# HTA as a High-Level Programming Model for Codelet Execution

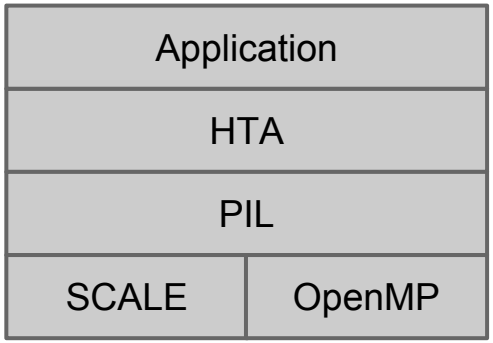
Chih-Chieh Yang
May 2014

# Outline

- Overview
- Parallel Intermediate Language (PIL)
- Hierarchically Tiled Array (HTA)
  - Semantics
  - Execution Model
- Experiments
  - NAS Benchmarks
  - Mini-benchmark
- Conclusion

# Overview

- Hierarchically Tiled Array (HTA) is a high-level programming model for expressing parallel computation with operations on tiled arrays
- We've implemented the HTA library based on Parallel Intermediate Language (PIL)
- Users write application code in HTA
- At compile time, PIL compiler translates the code into SCALE code

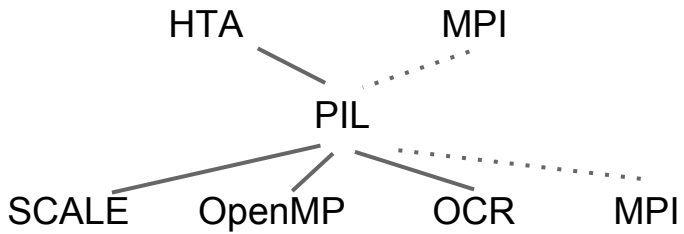


# Outline

- Overview
- Parallel Intermediate Language (PIL)
- Hierarchically Tiled Array (HTA)
  - Semantics
  - Execution Model
- Experiments
  - NAS Benchmark
  - Mini-benchmark
- Conclusion

# PIL - Parallel Intermediate Language

- An intermediate language for realizing any-to-any parallel programming language conversion




- PIL accepts a task graph as input
  - A PIL node is the specification of a parallel task which is either a single computation task or data parallel computation
  - Users can create reusable task graphs and use as parallel library function

# PIL Node Syntax

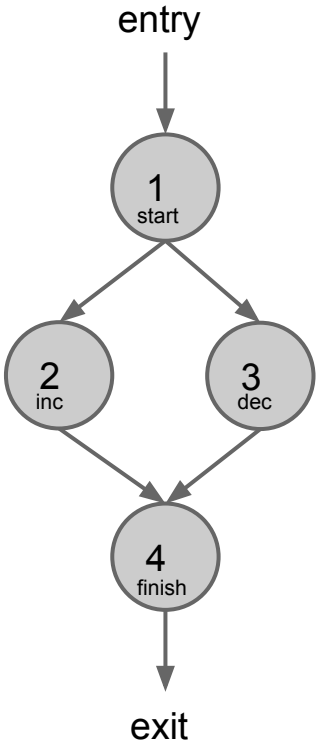
```
node(label, index, [lower:step:upper], target, [label1, label2, ..., labelN], func(arg1, arg2, ..., argN))
```

- **label**: the identifier for the node
- **index**: a variable used to identify execution instance in data parallel computation
- **[bounds]**: Iteration space used to determine the number of data parallel computation execution instances to spawn
- **target**: a variable that must be assigned during execution of the task to determine the successor of the task
- **[labels]**: possible successor nodes
- **func**: the serial function that performs the computation

# An Example

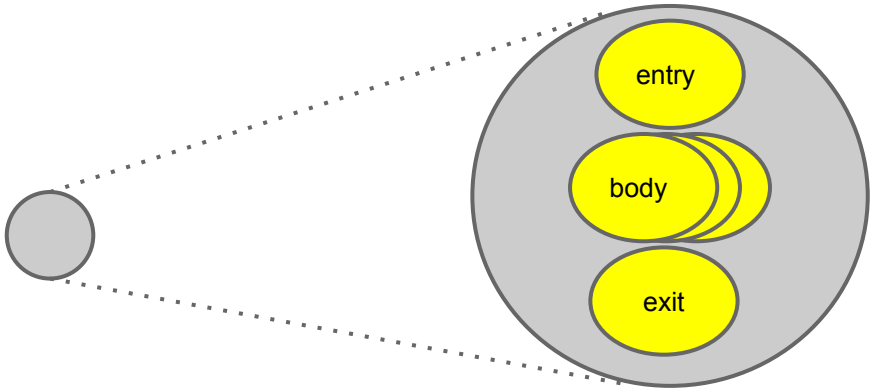
```
#define NUM_ELEM (8)
void start(int *target, gpp_t index_array, gpp_t data_array, int inc_or_dec, int* array){
        printf("start: inc_or_dec = %d.'\n", inc_or_dec);
        *target = (inc_or_dec) ? (2) : (3);
}
void inc(int *target, gpp_t index_array, gpp_t data_array, int i, int* array){
        printf("Increment array[%d]\n", i);
        array[i]++;
        *target = 4;
}
void dec(int *target, gpp_t index_array, gpp_t data_array, int i, int* array){
        printf("Decrement array[%d]\n", i);
        array[i]--;
        *target = 4;
}
void finish(int *target, gpp_t index_array, gpp_t data_array){
        printf("Computation is done\n");
        *target = 0;
}

node(1, NULL, [1:1:1], target, [2, 3], start(&target, index_array, data_array, inc_or_dec, &array))
node(2, i, [0:1:NUM_ELEM-1], target, [4], inc(&target, index_array, data_array, i, &array))
node(3, i, [0:1:NUM_ELEM-1], target, [4], dec(&target, index_array, data_array, i, &array)))
node(4, NULL, [1:1:1], target, [0], finish(&target, index_array, data_array))
```



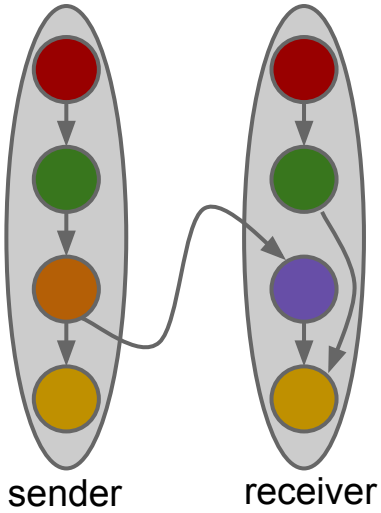
# PIL-to-SCALE Translation

- PIL compiler translates source-to-source from PIL program to SCALE
- For each PIL node, a swarm procedure is generated which contains a few codelets
  - Entry codelet creates all the body instances and the exit, and it also creates the dependencies
  - Body instances execute in parallel and satisfies one of the dependencies of the exit
  - Exit codelet is queued and wait for dependencies to be satisfied by all body instances



# Communication in PIL

- Communication takes place in special PIL nodes




```
swarm_procedure void pil_communicate(
  pil_comm_param *input) {

  swarm_codelet entry() {
    /* register recv codelet */
  }
  swarm_codelet setup() {
    if (input->recv) {
      /* create dependence to fire exit()
         after the recv completes */
    } else { /* fire send */ }
  }
  swarm_codelet send() {
    /* nw_call recv on remote machine */
  }
  swarm_codelet recv() {
    /* unpack data */
    /* satisfy dependence to fire exit */
  }
  swarm codelet exit() {
    /* call next swarm_procedure */
  }
}
```

# Outline

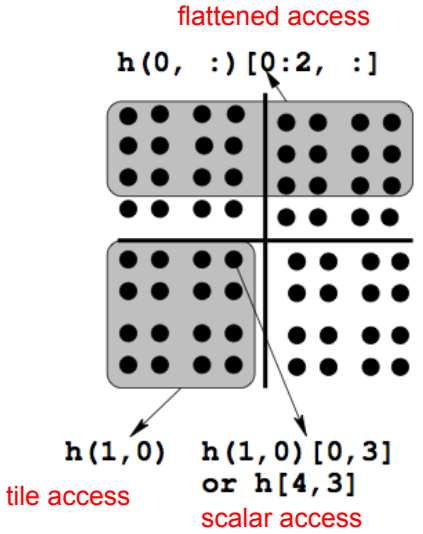
- Overview
- Parallel Intermediate Language (PIL)
- Hierarchically Tiled Array (HTA)
  - Semantics
  - Execution Model
- Experiments
  - NAS Benchmark
  - Mini-benchmark
- Conclusion

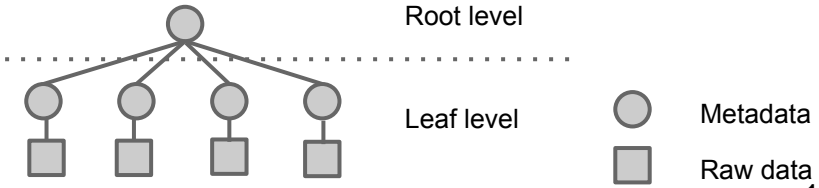
# HTA Programming Model

- Parallel computation is formulated as operations on tiled arrays
- In X-Stack projects, it is implemented as a library written in PIL
  - Facilitates application development through re-usable operations
  - Allows users to control locality and provide hints to the codelet runtime system
- Low-level details hidden from the user
  - Initiation of parallel tasks
  - Communication/synchronization between nodes

# HTA Notations

- h = new hta(2, ((2, 2), (4, 4)))
  - 2x2 at the root level
  - 4x4 at the leaf level
- Access operators
  - () is used to access tiles
  - [] is used to index scalar elements directly



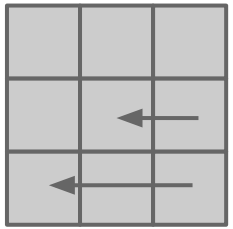




* In this project, we implement in C, so the chained access will be like:
access_scalar(access_tile(h, 1, 0), 0, 3)

# Cannon's Algorithm

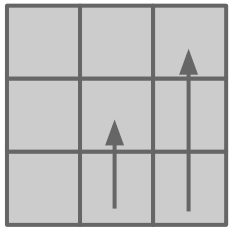
## Initial shift

A



Row i shift left by i

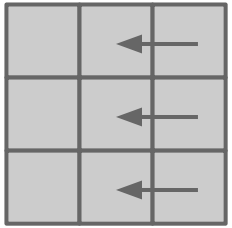
B



Column i shift up by i

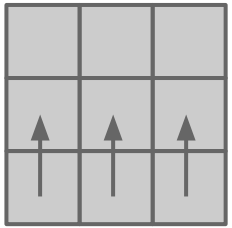
## 2nd for loop

A



Each row shift left by 1

B



Each column shift up by 1

# Cannon's Matrix-Matrix Multiplication in HTA

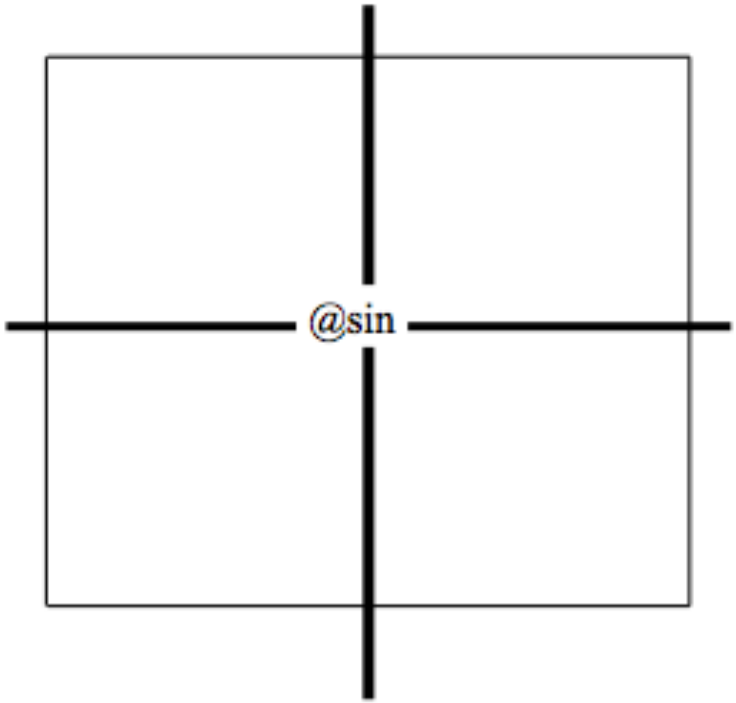
```
01 function C = cannon(A,B,C)
02   for i=2:m   // Initial shift                          sequential
03     A(i,:) = circshift(A(i,:), [0,      -(i-1)]);
04     B(:,i) = circshift(B(:,i), [-(i-1),      0]);      parallel
05   end
06   for k=1:m-1                                          sequential
07     C = C + A * B;  //(A*B) performs matmul on blocks
08                     // with the same indices
09     A = circshift(A, [0,  -1]);
10     B = circshift(B, [-1,  0]);                        parallel
11   end                                                  sequential
12 end
```

# HTA Operations

- Creation
  - Allocates space for metadata and raw data
- Access
  - Indexing in the hierarchy
  - Can either access a scalar element, a tile, or a flattened array
- Assignment
  - Modify values in tiles
  - Legal if the shape of RHS is conformable to the LHS

# Map

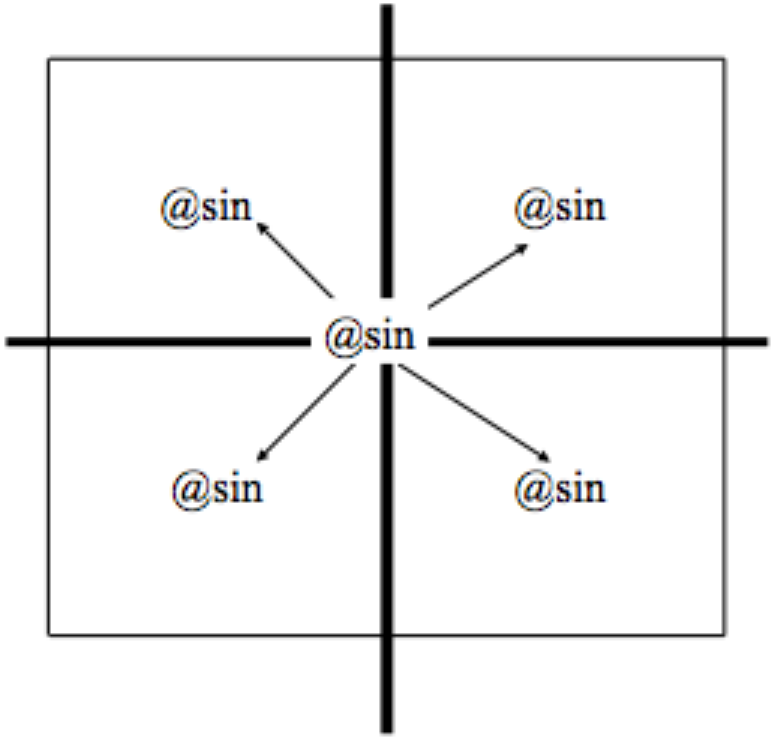
r = map (@sin, h)




Recursive

# Map

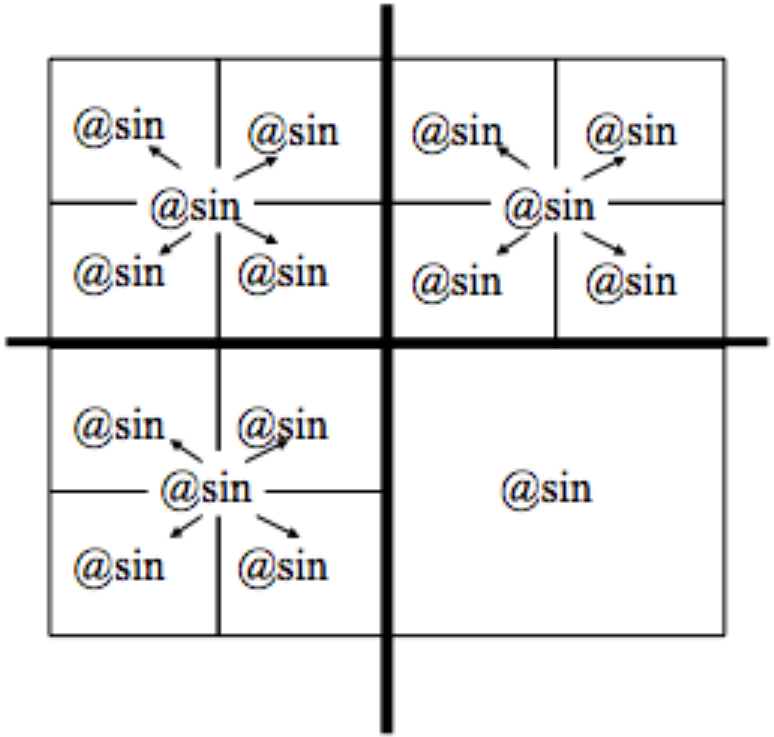
r = map (@sin, h)




Recursive

# Map

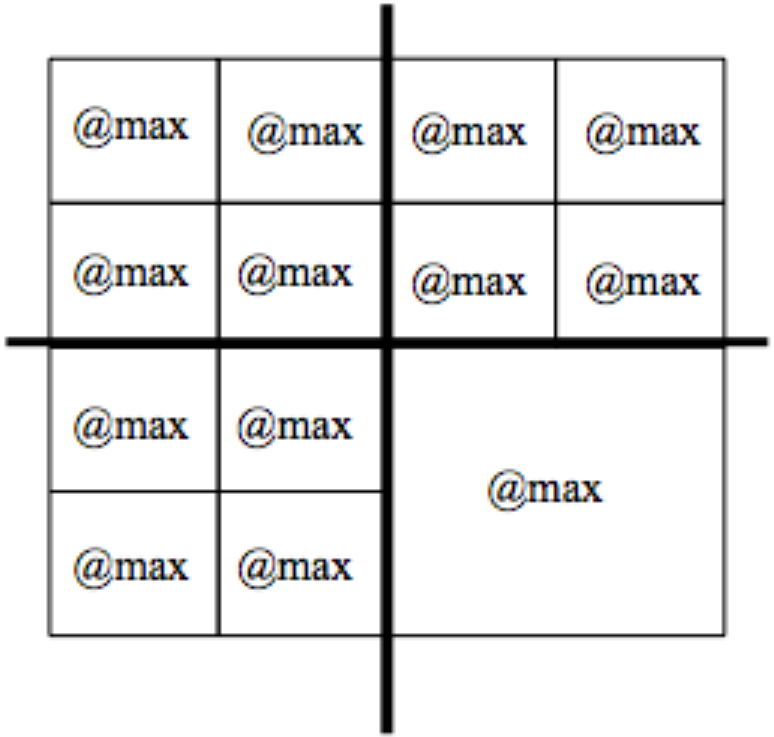
r = map (@sin, h)




Recursive

# Reduce

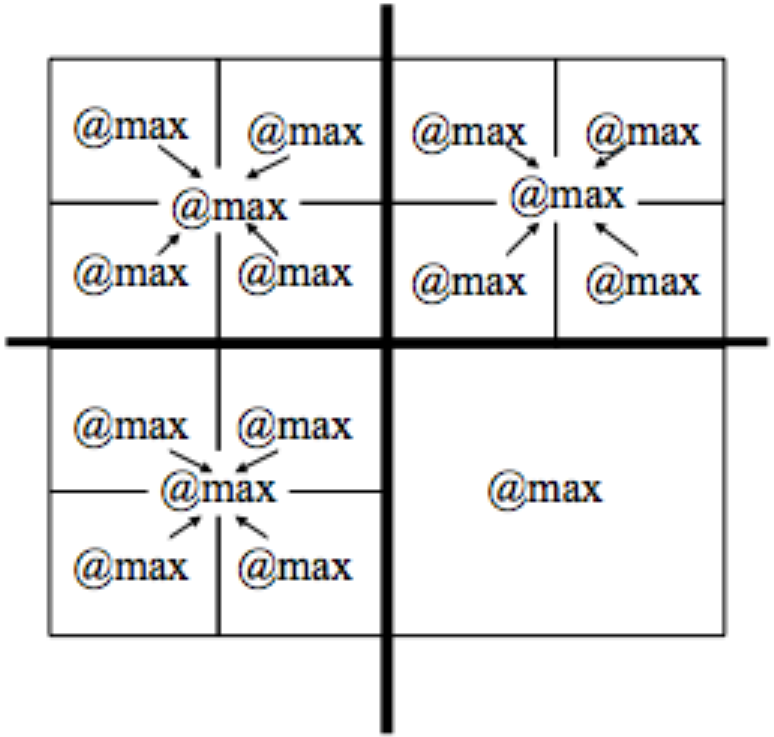
r = reduce (@max, h)




Recursive

# Reduce

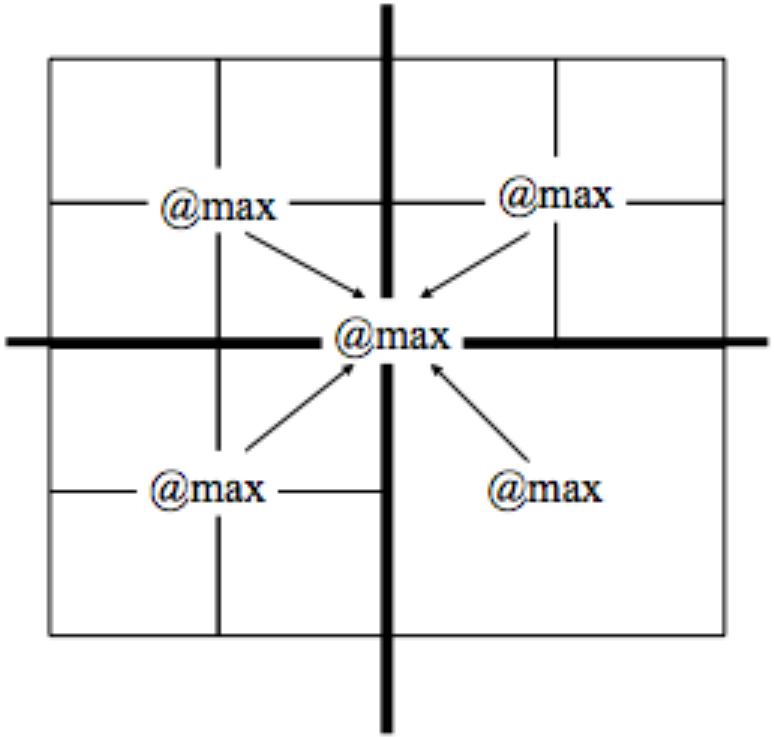
r = reduce (@max, h)




Recursive

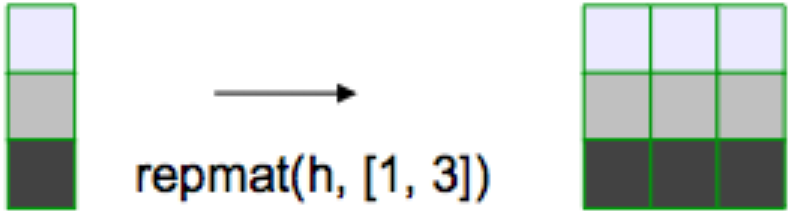
# Reduce

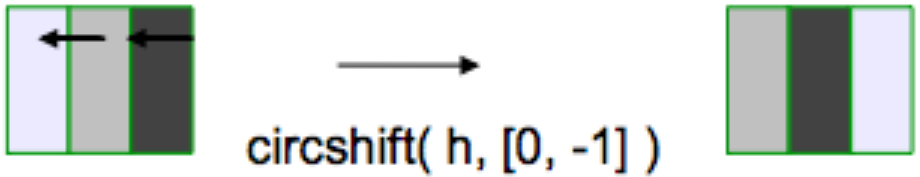
r = reduce (@max, h)

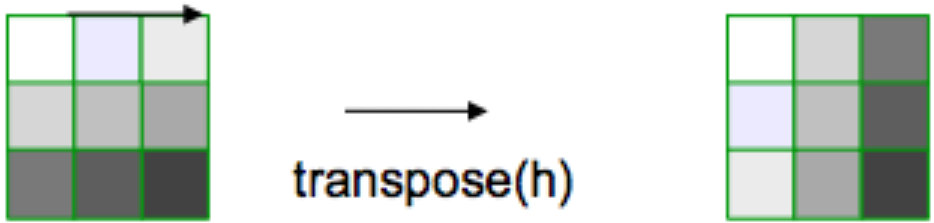



Recursive

# Higher level operations



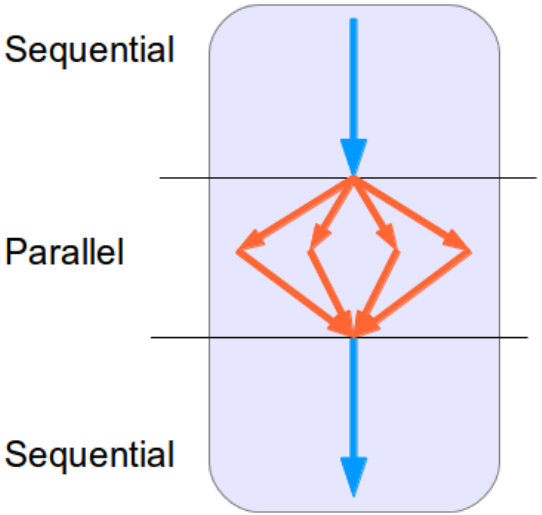



# Execution Model

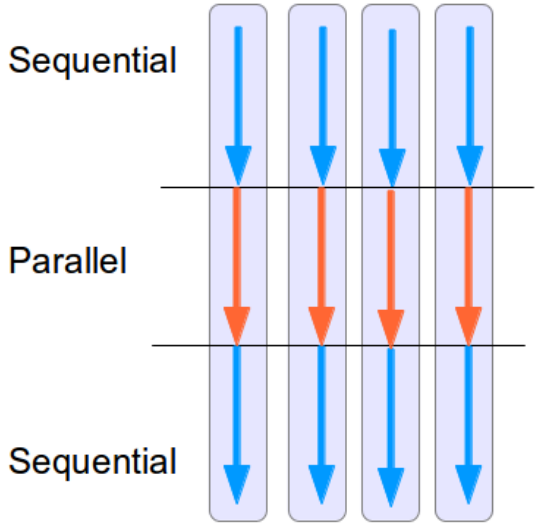
- Shared memory -> fork-join
  - Sequential part is executed by a master process
  - Whenever an HTA operation is encountered, the master spawns worker processes to perform tasks in parallel
  - Synchronization barrier is not always needed and can be relaxed
- Distributed memory -> SPMD
  - Sequential part is redundantly computed on all processes
  - HTA operations are executed in parallel by the processes involved

# Execution Models

Shared Memory Fork/Join




Distributed Memory SPMD

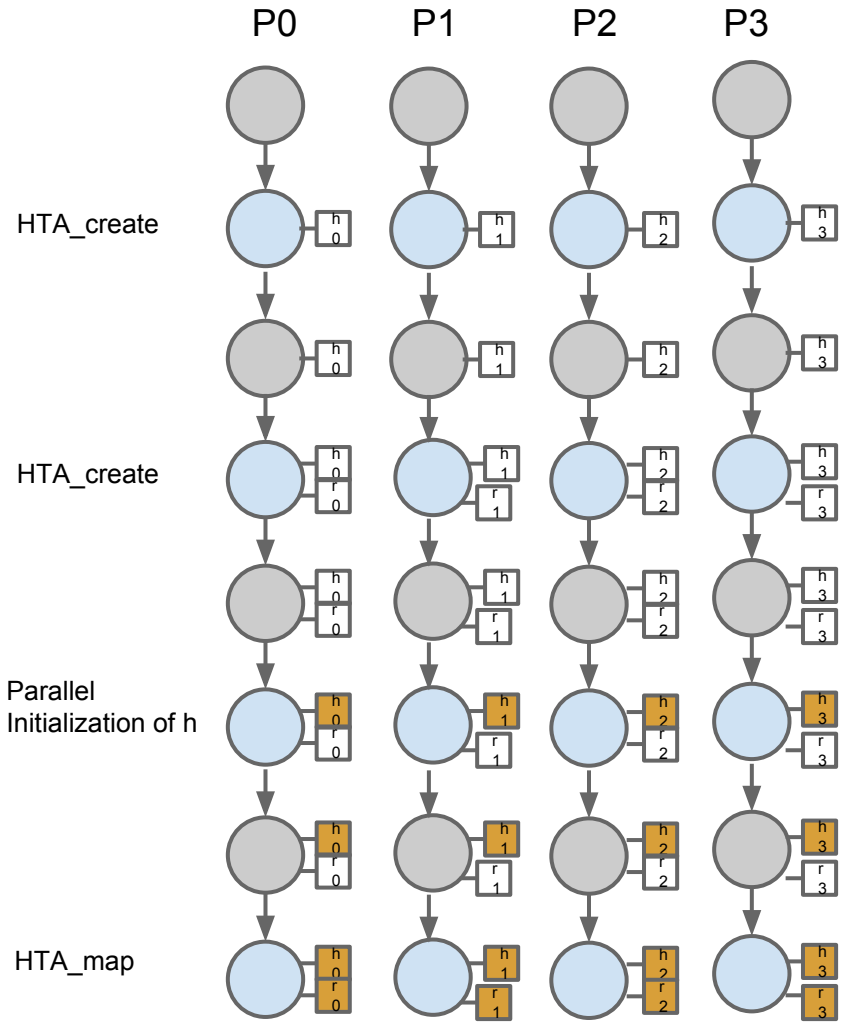


# Data Distribution

- Data placement
  - The tree structure is duplicated on all processes
  - The leaf raw data is distributed and each leaf tile has a single owner
- Communication
  - Implemented with point-to-point communication mechanism provided by PIL
    - pil_send and pil_recv
  - Possible to implement optimized collective operations in PIL
    - Broadcast, reduction, all-to-all exchange, ... etc.

# SPMD Execution

```
01  #define EXP (8)
02  void power(HTA *r, HTA* h) {
03      double* data1 = r->leaf.raw, data2 = h->leaf.raw;
04      int num_elem = Tuple_product(&r->flat_size);
05      for(int i = 0; i < num_elem; i++) {
06          double x = data2[i];
07          POW(x, EXP);
08          data1[i] = x;
09      }
10  }
11  int hta_main() {
12      HTA *h = HTA_create(...);
13      HTA *r = HTA_create(...);
14      HTA_map(init, h);
15      HTA_map(power, r, h);  // r = pow(h, exp)
16  }
```



# Communication Patterns

```
h(0, 1) = 0;
```

- Assign scalar value 0 to all elements in tile h(0, 1)
- Only the process that owns h(0, 1) will perform the assignments

```
t(5) = x(3);
```

- Overwrite tile t(5) with tile x(3)
- The process that owns x(3) ***sends*** the tile and t(5)'s owner ***receives*** and overwrites raw data

```
t(1:n) = x(3) + 1;
```

- Assign all tiles in t of with x(3) + 1
- First the owner of x(3) increments all elements
- It then ***broadcasts*** the resulting tile to owners of t(1) to t(n)
- When the broadcast is completed, the owners of t(1) to t(n) overwrites the tile with newly received tile

```
int w = x(3)[2];
```

- Owner of x(3) reads x(3)[2] and ***broadcasts*** the scalar value to all others
- All others assign the scalar value to the variable w

# HTA Operations and Their Corresponding Communication Patterns

| HTA Operation | Communication Pattern |
|---|---|
| Assignment | Send/receive |
| Access | Broadcast |
| Reduce | Reduce |
| Scan | Send/receive |
| Circular shift | Send/receive |
| Repmat | Broadcast |
| Transpose | All-to-all |

All of the patterns can be implemented with point-to-point send/receive

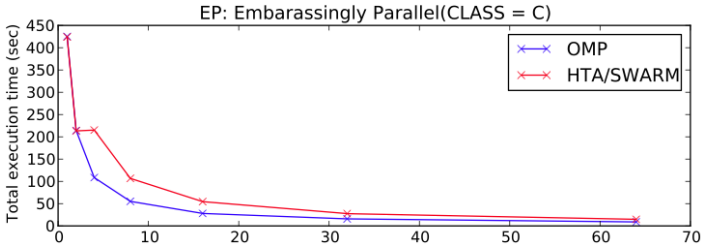
# Outline

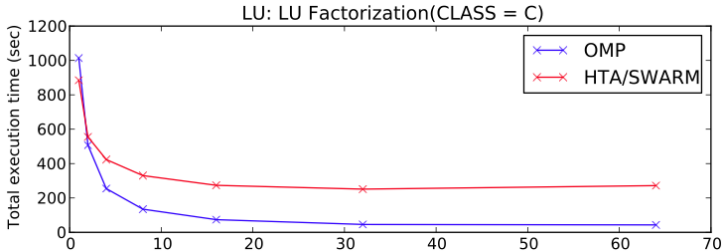
- Overview
- Parallel Intermediate Language (PIL)
- Hierarchically Tiled Array (HTA)
  - Semantics
  - Execution Model
- Experiments
  - NAS Benchmark
  - Mini-benchmark
- Conclusion

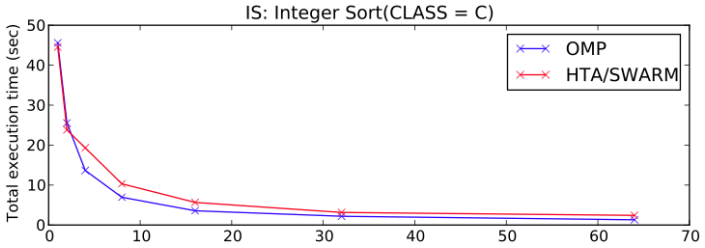
# NAS Parallel Benchmarks Implementation with HTA

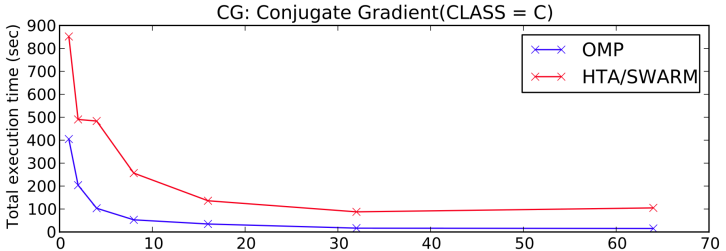
- We have implemented six of the NAS Parallel Benchmarks with HTA running on ETI SWARM runtime (shared memory)
  - EP, IS, CG, LU, MG, FT
- Experiments conducted on a multi-core shared memory machine using up to 64 threads
  - 4 Intel Xeon E7-4860 CPU, each with 10 cores (80 hardware threads)
- Preliminary performance results obtained
  - Execution time compared with highly tuned OpenMP implementation
  - Overhead analysis on-going

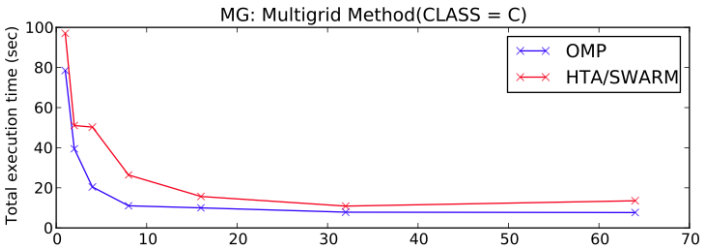
# Performance Results

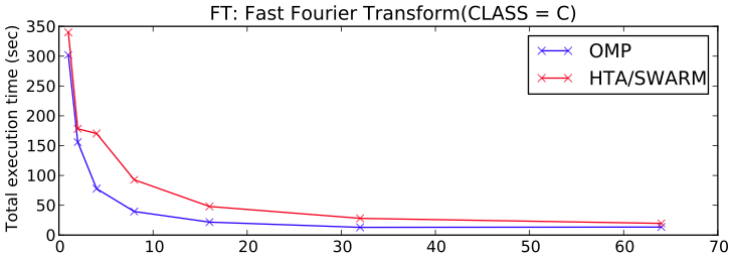


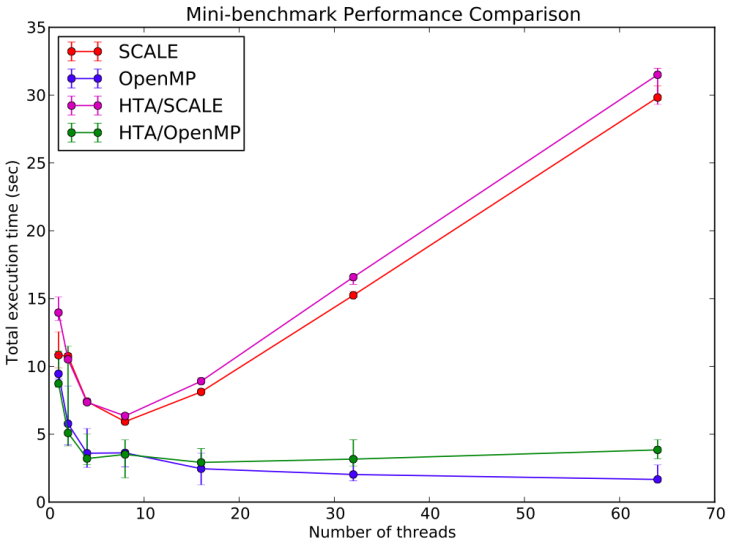




Slow-down due to:
1. Algorithm differences (programming model dependent)
2. Unidentified overhead in different levels of the software stack

# Mini-benchmark

- We created mini-benchmark programs that are written directly in SCALE and OpenMP and compare with the HTA versions
- The benchmark performs a large number of parallel operation invocations in a for loop
  - Memory bound
  - Fix-sized data set
- Pure-SCALE version shows significant overhead
  - SWARM runtime startup/finishing overhead
  - Serialization in spawning new codelets




Invoking a parallel operation for 10000 times

# Overhead in Invoking Parallel Operation

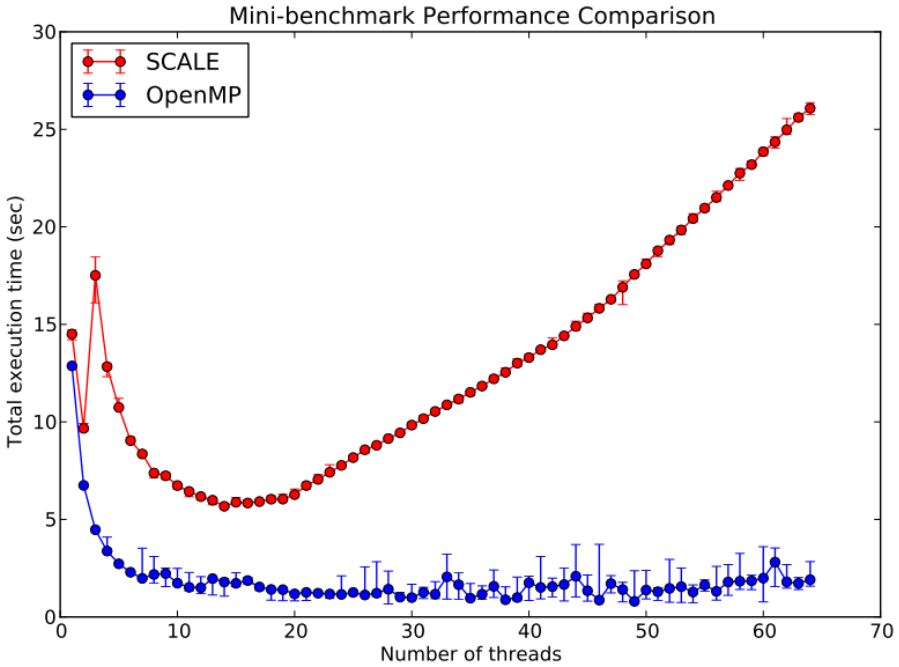
```
swarm_codelet entry() {
    int np = h1->size;
    dep.requires(np + 1) =>exit;
    for(int i = 0; i < np; i++)
        do => pwmul(h1->tiles[i], h2->tiles[i], h3->tiles[i]);
    swarm_Dep_satisfy(&dep, 1U);
}
```

Sequentially spawning workers with argument boxing (memory copying)

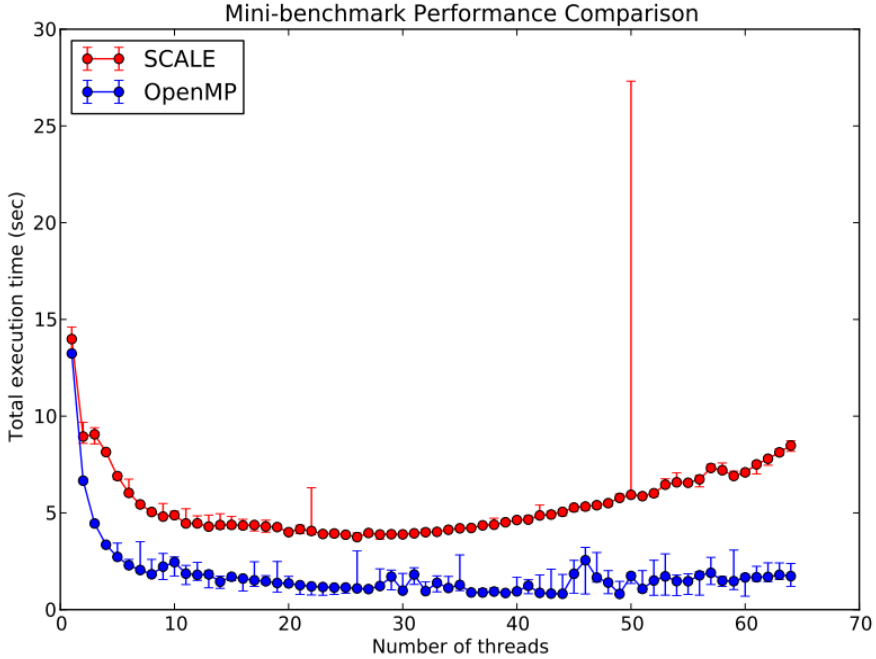
```
swarm_codelet entry() {
    int np = _num = h1->size;
    _ctr = 0;
    _h1 = h1; _h2 = h2; _h3 = h3;
    dep.requires(np + 1) => exit;
    swarm_Locale_scheduleToLeaves(swarm_getRootLocale(NULL), np, swarm_cargs(pwmul),
        NULL, NULL, NULL, swarm_Scheduler_ORDER_FIFO);
    swarm_Dep_satisfy(&dep, 1U);
}
```

Spawn a task for each worker thread w/o argument boxing

# Performance Improvement of the Mini-benchmark




Original




Using swarm_Locale_scheduleToLeaves

# Conclusion

- Current Status
  - Further analysis of the overhead in the HTA library and PIL generated code
- Future Work
  - Implement SPMD execution
  - Optimize PIL collective communication API
  - Extend HTA with CnC

# Backup Slides

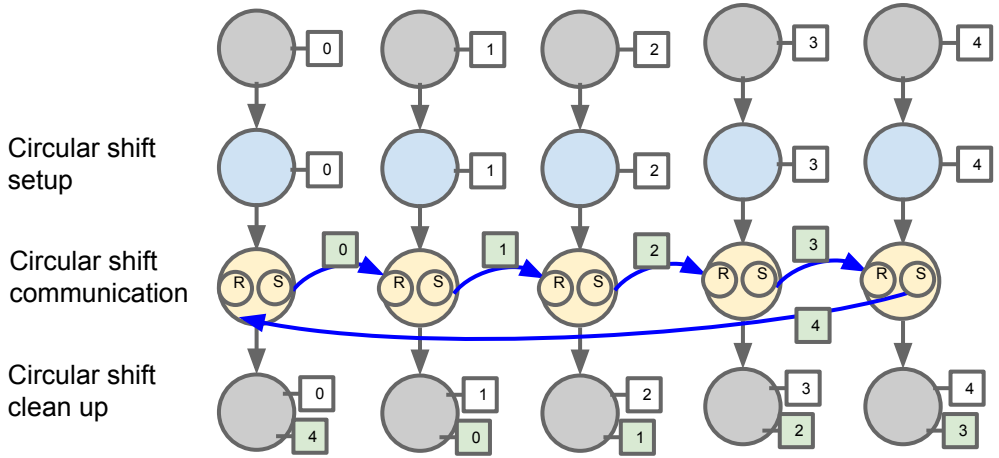
# Circular Shift

- A communication operation
- It shifts data tiles along the selected dimension by the specified distance

```
HTA_circular_shift(h(:, 1), 0, 1);
```

dimension

distance



# SPMD Execution

- On the distributed memory machine, HTA programs execute in SPMD fashion
  - Serial part is redundantly computed on all processes
  - HTA operations are executed in parallel by the processes involved
    - Each process can determine if it is involved
    - Owner computes: computation happens at the owner of the tile being modified

# Creation

```
HTA *h = HTA_create(...);
```

- Each process allocates space locally and no communication is required
- Tree structure (metadata) is cloned on each process
  - A process knows about where to look for the data if not locally owned
- Only the owner allocates space for leaf level raw data tile