Initial Kernel Timing Using a Simple PIM Performance Model

Daniel S. Katz[1*], Gary L. Block[1], Jay B. Brockman[2], David Callahan[3], Paul L. Springer[1,] Thomas Sterling[1,4]

[1]*Jet Propulsion Laboratory, California Institute of Technology, USA*

[2]*University of Notre Dame, USA*

[3]*Cray Inc., USA*

[4]*California 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

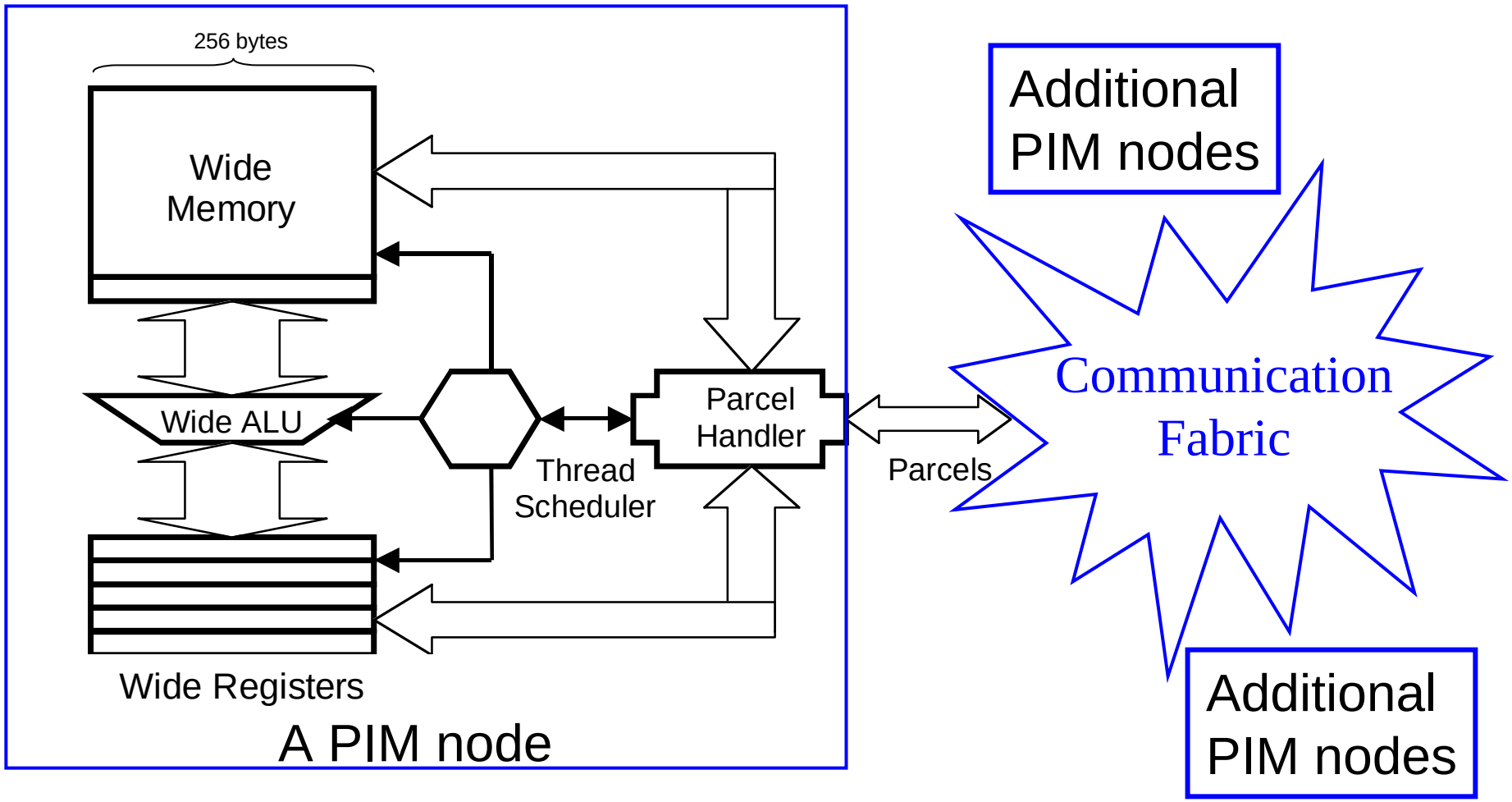
- Intent of work is to guide and validate work on Cascade in the areas of compilers, simulators, and languages

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

Generic Multi-PIM Structure



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

Performance Parameters	Time (cycles)
Function Unit	1
Memory Cycle	16
Parcel Accept	4
Parcel Create	4
Parcel Transport	256
Thread Create	2
Instruction Cycle	4

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

Synchronization

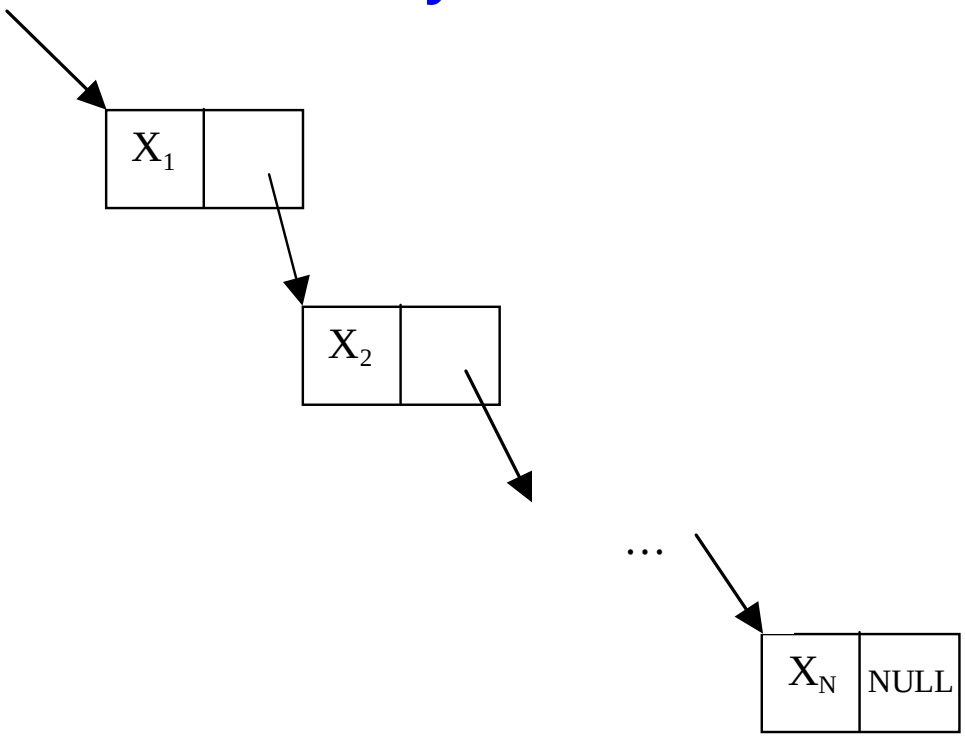
- Producer/Consumer synchronization implemented through **full/empty** semantics
- Each memory location is considered either **full** or **empty**
  - This has no other impact on the content of the location
- Stores make a location **full** by default, but can have other behavior if needed
- Loads can 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

Function	Description
`val = readfe(&loc)`	Block until **`loc`** is full, then read **`val`**, leaving **`loc`** empty after read
`val = readff(&loc)`	Block until **`loc`** is full, then read **`val`**, leaving **`loc`** full after read
`(void) writeef(&loc, val)`	Block until **`loc`** is empty, then write **`val`**, leaving **`loc`** full after write
`(void) writexf(&loc, val)`	Write **`val`**, leaving **`loc`** full after write
`(void) purge(&loc)`	Set **`loc`** to empty
`flag = REMOTE(&loc)`	Set flag to true if **`loc`** is remote, false otherwise
`flag = LOCAL(&loc)`	Set flag to true if **`loc`** is local, false otherwise
…	…

Linked List Traversal

Linked list as stored in memory:



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

Code	Timing (cycles)	Comment
`void thread f(int *ptr, int *y) {`	6	This requires thread creation and one instruction cycle
`int *x;`	0	Declaration
`tag1: x = ptr;`	20	One memory cycle (16) needed to load a value from `&ptr` and one instruction cycle (4)
`ptr = *(x+1);`	5	Since the load from `&ptr` actually loaded a full wide word, the value at (`&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 `&ptr` is even.
`if (ptr == NULL) {`	5	one functional operation and one instruction cycle
`*y = *x;`	20	one memory access and one instruction cycle, or 20 cycles
`stop;`	0	Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)
`} else {`	5	Branch, which is a single functional operation/instruction cycle
`goto tag1;`	0	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
`}`		
`}`		

- 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

Code	Timing (cycles)	Comment
void thread f(int *ptr, int *y) {	6	This requires thread creation and one instruction cycle
int *x;	0	Declaration
tag1: if (LOCAL(ptr)) {	5	one functional operation and one instruction cycle
x = ptr;	20	One memory cycle (16) needed to load a value from &ptr and one instruction cycle (4)
ptr = *(x+1);	5	Since the load from &ptr actually loaded a full wide word, the value at (&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 &ptr is even.
if (ptr == NULL) {	5	one functional operation and one instruction cycle
if (REMOTE(y)) {	5	one functional operation and one instruction cycle
writexf(y,*x);	296	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)
stop;	0	Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)
} else {	5	Branch, which is a single functional operation/instruction cycle
*y = *x;	20	one memory access and one instruction cycle, or 20 cycles
stop;	0	Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)
}		
} else {	5	Branch, which is a single functional operation/instruction cycle
goto tag1;	0	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
} else {	5	Branch, which is a single functional operation/instruction cycle
f(ptr, y);	276	parcel creation (4), parcel transport (256), and instruction cycle (4), plus parcel decode (8) on remote node
stop;	0	Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)
}		
}		

Here we send the thread to the data

Linked List Traversal, Multiple PIM Case 1 Analysis

- The time required to run this thread is:

  $$6 + N*R1*40 + N*(1-R1)*328 + R2*20 + (1-R2)*296,$$

  - 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

Code	Timing (cycles)	Comment
void thread f(int *ptr, int *y) {	6	This requires thread creation and one instruction cycle
int tmp[2], x;	0	Declaration
tag1: if (LOCAL(ptr)) {	5	one functional operation and one instruction cycle
x = ptr[0];	20	One memory cycle (16) needed to load a value from &ptr and one instruction cycle (4)
ptr = *(ptr[1]);	5	Since the load from &ptr actually loaded a full wide word, the value at (&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 &ptr is even.
goto tag2;	5	one functional operation and one instruction cycle
} else {	5	one functional operation and one instruction cycle
purge tmp[0];	5	one functional operation and one instruction cycle
tmp[0] = ptr[0];	8	parcel create (4) and instruction cycle (4)
tmp[1] = ptr[1];	0	assumes the compiler can combine this with the previous line
x = readff(&tmp[0])	572	for this to complete, the previously generated parcel must be transported (256), the parcel accepted/decoded (16), a memory access completed (20), a parcel generated to send that data back to this processor (4), transport of that second parcel (256), parcel accept on this node (16), and an instruction cycle (4)
ptr = *(tmp[1]);	5	one functional operation and one cycle, partially combined with previous line
}		
tag2: if (ptr== NULL) {	5	one functional operation and one cycle, partially combined with previous line
if REMOTE(y)) {	5	one functional operation and one cycle, partially combined with previous line
writexf(y,x);	296	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
stop;	0	Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)
} else {	5	one functional operation and one cycle, partially combined with previous line
*y = x;	5	one functional operation and one cycle, partially combined with previous line
stop;	0	Once the previous write is done, this time doesn't matter. (It takes 5 cycles, however.)
} else {	5	one functional operation and one cycle, partially combined with previous line
goto tag1;	0	combined with line above
}		
}		

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 a costly as the first multi-PIM case for a blocked list

- Summary for 100 element list and 10 PIM nodes:

Case Description	**Number of Cycles (x 1000)**
Single PIM node, all elements on 1 node	3.5
Multi-PIM node case 1, all elements on 1 node	4
Multi-PIM node case 1, elements block distributed	7
Multi-PIM node case 1, elements randomly distributed	29
Multi-PIM node case 2, all elements on 1 node	4.5
Multi-PIM node case 2, elements block distributed	55
Multi-PIM node case 2 modified to move elements to thread in blocks, elements block distributed	~14
Multi-PIM node case 2, elements randomly distributed	55

Vector Sum, Single PIM Case

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

- 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:

```
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 thead 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:

```
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;
     }
     *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 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 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

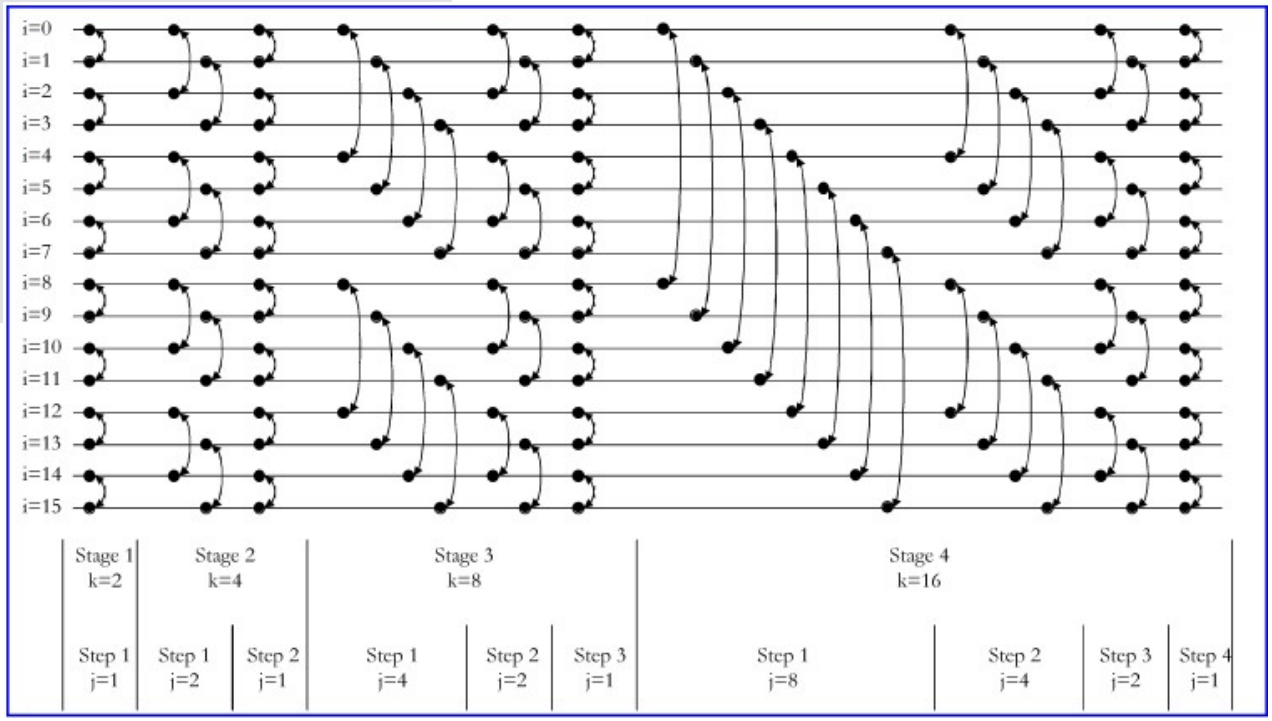
- Timing of both cases likely similar
  - Both dominated by BLOCKSIZE (the actual sums)
    (Assuming that n/BLOCKSIZE is big)
  - Option chosen depends on resource issues and relative cost of thread creates, memory operations, network issues, etc.

Bitonic Sort

Pseudocode:

```
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

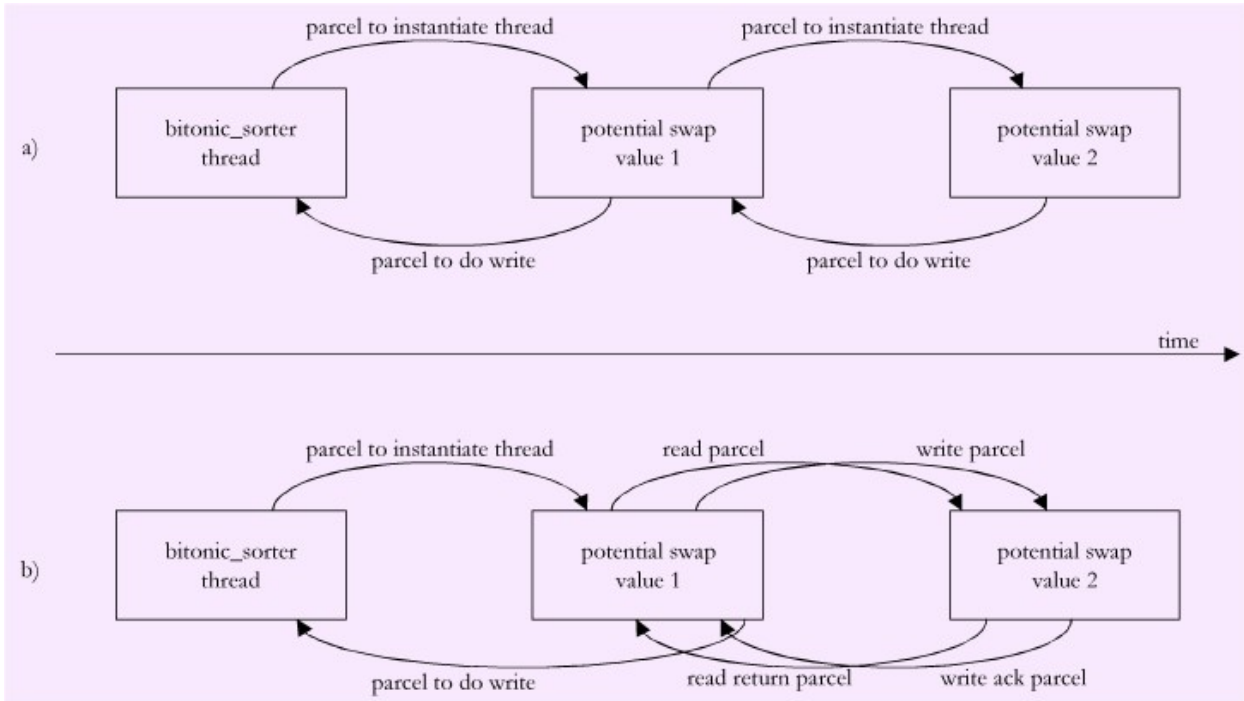
- Single thread
- Reads each pair of potential swap values, then writes them back in potentially swapped order

- 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…

```
void thread bitonic_sorter(int *y, int *data, int N) {

  int i,j,k,ij,x1,x2;

  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) {
      // get the two values
      purge(x1);
      purge(x2);
      x1 = readfe(&(data[i]));
      x2 = readfe(&(data[ij]));
      //check for a swap
      p1 = ((i&k)==0);
      p2 = (readff(&x1) > readff(&x2));
      if (p1 == p2) {
       // send back the swapped values
       writexf(&(data[i]), x2);
       writexf(&(data[ij]), x1);
      } else {
       writexf(&(data[i]), x1);
       writexf(&(data[ij]), x2);
      }
     }
    }
   }
  }
  writexf(y,1);
}
```

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



Bitonic Sort Code - Multiple threads

```
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);
     }
    }
   }
  }
}
```

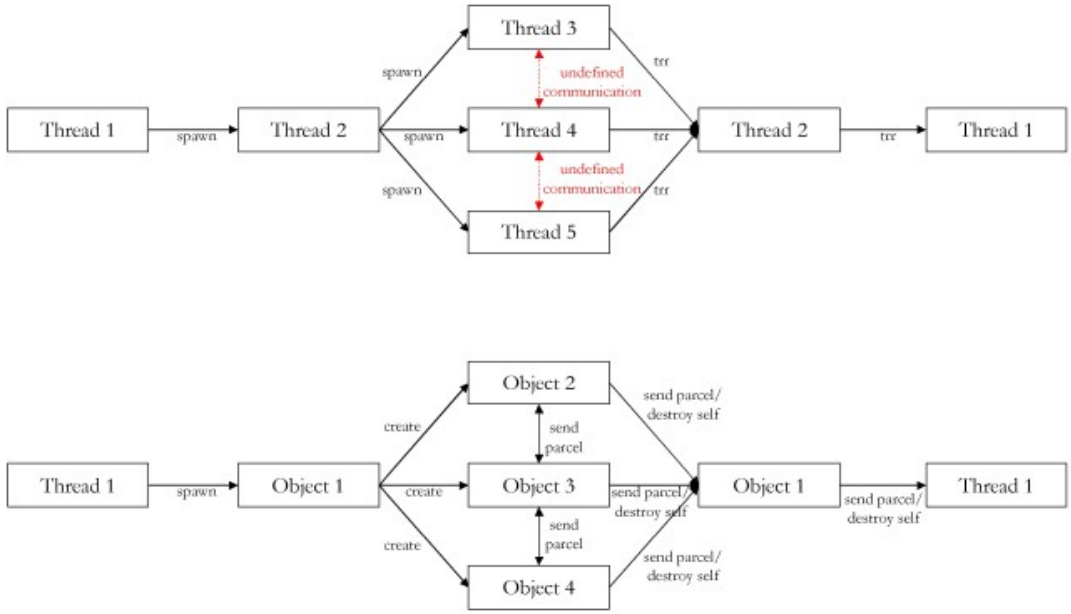
```
void comp_swap1 (int *my_x_loc,
     int *other_x_loc, logical order,
     int *end) {

  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
  - log2(N) stages w/ between 1 and log2(N) steps w/ N/2 compare/exchange operations => $\frac{(\log_2 N)(\log_2 N+1)N}{4}$ threads
    - only ~ 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 umber of objects created ~= 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)