Introducing the Cray XMT

Petr Konecny
May 4th 2007
Agenda

- Origins of the Cray XMT
- Cray XMT system architecture
  - Cray XT infrastructure
  - Cray Threadstorm processor
- Shared memory programming model
  - Benefits/drawbacks/solutions
  - Basic programming environment features
- Strong Cray XMT application areas
- Examples
  - HPCC Random Access
  - Hash tables
- Summary
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

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)

Network
- PCI-X
- 10 GigE
- Fiber Channel
- RAID Controllers
Cray XMT Speeds and feeds

- 500M instructions/s
- 500M memory op/s
- 66M cache lines/s
- 4,8 or 16 GB DDR DRAM
- 140M memory op/s
- 110M→30M memory op/s (1→4K processors); bisection bandwidth impact

Diagram:
- Execution Pipes
- Threadstorm ASIC
- Switch
- Mem Cache
- DDR DRAM
- Seastar
- 3D Torus
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 (three per processor)

- Memory buffer (cache)
  - Only store data of the DIMMs attached to the processor
  - All requests go through the buffer
  - 128 KB, 4-way associative, 64 byte cache lines
Threadstorm CPU architecture (continued)
Shared memory model

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

- **Drawbacks**
  - Latency: time for a single operation
  - Network bandwidth limits performance
  - Legacy MPI codes
Cray XMT addresses shared memory drawbacks

- **Latency**
  - Little’s law: Concurrency = Bandwidth \* Latency
    - e.g.: 800 MB/s, 2\(\mu\)s 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
  - Efficient for small messages
  - Software controlled caching (registers, nearby memory)
    - Reduces network bandwidth
    - Eliminates cache coherency traffic
XMT Programming Environment supports multithreading

- Flat distributed shared memory!
- Rely on the parallelizing compilers
  - They do great with loop level parallelism
- Some 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
- Performance tools
  - Apprentice2 – parse compiler annotations, visualize runtime behavior
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
Several Cray XMT application areas

- Graph problems (intelligence, protein folding, bioinformatics)
- Optimization problems (branch-and-bound, linear programming)
- Computational geometry (graphics, scene recognition and tracking)
- Coupled physics with multiple materials and time scales

Let’s look deeper at two kernels common to these apps....
HPCC Random Access (part 1)

- Update a large table based on a random number generator
- **NEXTRND** returns next value of RNG
  ```c
  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
  ```c
  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 (part 2)

- HPCC\_starts is expensive
- Restructure loop to amortize cost
  
  ```c
  for(i=0; i<NUPDATE; i+=bigstep) {
    unsigned v = HPCC\_starts(i);
    for(j=0; j<bigstep; i++) {
      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 (part 3)

- **Performance analysis**
  - Most cache lines are dirty
  - Loads usually miss the memory buffer
    - Write back and evict some cache line
    - Load new data into freed cache line
  - The data is usually still in the buffer at the time of store
  - Two DIMM transfers per update
    - Peak of 33 M updates/s/processor

- **On 64 CPU pre-production hardware**
  - Hardware bug workarounds limit memory performance
  - 1.3 Gup/s on 64P: about 60% of peak
  - 95% scaling efficiency (from 1P to 64P)
Chained Hash Table

- **Table** is an array of pointers to **Node**
- **Node** contains **key** and a pointer to the next **Node**
Key Lookup

Node* lookup(KeyType key) {
    Node *node = Table[hash(key)];
    while (node && node->key != key) {
        node = node->next;
    }
    return node;
}

- Low concurrency per thread (a node at a time)
- Poor cache reuse
- Can perform multiple concurrent lookups
- Control dependency in the loop complicates vectorization
Using Hash table

- Use large table
  - To get O(1) amortized cost per operation
  - To avoid contention
- Single lookup is still limited by latency
- Insert random elements
- Lookup all elements, count how many are absent
  
  ```c
  for(i=0; i<DSIZE; i++)
      bad += (lookup(data[i]) == 0);
  ```
- The compiler parallelizes the loop across all processors
- It turns `+=` into a reduction
Software 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)
Summary

- Cray XMT adds value for an important class of problems
  - Terabytes of memory
  - Irregular access with small granularity
  - Lots of parallelism
- Shared memory programming is good for productivity
- Starting to gather numbers now on the pre-production system