Parameterized Diamond Tiling for Stencil Computations with Chapel Parallel Iterators

Ian Bertolacci – Colorado State University
Catherine Olschanowsky – Colorado State University
Ben Harshbarger – Cray Inc.
Bradford L. Chamerlain – Cray Inc.
David G. Wonnacott – Haverford College
Michelle Mills Strout – Colorado State University

This project is supported by
Department of Energy Early Career Grant DE-SC0003956
and
National Science Foundation Grant CCF-1422725

HPC is a Software Engineering Nightmare

- Performance code is a nightmare to develop.
- Hardware is evolving rapidly.

Can your software be easily adapted to perform on the next big system?

Our work demonstrates that it is possible to develop code that is performant and adaptable.

  – Parameterized Diamond Tiling
  – Chapel iterators as tiling schedule abstraction

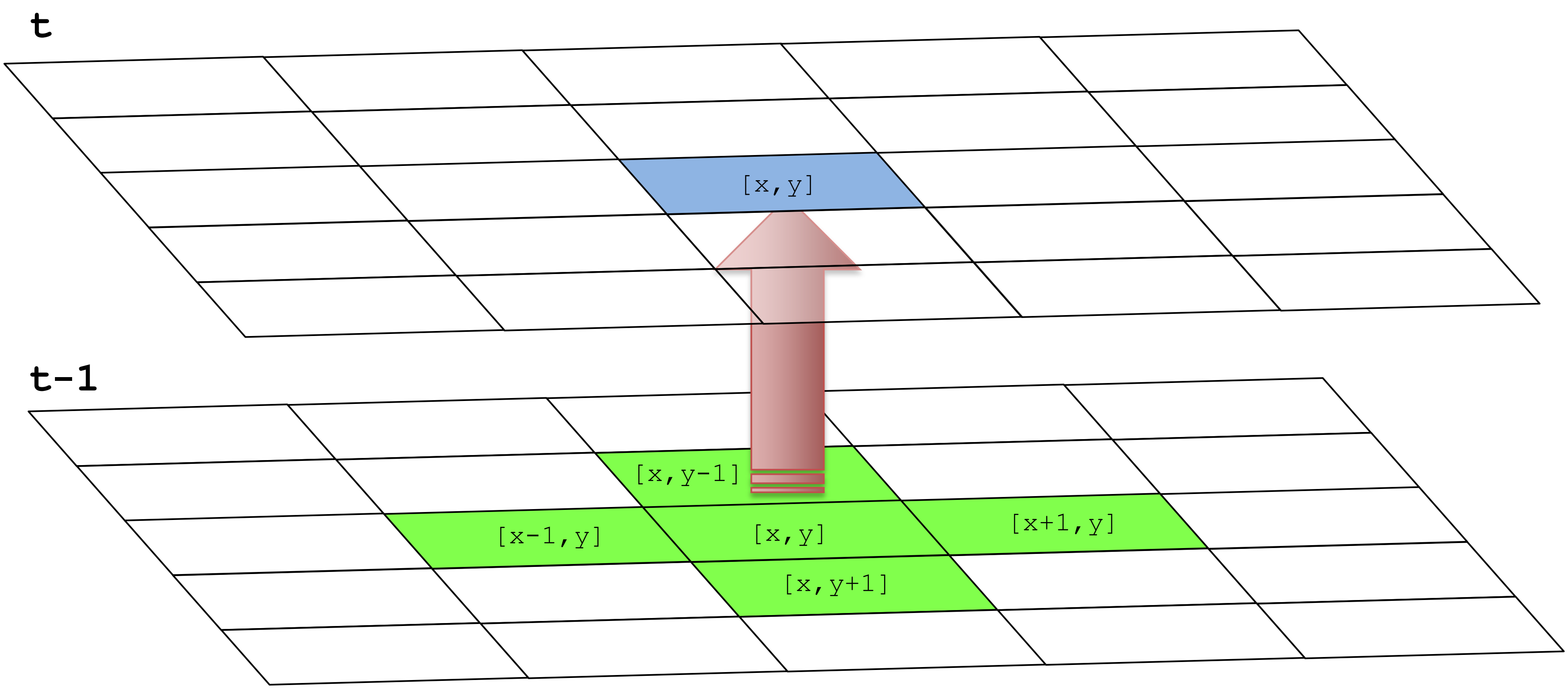
Chapel Programming Language

- A new parallel programming language
  - Design and development led by Cray Inc.
  - Initiated under the DARPA HPCS program
- Overall goal: Improve programmer productivity
  - Improve the **programmability** of parallel computers
  - Match or beat the **performance** of current programming models
  - Support better **portability** than current programming models
  - Improve the **robustness** of parallel codes

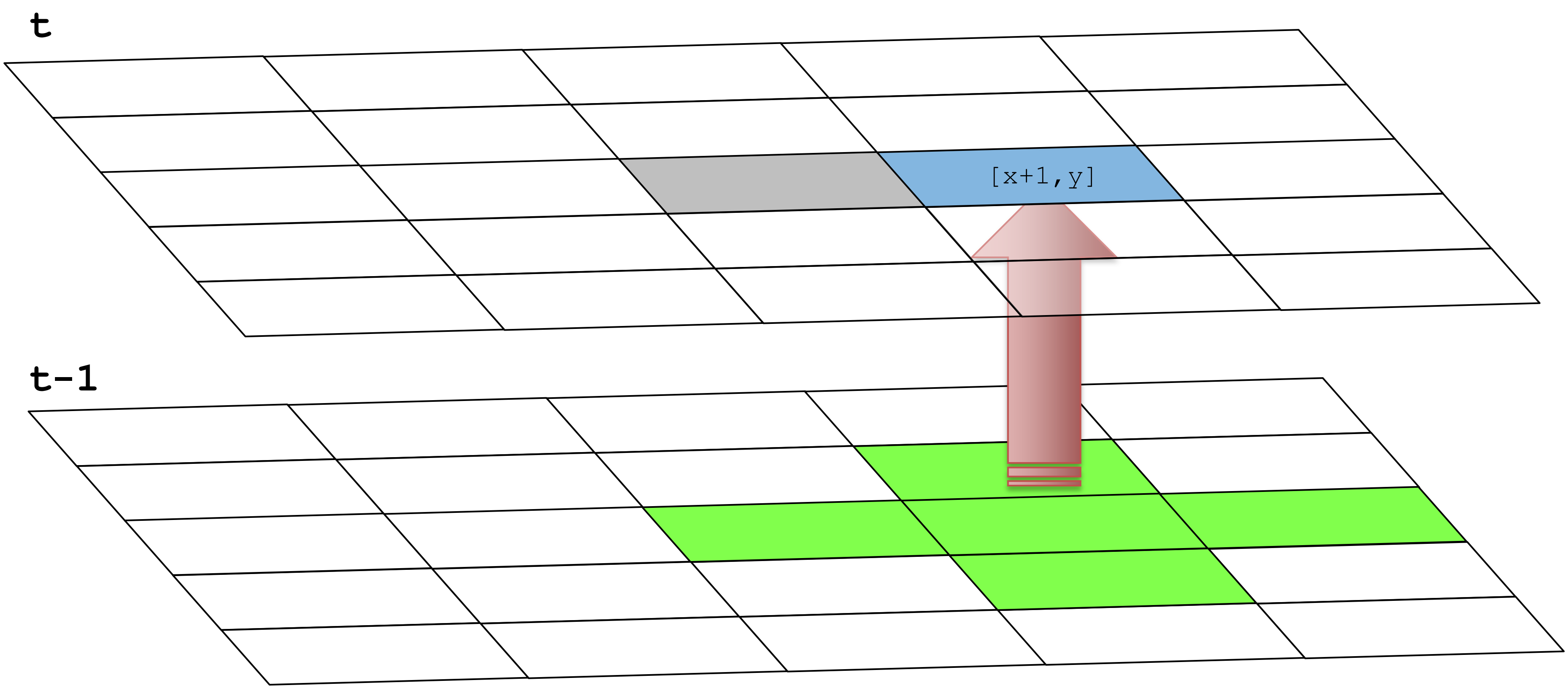
Target: Stencil Computations

- Partial Differential Equation solvers
  - Air and fluid flow simulation
  - Seismic wave models and damage simulation
  - Blast-wave equations
  - Heat equations
  - Atmospheric modeling
  - Magnetic field simulations

Stencil Computations



Stencil Computations



Stencil Computations

C + OpenMP:

Chapel:

Stencil Computations

C + OpenMP:

```
A[t][x][y]=( A[t-1][x-1][y] + A[t-1][x][y-1]
           + A[t-1][x+1][y] + A[t-1][x][y+1]
           + A[t-1][x  ][y] )/5;
```

Chapel:

```
A[t,x,y]=( A[t-1, x-1, y] + A[t-1, x, y-1]
         + A[t-1, x+1, y] + A[t-1, x, y+1]
         + A[t-1, x  , y] )/5;
```

Stencil Computations

C + OpenMP:

```
for( int x = 1; x <= N; x += 1 )
  for( int y = 1; y <= M; y += 1 )
    A[t][x][y]=( A[t-1][x-1][y] + A[t-1][x][y-1]
               + A[t-1][x+1][y] + A[t-1][x][y+1]
               + A[t-1][x  ][y] )/5;
```

Chapel:

```
for (x,y) in {1..N, 1..M} do
  A[t,x,y]=( A[t-1, x-1, y] + A[t-1, x, y-1]
           + A[t-1, x+1, y] + A[t-1, x, y+1]
           + A[t-1, x  , y] )/5;
```

Stencil Computations

C + OpenMP:

```
for( int t = 1; t <= T; t += 1 )
  for( int x = 1; x <= N; x += 1 )
    for( int y = 1; y <= M; y += 1 )
      A[t][x][y]=( A[t-1][x-1][y] + A[t-1][x][y-1]
                 + A[t-1][x+1][y] + A[t-1][x][y+1]
                 + A[t-1][x  ][y] )/5;
```

Chapel:

```
for t in 1..T do
  for (x,y) in {1..N, 1..M} do
    A[t,x,y]=( A[t-1, x-1, y] + A[t-1, x, y-1]
             + A[t-1, x+1, y] + A[t-1, x, y+1]
             + A[t-1, x  , y] )/5;
```

Stencil Computations

C + OpenMP:

```
for( int t = 1; t <= T; t += 1 )
  #pragma omp parallel for
  for( int x = 1; x <= N; x += 1 )
    for( int y = 1; y <= M; y += 1 )
      A[t][x][y]=( A[t-1][x-1][y] + A[t-1][x][y-1]
                 + A[t-1][x+1][y] + A[t-1][x][y+1]
                 + A[t-1][x][y] )/5;
```

Chapel:

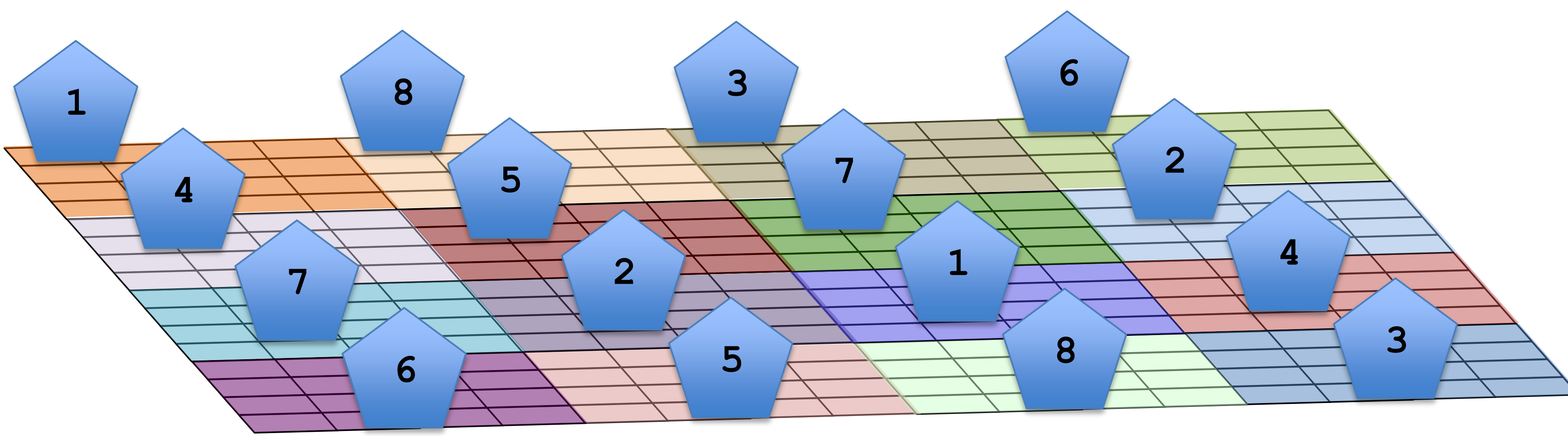
```
for t in 1..T do
  forall (x,y) in {1..N, 1..M} do
    A[t,x,y]=( A[t-1, x-1, y] + A[t-1, x, y-1]
             + A[t-1, x+1, y] + A[t-1, x, y+1]
             + A[t-1, x  , y] )/5;
```

Stencil Computations

- Naïve parallelization performance does not scale with additional cores!
  - Bandwidth-bound computation.
  - Naïve parallelism under utilizes the memory-hierarchy.

Traditional Solution: Space Tiling

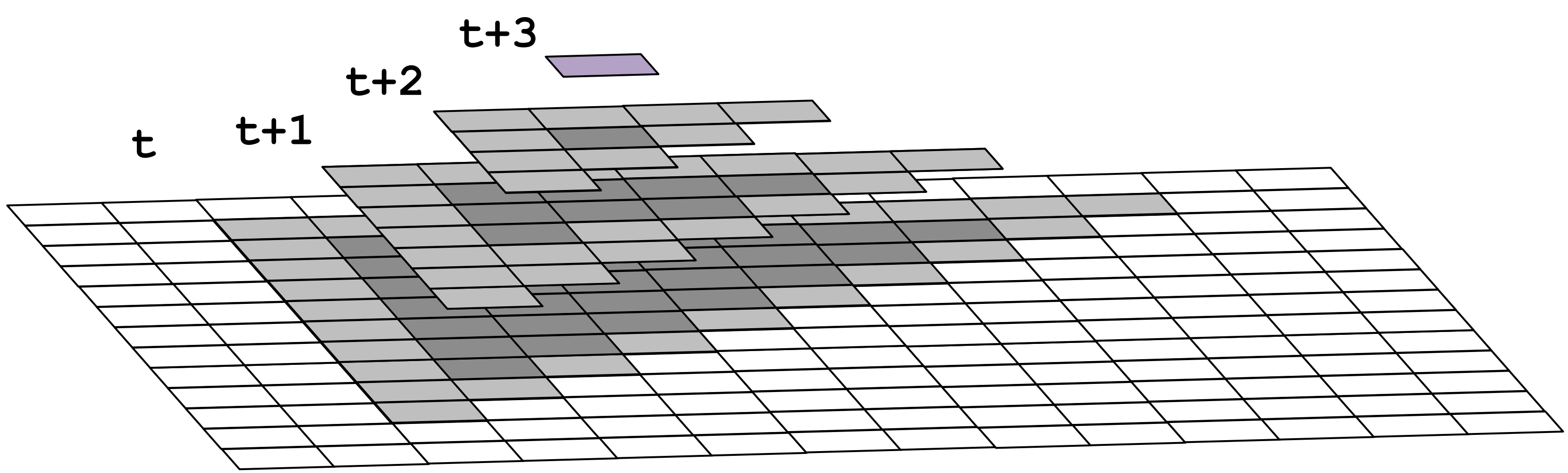
- Group spatial iterations together to reuse shared input data
- No reuse of just-computed values.



Modern Solution: Diamond Tiling

[Bandishi et al. SC12]

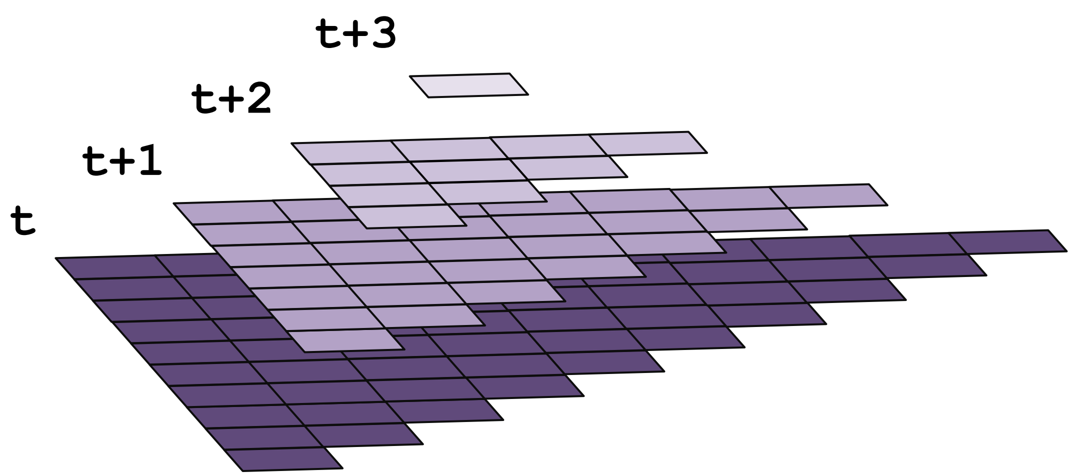
- Mixes space and time tiling
- A single tile will perform multiple time steps
  – Reuse just-computed values



Modern Solution: Diamond Tiling

[Bandishi et al. SC12]

- Mixes space and time tiling
- A single tile will perform multiple time steps
  - Reuse just-computed values



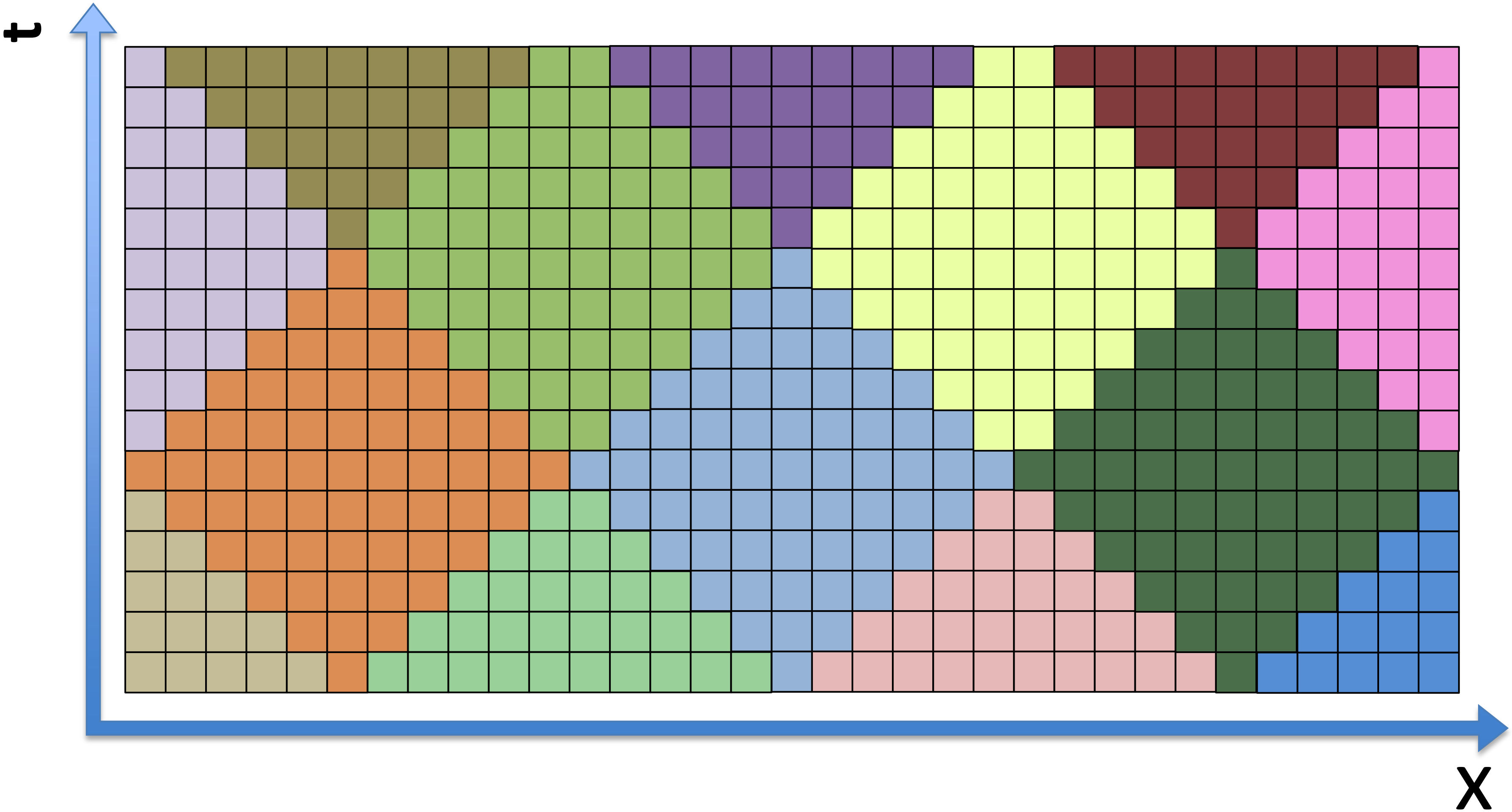
Modern Solution: Diamond Tiling

[Bandishi et al. SC12]

- Mixes space and time tiling
- A single tile will perform multiple time steps
  - Reuse just-computed values
- Concurrent Startup
  - Many tiles can start in parallel

Modern Solution: Diamond Tiling

[Bandishi et al. SC12]



Software Engineering Limitations

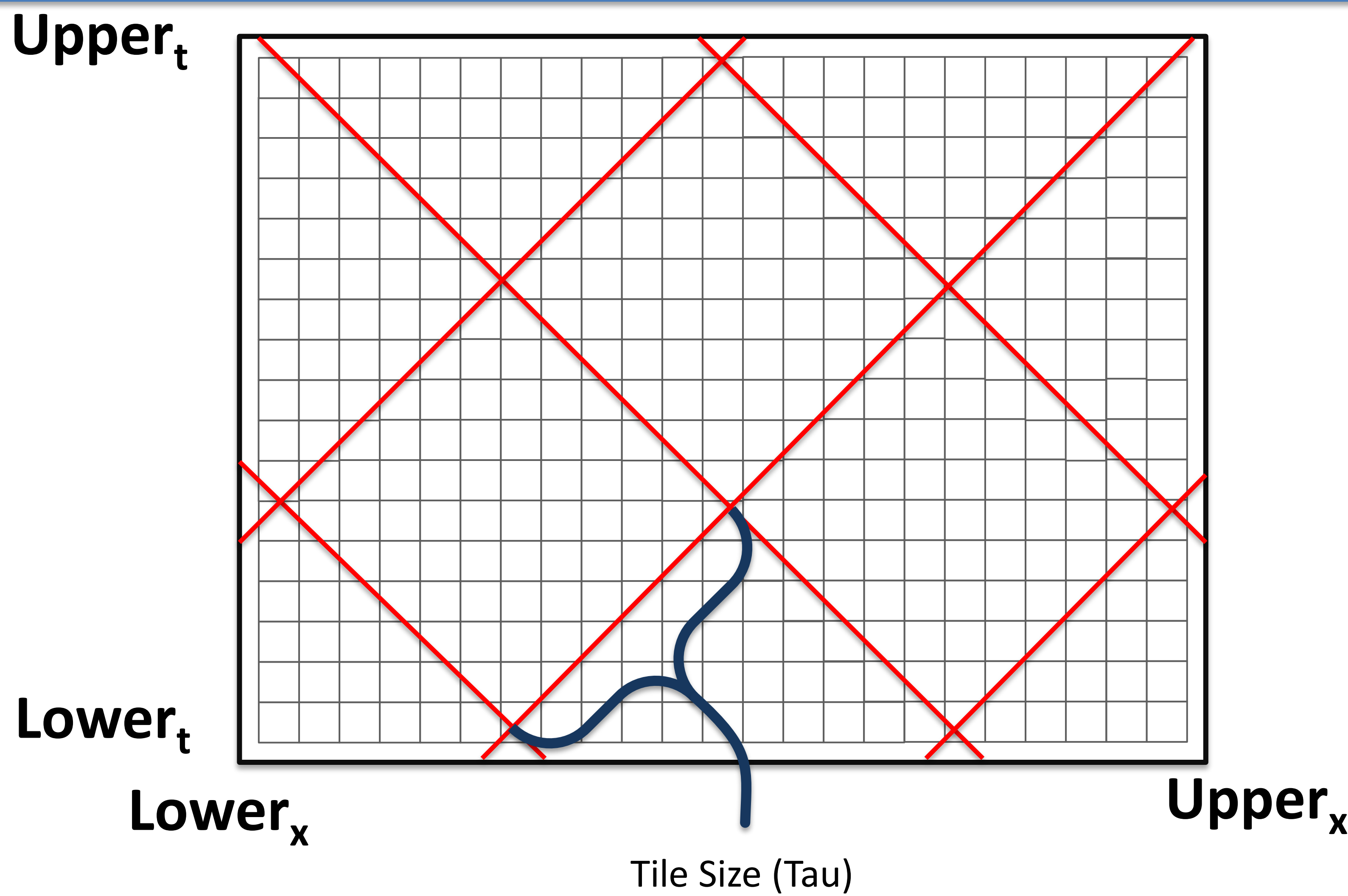
- Code generation requires a constant tile size.
  - Severely complicates application portability.
  - Must generate different code for each tile size that’s optimal for different stencils within application.
  - Lengthens performance profiling process.
- Generated code is not human friendly
  - Multiple stencil-computations results in chains of convoluted loop nests.
  - Difficult to modify, debug, and improve.

Technical Contributions

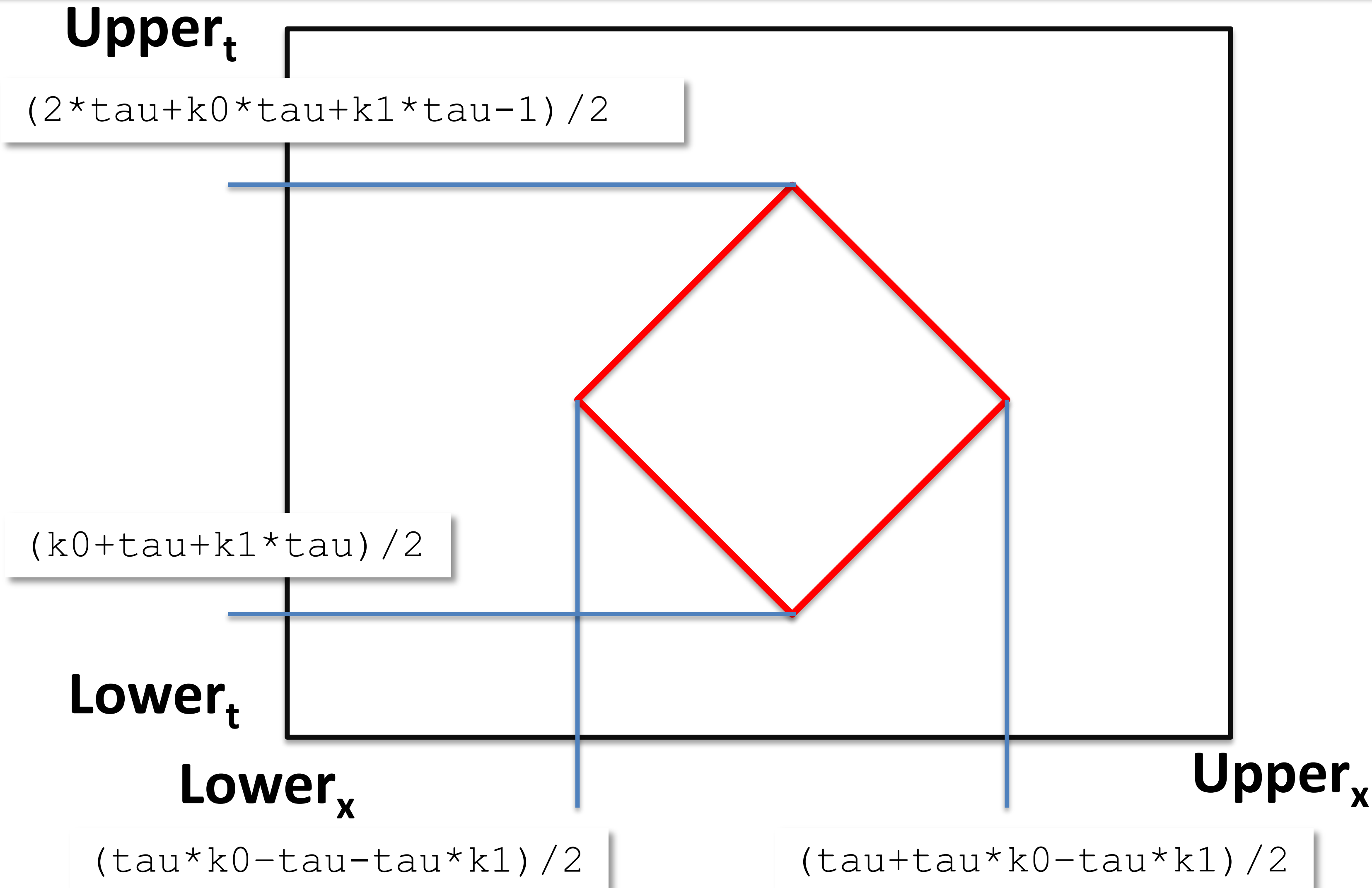
- Parameterized Diamond Tiling

- Demonstration of Chapel iterators as effective tiling schedule deployment mechanism

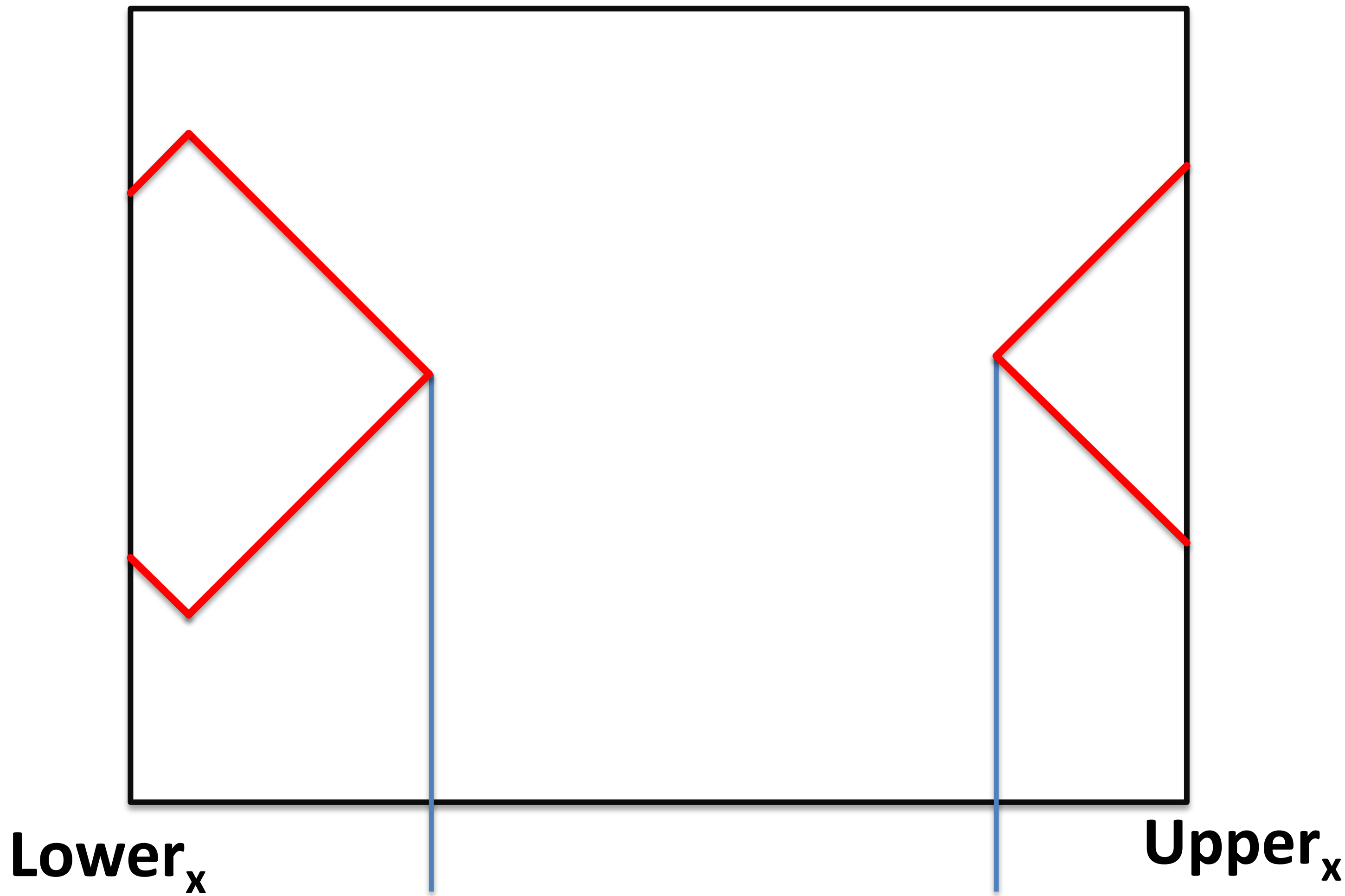
Parameterized Diamond Tiling



Parameterized Diamond Tiling



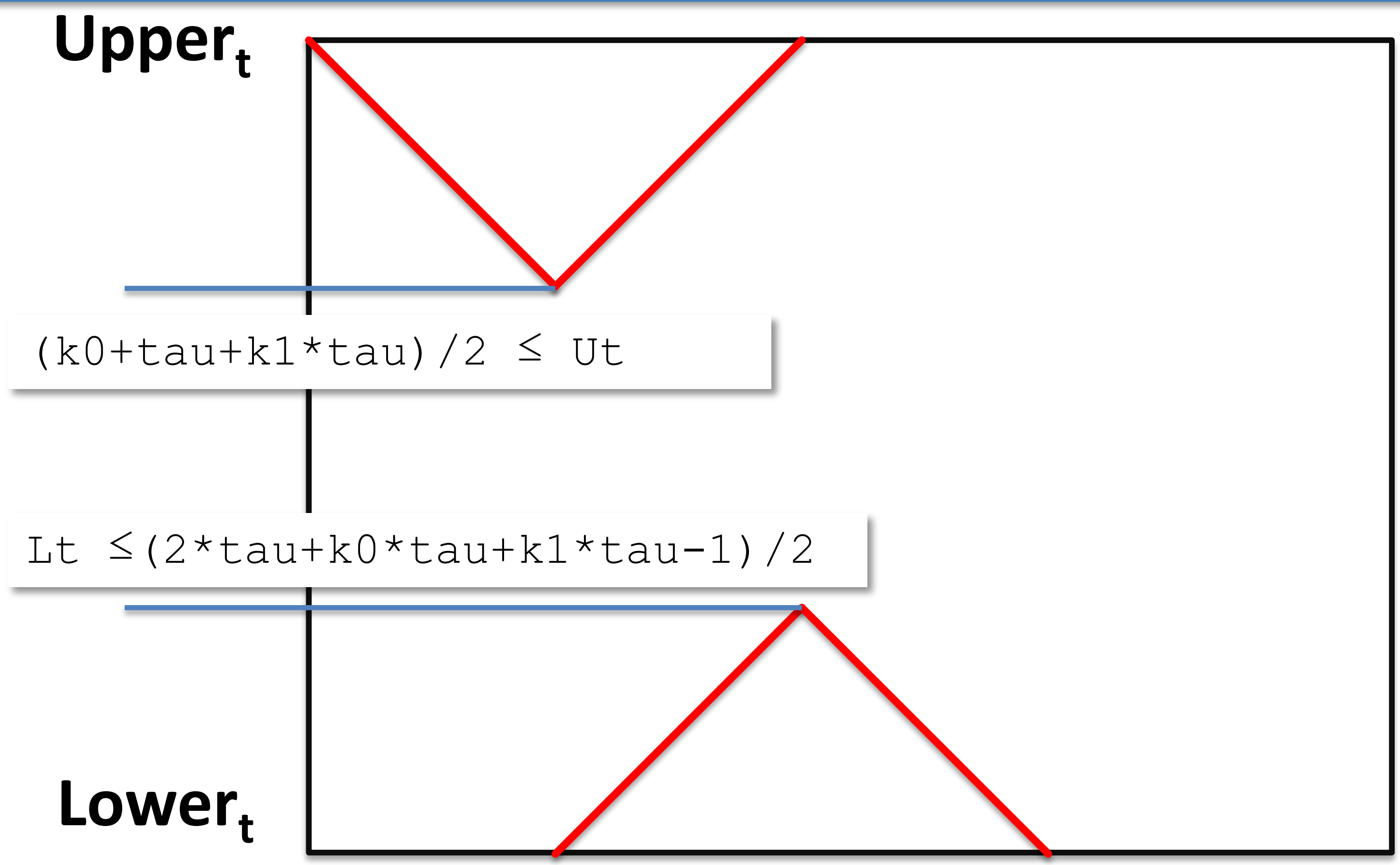
Parameterized Diamond Tiling



`Lx ≤ (tau*k0-tau-tau*k1)/2`

`(tau+tau*k0-tau*k1)/2 ≤ Ux`

Parameterized Diamond Tiling



Parameterized Diamond Tiling

- Generalizable to higher dimensionality, different tiling hyperplanes.
- Paper presents methodology to do this by hand.

Generated Code

C + OpenMP:

```
for (kt=ceild(3,tau)-3; kt<=floord(3*T,tau); kt++) {
 int k1_lb = ceild(3*Lj+2+(kt-2)*tau,tau*3);
 int k1_ub = floord(3*Uj+(kt+2)*tau,tau*3);
 int k2_lb = floord((2*kt-2)*tau-3*Ui+2,tau*3);
 int k2_ub = floord((2+2*kt)*tau-3*Li-2,tau*3);
 #pragma omp parallel for
 for (k1 = k1_lb; k1 <= k1_ub; k1++) {
  for (x = k2_lb; x <= k2_ub; x++) {
   k2 = x - k1;
   for (t = max(1, floord(kt*tau-1, 3));
        t < min(T+1, tau + floord(kt*tau, 3));
        t++) {
    write = t & 1;
    read = 1 - write;
    for (x = max(Li,max((kt-k1-k2)*tau-t, 2*t-(2+k1+k2)*tau+2));
         x <= min(Ui,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau));
        x++) {
     for (y = max(Lj,max(tau*k1-t, t-i-(1+k2)*tau+1));
          y <= min(Uj,min((1+k1)*tau-t-1, t-i-k2*tau));
          y++) {
      A[write][x][y] = ( A[read][x-1][y] + A[read][x][y-1]+
                         A[read][x+1][y] + A[read][x][y+1]+
                         A[read][x  ][y] )/5;
} } } } } }
```

Tile Wave-fronts

Tile Coordinates

Time-Steps in Tile

Read/Write buffer

X,Y Coordinates

Stencil Operation

Generated Code

Chapel:

```
for kt in -2 .. floord(3*T,tau) {
 var k1_lb: int = floord(3*Lj+2+(kt-2)*tau,tau_times_3);
 var k1_ub: int = floord(3*Uj+(kt+2)*tau-2,tau_times_3);
 var k2_lb: int = floord((2*kt-2)*tau-3*Ui+2,tau_times_3);
 var k2_ub: int = floord((2+2*kt)*tau-2-3*Li,tau_times_3);

 forall k1 in k1_lb .. k1_ub {
  for x in k2_lb .. k2_ub {
   var k2 = x-k1;
   for t in max(1,floord(kt*tau,3))
         .. min(T,floord((3+kt)*tau-3,3)){

    write = t & 1;
    read = 1 - write;
    for x in max(Li,(kt-k1-k2)*tau-t,2*t-(2+k1+k2)*tau+2))
          .. min(Ui,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau)) {

     for y in max(Lj,tau*k1-t,t-x-(1+k2)*tau+1)
           .. min(Uj,(1+k1)*tau-t-1,t-x-k2*tau){

      A[write,x,y]=( A[read, x-1, y] + A[read, x, y-1]+
                     A[read, x+1, y] + A[read, x, y+1]+
                     A[read, x  , y] )/5;
} } } } } }
```

Tile Wave-fronts

Tile Coordinates

Time-Steps in Tile

Read/Write buffer

X,Y Coordinates

Stencil Operation

Chapel Iterators

```
iter my_iter( N: int ): int {
  for i in 1..N do yield i;
  for i in N..1 by -1 do yield
i;
}

for j in my_iter( 10 ) do
  writeln( j );
```

Iterator Abstraction

Chapel

```
iter DiamondTileIterator( lowerBound: int, upperBound: int, T: int,
                          tau: int,
                          param tag: iterKind): 4*int
                          where tag == iterKind.standalone {
 for kt in -2 .. floord(3*T,tau) {
  var k1_lb: int = floord(3*Lj+2+(kt-2)*tau,tau_times_3);
  var k1_ub: int = floord(3*Uj+(kt+2)*tau-2,tau_times_3);
  var k2_lb: int = floord((2*kt-2)*tau-3*Ui+2,tau_times_3);
  var k2_ub: int = floord((2+2*kt)*tau-2-3*Li,tau_times_3);

  forall k1 in k1_lb .. k1_ub {
   for x in k2_lb .. k2_ub {
    var k2 = x-k1;
    for t in max(1,floord(kt*tau,3))
          .. min(T,floord((3+kt)*tau-3,3)){

     write = t & 1;
     read = 1 - write;
     for x in max(Li,(kt-k1-k2)*tau-t,2*t-(2+k1+k2)*tau+2))
           .. min(Ui,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau)) {

      for y in max(Lj,tau*k1-t,t-x-(1+k2)*tau+1)
           .. min(Uj,(1+k1)*tau-t-1,t-x-k2*tau){

       yield (read, write, x, y);
       A[write,x,y]=(A[read, x-1, y] + A[read, x, y-1]+
                      A[read, x+1, y] + A[read, x, y+1]+
                      A[read, x  , y] )/5;
} } } } } } }
```

Reduction of Code Complexity

Without Iterator

```
for kt in -2 .. floord(3*T,tau) {
 var k1_lb: int = floord(3*Lj+2+(kt-2)*tau,tau_times_3);
 var k1_ub: int = floord(3*Uj+(kt+2)*tau-2,tau_times_3);
 var k2_lb: int = floord((2*kt-2)*tau-3*Ui+2,tau_times_3);
 var k2_ub: int = floord((2+2*kt)*tau-2-3*Li,tau_times_3);

 forall k1 in k1_lb .. k1_ub {
  for x in k2_lb .. k2_ub {
   var k2 = x-k1;
   for t in max(1,floord(kt*tau,3))
         .. min(T,floord((3+kt)*tau-3,3)){

    write = t & 1;
    read = 1 - write;
    for x in max(Li,(kt-k1-k2)*tau-t,2*t-(2+k1+k2)*tau+2))
          .. min(Ui,min((1+kt-k1-k2)*tau-t-1, 2*t-
                 (k1+k2)*tau)) {

     for y in max(Lj,tau*k1-t,t-x-(1+k2)*tau+1)
           .. min(Uj,(1+k1)*tau-t-1,t-x-k2*tau){

      A[write,x,y]=( A[read, x-1, y] + A[read, x, y-1]+
                     A[read, x+1, y] + A[read, x, y+1]+
                     A[read, x  , y] )/5;
} } } } } }
```

With Iterator

```
forall (read, write, x ,y) in
 DiamondTileIterator(L, U, T, tau) {
  A[write, x, y] = ( A[read, x-1, y] +
                     A[read, x, y-1] +
                     A[read, x, y  ] +
                     A[read, x, y+1] +
                     A[read, x+1, y] )/5;
}
```

Metrics of Success

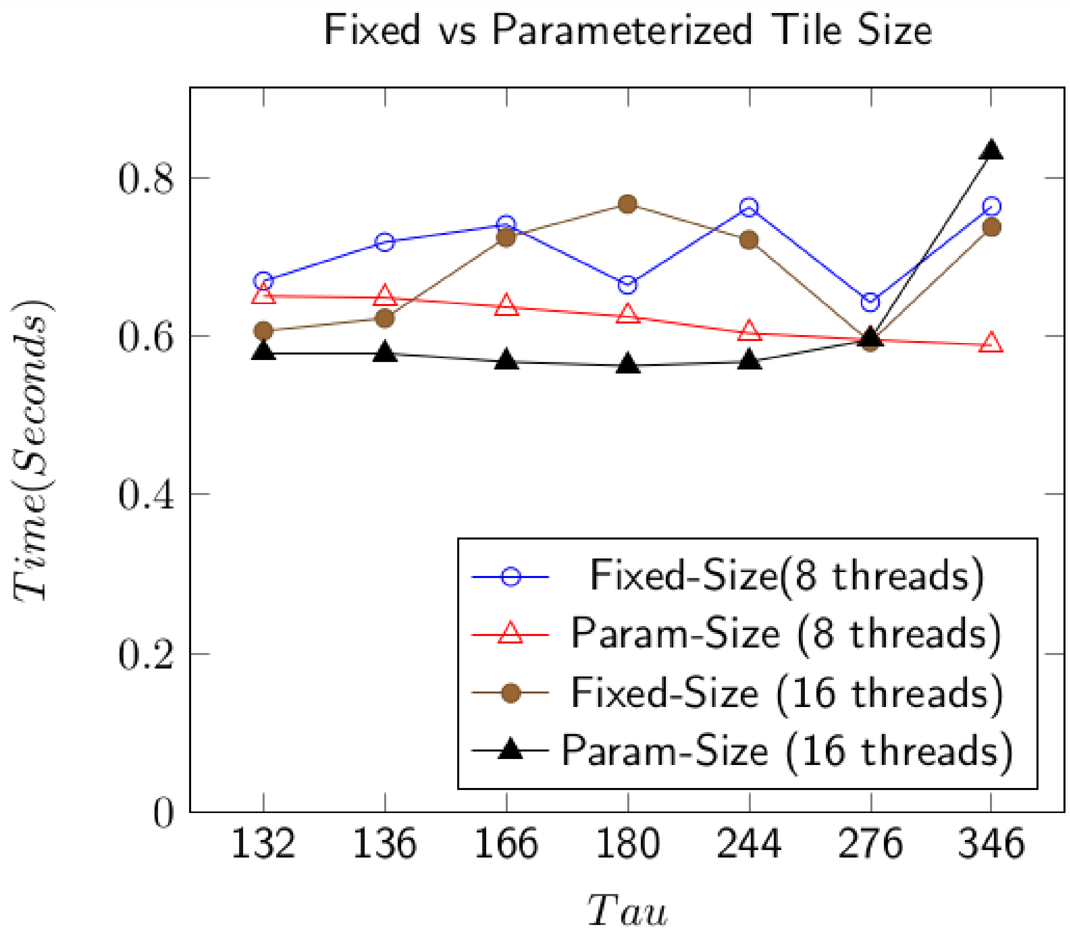
- Parameterized Diamond Tiling is competitive with fixed size Diamond Tiling.

- Chapel iterator performance is competitive with C + OpenMP implementation.

Methodology

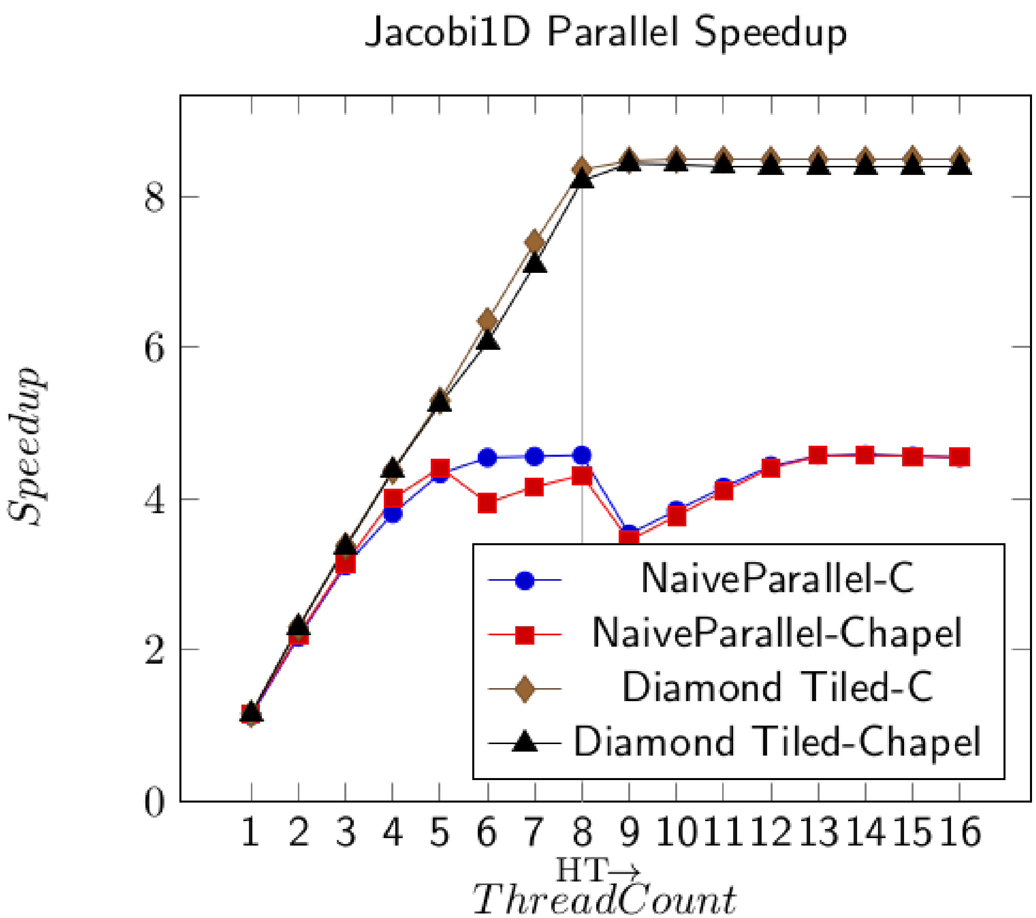
- Hardware:
  - Workstation Machine
  - Single socket Intel Xeon E5
    - 8 Core (16 Hyper-Threads) 2.6GHz
    - 32Kb L1 data, 256Kb L2, 20Mb L3 Cache
  - 32 Gb RAM
- Benchmarks:
  - Jacobi 1D & 2D
  - Problem sizes 2x L3 cache

Parameterized vs Fixed Tile Sizes



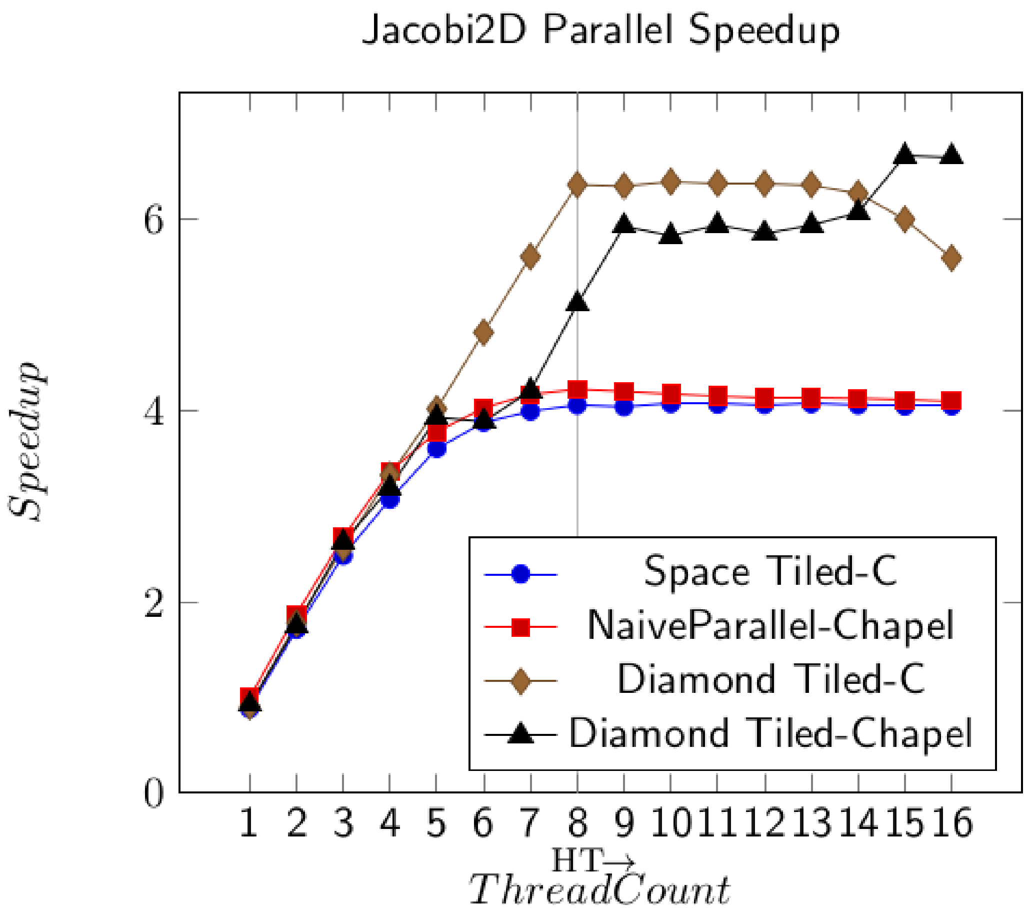
- Parameterized tile size does *better* than fixed.

Competitive Performance



- Maximum Speedup:
  - Chapel: 8.4x
  - C + OpenMP: 8.5x

Competitive Performance



- Maximum Speedup:
  - Chapel: 6.7x
  - C + OpenMP: 6.4x

Conclusion

- Parameterized tile size Diamond Tiling is just as effective as fixed tile size Diamond Tiling.

- Diamond Tiling implemented in Chapel iterators is competitive with Diamond Tiling in C + OpenMP.

- Chapel iterators make advanced tiling schedules much easier to adopt and use.