The Utility of Fast Active Messages on Many-Core Chips

Efficient Supercomputing Project
Stanford University

Overview
In the many core era, power has become the limiting factor in performance scaling. This paper demonstrates the ability of active messages to increase the energy efficiency of parallel code.

• Active messages allow the user to manage data locality and communication.
• Integrating active messages with cache coherence simplifies programming.
• We have targeted and improved three key parallel programming idioms: reductions, contention, and data walks.
• Active messages enable significant runtime, efficiency, and scalability improvements in benchmarks.

Active Messages
Active messages invoke an atomic software handler at their destination. They allow the user to manage locality and overlap computation with communication, increasing energy efficiency and decreasing execution time.

Active Messaging Semantics
• Messages are sent to the home node of the targeted address.
• Handlers are atomic at destination, but must run to completion.
• Cache coherent shared memory programming model and hardware.
• User visible and customizable.

The sending core sends an AM with a specified destination address to the remote core.

Optimized Programming Idioms

Barriers & Reductions
In many programs, the end of a computation iteration is marked by a barrier or reduction step. Each core atomically modifies one or more global cache lines, signaling completion. This reduction computation can be simple or complex, ranging from a barrier to the bucket count updates in radix sort.

Optional Arguments
• barrier: Comparing a sequential barrier made with pipebar calls, barrier implementations vary.
• Contended: Each thread randomly updates 1 of 256 random variables.
• Walking: Each thread randomly selects a block of 64 or 16 entries in a large array and sums (or writes) them.

Software API

AM_Send(AM_Header* head)

User visible and cache coherent shared memory programming model and hardware.

• AM_Send
• Cache coherent shared memory programming model and hardware.
• User visible and customizable.

Using Cache Coherency

• We provide the following API functions and libraries for active messaging:
  • AM_Send(Am_Header* head): Sends the active message pointed to in the argument. The hardware reads the length field and immediately copies the message into the network. After this function call, the caller can overwrite the message with no side effects.
  • AM_Wait_For_Reply(AM_Header* reply): Causes the thread to sleep until the reply is delivered to the caller. The user written handler function that runs atomically and may not block. The arguments, which must be statically casted, are the destination object and message itself. Called by the hardware via the handler selector.

Software Example

This code example shows the implementation of a hash table insert function with active messages.

Hardware Implementation

• Each core is a 2-way multithreaded: One thread for the AM handler, and a second thread for execution.
• Short Active Messages are assembled in a specialized Active Message Reassembly Filter (AMRF). The AMRF has a much lower energy per access than the L1 cache.
• Incoming messages are queued into the L1 cache. If necessary, messages are buffered into the memory hierarchy.

Experimental Methodology

• We use a custom timing simulator with a PMP frontend.
• Each benchmark was hand-optimized with and without active messages.
• Unless noted, our baseline configuration is 256 cores with 256-16kB L1 caches, 16-100kB L2 caches, and a 16MB L3 cache.

Motivation

Energy Usage of Splash 2: Rate Sort

- Energy efficiency constraints performance.
- Data movement significantly impacts program execution energy and latency.
- Cache coherence enables programmability, but obfuscates locality.
- Active messages – the act of sending a message that triggers a handler at a remote node – allows for programmers to reason about locality while still maintaining the programmability of cache coherence.

Results

• Micro-benchmarks
  • BFS: Speedup from overlapping queries and removing globally contended variables.
  • Hash Table: Moving the data without active messages thrashes the L1 cache.
• Benchmarks
  • Kmeans: Significant reduction in the energy and time to do the reduction step.

Optimized Micro-benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AM Speedup</th>
<th>PT Speedup</th>
<th>AM Energy</th>
<th>PT Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduc</td>
<td>3.2</td>
<td>1.8</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>Pthread</td>
<td>4.5</td>
<td>3.0</td>
<td>1.2</td>
<td>1.8</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>6.0</td>
<td>4.2</td>
<td>1.8</td>
<td>2.4</td>
</tr>
<tr>
<td>Hash Table</td>
<td>7.1</td>
<td>5.3</td>
<td>2.1</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Scalability

- Active messages provide better performance scalability. Bottlenecks are a smaller part of the execution time, lessening the effect of Amdahl’s Law. The energy of the hash table increases with more cores in the baseline, as data must be moved longer distances. The AM version consumes less energy because the distributed hash table can fit into the L1 cache. All graphs are normalized to the 16 core version of a specific (AM or PT) implementation.