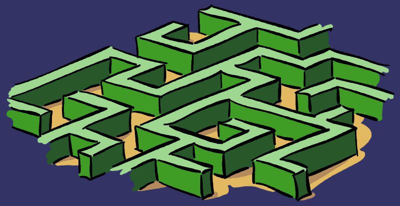
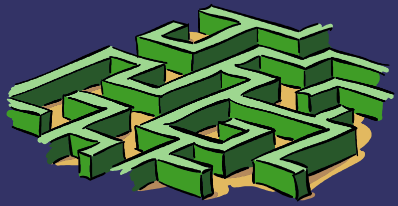
An Analysis of Multicore Specific Optimization in MPI Implementations

Pengqi Cheng & Yan Gu
Tsinghua University



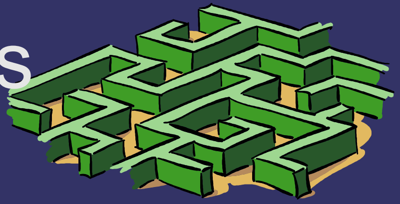
Introduction

- CPU frequency stalled
- Solution: Multicore
- OpenMP – shared memory
- MPI – Message Passing Interface
- MPI will be more efficient than OpenMP for manycore – memory wall



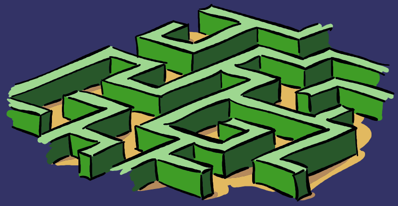
Thread-Level Parallelism

- Hybrid Programming
- Lowering MPI – lack of scalability
- MPI + OpenMP / Pthreads / etc.
- Advantage
    - More control
- Disadvantage
    - More complexity
    - Close to hardware instead of algorithm
    - Hard to reuse existed codes



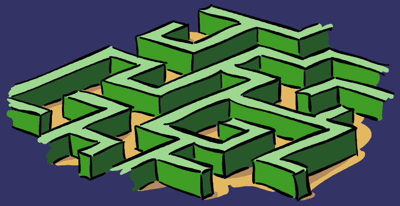
MPICH2 – Implementation

- Communication Subsystem – Nemesis
- One lock-free *receive* queue per process
-



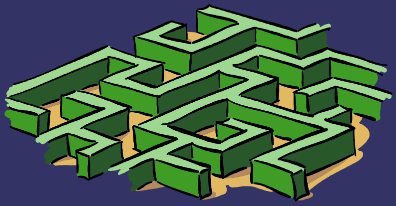
MPICH2 – Location of free queue

- One global
  - Good for balance on multicore
  - Lack of scalability
- One per process deq. by one side
  - Good for NUMA – less remote access
  - Inevitable imbalance
- MPICH2 uses the latter
- Dequeued by the sender itself



MPICH2 – pseudocode of queue

```
Enqueue (queue, element)
    prev = SWAP (queue->tail, element); //atomic swap
    if (prev == NULL)
        queue->head = element;
    else
        prev->next = element;
Dequeue (queue, &element)
    element = queue->head;
    if (element->next != NULL)
        queue->head = element->next;
    else
        queue->head = NULL; //CAS – atomic compare and swap
        old = CAS (queue->tail, element, NULL);
        if (old != element)
            while (element->next == NULL)
                SKIP;
            queue->head = element->next;
```



MPICH2 – Optimizations

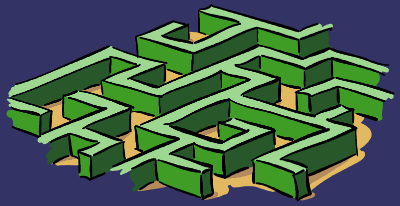
- Reducing L2 cache miss
  - Both head and tail accessed when
    - Enqueuing onto an empty queue
    - Dequeuing the last element
  - One miss less if head and tail are in the same cache line
  - False sharing if more elements
  - With a *shadow* head copy, miss only when enqueuing onto an empty queue or dequeuing from a queue with only one element

MPICH2 – Optimizations

- Bypassing Queues
  - Fastbox – single buffer
  - One per pair of process
  - Check fastbox first and then the queue
- Memory Copy
  - Assembly/MMX in place of memcpy()
- Bypassing the Posted Receive Queue
  - Checks all send/recv pair instead of matching send to current recv

MPICH2 – Large Message Transfer

- Queues have to store unsent data
- What if the message is large?
    - Bandwidth pressure
    - Cache pollution
- Rendezvous instead of eager

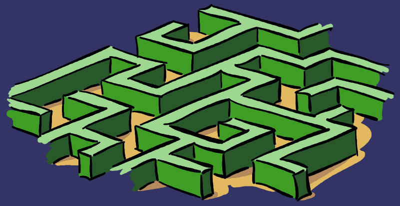


OpenMPI – sm BTL

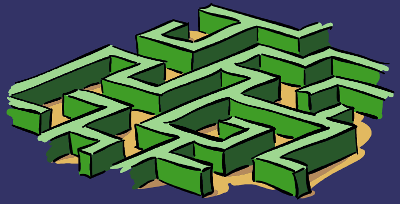
- Shared Memory Byte Transfer Layer
- Transfer fragments of broken messages
- Sender fills a sm fragment in its free lists
    - Two free lists, for small/large msg.
- Sender packs the user-message fragment into sm fragment.
- Sender posts a pointer to this shared frag into FIFO queue of receiver.
- Receiver polls its FIFO(s). Unpack data when it finds a new fragment pointer and notifies the sender

KNEM – Kernel Nemesis

- Linux Kernel Module
- Problems of traditional buffer copying
  - Cache pollution
  - Waste of memory space
  - High CPU use
- Solution
  - Direct single copying in kernel space

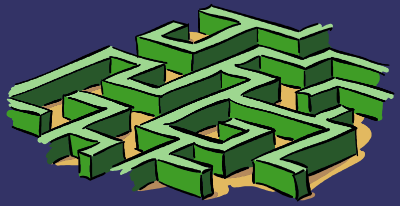


KNEM – Implemetation



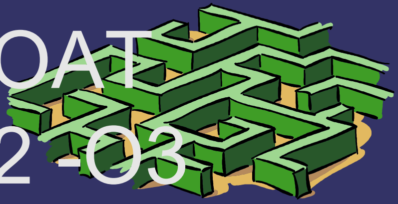
Experiment Platform

- Hardware
  - Quad-Core Intel Core i5 750 2.67GHz
    - L1: 32KB+32KB per core
    - L2: 256KB per core
    - L3: 8MB shared
  - 4GB DDR3 @ 1333MHz

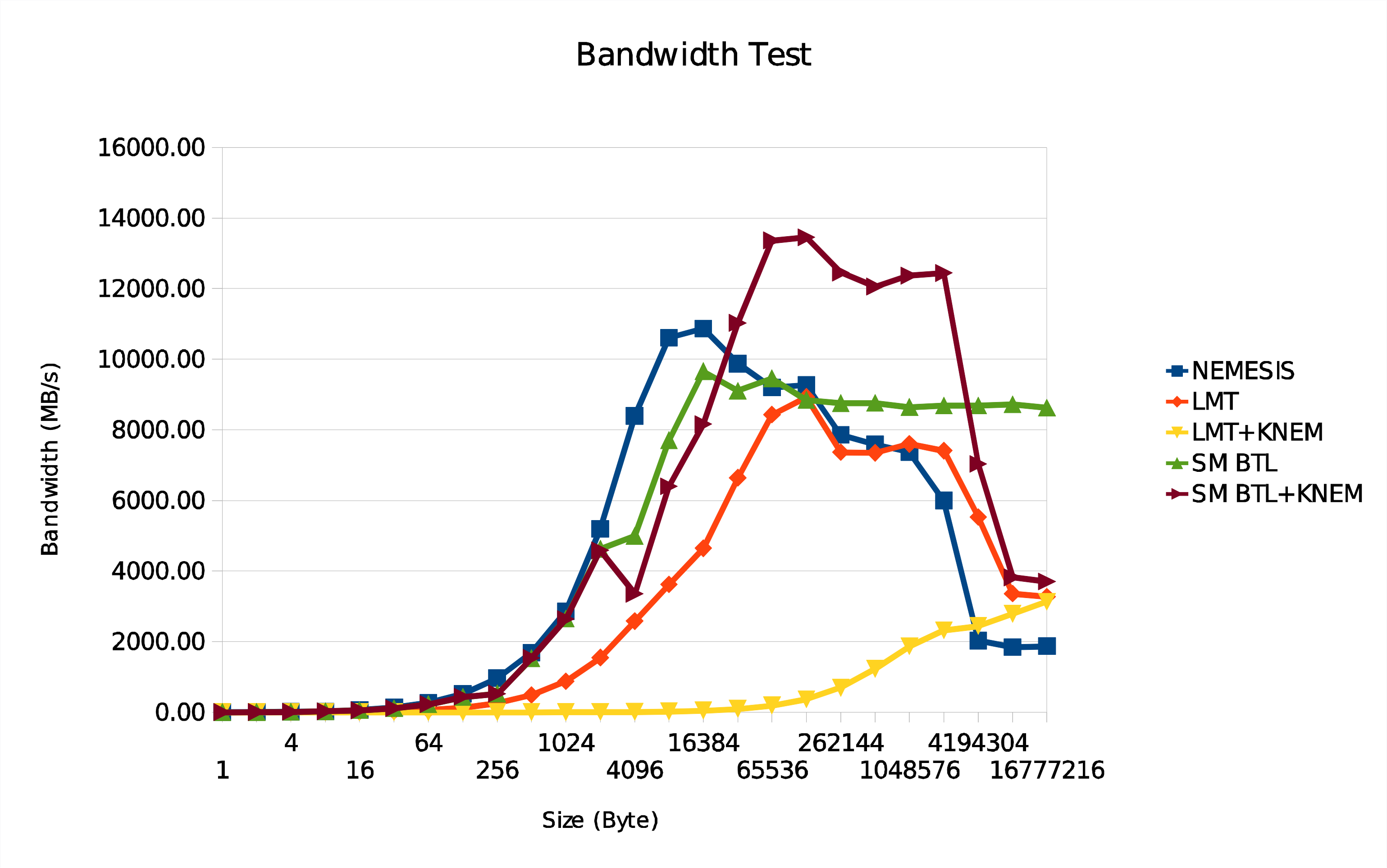


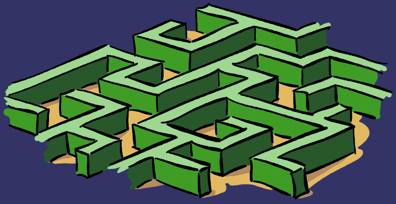
Experiment Platform

- Software
  - Arch Linux x86-64 with Kernel 2.6.36
  - GCC 4.2.4
  - MPICH2 1.3.1 -O2
    - No LMT / LMT Only / LMT + KNEM
  - OpenMPI 1.5.1 -O2
    - sm BTL, with and without KNEM
  - KNEM 0.9.4 -O2, without I/OAT
  - OSU Micro-Benchmarks 3.2 -O3
  - 2 processes for one-to-one

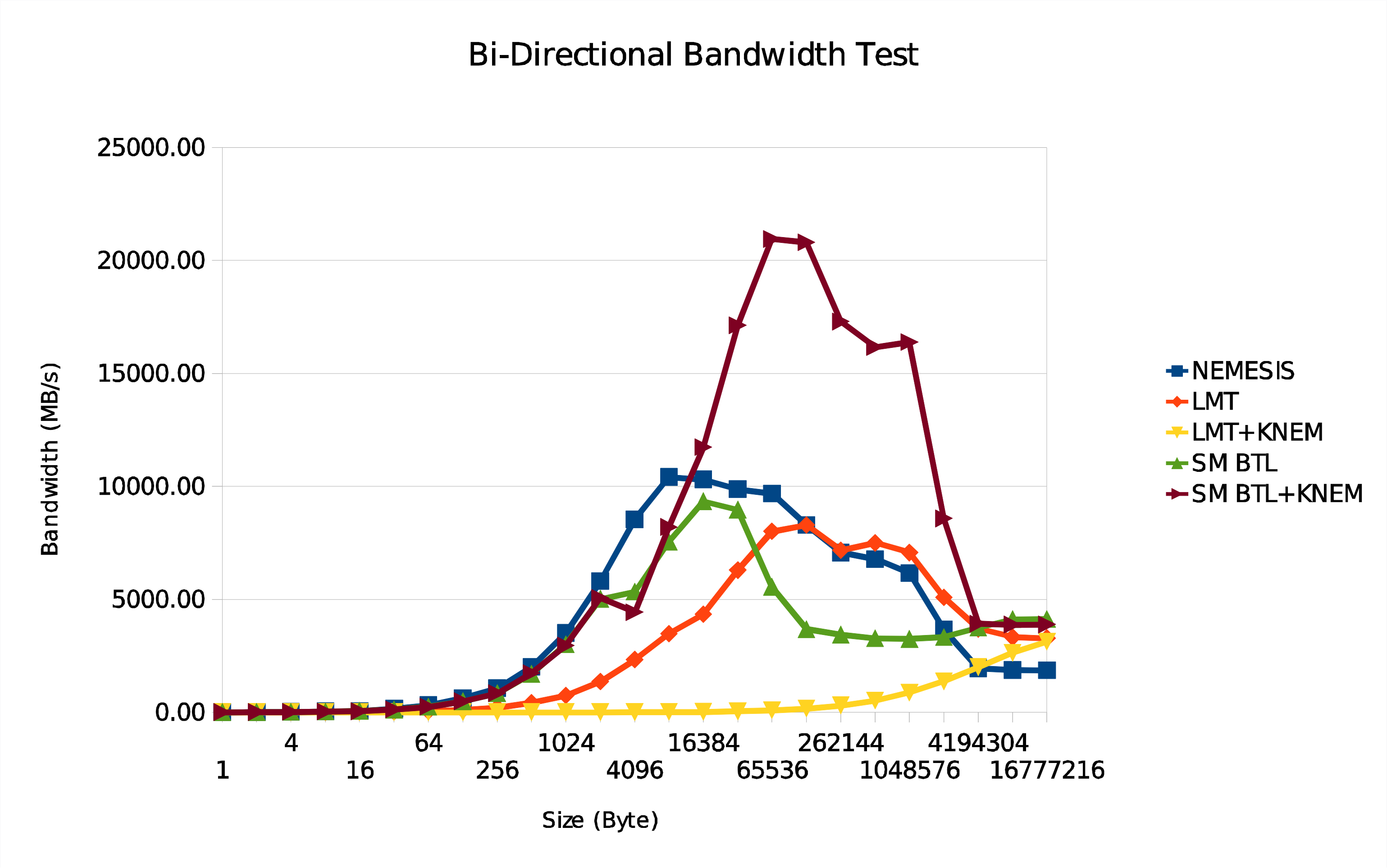


Results

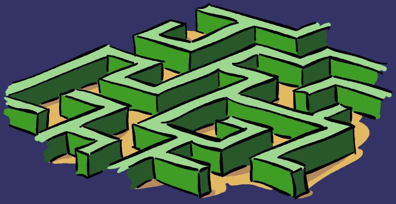

Bandwidth Test
Bandwidth (MB/s)
16000.00
14000.00
12000.00
10000.00
8000.00
6000.00
4000.00
2000.00
0.00
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
4194304
16777216
Size (Byte)
NEMESIS
LMT
LMT+KNEM
SM BTL
SM BTL+KNEM



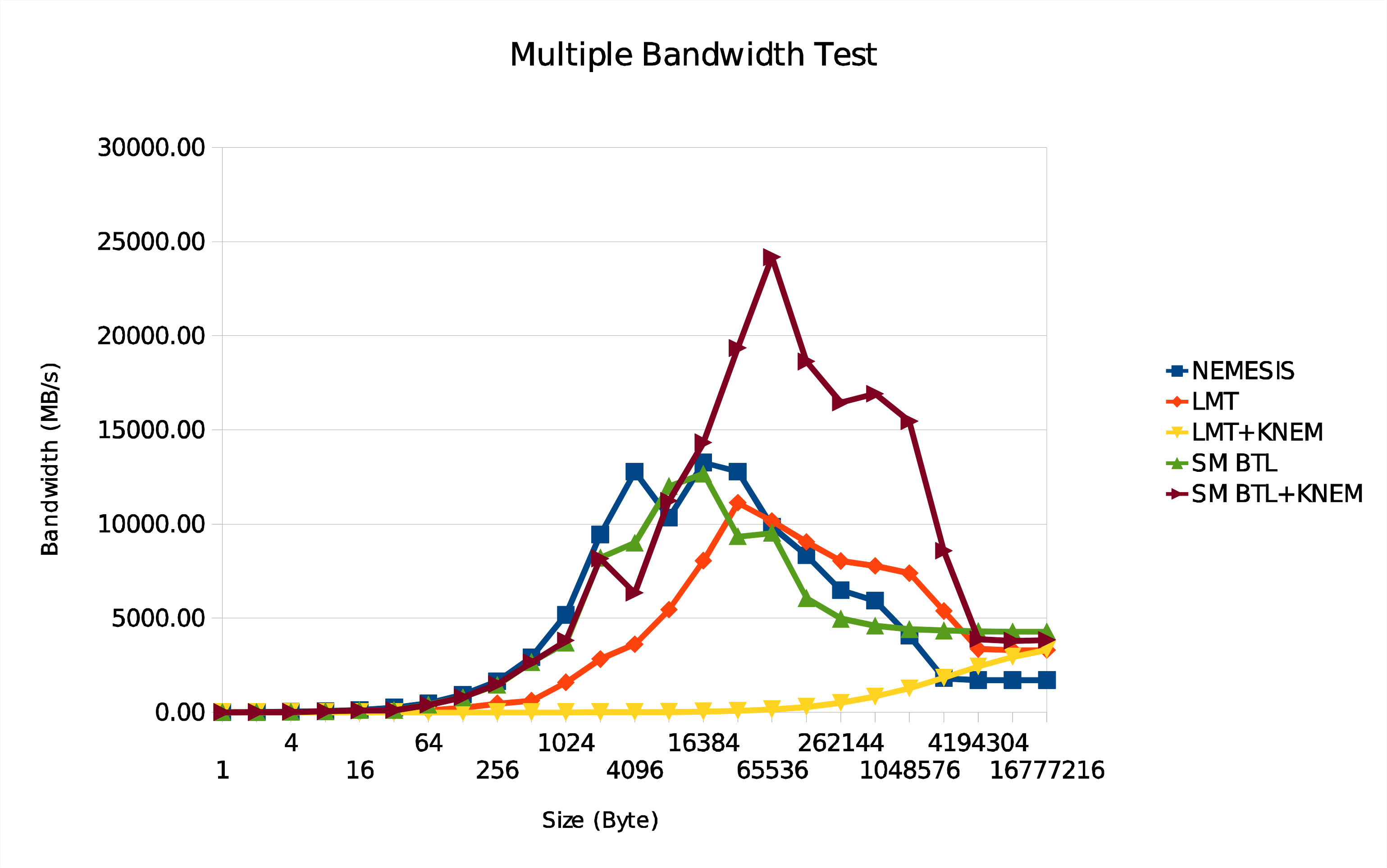
Results

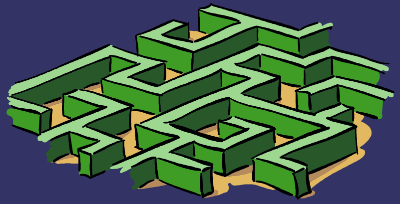


Bi-Directional Bandwidth Test
25000.00
20000.00
15000.00
10000.00
5000.00
0.00
Bandwidth (MB/s)
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
4194304
16777216
Size (Byte)
NEMESIS
LMT
LMT+KNEM
SM BTL
SM BTL+KNEM

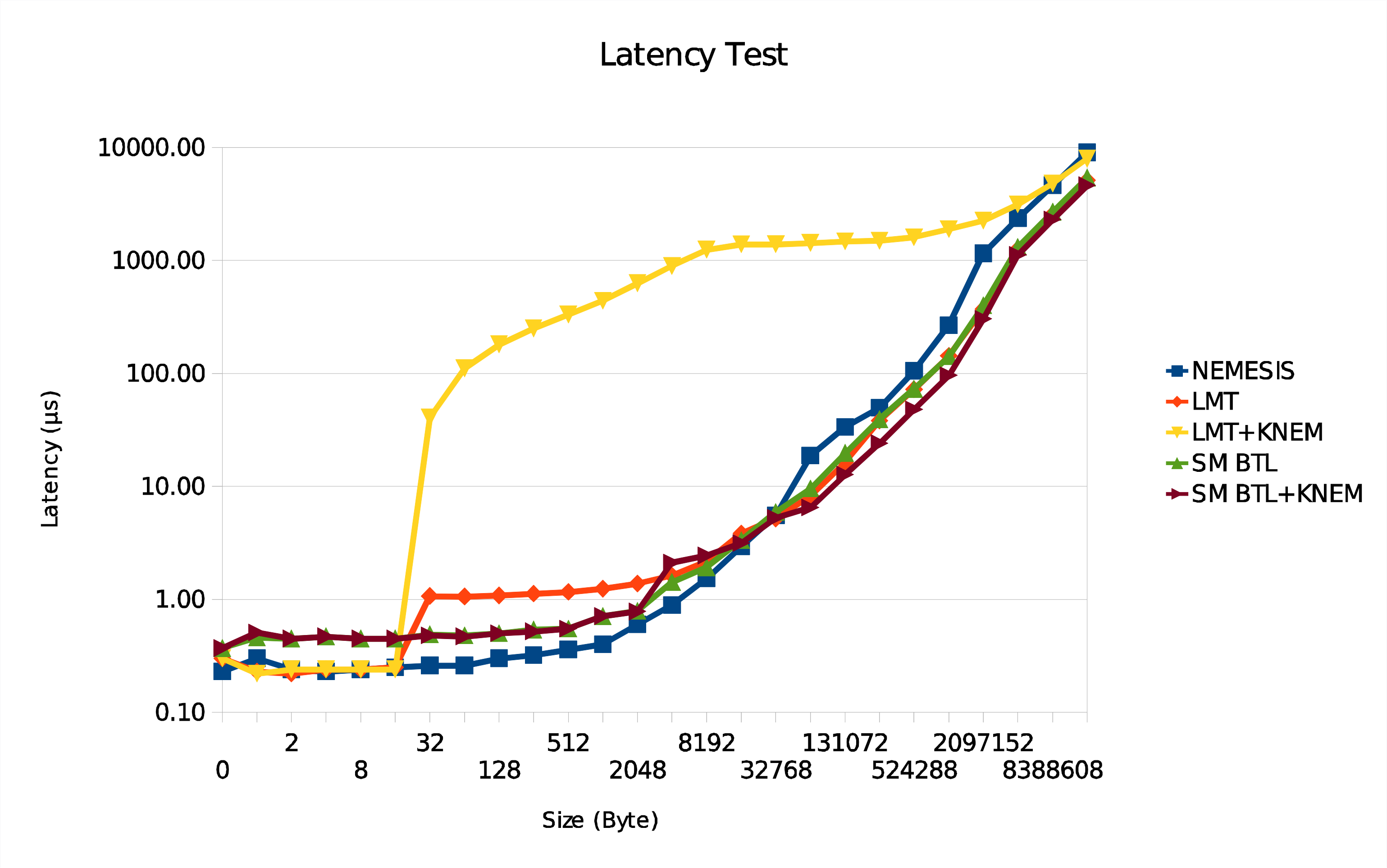


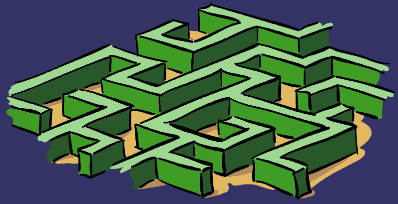
Results


Multiple Bandwidth Test
Bandwidth (MB/s)
30000.00
25000.00
20000.00
15000.00
10000.00
5000.00
0.00
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
4194304
16777216
Size (Byte)
NEMESIS
LMT
LMT+KNEM
SM BTL
SM BTL+KNEM

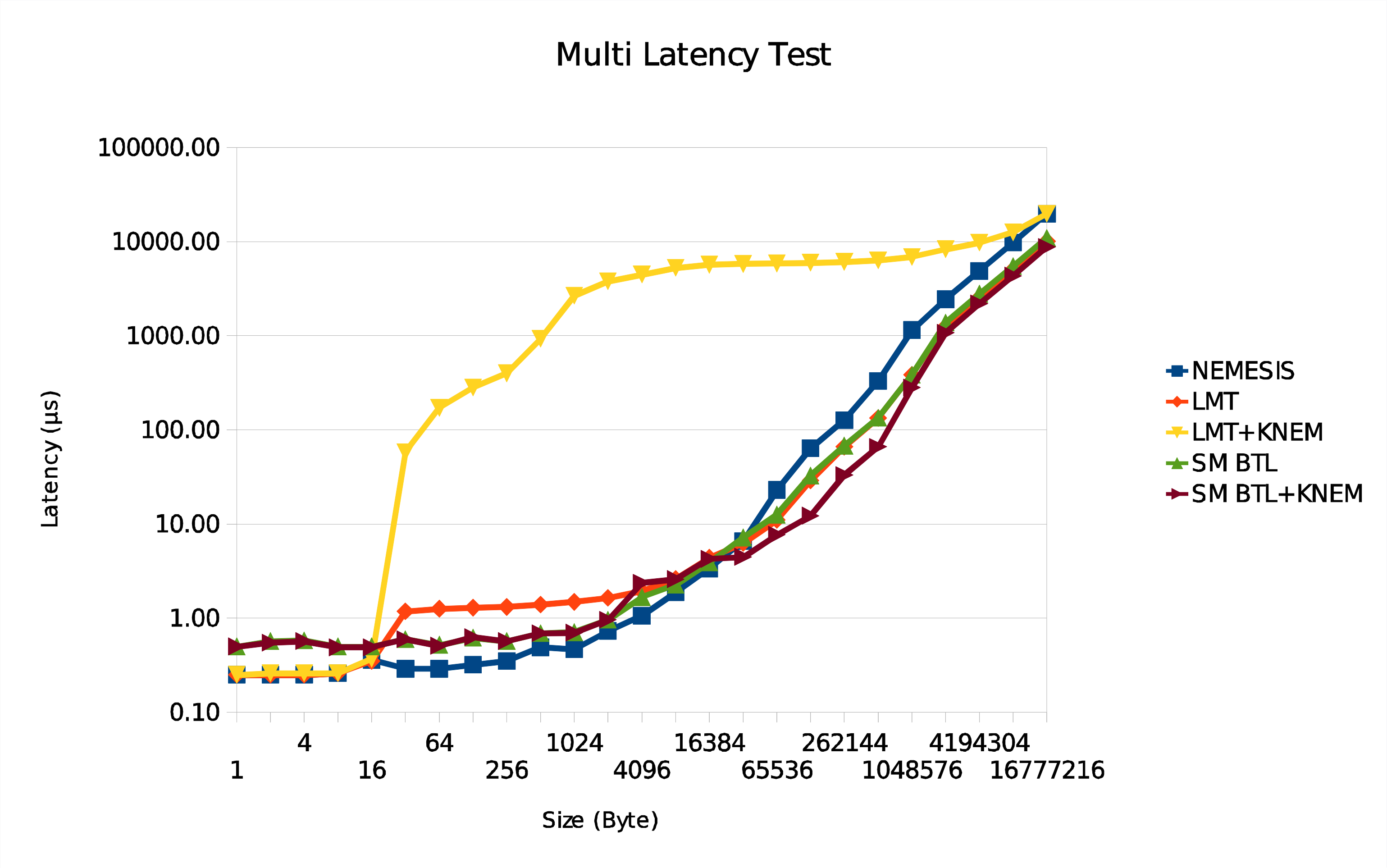


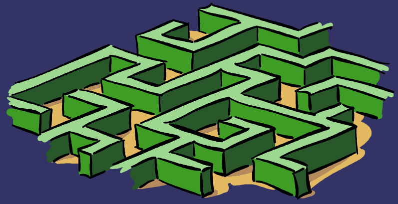
Results


Latency Test
Latency (µs)
10000.00
1000.00
100.00
10.00
1.00
0.10
0
2
8
32
128
512
2048
8192
32768
131072
524288
2097152
8388608
Size (Byte)
NEMESIS
LMT
LMT+KNEM
SM BTL
SM BTL+KNEM

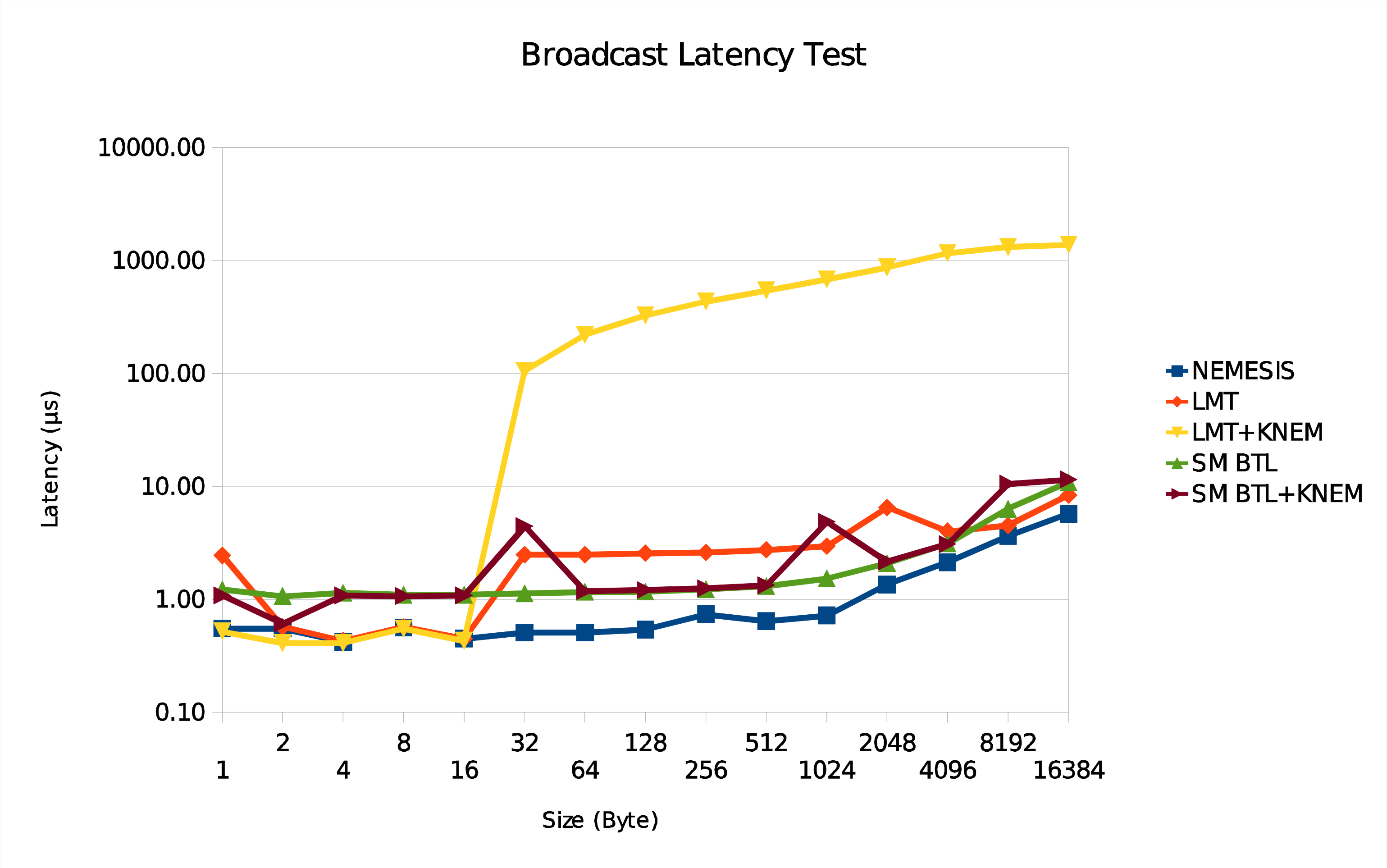


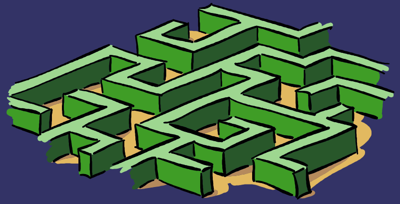
Results


Multi Latency Test
Latency (µs)
100000.00
10000.00
1000.00
100.00
10.00
1.00
0.10
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
4194304
16777216
Size (Byte)
NEMESIS
LMT
LMT+KNEM
SM BTL
SM BTL+KNEM

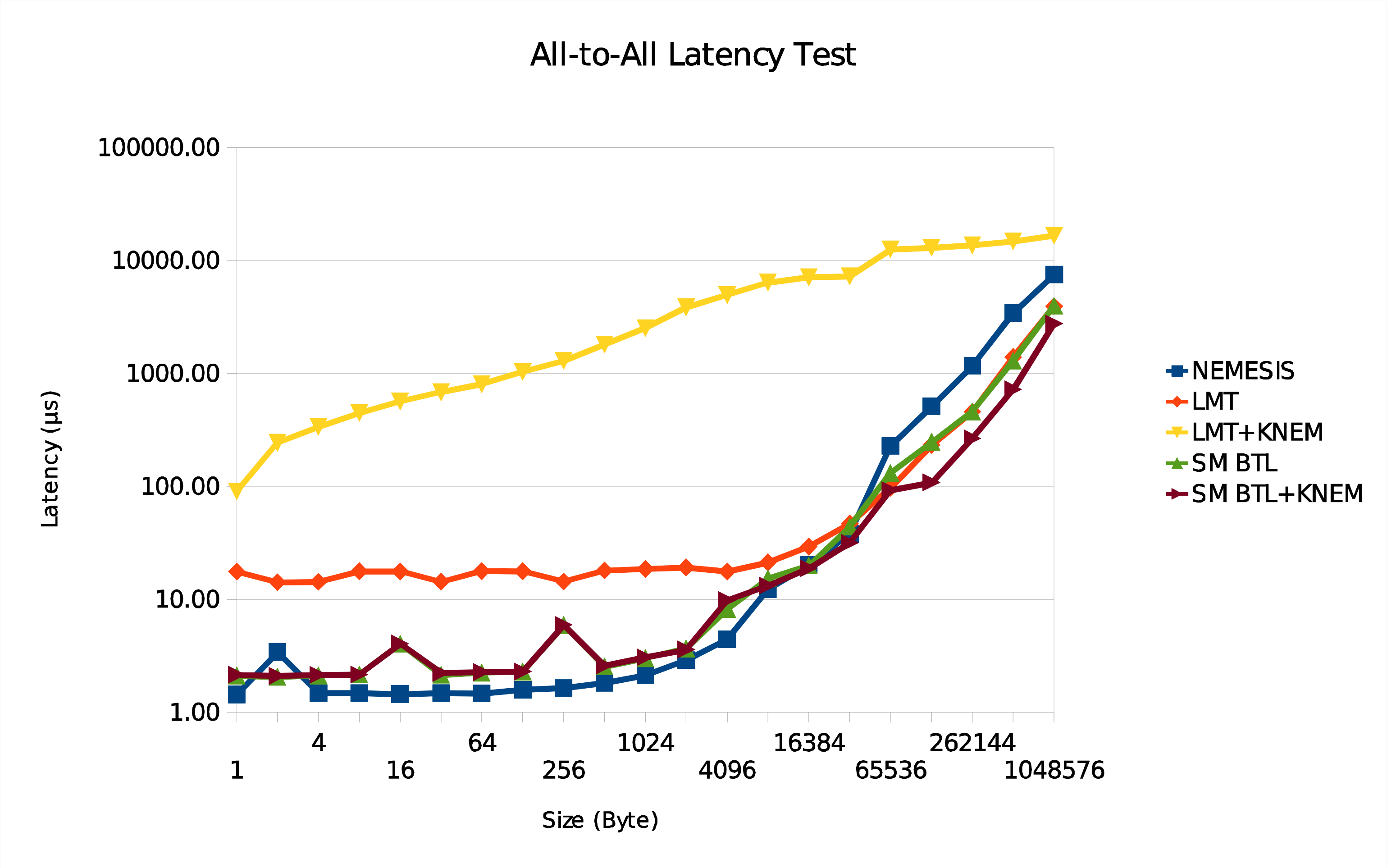


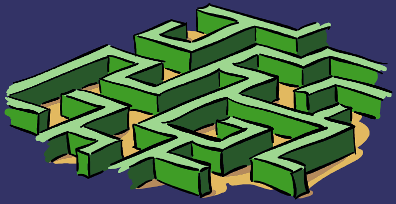
Results


Broadcast Latency Test
Latency (µs)
10000.00
1000.00
100.00
10.00
1.00
0.10
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
Size (Byte)
NEMESIS
LMT
LMT+KNEM
SM BTL
SM BTL+KNEM



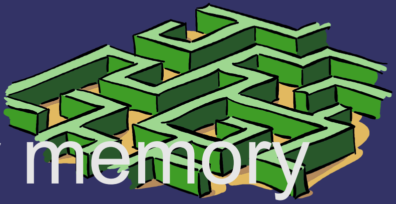
Results


All-to-All Latency Test
Latency (µs)
100000.00
10000.00
1000.00
100.00
10.00
1.00
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
Size (Byte)
NEMESIS
LMT
LMT+KNEM
SM BTL
SM BTL+KNEM



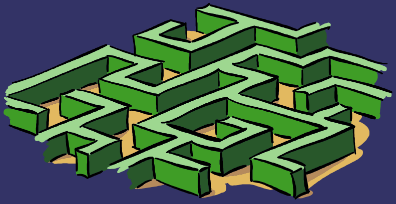
Analysis

- Nemesis (without LMT/KNEM)
  - Best for small messages
- sm BTL – best for large messages
- Watershed: about 16KB
- 16KB~4MB
  - KNEM accelerates sm BTL
  - But slower for LMT
- 4MB+ (larger than L3 cache)
  - KNEM makes sm BTL slower
  - But improves LMT
  - sm BTL > KNEM > LMT for memory
  - Will KNEM be better with DMA?



Analysis

- LMT > Original Nemesis
  - Threshold: 32KB~256KB
  - Smaller if more concurrent accesses
  - Steep Slopes at 32KB – LMT disabled
- How about
  - More cores?
    - Difference between 1-1 and all-all
  - Private cache?
  - I/OAT & DMA?
    - Will KNEM be faster?



Thank you!

Any Questions?