Eldorado

John Feo
Cray Inc

Outline

- Why multithreaded architectures
- The Cray Eldorado
- Programming environment
- Program examples
Overview

- Eldorado is a peak in the North Cascades.
- Internal Cray project name for the next MTA system.
  - Make the MTA-2 cost-effective and supportable.
  - Retain proven performance and programmability advantages of the MTA-2.
- Eldorado
  - Hardware based on Red Storm.
  - Programming model and software based on MTA-2.

~1970’s Generic computer

Instructions

CPU

Memory

Memory keeps pace with CPU

Everything is in Balance…
... but balance is short-lived

- Processors have gotten much faster (~ 1000x)
- Memories have gotten a little faster (~ 10x), and much larger (~ 1,000,000x)
- System diameters have gotten much larger (~ 1000x)
- Flops are for free, but bandwidth is very expensive
- *Systems are no longer balanced, processors are starved for data*

20th Century parallel computer

Avoid the straws and there are no flaws; otherwise, good luck to you
Constraints on parallel programs

- Place data near computation
- Avoid modifying shared data
- Access data in order and reuse
- Avoid indirection and linked data-structures
- Partition program into independent, balanced computations
- Avoid adaptive and dynamic computations
- Avoid synchronization and minimize inter-process communications

A tiny solution space
Multithreading

- Hide latencies via parallelism
- Maintain multiple active threads per processor, so that *gaps introduced by long latency operations in one thread are filled by instructions in other threads*
- Requires special hardware support
  - Multiple threads
  - Single-cycle context switch
  - Multiple outstanding memory requests per thread
  - Fine-grain synchronization

The generic transaction

- Allocate memory
- Write/read memory
- Deallocate memory
Execution time line – 3 threads

Transaction per second on MTA

<table>
<thead>
<tr>
<th>Threads</th>
<th>P1</th>
<th>P2</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,490.55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>14,438.27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>26,209.85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>31,300.51</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>32,629.65</td>
<td>65,890.24</td>
<td>123,261.82</td>
</tr>
<tr>
<td>50</td>
<td>32,896.10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is not important

- Placing data near computation
- Modifying shared data
- Accessing data in order
- Using indirection or linked data-structures
- Partitioning program into independent, balanced computations
- Using adaptive or dynamic computations
- Minimizing synchronization operations

What is important

Parallelism
Eldorado system goals

- Make the Cray MTA-2 cost-effective and supportable
  - Mainstream Cray manufacturing
  - Mainstream Cray service and support

- Eldorado is a Red Storm with MTA-2 processors
  - Eldorado leverages Red Storm technology
  - Red Storm cabinet, cooling, power distribution
  - Red Storm circuit boards and system interconnect
  - Red Storm RAS system and I/O system (almost)
  - Red Storm manufacturing, service, and support teams

- Eldorado retains key MTA technology
  - Instruction set
  - Operating system
  - Programming model and environment

Red Storm

- Red Storm consists of over 10,000 AMD Opteron™ processors connected by an innovative high speed, high bandwidth 3D mesh interconnect designed by Cray (Seastar)

- Cray is responsible for the design, development, and delivery of the Red Storm system to support the Department of Energy's Nuclear stockpile stewardship program for advanced 3D modeling and simulation

- Red Storm uses a distributed memory programming model (MPI)
Eldorado 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
- MTX (BSD)

Eldorado system architecture

Eldorado CPU

Eldorado CPU
**Speeds and Feeds**

- **1.5 GFlops**
- **500M memory ops**
- **100M memory ops**
- **16 GB DDR DRAM**

Sustained memory rates are for random single word accesses over entire address space.

**Eldorado memory**

- **Shared memory**
  - Some memory can be reserved as local memory at boot time
  - Only compiler and runtime system have access to local memory

- **Memory module cache**
  - Decreases latency and increases bandwidth
  - No coherency issues

- **8 word data segments randomly distributed across the memory system**
  - Eliminates stride sensitivity and hotspots
  - Makes programming for data locality impossible
  - Segment moves to cache, but only word moves to processor

- **Full/empty bits on all data words**

```
  +------+
  | 63   |
  +------+
  | 0    |
  +------+
  ^ tag bits  | data values
  ^ full-empty  | forward
  ^ trap 1  | trap 2
```
### MTA-2 / Eldorado Comparisons

<table>
<thead>
<tr>
<th></th>
<th>MTA-2</th>
<th>Eldorado</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU clock speed</td>
<td>220 MHz</td>
<td>500 MHz</td>
</tr>
<tr>
<td>Max system size</td>
<td>256 P</td>
<td>8192 P</td>
</tr>
<tr>
<td>Max memory capacity</td>
<td>1 TB (4 GB/P)</td>
<td>128 TB (16 GB/P)</td>
</tr>
<tr>
<td>TLB reach</td>
<td>128 GB</td>
<td>128 TB</td>
</tr>
<tr>
<td>Network topology</td>
<td>Modified Cayley graph</td>
<td>3D torus</td>
</tr>
<tr>
<td>Network bisection bandwidth</td>
<td>3.5 * P GB/s</td>
<td>15.36 * P^23 GB/s</td>
</tr>
<tr>
<td>Network injection rate</td>
<td>220 MW/s per processor</td>
<td>Variable (next slide)</td>
</tr>
</tbody>
</table>

### Eldorado Scaling

<table>
<thead>
<tr>
<th>Example Topology</th>
<th>6x12x8</th>
<th>11x12x8</th>
<th>11x12x16</th>
<th>22x12x16</th>
<th>14x24x24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>576</td>
<td>1056</td>
<td>2112</td>
<td>4224</td>
<td>8064</td>
</tr>
<tr>
<td>Memory capacity</td>
<td>9 TB</td>
<td>16.5 TB</td>
<td>33 TB</td>
<td>66 TB</td>
<td>126 TB</td>
</tr>
<tr>
<td>Sustainable remote memory reference rate (per processor)</td>
<td>60 MW/s</td>
<td>60 MW/s</td>
<td>45 MW/s</td>
<td>33 MW/s</td>
<td>30 MW/s</td>
</tr>
<tr>
<td>Sustainable remote memory reference rate (aggregate)</td>
<td>34.6 GW/s</td>
<td>63.4 GW/s</td>
<td>95.0 GW/s</td>
<td>139.4 GW/s</td>
<td>241.9 GW/s</td>
</tr>
<tr>
<td>Relative size</td>
<td>1.0</td>
<td>1.8</td>
<td>3.7</td>
<td>7.3</td>
<td>14.0</td>
</tr>
<tr>
<td>Relative performance</td>
<td>1.0</td>
<td>1.8</td>
<td>2.6</td>
<td>4.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>
Software

- The compute nodes run MTX, a multithreaded unix operating system
- The service nodes run Linux
- I/O service calls are offloaded to the service nodes
- The programming environment runs on the service nodes

- Operating Systems
  - LINUX on service and I/O nodes
  - MTX on compute nodes
  - Syscall offload for I/O

- Run-Time System
  - Job launch
  - Node allocator
  - Logarithmic loader
  - Batch system – PBS Pro

- Programming Environment
  - Cray MTA compiler - Fortran, C, C++
  - Debugger - mdb
  - Performance tools: canal, traceview

- High Performance File Systems
  - Lustre

- System Mgmt and Admin
  - Accounting
  - Red Storm Management System
  - RSMS Graphical User Interface

CANAL

- Compiler ANALysis
- Static tool
- Shows how the code is compiled and why
What is Eldorado’s sweet spot?

- Any cache-unfriendly parallel application is likely to outperform on Eldorado
- Any application whose performance depends upon...
  - Random access tables (GUPS, hash tables)
  - Linked data structures (binary trees, relational graphs)
  - Highly unstructured, sparse methods
  - Sorting
- Some candidate application areas:
  - Adaptive meshes
  - Graph problems (intelligence, protein folding, bioinformatics)
  - Optimization problems (branch-and-bound, linear programming)
  - Computational geometry (graphics, scene recognition and tracking)

Sparse Matrix – Vector Multiply

- \( \mathbf{C}_{n \times 1} = \mathbf{A}_{n \times m} \cdot \mathbf{B}_{m \times 1} \)
- Store \( \mathbf{A} \) in packed row form
  - \( \mathbf{A}[nz] \), where \( nz \) is the number of non-zeros
  - \( \mathbf{cols}[nz] \) stores the column index of the non-zeros
  - \( \mathbf{rows}[n] \) stores the start index of each row in \( \mathbf{A} \)

```c
#pragma mta use 100 streams
#pragma mta assert no dependence
for (i = 0; i < n; i++) {
    int j;
    double sum = 0.0;
    for (j = rows[i]; j < rows[i+1]; j++)
        sum += A[j] * B[cols[j]];
    C[i] = sum;
}
```
Canal report

```c
#pragma mta use 100 streams
#pragma mta assert no dependence
for (i = 0; i < n; i++) {
    int j;
    double sum = 0.0;
    for (j = rows[i]; j < rows[i+1]; j++)
        sum += A[j] * B[cols[j]];
    C[i] = sum;
}
```

Parallel region 2 in SpMV
Multiple processor implementation
Requesting at least 100 streams

Loop 3 in SpMV at line 33 in region 2
In parallel phase 1
Dynamically scheduled

Loop 4 in SpMV at line 34 in loop 3
Loop summary: 3 memory operations, 2 floating point operations
3 instructions, needs 30 streams for full utilization, pipelined

Performance

- \( N = M = 1,000,000 \)
- Non-zeros 0 to 1000 per row, uniform distribution
  - \( N_z = 499,902,410 \)

<table>
<thead>
<tr>
<th>P</th>
<th>T</th>
<th>Sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.11</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>3.59</td>
<td>1.98</td>
</tr>
<tr>
<td>4</td>
<td>1.83</td>
<td>3.88</td>
</tr>
<tr>
<td>8</td>
<td>0.94</td>
<td>7.56</td>
</tr>
</tbody>
</table>

**IBM Power4 1.7 GHz**

\[ 26.1 \text{ s} \]

\[ \text{Time} = (3 \text{ cycles} \times 499902410 \text{ iterations}) / 220000000 \text{ cycles/sec} = 6.82 \text{ sec} \]

96% utilization
Any parallel sort works well on Eldorado

- Bucket sort is best if universe is large and elements don’t cluster in just a few buckets

**Bucket Sort**

*Count the number of elements in each bucket*

*Calculate the start of each bucket in dst*

*Copy elements from src to dst, placing each element in the correct segment*

*Sort the elements in each segment*

```c
for (i = 0; i < n; i++)
    count[src[i] >> shift] += 1;
```

- Compiler automatically parallelizes the loop using an `int_fetch_add` instruction to increment `count`
Step 2

```c
start(0) = 0;
for (i = 0; i < n_buckets; i++)
    start(i) = start(i-1) + count(i)
```

- Compiler automatically parallelizes most first-order recurrences

```
src
```
```
dst
```
```
start
```

Step 3

```c
#pragma mta assert parallel
#pragma mta assert noalias *src, *dst
for (i = 0; i < n; i++) {
    bucket = src(i) >> shift;
    index  = start[bucket]++;  
    dst(index) = src(i);
}
```

- Compiler can not automatically parallelize the loop because the order in which elements are copied to `dst` is not determinant
- `noalias` pragma lets compiler generate optimal code (3 instructions)
- The compiler uses an `int_fetch_add` instruction to increment `start`
Step 4

- Sort each segment
- Since there are many more buckets than streams and we assume the number of elements per bucket is small, any sort algorithm will do
  - we used a simple insertion sort
- Most of the time is spent skipping through buckets of size 0 or 1

Performance

- $N = 100,000,000$

<table>
<thead>
<tr>
<th>P</th>
<th>T</th>
<th>$Sp$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.14</td>
<td>1.0</td>
</tr>
<tr>
<td>2</td>
<td>4.56</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>2.41</td>
<td>3.79</td>
</tr>
<tr>
<td>8</td>
<td>1.26</td>
<td>7.25</td>
</tr>
</tbody>
</table>

$9.14 \text{ sec} \times \frac{220000000 \text{ cycles/sec}}{100000000 \text{ elements}} = 20.1 \text{ cycles/elements}$
Prefix operations on lists

- \( P(i) = P(i - 1) + V(i) \)

- **List ranking** - determine the rank of each node in the list
  - A common procedure that occurs in many graph algorithms

### Hellman and JaJa algorithm

- Mark \( n_{walk} \) nodes including the first node
  - This step divides the list into \( n_{walk} \) sublists

- Traverse the sublists computing each node's rank in the sublist

- From the lengths of the sublists, compute the start of each sublist

- Re-traverse each sublist, incrementing each node's local rank by the sublist's start
Steps 1 and 2

Step 3
Two more kernels

- Linked List Search

<table>
<thead>
<tr>
<th></th>
<th>N = 1000</th>
<th>N = 10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunFire 880 MHz (1P)</td>
<td>9.30</td>
<td>107.0</td>
</tr>
<tr>
<td>Intel Xeon 2.8 GHz (32b) (1P)</td>
<td>7.15</td>
<td>40.0</td>
</tr>
<tr>
<td>Cray MTA-2 (1P)</td>
<td>0.49</td>
<td>1.98</td>
</tr>
<tr>
<td>Cray MTA-2 (10P)</td>
<td>0.05</td>
<td>0.20</td>
</tr>
</tbody>
</table>

- Random Access (GUPs)

<table>
<thead>
<tr>
<th></th>
<th>Giga updates per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Power4 1.7 GHz (256P)</td>
<td>0.0055</td>
</tr>
<tr>
<td>Cray MTA-2 (1P)</td>
<td>0.41</td>
</tr>
<tr>
<td>Cray MTA-2 (5P)</td>
<td>0.204</td>
</tr>
<tr>
<td>Cray MTA-2 (10P)</td>
<td>0.405</td>
</tr>
</tbody>
</table>

Eldorado Timeline

- Early Systems (moderate-size of 500-1000P)
- Prototype Systems
- ASIC Tape-out

<table>
<thead>
<tr>
<th>Year</th>
<th>2004</th>
<th>2005</th>
<th>2006</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Eldorado Summary

- Eldorado is a low-risk development project because it reuses much of the Red Storm infrastructure which is part of Cray’s standard product path
  - Ranier module
  - Cascade LWP

- Eldorado retains the MTA-2’s easy to program model with parallelism managed by compiler and run-time
  - Ideal for applications that do not run well on SMP clusters
  - High productivity system for algorithms research and development

- Applications scale to very large system sizes

- 15-18 months to system availability