Programming for Parallelism and Locality with Hierarchically Tiled Arrays

Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi , Basilio B. Fraguela , María J. Garzarán, David Padua and Christoph von Praun

Insight

- Advantages of tiling:
  - Increased locality
  - Improves parallelism

- But, most programming languages lack language constructs for tiling

Application Areas

- Multi-level tiling:
  - Cache-oblivious / recursive algorithms
    - Numerical/linear algebra
    - Sorting
    - Scanning
  - Stencil codes
    - ODEs and PDEs
- Single-level tiling
  - Wide range of applications

Impact of Work

- Good number of citations ~150
- But, idea didn't really catch on
  - Not much work on multiple level tiling since 2010
- Work has been much more focused on single-tiling:
  - Automatic tiling
  - Optimizing & Dynamic Tiling
  - Tiling for GPUs & Distributed Systems
  - Overlapping tiling
  - Tiling across the memory hierarchy
  - Edge cases

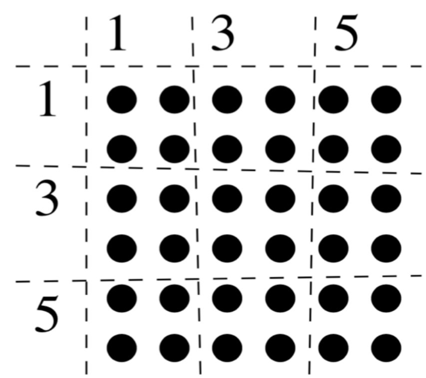
What is an HTA?

- An array partitioned into tiles.
  - Tiles are either conventional arrays or lower level HTAs
  - Can have any number of dimensions
- Tiles can be distributed across processors or stored locally

Local HTA

hta(M, { [1,3,5], [1,3,5] })

(Ref - Construction of an HTA [1, p.49])



(1)

matrix M

Distributed HTA

hta(M, { [1,3,5], [1,3,5] }, [2,2] , “cyclic”)

Processor Grid

P1	**P2**
P3	**P4**

P1	**P2**	**P1**
P3	**P4**	**P3**
P1	**P2**	**P1**

(1) Construction of an HTA [1, p. 49]

Distributed Programming Model

- Follows SPMD model
  - Communication: 2-sided message passing (MPI)
  - Computation: each processor applies on locally owned tiles

$$C = \begin{array}{|c|c|c|} \hline \mathbf{P1} & \mathbf{P2} & \mathbf{P1} \\ \hline \mathbf{P3} & \mathbf{P4} & \mathbf{P3} \\ \hline \mathbf{P1} & \mathbf{P2} & \mathbf{P1} \\ \hline \end{array} * 2$$

Distribution Types

Processor Grid (1x2)

P1	P2

HTA (3x8)

Cyclic

P1	P2	P1	P2	P1	P2	P1	P2
P1	P2	P1	P2	P1	P2	P1	P2
P1	P2	P1	P2	P1	P2	P1	P2

Block

P1	P1	P1	P1	P2	P2	P2	P2
P1	P1	P1	P1	P2	P2	P2	P2
P1	P1	P1	P1	P2	P2	P2	P2

Block-Cyclic

P1	P1	P2	P2	P1	P1	P2	P2
P1	P1	P2	P2	P1	P1	P2	P2
P1	P1	P2	P2	P1	P1	P2	P2

Distribution Types (Continued)

Processor Grid

P1	P2
P3	P4

Cyclic

P1	P2	P1	P2
P3	P4	P3	P4
P1	P2	P1	P2
P3	P4	P3	P4

HTA (4x4)

Block

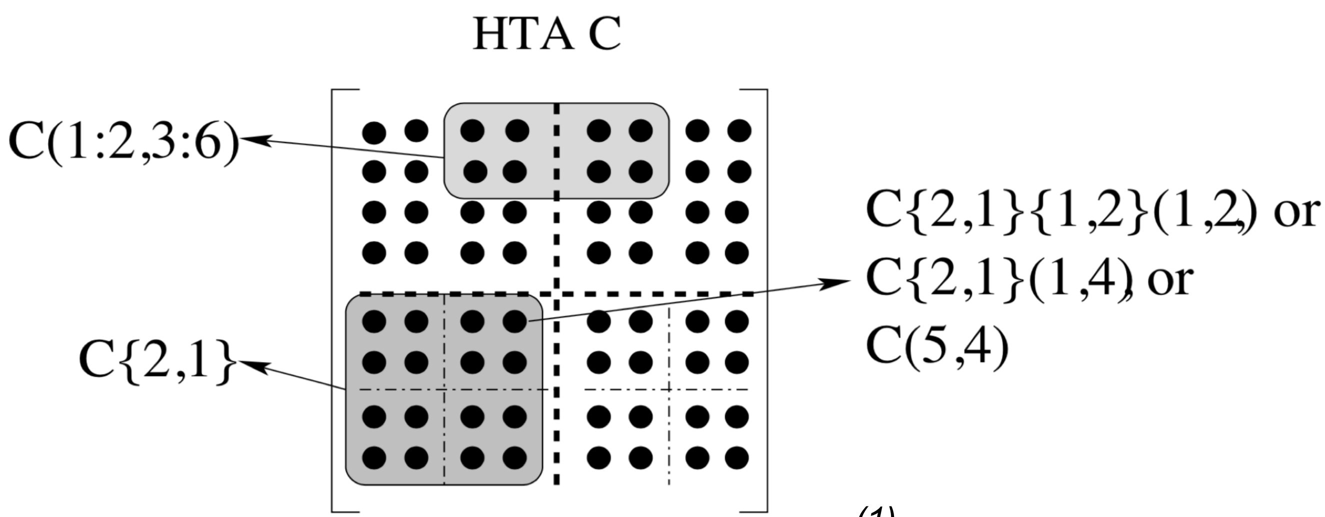
P1	P1	P2	P2
P1	P1	P2	P2
P3	P3	P4	P4
P3	P3	P4	P4

Machine Mapping

- Not addressed in this paper

- HTAs have a machine mapping that specifies:
  - where the HTA is allocated in a distributed system
    - *Distribution* class: specifies the home location of the scalar data for each of the tiles of an HTA
  - the memory layout of the scalar data array underlying the HTA
    - *MemoryMapping* class: specifies the layout (row-major across tiles, row-major per tile etc.), size and stride of the flat array data underlying the HTA

Accessing HTAs



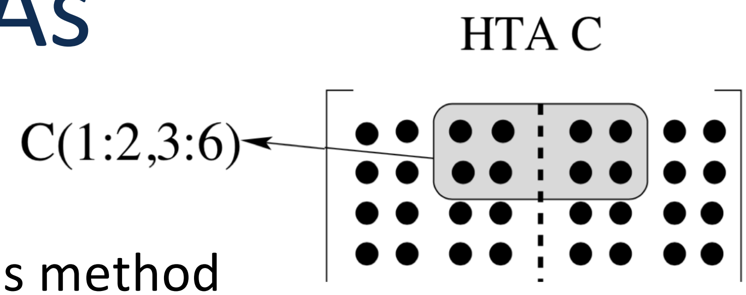
(1)

- 3 methods
  - Hierarchically – addressing using each level of tile
  - Flat – addresses the elements of an HTA by their absolute indices, as a normal array
  - A combination of the two – applying flattening at any level of the hierarchy

- Takeaway – provides a simple method for selecting elements of HTAs that bridges the gap between HTA and non-HTA applications

(1) Accessing the contents of an HTA [1, p.49])

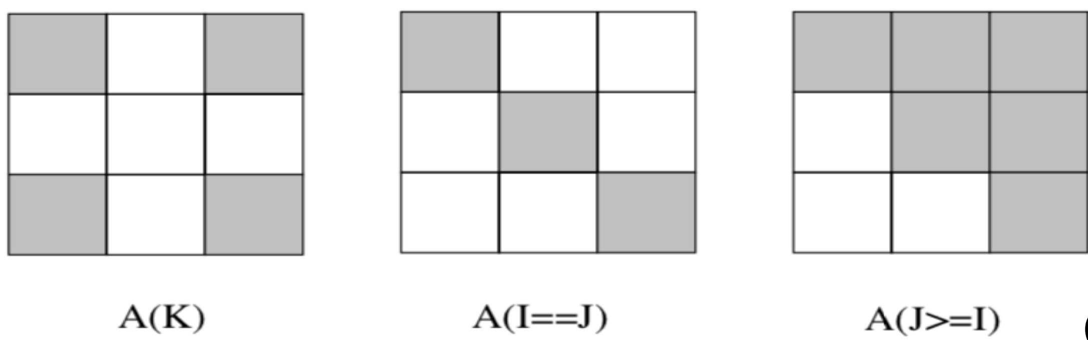
Accessing Regions of HTAs



(1)

- 3 methods
  - Can use begin:step:end indexing for any access method
  - Can use : notation to refer to the whole range of values for an index
  - Can use Boolean arrays for logical indexing

$$K = \begin{bmatrix} true & false & true \\ false & false & false \\ true & false & true \end{bmatrix} \quad I = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} \quad J = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$



(2)

- Takeaway: access methods cover all use cases

(1) Accessing the contents of an HTA [1, p. 49]
(2) Logical indexing in HTA [1, p. 50]

Rules for Binary Operations & Assignment

- HTA ⊕ Scalar
  - each scalar of the HTA is operated with the scalar

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} * 2 = \begin{bmatrix}2 & 4\\ 6 & 8\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = 2 \rightarrow \begin{bmatrix}2 & 2\\ 2 & 2\end{bmatrix}$$

- HTA ⊕ Matrix
  - each lowest level tile of the HTA is operated with the matrix

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} * \begin{bmatrix}2\\ 3\end{bmatrix} = \begin{bmatrix}8\\ 18\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = \begin{bmatrix}5\\ 6\end{bmatrix} \rightarrow \begin{bmatrix}5 & 5\\ 6 & 6\end{bmatrix}$$

- HTA ⊕ HTA
  - Same topology -> corresponding tiles are operated on
    - Produces an HTA with the same topology
  - Otherwise, the operation acts like HTA ⊕ Matrix

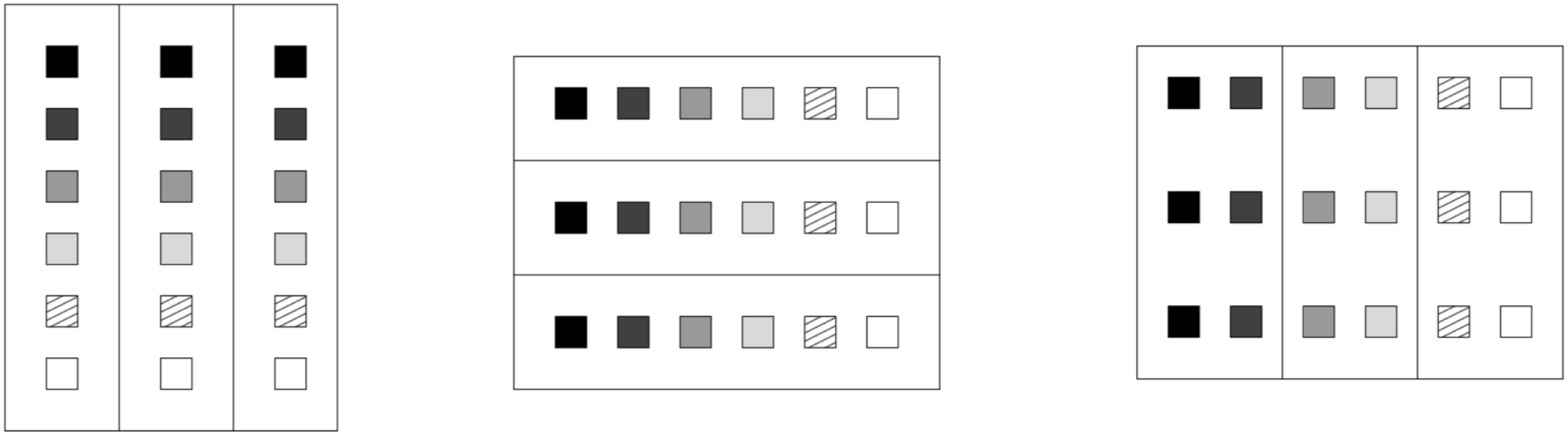
$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} * \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = \begin{bmatrix}7 & 10\\ 15 & 22\end{bmatrix}$$

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = \begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix} \rightarrow \begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}$$

HTA Methods

- Overloaded array operations that, when applied to HTAs, operate on the tile level (instead of individual array elements)
  - Assignment
  - Binary operations
  - Indexing
  - Other frequently-used array functions
    - transpose
    - permute
    - circshift
    - repmat
  - Methods that apply only to HTAs
    - reduceHTA - a generalized reduction method that operates on HTA tiles
    - parHTA - applies in parallel the same function to each tile of an HTA

Example – transpose and permute



h = hta(1,3)

(a)

h = transpose(h) *or* h'

h = permute(h,[2,1])

(b)

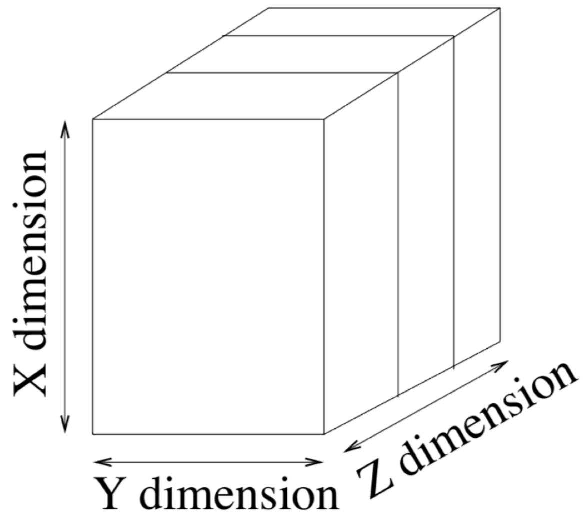
h = dpermute{h,[2,1]}

(c) (1)

- dpermute – the data permuted, but the shape of the containing HTA remains the same (# of tiles in each dimension)

(1) Transpose and dpermute [1, p. 50]

Example – 3D matrix & dpermute



(a)

```
X=hta(A,{[1],[1],[partition-z]},
            [1,1,nprocs])

X=fft(X,[],1)
X=fft(X,[],2)
X=dpermute(X,[3,1,2])
X=fft(X,[],1)
```

(b)

(1)

- Implicit parallel communication from HTA assignment
- fft is applied in parallel on local tiles
- 1st and 2nd dimensions are local, so use dpermute to make the 3rd dimension local to the processor for fft can be applied

(1) Data Permutation in FFT.(a)-Pictorial view.(b)-code [1, p. 50]

Example – circshift

```
function C = cannon(A,B,C)
  for i=2:m
    A{i,:} = circshift(A{i,:}, [0, -(i-1)]);
    B{:,i} = circshift(B{:,i}, [-(i-1), 0]);
  end
  for k=1:m-1
    C = C + A * B;
    A = circshift(A, [0, -1]);
    B = circshift(B, [-1, 0]);
  end
```

- Cannon's Algorithm – MMM
- Shifts tiles in row i of A to the left i-1 times
- Shifts tiles in column i of B up i-1 times
- Matrix multiplication is done locally -> C is left distributed

Advantages of Tiled Cannon's and parHTA Example

- Aggregates data into a tile for communication
- Increased locality from matrix-matrix multiplication (instead of element by element multiplication)
- Can further increase cache locality by using HTAs with more levels, and applying matrix multiplication recursively
  - C = parHTA (@matmul, A, B, C)

```
function C = matmul (A, B, C)
  if (level(A) == 0)
      C = C + A * B;
  else
    for i=1:size(A,1)
      for k=1:size(A,2)
        for j=1:size(B,2)
          C{i, j} = matmul(A{i,k}, B{k,j}, C{i,j});
```

Example - repmat

- Normal Summa algorithm

  For (k = 1 … M)

  For (i = 1 … M)

  For (j = 1 … M)

  C (i, j) = C (i , j) + a(i, k) * b (k, j)

- Tiled version:

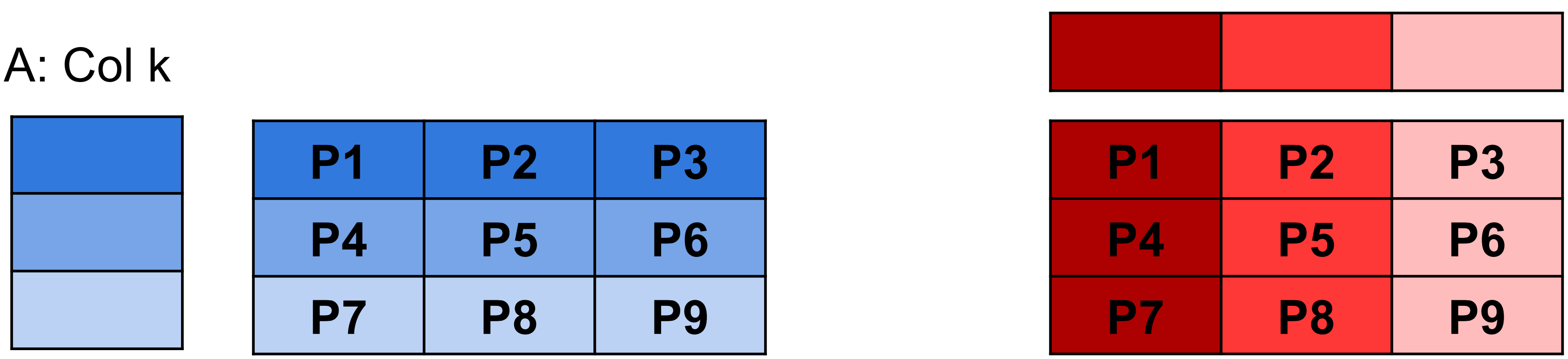
```
function C = summa (A, B, C)
  for k=1:m
     T1 = repmat(A{:, k}, 1, m);
     T2 = repmat(B{k, :}, m, 1);
     C = C + T1 * T2;
  end
```

Example – repmat (continued)

- Tiled version:

```
function C = summa (A, B, C)
  for k=1:m
     T1 = repmat(A{:, k}, 1, m);
     T2 = repmat(B{k, :}, m, 1);
     C = C + T1 * T2;
  end
```

B: Row k



- Multiplication is then done locally

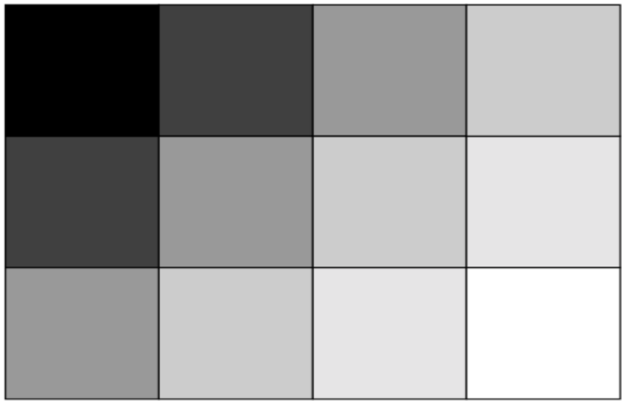
Example – logical indexing

- Wavefront computation - Normal code:

```
for i=2:m-1
  for j=2:n-1
    A(i,j)= A(i-1,j) + A(i,j-1);
```

- Can parallelize by computing in parallel the element of each diagonal of the matrix:

(1)



(1)

(1) 2-D wavefront computation [1, p. 52]

Example – logical indexing

- Wavefront computation

$$x = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix} \quad y = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$ (1)

```
for k=2:m+n
   for i=2:dimx-1
      for j=2:dimy-1
         A{x+y == k}(i, j) = A{x+y == k}(i-1, j) +
                             A{x+y == k}(i, j-1);
      end
   end
   A{x+y == k+1 & x>1}(1, :) = A{x+y == k & x<m}(dimx-1,:);
   A{x+y == k+1 & y>1}(:, 1) = A{x+y == k & y<n}(:, dimy-1);
end
```

(1)

- Select tiles on the diagonal using “x+y == k”
- Implicit communication

(1) 2-D wavefront computation [1, p. 52]

parHTA and reduceHTA example

```
A = hta(MX, {partition_A}, [m n]);
V = hta(VX, {partition_B}, [m n]);
B = repmat(V, m, 1)
B = parHTA(@tranpose, B)
C = reduceHTA('sum', A * B, 2, true);
```

A

1	2	3
4	5	6
7	8	9

V

a	b	c

B

a	b	c
a	b	c
a	b	c

B = B^T

a	a	a
b	b	b
c	c	c

A * B

1*a	2*a	3*a
4*b	5*b	6*b
7*c	8*c	9*c

Sum(A * B)

1*a+2*a+3*a
4*b+5*b+6*b
7*c+8*c+9*c

all-to-all

d	d	d
e	e	e
f	f	f

Implementations

- Implemented as a class for:
  - MATLAB
  - C++
- Uses object oriented capabilities of these languages

Goals

- To identify the set of operations on tiles needed to develop parallel programs
  - Examples seem fairly comprehensive
- To implement these operations so that they are:
  - Readable / easy to use
  - Efficient

Results – Performance of MATLAB Implementation

- NAS Performance Benchmarks

Nprocs	EP (CLASS C)		FT (CLASS B)		CG (CLASS C)		MG (CLASS B)		LU (CLASS B)	
	Fortran+ MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA	Fortran + MPI	Matlab + HTA
1	901.6	3556.9	136.8	657.4	3606.9	3812.0	26.9	828.0	15.7	245.1
4	273.1	888.8	109.1	274.0	362.0	1750.9	17.0	273.8	6.3	60.5
8	136.3	447.0	65.5	159.3	123.4	823.6	9.6	151.3	2.9	29.9
16	68.6	224.8	37.2	87.2	89.5	375.2	4.8	87.0	1.2	16.0
32	34.7	112.0	20.7	42.9	48.4	250.3	3.3	54.9	1.1	9.8
64	17.1	56.7	10.4	24.0	44.5	148.0	1.6	50.4	1.3	7.1
128	8.5	29.1	5.9	15.6	30.8	123.0	1.4	38.5	1.6	N/A

(1)

- Takeaways:
  - For each configuration, MATLAB+HTA is slower
  - EP, FT, CG: 128 parallel MATLAB performs 30.9, 8.8, and 29.3X faster than sequential Fortran
  - MG, LU, BT(not shown): slow sequential, 128 parallel MATLAB performs close to the same or worse than sequential Fortran

(1) Execution times in seconds for some of the applications in the NAS benchmarks [1, p. 53]

C++ Implementation

- Largely the same
- Differences
  - Focused on performance
  - Array operators not overloaded (fixed later)

C++ Example

```
typedef Tuple<2> T;

HTA<double, 2, 1> A, B,C;
A =HTA<double, 2, 1>::alloc((T(xtiles, ytiles),
                T(tile_sz_x,tile_sz_y)),ROW);
B =HTA<double, 2, 1>::alloc((T(xtiles, ytiles),
                T(tile_sz_x,tile_sz_y)),ROW);
C =HTA<double, 2, 1>::alloc((T(xtiles, ytiles),
                T(tile_sz_x,tile_sz_y)),ROW);

template <int LEVEL> void mult(
                  HTA<double, 2, LEVEL> A,
                  HTA<double, 2, LEVEL> B,
                  HTA<double, 2, LEVEL> C) {
  int M = A.shape()[0].size();
  int N = B.shape()[0].size();
  int Q = B.shape()[1].size();
  for (int i = 0; i< M; i++) {
     for (int k = 0; k < N; k++) {
        for (int j = 0; j< Q; j++) {
           mult (A[T(i,k)], B[T(k,j)], C[T(i,j)]);
  }}}
}

void mult(double& A,double& B,double& C)
{
  C += A * B;
}
```

(1)

- Takeaway: C++ implementation, with type declarations and without array operators, is much less readable

(1) Recursive matrix multiplication in C++ using HTAs [1, p. 54]

Results - Performance of C++ Implementation

Matrix Size	Naïve 3 loops	Tiled 6 loops	HTA naïve	HTA+ATLAS	ATLAS	Intel MKL(1)
504	161	657	675	2069	2387	3624
1008	150	649	679	2192	2384	3762
2016	133	632	675	2216	2492	3821
3024	135	644	668	2245	2509	3716
4032	36	588	613	2217	2519	3752

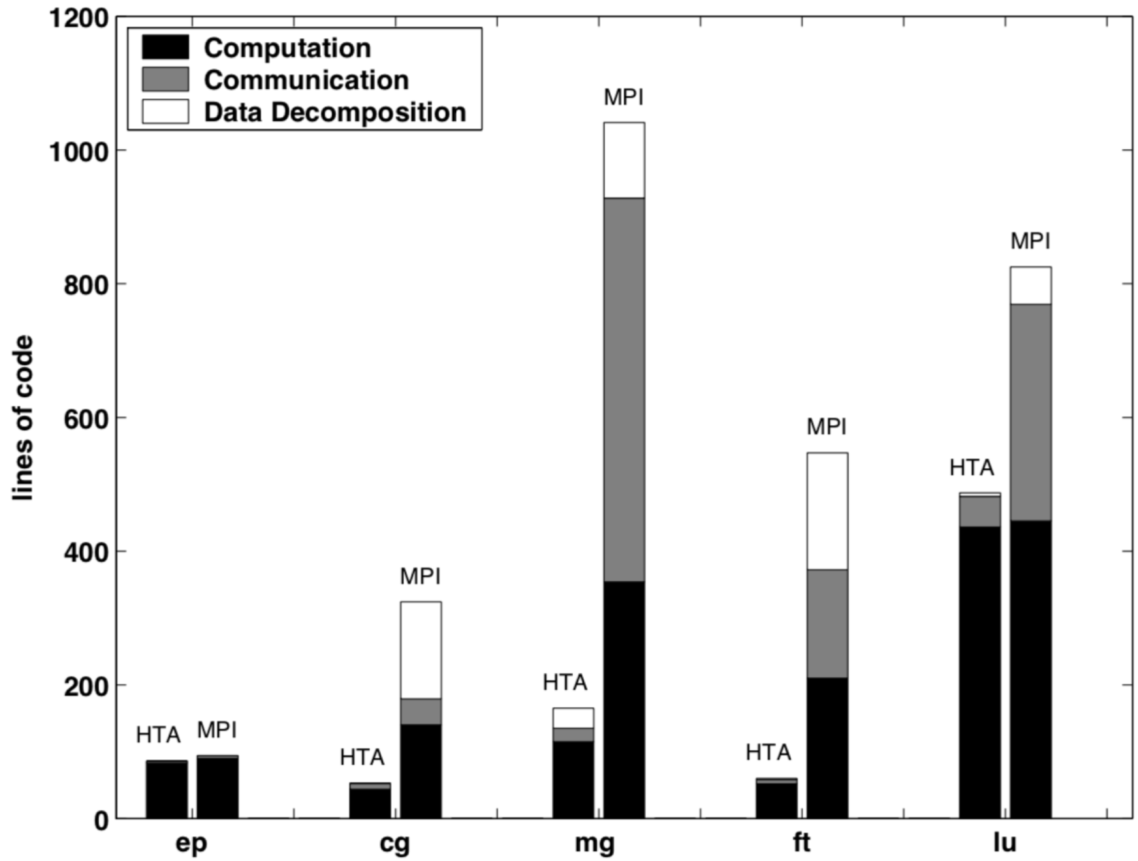
Performance in MFLOPS for different versions of matrix-matrix multiplication. (1) MKL uses SSE2 vector extension.

(1)

- One level of HTA introduces an overhead of between 8 – 13.5%
- Naïve use produces little benefit
- INTEL MKL is the only version that uses INTEL SSE2 vector extensions, all others use scalar code

(1) Performance in MFLOPS for different versions of matrix-matrix multiplication [1, p. 55[]

Results – Program Complexity



(1)

- Reduces complexity (measured in terms of lines of code)
  - Communication: assignments vs. message passing
  - Data decomposition: single HTA constructor vs. computing assignment

(1) Linecount of key sections of HTA and MPI programs [1, p. 55]

Advantages of HTAs

- Library approach
  - Easy to use, can make small modifications to existing code
- Global data view & single threaded view
  - Reduced program complexity
- Hierarchical tiling
  - Recursion
  - Can use multiple levels of parallelism

Disadvantages of HTAs

- Programming burden:
  - Creating distributed, tiled algorithms is challenging
  - Insufficient documentation
- Difficulty optimizing code:
  - Interprocess communication is hidden from the user
  - SPMD Model limits ability to use irregular parallelism
  - Library overhead

Future Work

- C++ implementation
  - Operator overloading
  - Optimizations
  - Additional functionality
    - Asynchronous communication
    - Map-reduce framework
    - Overlapped tiling
    - Data layering

References

- [1] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun, "Programming for parallelism and locality with hierarchically tiled arrays", in *PPoPP '06*, pages 48–57, New York, NY, USA, 2006. ACM.