Initial Kernel Timing Using a Simple PIM Performance Model

Daniel S. Katz1*, Gary L. Block1, Jay B. Brockman2, David Callahan3, Paul L. Springer1, Thomas Sterling1,4

1Jet Propulsion Laboratory, California Institute of Technology, USA
2University of Notre Dame, USA
3Cray Inc., USA
4California Institute of Technology, USA

*Technical Group Supervisor
Parallel Applications Technologies Group
http://pat.jpl.nasa.gov/
Daniel.S.Katz@jpl.nasa.gov

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under its Contract No. NBCH3039003.
Purpose of this Poster

- Discuss initial results of paper-and-pencil studies of 4 application kernels applied to a processor-in-memory (PIM) system roughly similar to the Cascade Lightweight Processor (LWP)

- Application kernels:
  - Linked list traversal
  - Vector sum
  - Bitonic sort

- Intent of work is to guide and validate work on Cascade in the areas of compilers, simulators, and languages
Poster Topics

- Generic PIM structure
- Concepts needed to program a parallel PIM system
  - Locality
  - Threads
  - Parcels
- Simple PIM performance model
- For each kernel:
  - Code(s) for a single PIM node
  - Code(s) for multiple PIM nodes that move data to threads
  - Code(s) for multiple PIM nodes that move threads to data
  - Hand-drafted timing forecasts, based on the simple PIM performance model
- Lessons learned
  - What programming styles seem to work best
    - Looking at both expressiveness and performance

Assembly  This Code  C  C++  Matlab
closer to h/w  more expressive
Generic Multi-PIM Structure

A PIM node

- Wide Memory
- Wide ALU
- Wide Registers
- Thread Scheduler
- Parcel Handler
- Parcels

Additional PIM nodes

Communication Fabric

256 bytes
Multi-PIM concepts

• **Locality**
  - Define a region in which items which can be operated upon in a single basic cycle
  - Things that are not local are remote

• **Threads**
  - Locus of local control and data
  - Associated with a region of memory referred to a set of registers
  - Ephemeral - creation and destruction are fast and easy
  - States: Active, Blocked
    - Active threads are scheduled and run
    - Blocked threads become active when some action occurs

• **Parcels**
  - Means of remote action
  - A packed version of a thread
### Performance Parameters

<table>
<thead>
<tr>
<th>Performance Parameters</th>
<th>Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Unit</td>
<td>1</td>
</tr>
<tr>
<td>Memory Cycle</td>
<td>16</td>
</tr>
<tr>
<td>Parcel Accept</td>
<td>4</td>
</tr>
<tr>
<td>Parcel Create</td>
<td>4</td>
</tr>
<tr>
<td>Parcel Transport</td>
<td>256</td>
</tr>
<tr>
<td>Thread Create</td>
<td>2</td>
</tr>
<tr>
<td>Instruction Cycle</td>
<td>4</td>
</tr>
</tbody>
</table>

- Note that these assumptions are not based on any particular hardware
- Specifically, they are not based on Cascade

Synchronization

- Producer/Consumer synchronization implemented through **full/empty** semantics
- Each memory location is considered either **full** or **empty**
  - This has no other impact on the content of the location
- Stores make a location **full** by default, but can have other behavior if needed
- Loads can block until a location is **full** or **empty**; they can make the location either **full** or **empty** when they complete
## Syntax for Full/Empty Loads and Stores

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val = readfe(&amp;loc)</code></td>
<td>Block until <code>loc</code> is full, then read <code>val</code>, leaving <code>loc</code> empty after read</td>
</tr>
<tr>
<td><code>val = readff(&amp;loc)</code></td>
<td>Block until <code>loc</code> is full, then read <code>val</code>, leaving <code>loc</code> full after read</td>
</tr>
<tr>
<td><code>(void) writeef(&amp;loc, val)</code></td>
<td>Block until <code>loc</code> is empty, then write <code>val</code>, leaving <code>loc</code> full after write</td>
</tr>
<tr>
<td><code>(void) writexf(&amp;loc, val)</code></td>
<td>Write <code>val</code>, leaving <code>loc</code> full after write</td>
</tr>
<tr>
<td><code>(void) purge(&amp;loc)</code></td>
<td>Set <code>loc</code> to empty</td>
</tr>
<tr>
<td><code>flag = REMOTE(&amp;loc)</code></td>
<td>Set flag to true if <code>loc</code> is remote, false otherwise</td>
</tr>
<tr>
<td><code>flag = LOCAL(&amp;loc)</code></td>
<td>Set flag to true if <code>loc</code> is local, false otherwise</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Linked List Traversal

Pseudocode:
Thread code:
void thread f(int *ptr, int *y) {
    int *x;

    tag1: x = ptr;
    ptr = *(x+1);
    if (ptr == NULL) {
        *y = *x;
        stop;
    } else {
        goto tag1;
    }
}

Calling thread’s code:
f(&head, &result);
last = readfe(&result);
## Linked List Traversal, Single PIM Case

<table>
<thead>
<tr>
<th>Code</th>
<th>Timing (cycles)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>void thread f(int *ptr, int *y) {</td>
<td>6</td>
<td>This requires thread creation and one instruction cycle</td>
</tr>
<tr>
<td>int *x;</td>
<td>0</td>
<td>Declaration</td>
</tr>
<tr>
<td>tag1: x = ptr;</td>
<td>20</td>
<td>One memory cycle (16) needed to load a value from &amp;ptr and one instruction cycle (4)</td>
</tr>
<tr>
<td>ptr = *(x+1);</td>
<td>5</td>
<td>Since the load from &amp;ptr actually loaded a full wide word, the value at (&amp;x+1) is already in a register, and copying it to the register we call ptr takes one functional operation and one instruction cycle. This assumes that &amp;ptr is even.</td>
</tr>
<tr>
<td>if (ptr == NULL) {</td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
<tr>
<td>*y = *x;</td>
<td>20</td>
<td>one memory access and one instruction cycle, or 20 cycles</td>
</tr>
<tr>
<td>stop;</td>
<td>0</td>
<td>Once the previous write is done, this time doesn’t matter. (It takes 5 cycles, however.)</td>
</tr>
<tr>
<td>} else {</td>
<td>5</td>
<td>Branch, which is a single functional operation/instruction cycle</td>
</tr>
<tr>
<td>goto tag1;</td>
<td>0</td>
<td>Also a branch, which is a single functional operation/instruction cycle that takes 5 cycles. The compiler should combine this operation with the previous branch, so this line is free</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The time required to run this thread is 6 cycles for startup, 35 cycles for each element of the list but the last, and 55 cycles for the last element of the list
- This is 35 cycles for each element of the list and 26 additional cycles in startup and shutdown
- This thread will take 3526 cycles to traverse a list containing 100 elements

Linked List Traversal, Multiple PIM Case 1

<table>
<thead>
<tr>
<th>Code</th>
<th>Timing (cycles)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>void thread f(int *ptr, int *y) { int *x; tag1: if (LOCAL(ptr)) { x = ptr; ptr = *(x+1); } else { } } else { goto tag1; } else { f(ptr, y); stop;</td>
<td>6</td>
<td>This requires thread creation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Declaration</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>One memory cycle (16) needed to load a value from &amp;ptr and one instruction cycle (4)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Since the load from &amp;ptr actually loaded a full wide word, the value at (&amp;x+1) is already in a register, and copying it to the register we call ptr takes one functional operation and one instruction cycle. This assumes that &amp;ptr is even.</td>
</tr>
<tr>
<td>if (ptr == NULL) { if (REMOTE(y)) { writexf(y,*x); stop;</td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>296</td>
<td>parcel creation (4), parcel transport (256), and instruction cycle (4) on the local node, plus parcel accept (8), memory operation (20), and instruction cycle (4) on the remote node (20)</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)</td>
</tr>
<tr>
<td>} else {</td>
<td>5</td>
<td>Branch, which is a single functional operation/instruction cycle</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>one memory access and one instruction cycle, or 20 cycles</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)</td>
</tr>
<tr>
<td>}</td>
<td>5</td>
<td>Branch, which is a single functional operation/instruction cycle</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Also a branch, which is a single functional operation/instruction cycle that takes 5 cycles. The compiler should combine this operation with the previous branch, so this line is free</td>
</tr>
<tr>
<td>} else {</td>
<td>5</td>
<td>Branch, which is a single functional operation/instruction cycle</td>
</tr>
<tr>
<td></td>
<td>276</td>
<td>parcel creation (4), parcel transport (256), and instruction cycle (4), plus parcel decode (8) on remote node</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here we send the thread to the data
Linked List Traversal, Multiple PIM Case 1 Analysis

- The time required to run this thread is:

\[ 6 + N \times R1 \times 40 + N \times (1-R1) \times 328 + R2 \times 20 + (1-R2) \times 296, \]

- N is the number of element in the list
- R1 is the frequency with which element j+1 is on the same PIM node as element j
- R2 is the frequency with which the last element is on the same PIM node as the register to which the last value is to be copied

- For 100-element list, with all elements on same PIM node, thread takes 4026 cycles
- Difference between this 4026 cycles and 3526 cycles in single PIM case is overhead of code used to check for local or remote references
- 100-element list with a blocked distribution on 10 PIM nodes (first 10 elements on one node, next 10 on another, etc., so R1 = 0.9 and R2 = 0.1), thread takes ~7000 cycles
- 100-element list with a random distribution on 10 PIM nodes (R1 = 0.1 and R2 = 0.1,) thread takes ~29000 cycles
# Linked List Traversal, Multiple PIM Case 2

Here we send the data to the thread.
Linked List Traversal, Multiple PIM Case 2 Analysis

- **Timing:**
  - Starting up a thread takes 6 cycles
  - Each local element of the list takes 45 cycles, and each remote element of the list takes 610 cycles
  - The final element of the list takes an additional 10 cycles if local and 296 cycles if remote

- For 100-element list with all elements on the same PIM node, thread takes 4506 cycles
  - Slightly longer than timing case 1, due to the slightly different way in which this code is written
  - Could be written to take ~ 4000 cycles, at the expense of clarity

- For case 2, the assumption of a blocked distribution or a random distribution is unimportant

- For a 100-element list in which 90 elements are on remote nodes, the time required for this thread about 55000 cycles, almost twice as much as case 1
  - In case 1, thread often had to move from one node to another
    - Time = parcel transport time x number of elements
  - Here, for each element, a parcel has to go to a remote node to get the data and another parcel has to bring the data back
    - Time = 2 x parcel transport time x number of elements
Linked List Traversal Summary

- Note that case 2 could be rewritten for a known blocked distribution of elements to gather more than one element at a time, but again, the round-trip parcel times would make this almost twice as costly as the first multi-PIM case for a blocked list.

- Summary for 100 element list and 10 PIM nodes:

<table>
<thead>
<tr>
<th>Case Description</th>
<th>Number of Cycles (x 1000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single PIM node, all elements on 1 node</td>
<td>3.5</td>
</tr>
<tr>
<td>Multi-PIM node case 1, all elements on 1 node</td>
<td>4</td>
</tr>
<tr>
<td>Multi-PIM node case 1, elements block distributed</td>
<td>7</td>
</tr>
<tr>
<td>Multi-PIM node case 1, elements randomly distributed</td>
<td>29</td>
</tr>
<tr>
<td>Multi-PIM node case 2, all elements on 1 node</td>
<td>4.5</td>
</tr>
<tr>
<td>Multi-PIM node case 2, elements block distributed</td>
<td>55</td>
</tr>
<tr>
<td>Multi-PIM node case 2 modified to move elements to thread in blocks, elements block distributed</td>
<td>~14</td>
</tr>
<tr>
<td>Multi-PIM node case 2, elements randomly distributed</td>
<td>55</td>
</tr>
</tbody>
</table>
## Vector Sum, Single PIM Case

<table>
<thead>
<tr>
<th>Code</th>
<th>Timing (cycles)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>void thread vector_sum(int *x, int n, int *result) {&lt;br&gt;int sum;&lt;br&gt;int *end = x + n;&lt;br&gt;tag1: if (x &lt; end) {&lt;br&gt;    sum += *x;&lt;br&gt;    x++;&lt;br&gt;    goto tag1;&lt;br&gt;} else {&lt;br&gt;    *result = sum;&lt;br&gt;} }</td>
<td>6</td>
<td>This requires thread creation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>Declaration</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>branch, which is a single functional operation/instruction cycle</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>branch, which is a single functional operation/instruction cycle</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>one functional operation and one instruction cycle</td>
</tr>
</tbody>
</table>

- Starting the thread takes 16 cycles, each element of the vector takes 20 cycles, and the final element takes an extra 15 cycles
- Time needed for vector of length N is $31+20*N$
- For vector of length 100,000, ~ 2,000,000 cycles
Vector Sum, Multiple PIMs

Case 1:

```c
void thread vector_sum(int *x, int n, int *result) {
    int *res;
    int i, sum, num_blocks;

    num_blocks = n/BLOCKSIZE;
    res = malloc(num_blocks*sizeof(int));
    for (i=0; i<num_blocks; i++) {
        purge(&res[i]);
        vector_sum0(x+i*BLOCkSIZE, &res[i]);
    }
    sum = 0;
    for (i=0; i<num_blocks; i++) {
        sum += readff(&res[i]);
    }
    *result = sum;
}

void thread vector_sum0(int *x, int *result) {
    int sum;
    int *end = x + BLOCKSIZE;
    tag1: if (x < end) {
        sum += *x;
        x++;
        goto tag1;
    } else {
        *result = sum;
    }
}
```

Case 2:

```c
void thread vector_sum(int *x, int n, int *result) {
    int right, left, k;
    int *end;
    if (n > BLOCKSIZE) {
        // if more than one block, recurse in parallel
        // then add results
        k = (n < 2*BLOCkSIZE) ? BLOCKSIZE :
            (n/2) & ~(BLOCKSIZE-1);
        purge(&left);
        purge(&right);
        vector_sum(x+k, n-k, &right);
        vector_sum(x, k, &left);
        *result = readff(&left) + readff(&right);
    } else {
        end = x + n;
        right = 0;
        tag1: if (x < end) {
            right += *x;
            x++;
            goto tag1;
        } else {
            *result = right;
        }
    }
}
```

Vector distributed by BLOCKSIZE contiguous elements on a node

Vector Sum Analysis and Discussion

- **Case 1:**
  - \# parcels = 2*n/BLOCKSIZE
  - \# threads = n/BLOCKSIZE+1
    \[(n/BLOCKSIZE \text{ threads do sums)}\]
  - First thread needs n/BLOCKSIZE extra words of memory
  - To avoid extra memory in thread 1:
    - Could use atomic memory operations in vector_sum0, where these threads would increase a running sum in vector_sum by their partial sum, then increment a counter in vector_sum
    - vector_sum would block on the counter until all vector_sum0 threads finished
  - 1/2 the parcels issued at single time from one PIM node
  - Likely other 1/2 will be sent back to the PIM node at about the same time as each other
  - Potential for network hotspots

- **Case 2:**
  - \# parcels = 4*n/BLOCKSIZE-4
  - \# threads = 2*n/BLOCKSIZE-1
    \[(n/BLOCKSIZE \text{ threads do sums)}\]
  - 2x threads of case 1
  - 2x parcels of case 1
  - Each thread uses only about 4 words of memory more than case 1
  - No hotspot issues

- **Timing of both cases likely similar**
  - Both dominated by BLOCKSIZE (the actual sums)
    \[(Assuming that n/BLOCKSIZE is big)\]
  - Option chosen depends on resource issues and relative cost of thread creates, memory operations, network issues, etc.
Bitonic Sort

Pseudocode:
```c
for (k = 2; k <= N; k = k * 2) {
    for (j = k / 2; j >= 1; j = j / 2) {
        for (i = 0; i < N; i = i + 1) {
            ij = i^j;
            if (ij==i) {
                p1 = (i&k)==0);
                p2 = (x(i) > x(ij));
                if (p1 == p2) { // if p1, want x(i) < x(ij), if not p1, want x(i) > x(ij)
                    tmp = x(i);
                    x(i) = x(ij);
                    x(ij) = tmp;
                }
            }
        }
    }
}
```

The comparisons and possible swaps in a bitonic sort of N(=16) elements:
Bitonic Sort - Single thread

- Single thread
- Reads each pair of potential swap values, then writes them back in potentially swapped order

An alternative is to start a thread at the first potential swapee’s location, and let it decide to do or not to do a swap, based on the value at the second potential swapee’s location

Two ways to write this code, as shown next…
Bitonic Sort with Parallelism

- Synchronization becomes important
- Must ensure that swaps from each stage use data that belongs to that stage
- Two methods below work, first (a) is used because it has less communication
- Bitonic_sorter thread could start each potential swap, block until potential swap completes
- Or, could start all potential swaps for a stage at once, wait for them all to return
  - Would have N+1 threads active at once (1 bitonic sorter, N/2 comp_swap1, N/2 comp_swap2)
  - See code on next panel
void thread bitonic_sorter(int *y, int *data, int N) {

    int i,j,k,ij,tmp;

    for (k = 2; k <= N; k = k * 2) {
        for (j = k / 2; j >= 1; j = j / 2) {
            for (i = 0; i < N; i = i + 1) {
                ij = i^j;
                if (ij>i) {
                    order = ((i&k)==0);
                    // start a thread to do the
                    // potential swap
                    purge(&tmp);
                    (void) comp_swap1(&(x[i]),&(x[ij]), order, &tmp);
                    ij = readfe(&tmp);
                }
            }
        }
    }
}

void comp_swap1 (int *my_x_loc, int *other_x_loc, logical order, int *end) {

    purge (my_x_loc);
    comp_swap2(other_x_loc, *my_x_loc, order, my_x_loc);
    (void) readff(my_x_loc);
    writexf(end, 1);
}

void comp_swap2 (int *my_x, int other_x, logical order, int *other_x_loc) {
    int tmp;

    if (order == (other_x > *my_x)) {
        tmp = *my_x;
        *my_x = other_x;
    } else {
        tmp = other_x;
    }
    writexf(other_x_loc, tmp);
}
Bitonic Sort - Other Options

- Could create a thread for each comparison/exchange operation (one for each pair of i iterations)
  - Each thread could execute when its predecessors had completed
  - \( \log_2(N) \) stages w/ between 1 and \( \log_2(N) \) steps w/ \( N/2 \) compare/exchange operations => \( \frac{(\log_2 N)(\log_2 N + 1)}{4} \) threads
    - only \( \sim N/2 \) threads active at any time
- Could create all the needed threads at once, where each blocks until two parcels are received from its predecessors
  - Do this by working backwards, spawning threads for last stage of sorts, then next to last stage of sorts, etc.
  - Syntax issue: how to tell thread where to send parcels when thread creator doesn’t have info about thread’s frame
  - Other issue: creating this number of threads may be problematic.
- Could do this using objects
  - Create a sorter object that waits for \( 2^N \) messages before sending a parcel back to the creator thread
  - Create objects for last stage’s swaps, w/ each object created on PIM node holding first element of that swap; tell objects not to start until they receive two parcels, and to send two messages to the sorter object when they are done
  - Then create next-to-last set of swapper objects, tell them to wait for 2 parcels, and when they are done to send a message to each of the appropriate swapper objects in last set
  - This process continues until we reached first stage’s swapper objects, which are told to start immediately upon creation, and to send a message upon completion to each of the second stage’s swapper objects
  - Total number of objects created \( \sim \) number of threads in the previous example
  - Main differences: Objects may use fewer resources than threads; syntax issues with threads communicating with other threads doesn’t appear
- Could rewrite code on previous panel as one thread per data element
  - Equivalent to swapping order of the loops so that i is the outermost loop, and parallelizing across I
  - Could be written using threads or objects
  - Difficult to write with threads, because it requires threads to be created with the knowledge of other threads, where those other threads have not yet been created, but since these threads have not yet been created, the addresses of their registers do not exist
  - Easier to write with objects
Conclusions

• Moving thread to data has potential to shorten runtime
• Coding for parallelism introduces overhead even when no parallelism exists
• Fairly simple syntax can be used to express complicated synchronization behaviors
• Tradeoffs between recursive and non-recursive thread programming should be examined
• Resource issues are important to understand, but may be very implementation dependant

• Some communication patterns are difficult to express w/ threads, but may be easier to express w/ objects (as shown above)