The Utility of Fast Active Messages on Many-Core Chips

Efficient Supercomputing Project

Stanford University

R. Curtis Harting
Vishal Parikh
Prof. William J. Dally

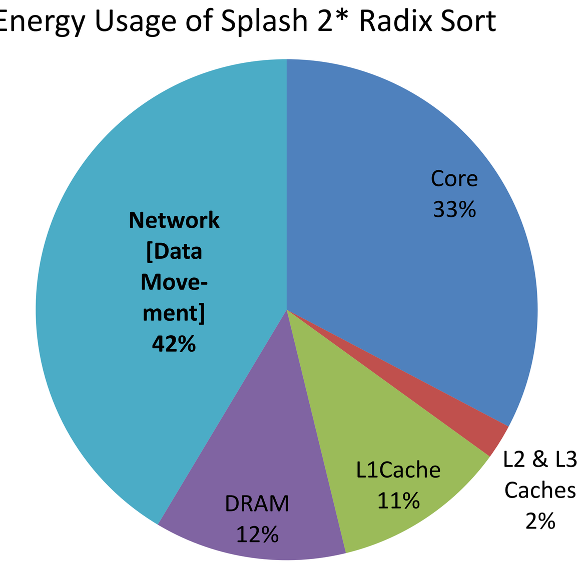
Overview

In the many core era, power has become the limiting factor in performance scaling. This poster 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 coherency 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.

Motivation

Energy Usage of Splash 2* Radix Sort



- Energy efficiency constrains performance
- Data movement significantly impacts program execution energy and latency
- Cache coherency 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

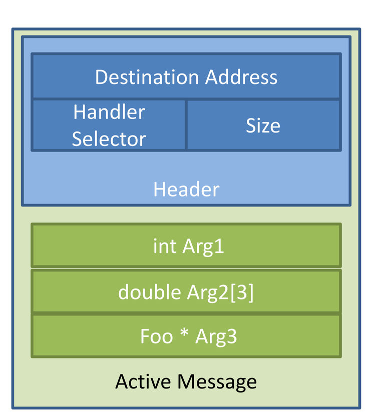
*S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995

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.

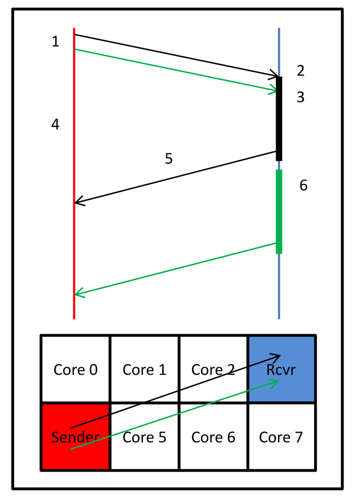
An active message itself is a user defined structure with the following fields:

- Destination Address – Messages are sent to the home node of this address, typically that of the targeted object
- Handler Selector – The handler function to be ran at the remote node
- Size – the size of the message
- Optional Arguments – Provided by the user



Active Messaging Semantics

- Messages are sent to the home node of the destination 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 2 messages to the receiver in rapid succession (1). When the first message arrives (2), the handler executes. The second message is queued (3) when it arrives. The sender can either sleep waiting for a response or continue executing code (4). The handler sends a response (5), completes, pops the waiting queue, and executes the second function (6).

Software API

We provide the following C++ function calls 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(int* replyAddr)**: Causes the thread to sleep until the int pointed to by replyAddr is non-zero and resets it to 0.
- **Handler_Function(void * daddr, void * msg)**: The user written handler function that runs atomically and may not block. The arguments, which must be statically cast, are the destination object and message itself. Called by the hardware via the handler selector
- We provide libraries for **barriers** and **locks**.

Software Example

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

```
struct AM_Header{
   void * daddr;
   void (*ip)(void *, void *);
   int size;
};
```

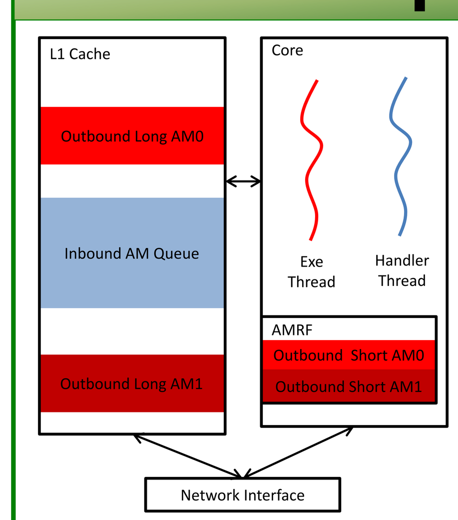
```
struct AM_hash{
   AM_Header head;
   AM_Reply* replyAddr;
   long key;
   long value;
   [enum] FUNC function;
};
```

```
bool HashAM::insert(long key,
long value){
 long hashVal =
      hashFunction(key);
 AM_hash* amh;
 //Setup the active message
 AM_Assemble(amh,
  /*destination*/
  &(hash_bkts[hashVal]),
  key, value, INSERT);
 AM_Send(&(amh->head));
 AM_wait_for_reply(replyAddr);
 return (reply != 0);
}
```

```
void am_handler(void * daddr,
        void * msg){
  AM_hash * amh = msg;
  switch(amh->function){
   case GET:  …
   case CONTAINS: ...
   case INSERT:
   retVal =  amh->hashTable->
    insert(amh->key,
     amh->value);
  break;
  case DEL:  …
  }
  AM_SendReply(amh, retVal);
}
```

- Sender hashes the value, assembles the AM, sends the AM, and waits for a response
- Handler parses the message and calls single threaded version of insert
- The handler sends a reply indicating insert success

Hardware Implementation

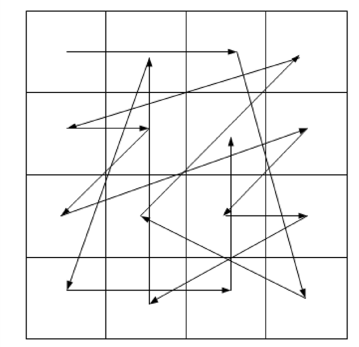


- Each core is 2-way multithreaded: One thread for the AM handler, and one thread for execution
- Short Active Messages are assembled in a specialized Active Message Register File (AMRF). The AMRF has a much lower energy per access than the L1 cache
- Incoming messages are queued into the L1 cache. If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

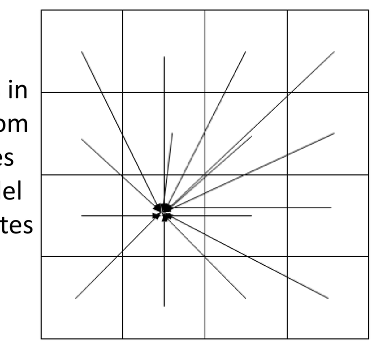
Optimized Programming Idioms

Barriers & Reductions

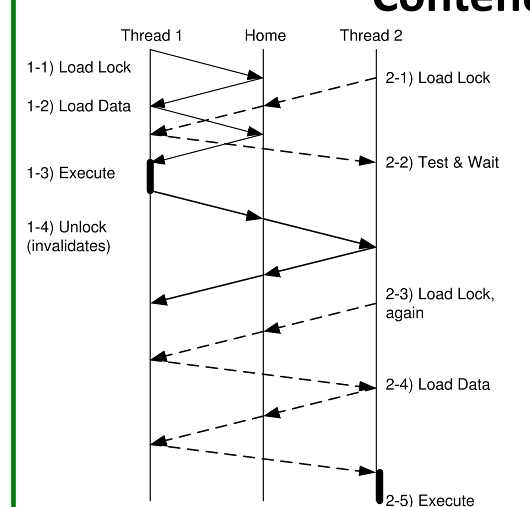
In many programs, the end of a computation iteration is marked by a barrier or reduction step. Each core atomically modifies one or more global cache lines, signaling completion. This reduction computation can be simple or complex, ranging from a barrier to the bucket count updates in radix sort.

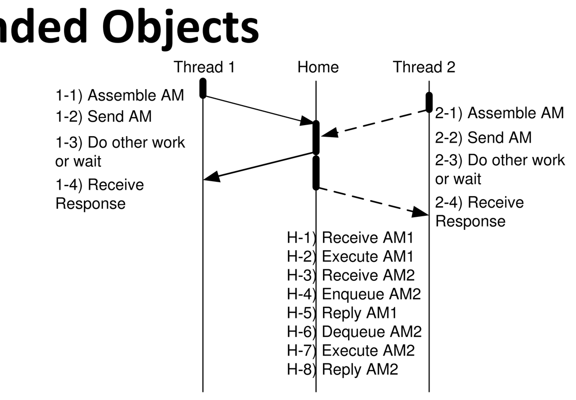


When threads hit a standard barrier (left), the data moves in serial and random fashion from core to core. Active messages (right) send the data in parallel to a single core. This eliminates the energy and delay of data movement.



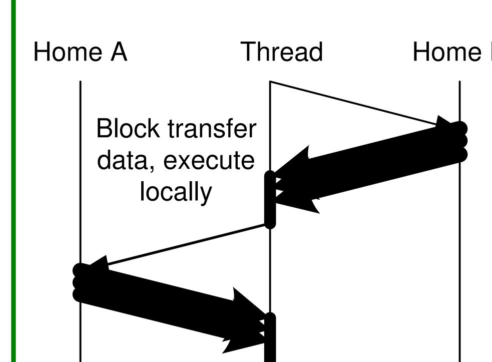
Contended Objects

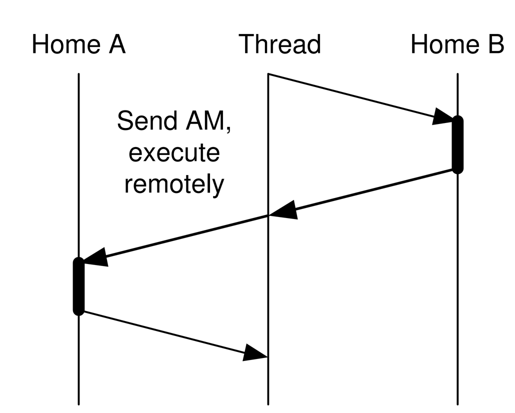




When multiple threads update a contended object, they must each acquire a lock and move the object to the L1 cache. Multiple threads performing this sequence at once leads to cache thrashing. Active messages remove this problem by atomically updating the object in a single location.

Data Walks





Traversing data structures without reuse wastes energy as the data is brought across the network and into the L1, polluting the cache. Active messages can be sent to the data, removing this costly movement.

Experimental Methodology

- We use a custom timing simulator with a PIN* frontend
- Each benchmark was hand coded with and without active messages
- Unless noted, our baseline configuration has 256 cores with 256-16kB L1 caches, 16-500kB L2 caches, and a 16MB L3 cache

Benchmarks

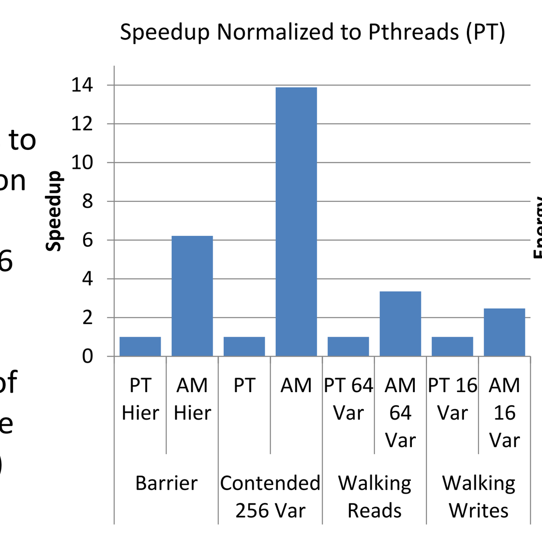
Name	Size	Reduction	Contended	Data Walk
Hash Table	512k Operations		✓	✓
Kmeans	8k & 131k Points	✓		
Radix Sort	1M elements	✓	✓	
Breadth First Search	262k Nodes	✓	✓	

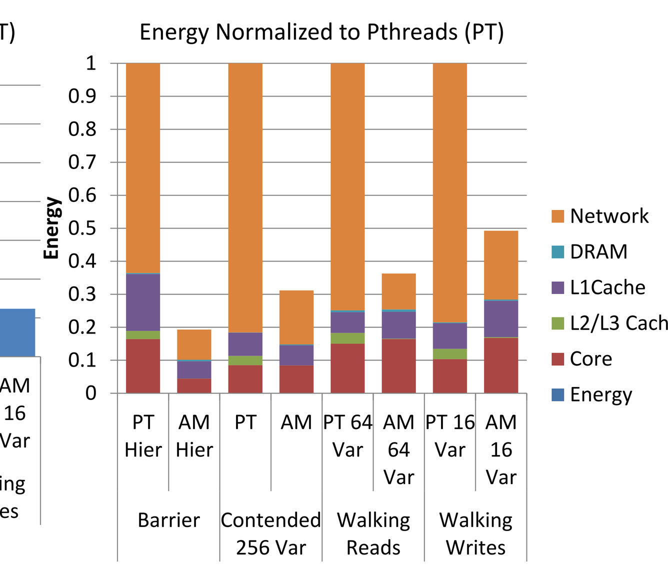
*C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40:190–200, June 2005.

Results

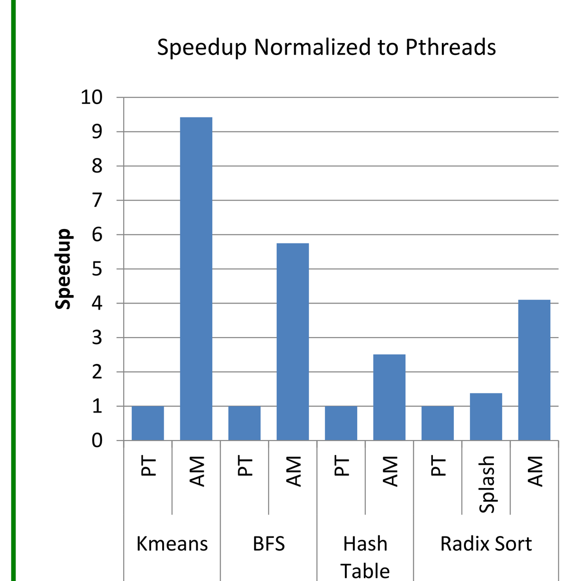
Micro-benchmarks

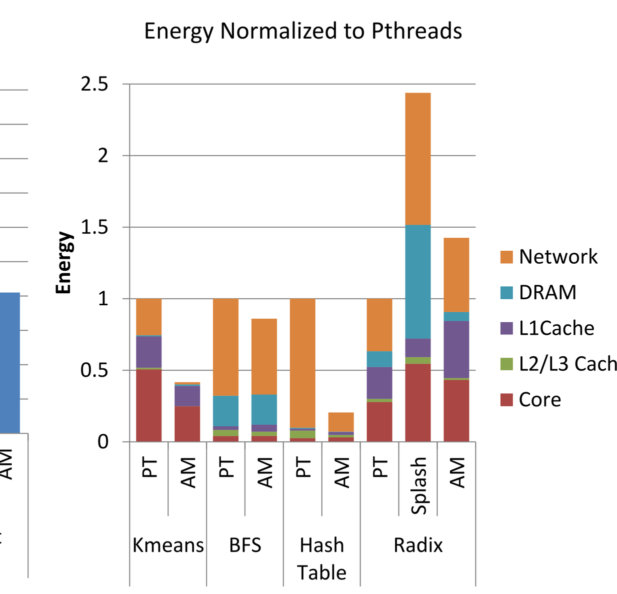
- **Barrier:** Comparing a hierarchical barrier made with pthread_barrier calls to an active messaging version
- **Contended:** Each thread randomly updates 1 of 256 random variables
- **Walking:** Each thread randomly selects a block of 64 (or 16) entries in a large array and sums (or writes) them.





Benchmarks

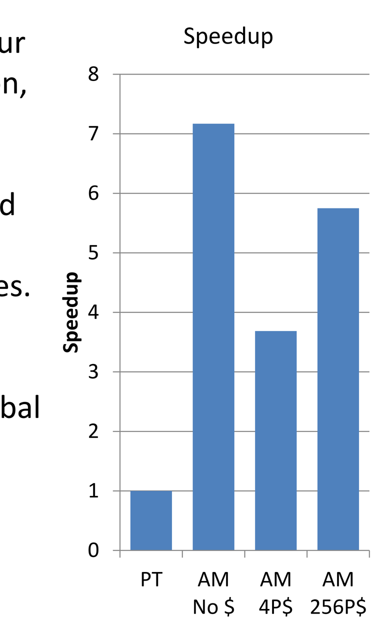


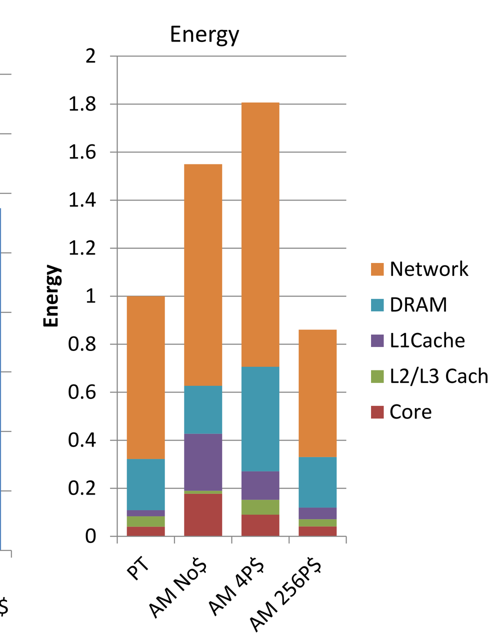


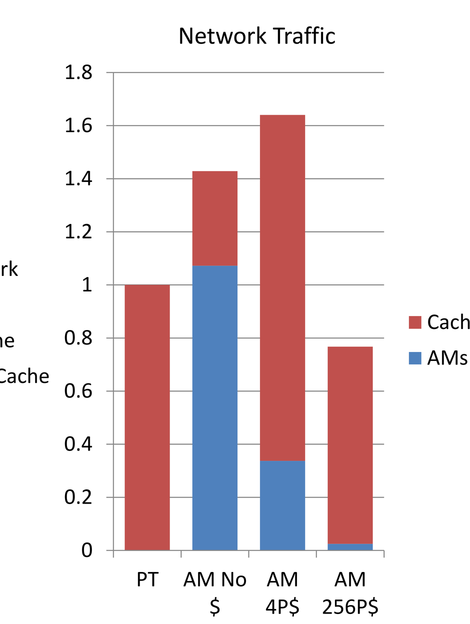
- **Kmeans:** Significant reduction in the energy and time to do the reduction step
- **BFS:** Speedup from overlapping queries and removing globally contended variables
- **Hash Table:** Moving the data without active messages thrashes the L1 cache
- **Radix:** The Splash version does not fit in the cache. The AMs suffer from overhead in doing updates

Using Cache Coherency

In the first version of our AM BFS implementation, we sent a message for each neighbor node, regardless of if they had been discovered. This sent too many messages. We now use the cache coherency protocol to keep a read-shared global array of visited nodes, only sending messages when necessary.







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.