Optimizing a Parallel Runtime System for Multicore Clusters: A Case Study

Chao Mei, Gengbin Zheng, Fillipo Gioachin, Laxmikant V. Kale
08/03/2010, TeraGrid'10
Motivation

- Almost all clusters consist of multicore nodes
  - Node size continues to grow

- The whole software stack needs to be adapted to the multicore architecture
  - Application-level
  - Parallel languages (including its runtime system)
  - System-level

- Potential benefits
  - Latency is much reduced for intra-node messages
  - Shared-memory data structure

Initial porting of a runtime system doesn’t necessarily lead to benefits!
Outline

- Introduction to the runtime system
  - Charm++

- Experiment Setup
  - Benchmark
  - 5 multicore machines

- Issues and Optimization Techniques
  - Synchronization overhead
  - Affinity settings
  - ...

- Performance for real applications
The Runtime System Case: Charm++

- Objected oriented C++ based

- Message driven execution
  - Asynchronous non-blocking remote method invocation

Chao Mei (chaomei2@illinois.edu)
Parallel Programming Lab, UIUC
Architectures of Runtime System

- **non-SMP, process**
  - Network stack
  - POSIX shared memory

- **SMP, process + system thread**
  - Shared memory address space
Initial Experiments Result

- Applications do not have any performance improvement
  - NAMD: ~10% degradation
  - ChaNGa: ~2% degradation

- Attack the problem in two steps
  - Issues on a single node
  - Issues on multiple nodes
Experiment Setup: Benchmark

- kNeighbor (k=3 in our study)

Benchmark one iteration time
- Touch every byte of the message when received
- Emphasize the performance of message latency in the presence of contention
Experiment Setup: Multicore Machines

- **Five multicore machines**
  - A: AIX 6.1/IBM Power 5, a 16-core (SMT=2) node
  - B: Ubuntu 8.04/Intel Nehelem Xeon E5520, a 8-core (SMT=2) node
  - C: Ubuntu 8.04/Intel Harpertown Xeon E5405, a 8-core node
  - D: Ubuntu 8.04/AMD Barcelona Opteron 2356, a 8-core node
  - E: CentOS 5.4/Intel Dunnington Xeon E7450, a 24-core node
Initial Comparison for $k$Neighbor

![Graph showing comparison between SMP, non-SMP, and PXSHM for different message sizes.](image-url)
Network Progress Engine Issue

- **Network progress engine**
  - Process incoming messages and send outgoing message immediately
  - Expensive

- **Initial Usage**
  - Invoked every time a message is sent
    - contention on the engine

- **Current Usage**
  - Not necessary for intra-node message
  - Only invoke network progress engine if it is an inter-node message
Not simply change processes to threads and make it thread safe, but re-think the overall design of the architecture.
Multi-threaded Performance Issues

- Efficient locking and synchronization among threads
  - key factor for fast fine-grained intra-node communication

- Three issues
  - Memory management
  - Granularity of critical sections
  - Message queues
Memory Management

- **Charm++** uses its own memory allocator
  - Based on a GNU memory allocator developed seven years ago
  - Every malloc/free is protected with a lock

- Switched to OS provided memory module

![Graph showing iteration time (us) vs message size (Bytes)]
Performance of OS-provided Memory Module

- Synthetic benchmark: every thread simultaneously allocates memory of the same size for 100,000 times, then free

<table>
<thead>
<tr>
<th>#thds</th>
<th>A(us)</th>
<th>B(us)</th>
<th>C(us)</th>
<th>D(us)</th>
<th>E(us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.06</td>
<td>0.78</td>
<td>0.80</td>
<td>1.13</td>
<td>0.68</td>
</tr>
<tr>
<td>2</td>
<td>2.23</td>
<td>1.30</td>
<td>1.44</td>
<td>1.53</td>
<td>2.03</td>
</tr>
<tr>
<td>4</td>
<td>6.06</td>
<td>3.95</td>
<td>2.14</td>
<td>2.36</td>
<td>2.73</td>
</tr>
<tr>
<td>8</td>
<td>15.35</td>
<td>8.71</td>
<td>3.69</td>
<td>4.72</td>
<td>7.06</td>
</tr>
<tr>
<td>16</td>
<td>36.89</td>
<td>22.63</td>
<td>n/a</td>
<td>n/a</td>
<td>14.58</td>
</tr>
<tr>
<td>24</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>21.31</td>
</tr>
<tr>
<td>32</td>
<td>210.96</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Performance of OS-provided Memory Module

- Synthetic benchmark: every thread simultaneously allocates memory of the same size for 100,000 times, then free

<table>
<thead>
<tr>
<th>#thds</th>
<th>A(us)</th>
<th>A/M(us)</th>
<th>B(us)</th>
<th>C(us)</th>
<th>D(us)</th>
<th>E(us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.06</td>
<td>1.03</td>
<td>0.78</td>
<td>0.80</td>
<td>1.13</td>
<td>0.68</td>
</tr>
<tr>
<td>2</td>
<td>2.23</td>
<td>1.02</td>
<td>1.30</td>
<td>1.44</td>
<td>1.53</td>
<td>2.03</td>
</tr>
<tr>
<td>4</td>
<td>6.06</td>
<td>1.05</td>
<td>3.95</td>
<td>2.14</td>
<td>2.36</td>
<td>2.73</td>
</tr>
<tr>
<td>8</td>
<td>15.35</td>
<td>1.03</td>
<td>8.71</td>
<td>3.69</td>
<td>4.72</td>
<td>7.06</td>
</tr>
<tr>
<td>16</td>
<td>36.89</td>
<td>1.06</td>
<td>22.63</td>
<td>n/a</td>
<td>n/a</td>
<td>14.58</td>
</tr>
<tr>
<td>24</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>21.31</td>
</tr>
<tr>
<td>32</td>
<td>210.96</td>
<td>1.02</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Granularity of Critical Sections

- Trade-off between productivity and performance

![Graph showing iteration time vs message size](chart)

- PXSHM
- SMP with OS malloc (v2)
- SMP with smaller critical sections (v3)

Avg. 35.1%
Message Queues

- **Producer-Consumer Queues (PCQueue)**
  - Commonly used data structure for implementing scheduler queues

- **Scenario in Charm++**
  - Single consumer, multiple producers

- **Use memory fence instead of locks**
  - A general API across multiple platforms for read/write fence
  - Two steps of optimizations
    - Remove locks for consumer
    - Remove locks for producers by having a queue pair between the single consumer and each producer
      - Polling overhead increased
Perf. of Optimizing Message Queues

v3 vs. v4: avg. 9.7% gain
v4 vs. v5: avg. 19.5% gain
Handling Processor Private Variables

- Similar to the thread private variables in OpenMP
  - “Cpv” macros providing transparent usage in non-SMP/SMP mode, e.g. CpvAccess(var)

- Initial implementation is array-based:
  - CpvAccess(var) \(\rightarrow\) var[myrank]
  - false sharing 😞

- Solution: Thread Local Storage (TLS): explicit or implicit
  - pthread_setspecific/pthread_getspecific on Unix-like
  - TlsSetValue/TlsGetValue on Windows
  - “__thread” if supported by compiler and assembler

<table>
<thead>
<tr>
<th>Platform</th>
<th>A (ns)</th>
<th>B (ns)</th>
<th>C (ns)</th>
<th>D (ns)</th>
<th>E (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS</td>
<td>0.40</td>
<td>1.27</td>
<td>1.5</td>
<td>1.75</td>
<td>1.26</td>
</tr>
<tr>
<td>Array-based</td>
<td>51.58</td>
<td>17.52</td>
<td>10.03</td>
<td>9.61</td>
<td>8.50</td>
</tr>
</tbody>
</table>
Perf. Improvement After Using TLS

Avg. 26.5% gain
CPU Affinity (1)

- OS adopts natural affinity
  - Keep process/thread on the same CPU as long as possible
CPU Affinity (2)

- Just fixing the affinity shows performance improvement
  - Fewer L1 cache misses
  - Performance better and more stable

![Graph showing iteration time and L1 cache misses with and without CPU affinity]
CPU Affinity (3)

- How to set the CPU affinity generally?
  - A cross-platform function API in Charm++
  - Some TeraGrid sites also provide such functionality when launching the job

- What’s the optimal affinity setting?
  - Depends on the communication pattern of the program

- Example
  - $k$Neighbor in the case of $k=1$ with 7 elements
  - Message size: 256 bytes
  - Immediate neighbor communication
- \text{Elem}(0, 1, 2, 3, 4, 5, 6) \rightarrow \text{CPU}(0, 2, 4, 6, 1, 3, 5): 11.66 \text{ us}
- \text{Elem}(0, 1, 2, 3, 4, 5, 6) \rightarrow \text{CPU}(0, 1, 2, 3, 4, 5, 6): 13.37 \text{ us}
- \textbf{Why?}
  - Inter-chip: 8 vs. 24
  - Inter-die: 8 vs. 4
  - Intra-die: 12 vs. 0
Other Issues

- Reducing memory accesses in operations of message queues
  - Very fine-grained performance tuning

![Graph showing iteration time vs message size](image)

- PXSHM
- SMP with TLS (v6)
- SMP with simple queue (v7)

Avg. 8.1% gain
Overall Improvement for $k$Neighbor

- 14.4X over initial SMP
- 4.87X over non-SMP
- 1.21X over non-SMP in PXSHM
Application Performance: NAMD

Platform E (24-core)

Platform C (8-core)
Application Performance: ChaNGa

Platform C (cube300)

Platform C (dwf1)
Conclusion

- Studied the parallelization of a parallel language runtime system for multicore platforms via Charm++
  - Described various issues for the initial implementation
  - Applied optimization techniques correspondingly
    - Lock and synchronization overhead
    - CPU affinity
    - False sharing

- Should be general enough and useful to other runtime systems
Thank you!

http://charm.cs.uiuc.edu