Introducing the Cray XMT

Petr Konecny

November 29th 2007

CRAY
THE SUPERCOMPUTER COMPANY

Agenda

- Shared memory programming model
  - Benefits/challenges/solutions
- Origins of the Cray XMT
- Cray XMT system architecture
  - Cray XT infrastructure
  - Cray Threadstorm processor
- Basic programming environment features
- Examples
  - HPCC Random Access
  - Breadth first search
- Rules of thumb
- Summary

Shared memory model

- Benefits
  - Uniform memory access
  - Memory is distributed across all nodes
  - No (need for) explicit message passing
  - Productivity advantage over MPI

- Challenges
  - Latency: time for a single operation
  - Network bandwidth limits performance
  - Legacy MPI codes

Addressing shared memory challenges

- Latency
  - Little's law:
    - Parallelism is necessary !
    - Concurrency = Bandwidth * Latency
    - e.g.: 800 MB/s, 2μs latency => 200 concurrent 64-bit word ops
  - Need a lot of concurrency to maximize bandwidth
    - Concurrency per thread (ILP, vector, SSE) => SPMD
    - Many threads (MTA, XMT) => MPMD

- Network Bandwidth
  - Provision lots of bandwidth
    - ~1 GB/s per processor, ~5 GB/s per router on XMT
  - Efficient for small messages
  - Software controlled caching (registers, nearby memory)
    - Eliminates cache coherency traffic
    - Reduces network bandwidth

Origins of the Cray XMT

Cray XMT (a.k.a. Eldorado)
Upgrade Opteron to Threadstorm

Multithreaded Architecture (MTA)
Shared memory programming model
Thread level parallelism
Lightweight synchronization

Cray XT Infrastructure
Scalable
I/O, HSS, Support
Network efficient for small messages

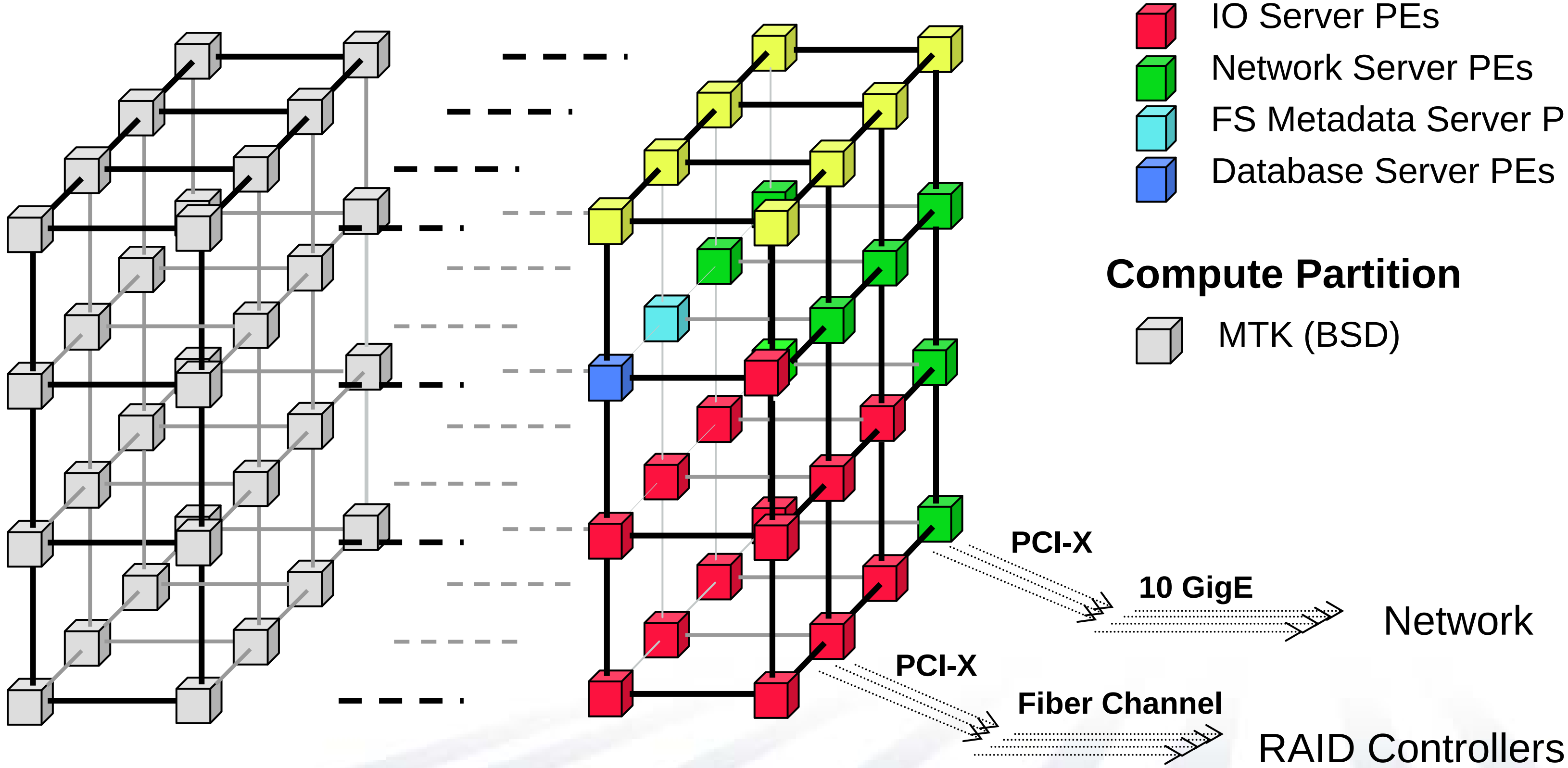
Cray XMT System Architecture

Compute

MTK

Service & IO

Linux



Service Partition

- ***Linux OS***
- ***Specialized Linux nodes***

Login PEs

IO Server PEs

Network Server PEs

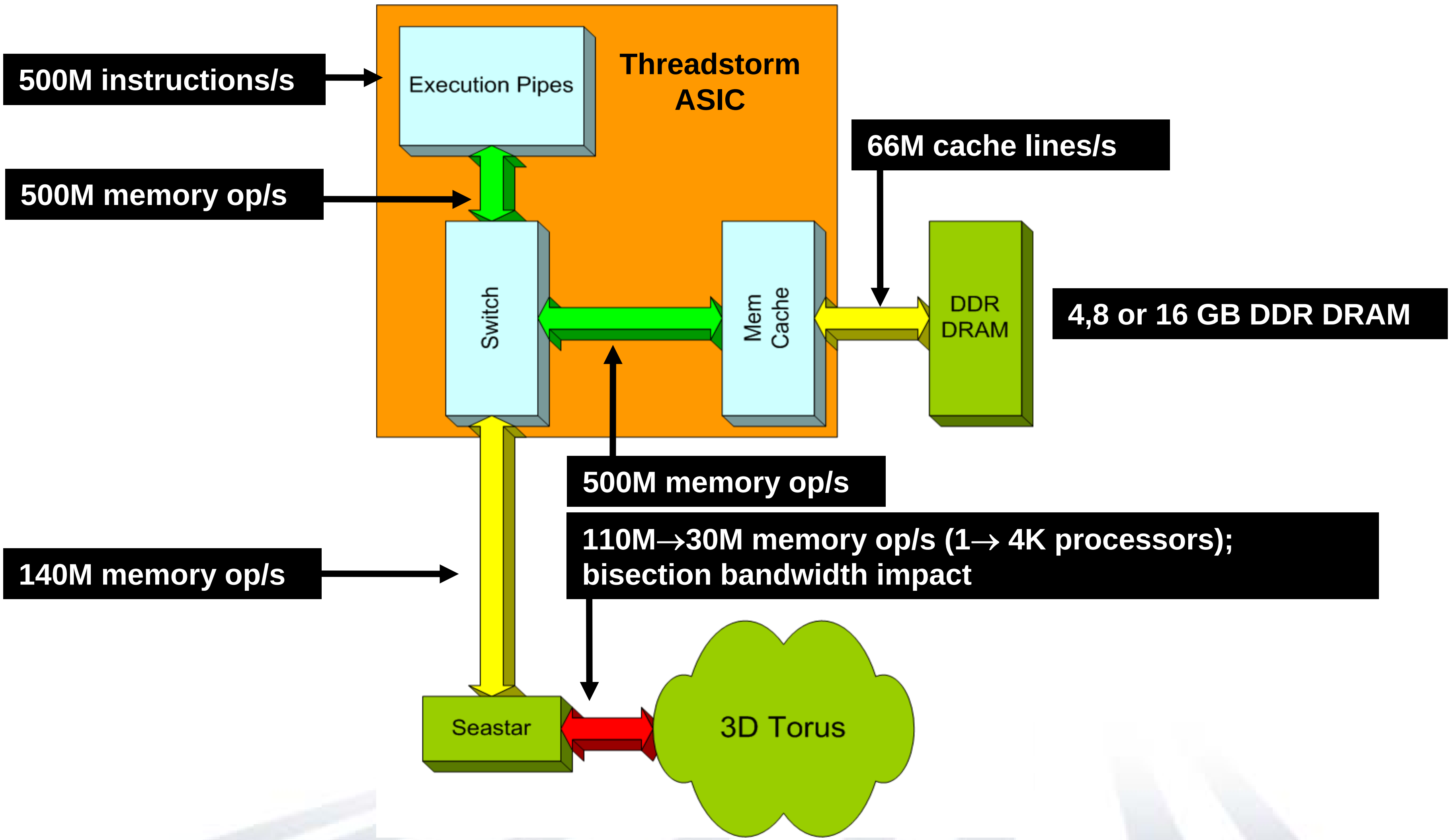
FS Metadata Server PEs

Database Server PEs

Compute Partition

MTK (BSD)

Cray XMT Speeds and feeds

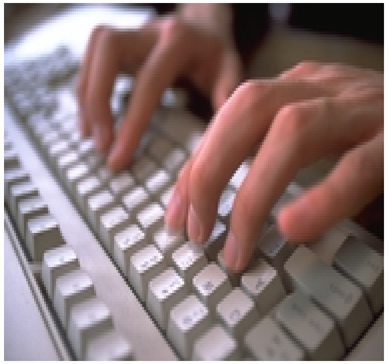


Cray Threadstorm architecture

- Streams (128 per processor)
  - Registers, program counter, other state
- Protection domain (16 per processor)
  - Provides address space
  - Each running stream belongs to exactly one protection domain
- Functional units
  - Memory
  - Arithmetic
  - Control
- Memory buffer (cache)
  - Only store data of the DIMMs attached to the processor
  - Never cache remote data (no coherency traffic)
  - All requests go through the buffer
  - 128 KB, 4-way associative, 64 byte cache lines

XMT Programming Environment supports multithreading

- Flat distributed shared memory!
- Rely on the parallelizing compilers
  - They do great with loop level parallelism
- Many computations need to be restructured
  - To expose parallelism
  - For thread safety
- Light-weight threading
  - Full/empty bit on every word
    - **`writeef`/`readfe`/`readff`/`writeff`**
  - Compact thread state
  - Low thread overhead
  - Low synchronization overhead
  - Futures (see LISP)
- Performance tools
  - Apprentice2 – parse compiler annotations, visualize runtime behavior



HPCC Random Access

- Update a large table based on a random number generator
- **`NEXTRND`** returns next value of RNG

```
unsigned rnd = 1;
for(i=0; i<NUPDATE; i++) {
  rnd = NEXTRND(rnd);
  Table[rnd&(size-1)] ^= rnd;
}
```

- **`HPCC_starts(k)`** returns k-th value of RNG

```
for(i=0; i<NUPDATE; i++) {
  unsigned rnd = HPCC_starts(i);
  Table[rnd&(size-1)] ^= rnd;
}
```

- Compiler can automatically parallelize this loop
- It generates **`readfe`**/**`writeef`** for atomicity

HPCC Random Access - tuning

- **`HPCC_starts`** is expensive
- Restructure loop to amortize cost

```
for(i=0; i<NUPDATE; i+=bigstep) {
  unsigned v = HPCC_starts(i);
  for(j=0;j<bigstep;j++) {
    v = NEXTRND(v);
    Table[(v&(size-1)] ^= v;
  }
}
```

- The compiler parallelizes outer loop across all processors
- Apprentice2 reports
  - Five instructions per update (includes NEXTRND)
  - Two (synchronized) memory operations per update

HPCC Random Access - performance

- Performance analysis
  - Each update requires a read from and a write to a DIMM
  - Peak of 66 M cachelines/s/processor =>
  - Peak of 33 M updates/s/processor
- Single processor performance
  - Measured 20.9 M updates/s
- On 64 CPU preproduction system
  - Measured 1.28 Gup/s
- 95% scaling efficiency from 1P to 64P

Breadth first search

- Algorithm to find shortest path tree in unweighted graph

```
Parent[*] = null
Enqueue(source)
Parent[source] = source
While queue not empty:
   For all u already in queue:
      Dequeue(u)
      For all neighbors v of u:
         If Parent[v] is null:
            Parent[v] = u
            Enqueue(v)
```

Breadth first search

- An algorithm to find shortest path tree in unweighted graph

```
parent[*] = null                         ← parallel
enqueue(source)
parent[source] = source
while queue not empty:                   ← serial
   for all u already in queue:           ← parallel
      dequeue(u)
      for all neighbors v of u:          ← possibly parallel
         if Parent[v] is null:           ← atomic (readfe)
            parent[v] = u                ← writeef
            enqueue(v)
```

Breadth first search - queue

- Each vertex can be enqueued at most once
- Use an array of size |V| with `head` and `tail` pointers

```
oldtail = tail;
oldhead = head;
head = tail;
#pragma mta assert parallel
for(int i = oldhead; i<oldtail; i++) {
  Node u = Queue[i];
  …
}
```

Breadth first search – tuning and performance

- Tune on sparse Erdös-Rényi graphs
- Reduce overhead of queue operations
- Eliminate contention for queue `tail` pointer
- Performance counters show:
  - 2 memory operations/edge
  - 8.45 memory operations/vertex
- 32p system
  - 1 billion nodes/10 billion edges: ~17s
- 128p system
  - 4 billion nodes/40 billion edges: ~20s

Performance – rules of thumb

- Instructions are cheap compared to memory ops
  - Most workloads will be limited by bandwidth
- Keep enough memory operations in flight at all times
  - Load balancing
  - Minimize synchronization
- Use moderately cache friendly algorithms
  - Cache hits are not necessary to hide latency
  - Cache can improve effective bandwidth
    - ~40% cache hit rate for distributed memory
    - ~80% cache hit rate for nearby memory
  - Reduce cache footprint
  - Be careful about speculative loads (bandwidth is scarce)
- Think of XMT as a lot of processors running at 1 MHz

Traits of strong Cray XMT applications

1. Use lots of memory
   - Cray XMT supports terabytes
2. Lots of parallelism
   - Amdahl's law
   - Parallelizing compiler
3. Fine granularity of memory access
   - Network is efficient for all (including short) packets
4. Data hard to partition
   - Uniform shared memory alleviates the need to partition
5. Difficult load balancing
   - Uniform shared memory enables work migration

Summary

- Shared memory programming is good for productivity
- Cray XMT adds value for an important class of problems
  - Terabytes of memory
  - Irregular access with small granularity
  - Lots of parallelism exploitable by programming environment
- Working on scaling the system

Future example: Tree search

```
struct Tree {
  Tree *llink;
  Tree *rlink;
  int data;
};

int search_tree(Tree *root, int target) {
  int sum = 0;
  if (root) {
    future int left$;
    future left$(root, target) {
      return search_tree(root->llink, target);
    }
    sum = (root->data == target ? 1 : 0);
    sum += search_tree(root->rlink, target);
    sum += left$;
  }
  return sum;
}
```

Wait for `left$` to be full before adding it to the sum.