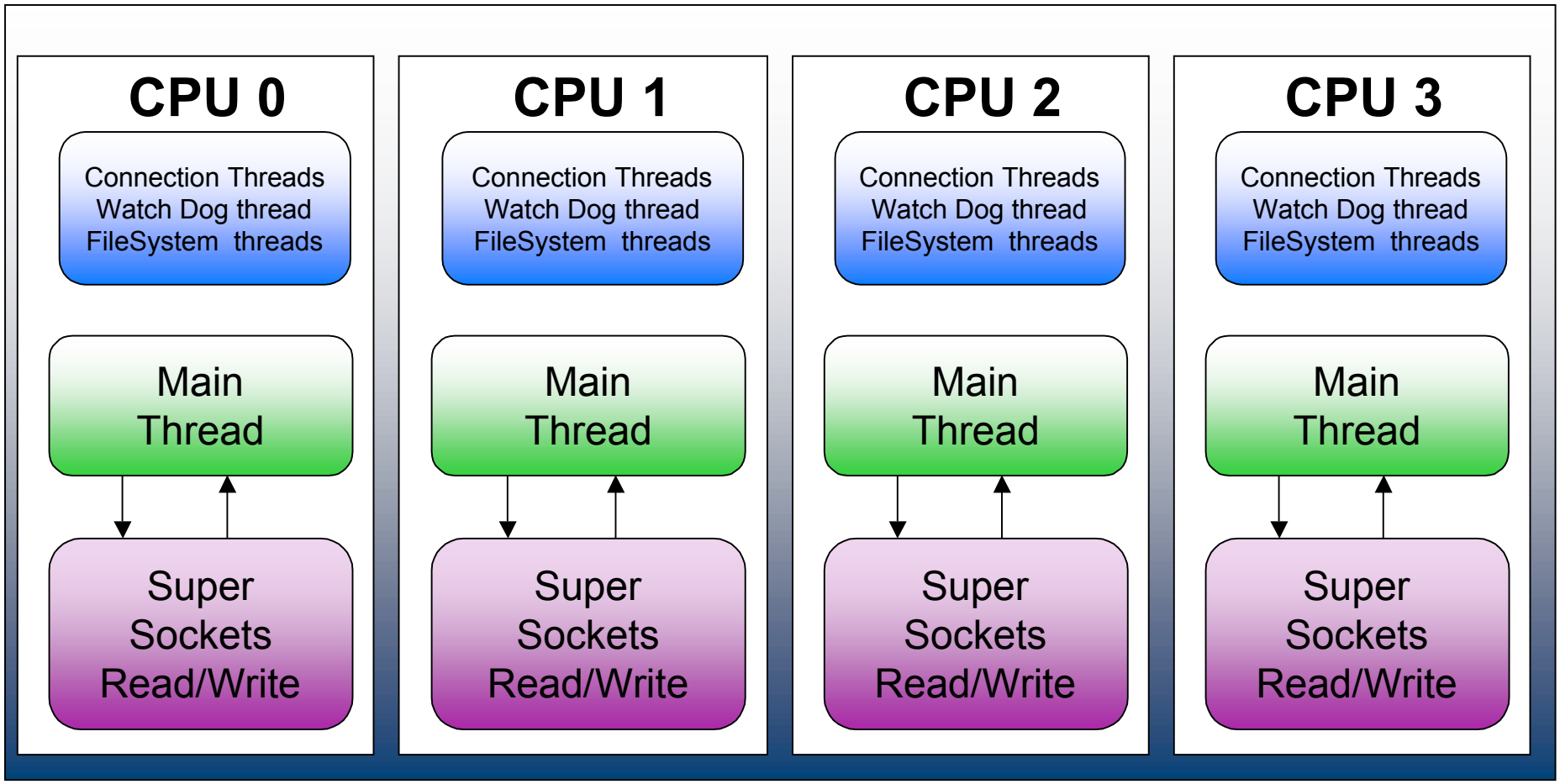
2 dump 504 0

(locks main thread on node 2 to CPU 0)

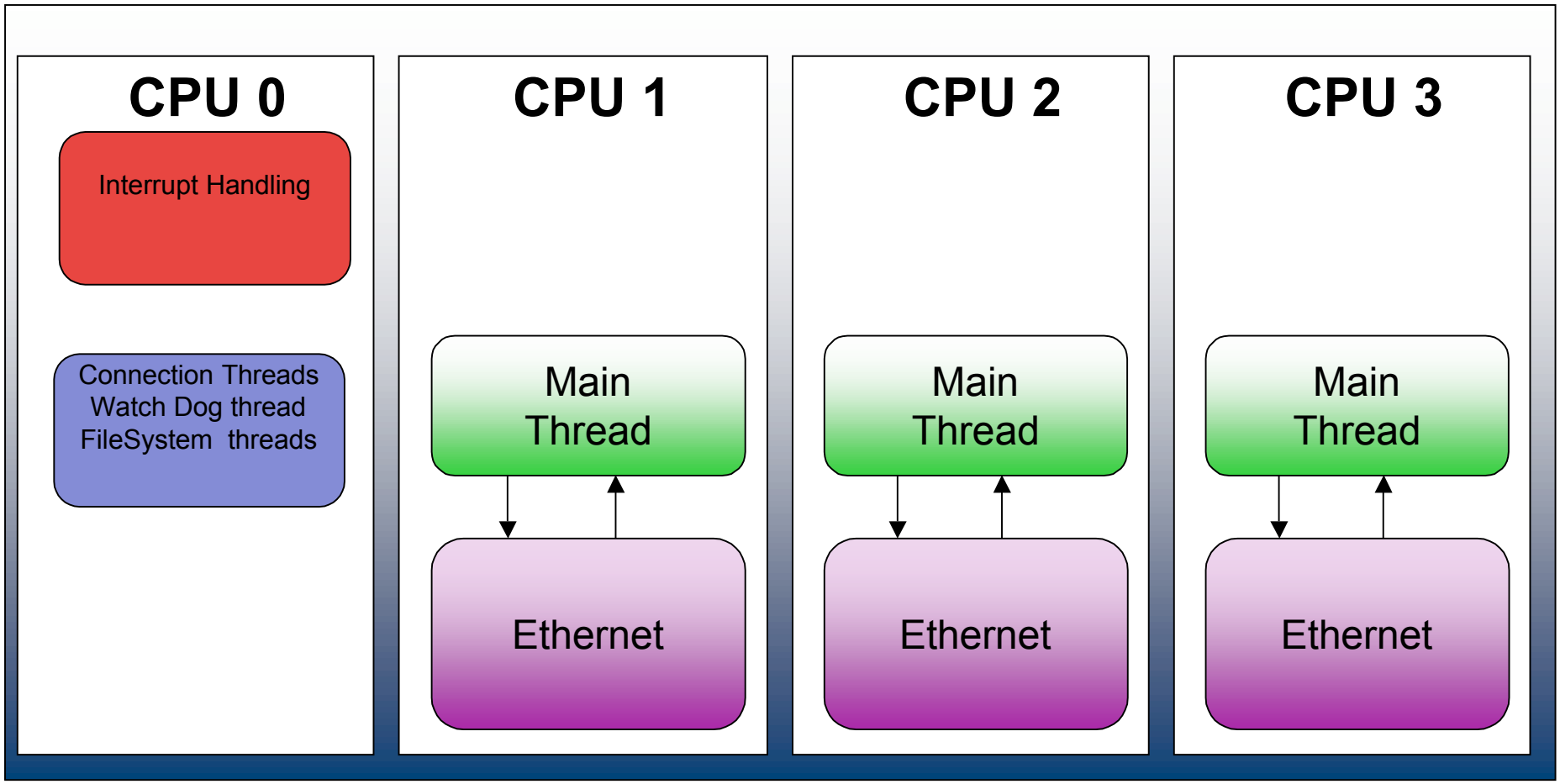
MySQL Cluster RT solution on Dual Core



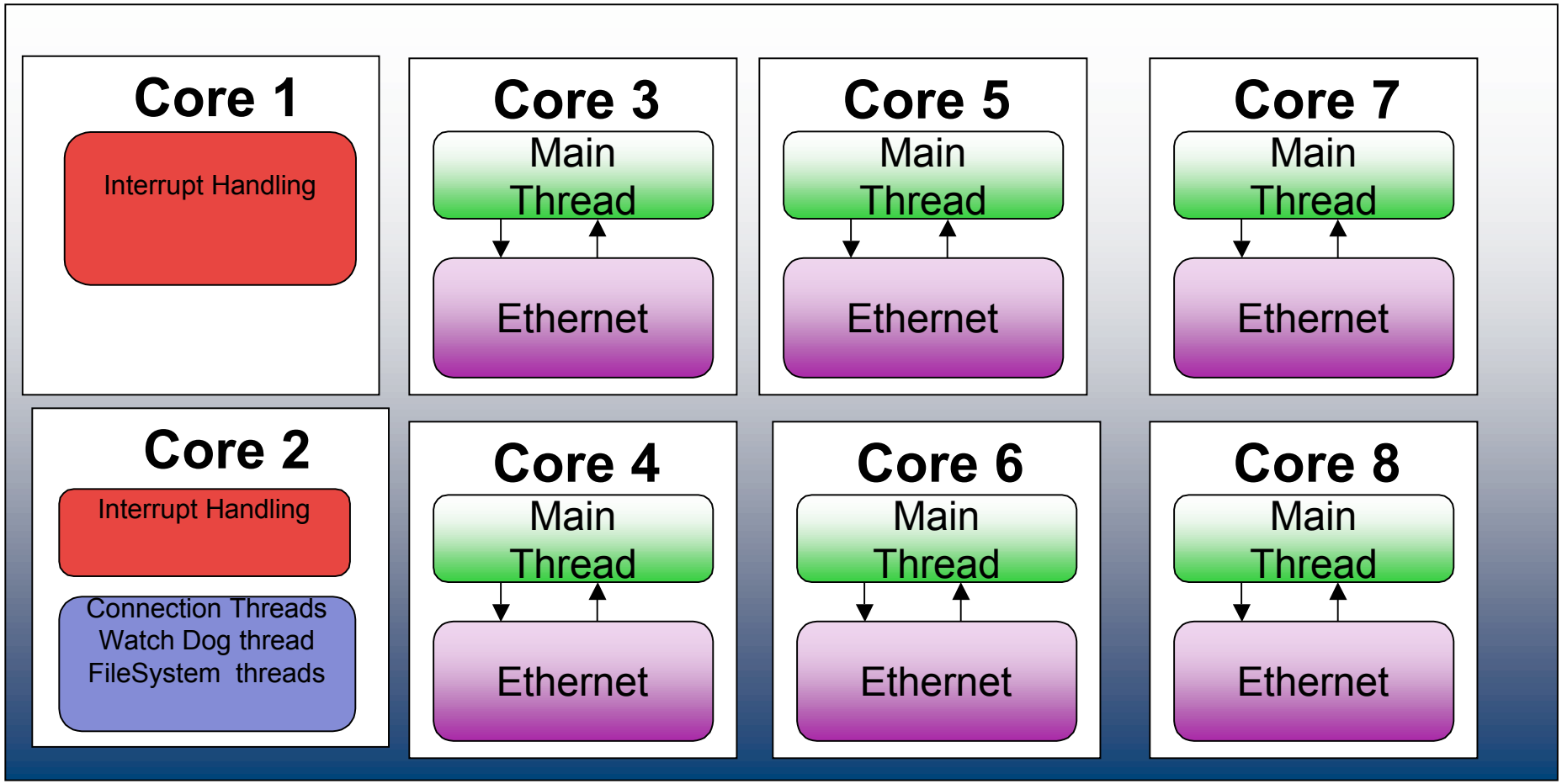
MySQL Cluster RT solution on Quad-Core computer using 4 data nodes CPU optimized architecture using Dolphin SuperSockets and Polling-based



MySQL Cluster RT solution on Quad-Core computer using 3 data nodes CPU optimized architecture using Ethernet



MySQL Cluster RT solution on Eight-Core computer using 6 data nodes CPU optimized architecture using Ethernet



Old ”thruths” revisited

- Previous recommendation was to run 1 data node per computer
- This was due to bugs in handling Multi-node failure handling
- This recommendation no longer exists since more than a year back
- Quality of multiple nodes per computer is good now

Distribution Awareness

- Start transaction coordinator on node which first query of transaction is using
- E.g. SELECT * from t WHERE pk=x

=> Map x into a partition, partition is then mapped into a node containing the primary replica of the record

- 100-200% improvement when application is distribution aware

Remove read before PK update

- UPDATE t SET a = const1 WHERE pk = x;
- No need to do a read before UPDATE, all data is already known
- ~10% improvement on DBT2
- Applies to DELETE as well

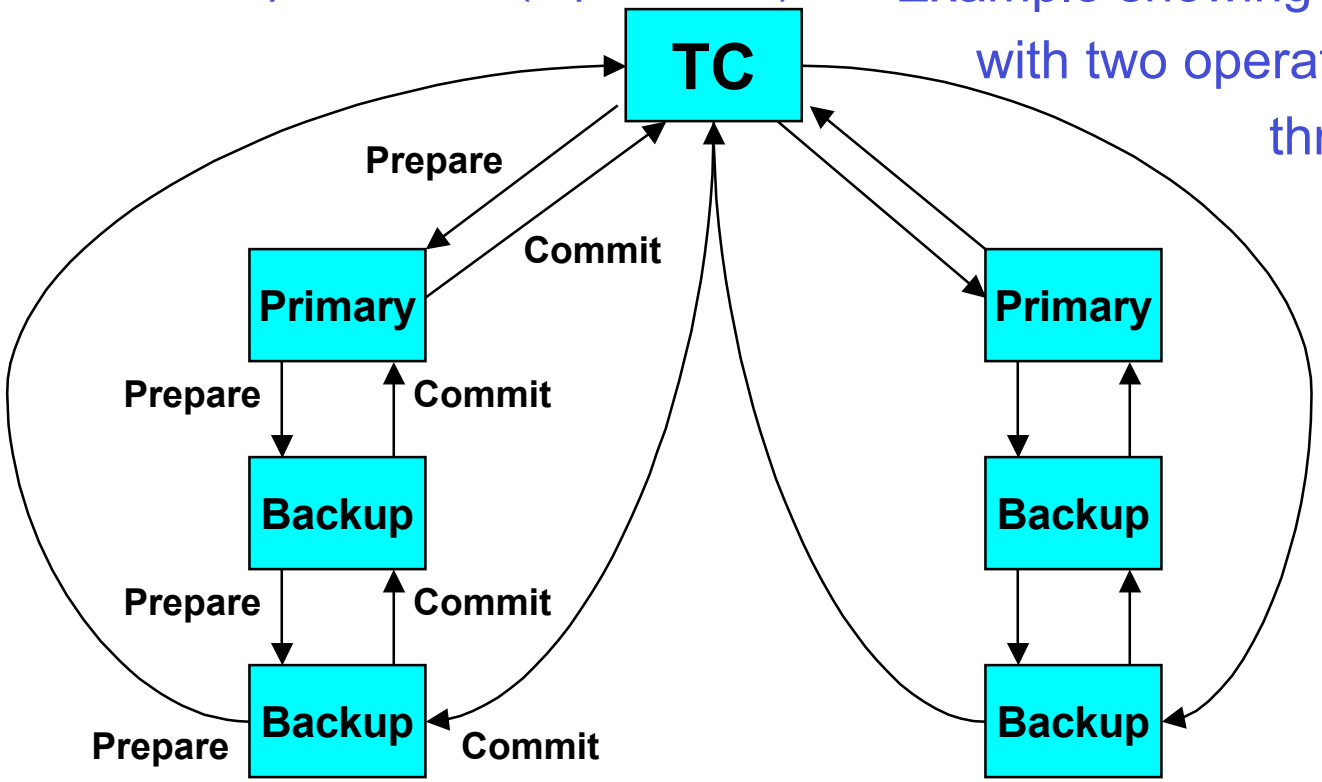
Ensure Proper Partitioning of Data Model

- Proper partitioning is important to ensure transaction execution is as localised to one nodegroup as possible (works together with Distribution Awareness)
- Transactions spanning several node groups means much more communication

Synchronous Replication:
Low failover time

messages = 2 x operations x (replicas + 1)

Example showing transaction with two operations using three replicas



1. Prepare F1 2. Commit F1 1. Prepare F2 2. Commit F2

Partitioning in DBT2 almost entirely on Warehouse ID

- Partitioning on primary key makes all transactions fully distributed over the entire cluster
- PARTITION BY KEY (warehouse_id)
- PARTITION BY HASH (warehouse_id)

=> Gives more or less perfect partitioning

Other Partitioning tricks

- If there is a table that has a lot of index scans (not primary key) on it

⇒Partitioning this table to only be in one node group can be a good idea

Partition syntax for this:

```
PARTITION BY KEY (id)
( PARTITION p0 NODEGROUP 0);
```

Use of features in MySQL Cluster version 5.0

- Lock Memory
- Batching of IN (..) primary key access
- INSERT batching
- Condition pushdown (faster table scans)

Lock Memory in Main Memory

- Ensure no swapping occurs in NDB Kernel

Batching IN (…) with primary keys

- 100 x SELECT * from t WHERE pk = x;
- SELECT * from t WHERE pk IN (x1,,,x100);
- IN-statement is around 10x faster than 100 SELECT single record PK access

Use of multi-INSERT

- 100 x INSERT INTO t (x)
- INSERT INTO t (x1),(x2),,,,,(x100)
- Multi-insert up to about 10x faster

Use of features in MySQL Cluster CGE version 6.4

- Multi-threaded Data nodes

⇒Currently no benefit using DBT2

⇒Have been shown to increase throughput by 40% for some NDB API benchmarks

Use of HW, CPU choice

- Pentium D @ 2.8GHz -> Core 2 Duo at 2.8GHz => 75% improvement
- Doubling of L2 cache size seem to double thread scalability of MySQL Cluster (experience using DBT2)
- Multi-core CPU's can be used, requires multiple node per Server

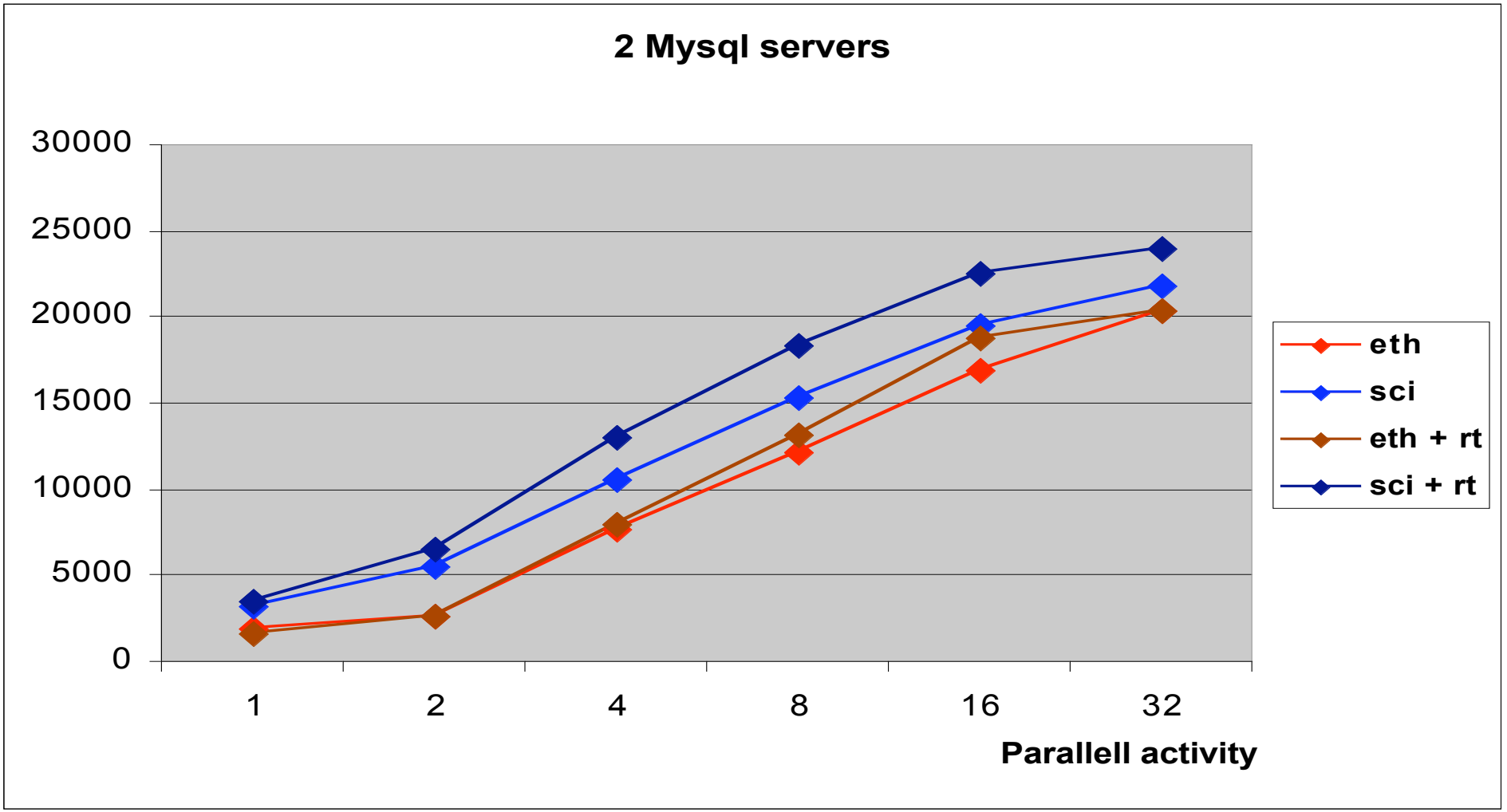
Use of HW, Interconnect choice

- Choice of Dolphin Express interconnect has been shown to increase throughput between 10% and 400% dependent on use case
- Response time improvements have been seen from 20% to 700%

Dolphin SuperSockets

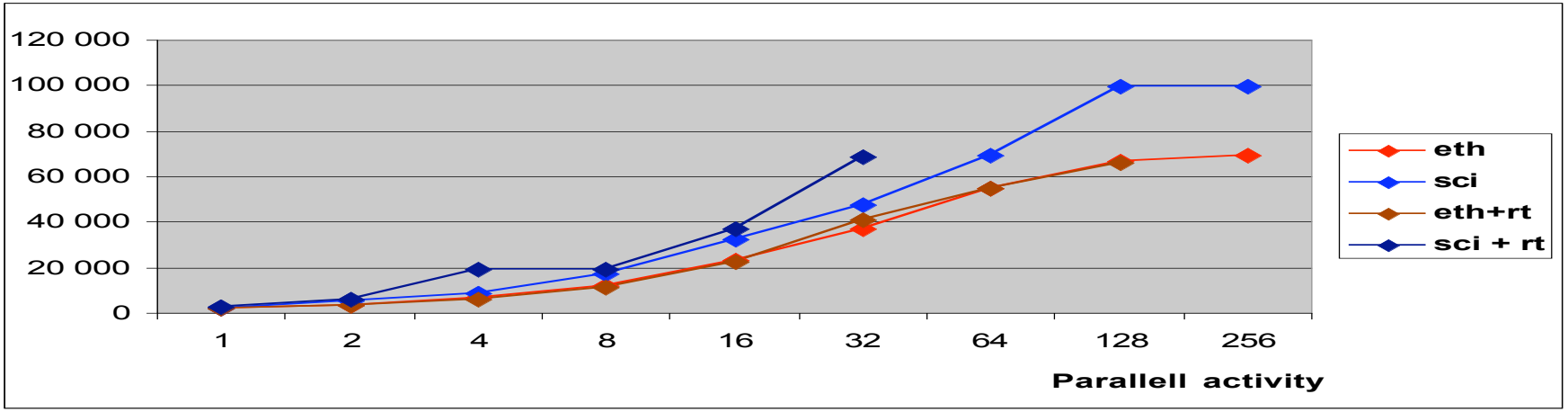
- Implementation of the Socket API using Dolphin Express Interconnect HW
- Latency of ping-pong on socket layer downto few microseconds
- High-Availability Features integrated
- Multi-Channel support integrated
- PCI Express Cards => 700 Mbyte/sec on Server Hardware

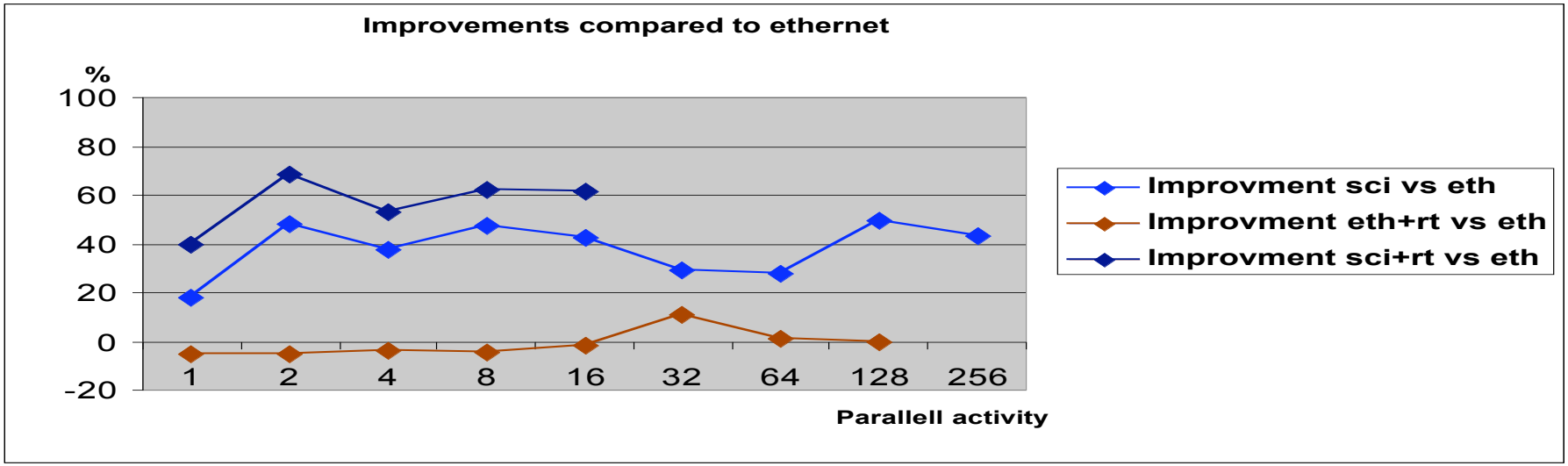
Minimal Cluster, 2 data nodes



Distribution aware

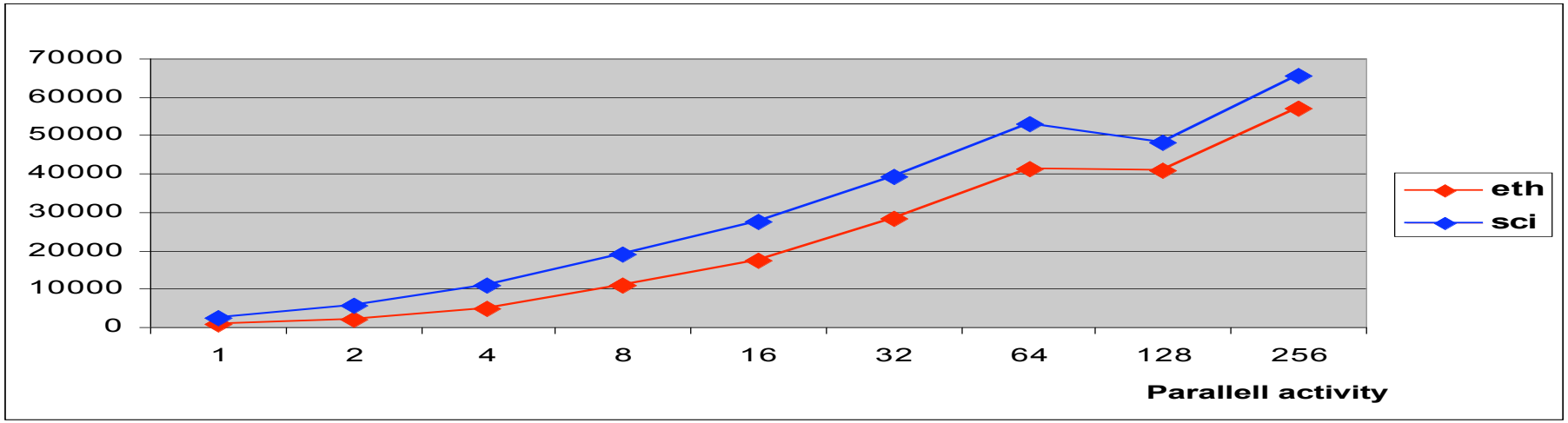
(8 data nodes on 2 Quad Core)

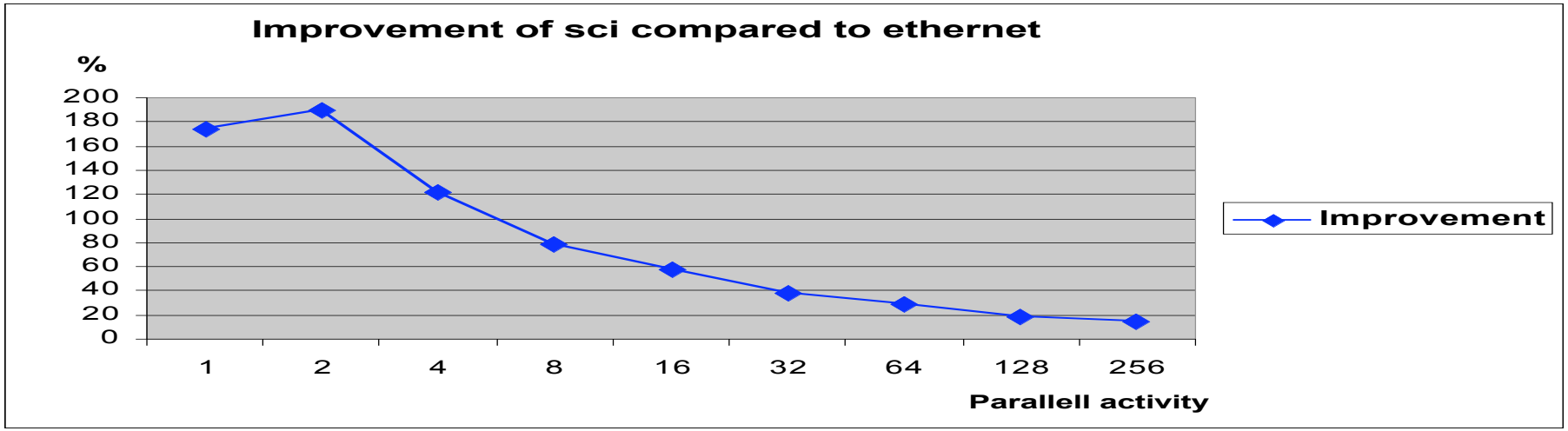




Non-distribution aware

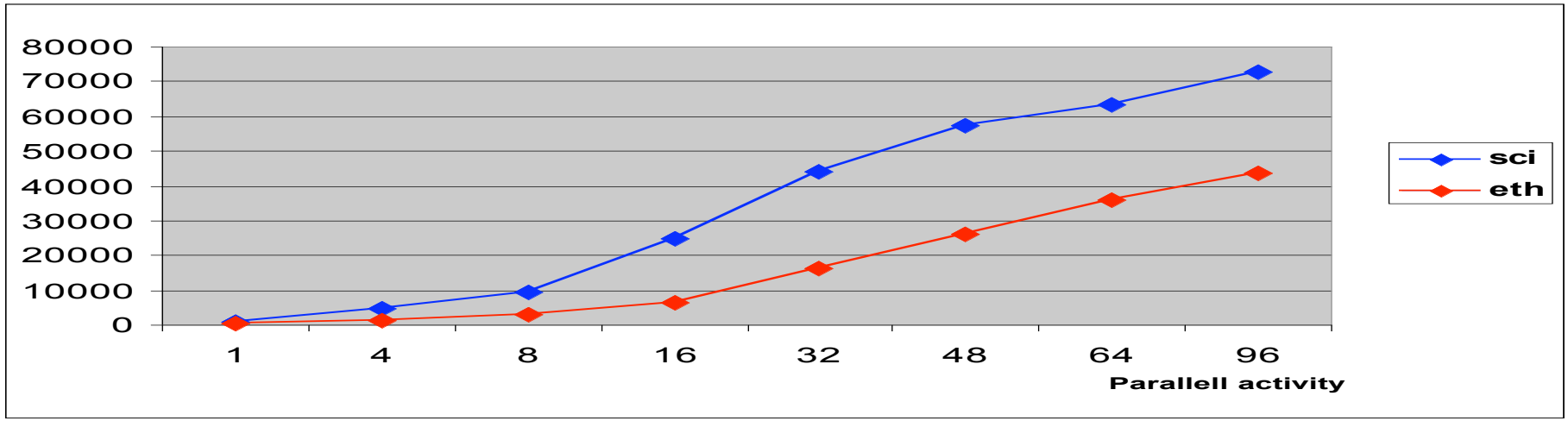
(4 data nodes on 4 Quad Cores)

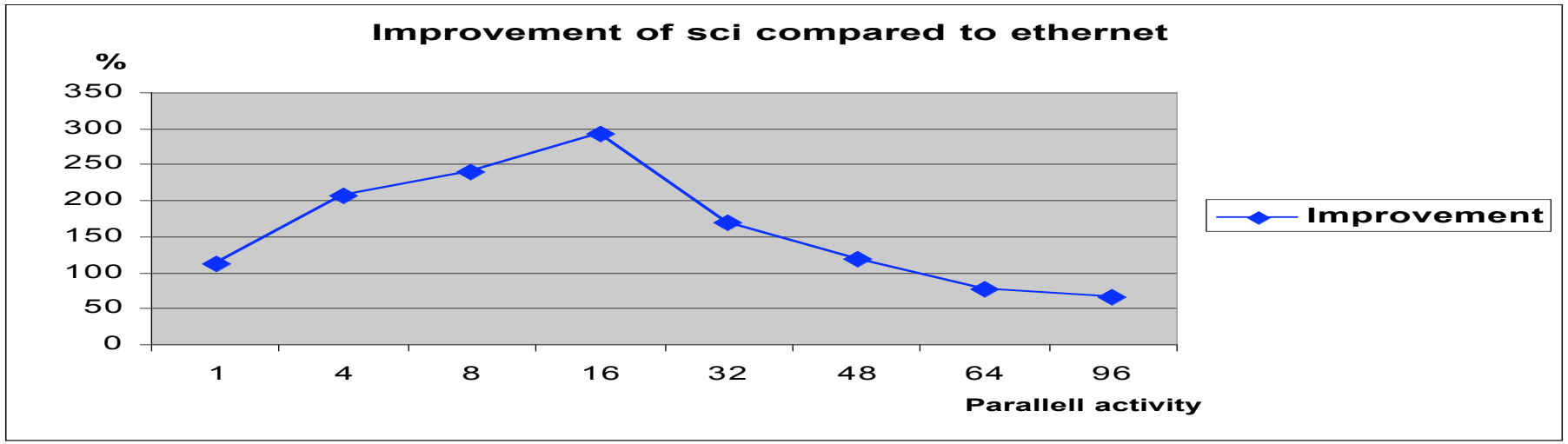




Non-distribution aware

(12 data nodes on 3 Quad Cores)





Important MySQL Server Parameters (5.1)

- --ndb-index-stat-enable=0 (Bug if enabled)
- --ndb-use-exact-count=0 (100%)
- --ndb-force-send=1 (20%)
- --engine-condition-pushdown=1 (~10%)

Scalability of Threads using DBT2

- Linear scalability 1->2->4 threads
- Special case of 1->2 threads on smaller clusters gives 200% increase
- ~40-70% increase 4->8 threads
- ~10-30% increase 8->16 threads
- Decreasing performance going beyond 16 threads

Scalability of MySQL Servers using DBT2

- Linear scalability adding MySQL Servers
- Maximum Performance where #MySQL Servers = 3 x Number of Data Nodes
- Number of MySQL Server = Number of Data Nodes 25% less maximum performance
- Number of MySQL Servers = 2 x Number of Data Nodes 5% less maximum performance

Scalability of Data Nodes using DBT2 with proper partitioning using Ethernet

- Measured using number of #Data Nodes == #MySQL Servers and at least 2 cores per data node
- 2-nodes Max = 27.000 tpm
- 4-nodes Max = 40.000 tpm (~50%)
- 8-nodes Max = 66.000 tpm (~65%)
- 16-nodes Max = 91.000 tpm (~40%)
- 32-nodes Max = 132.000 tpm (~40%)

Scalability of Data Nodes using DBT2 with proper partitioning using Dolphin Express

- 2-nodes 25.000 tpm
- 8-nodes 100.000 tpm
- Scalability using Dolphin Express much improved compared to Ethernet scalability

Future SW performance improvements (1)

- Batched Key Access, Improves execution of joins especially where joins use lookups of many primary key accesses (0-400%)

Preview of this feature already available

- Improved Scan protocol (~15%)
- Improved NDB Wire Protocol (decreases number of bits transported to almost half) (~20%)

⇒ Less cost for communication

⇒ Less cost for memory copying in NDB code

Future SW performance improvements (2)

- Incremental Backups
- Optimised backup code
- Parallel I/O on Index Scans Using disk data
- Various local code optimisations
- Using Solaris features for locking to CPU's, Fixed Scheduler priority, Interrupts on dedicated core
- Compiler improvements (see my blog for how this improved MySQL/InnoDB on Niagara boxes)
- Improved scalability inside of one MySQL Server
- Increase maximum number of data nodes from 48 to 128

So how will MySQL Cluster work on a Niagara-II with 256 GB memory? Unpublished results from 2002

- Benchmark load:
- Simple read, read 100 bytes of data through primary key
- Simple update, update 8 bytes of data through primary key
- Both are transactional
- HW: 72-CPU SunFire 15k, 256 GB memory
- CPU's: Ultra Sparc-III@900MHz
- 32-node NDB Cluster, 1 data node locked to 1 CPU
- Results (Database size = 88 Gbyte, ~900 million records):
- Simple Read: 1.5 million reads per second
- Simple update: 340.000 updates per second