The Pochoir Stencil Compiler

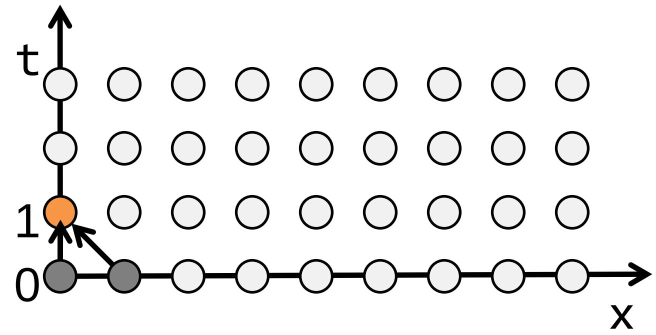
Yuan Tang‡*, Rezaul Chowdhury*, Bradley Kuszmaul*, CK Luk†, Charles Leiserson*

‡*Fudan University*, **MIT CSAIL* and†*Intel*

Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t−1`, `t−2`, …, `t−k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

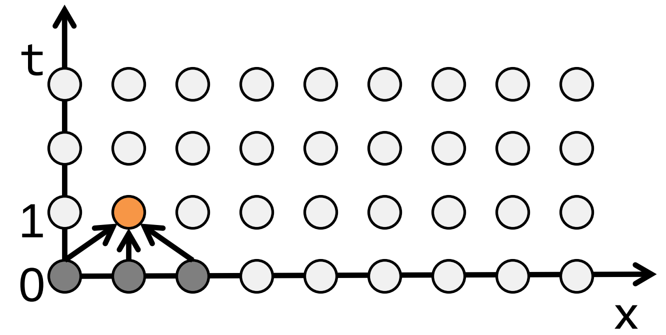
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t–1`, `t–2`, …, `t–k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

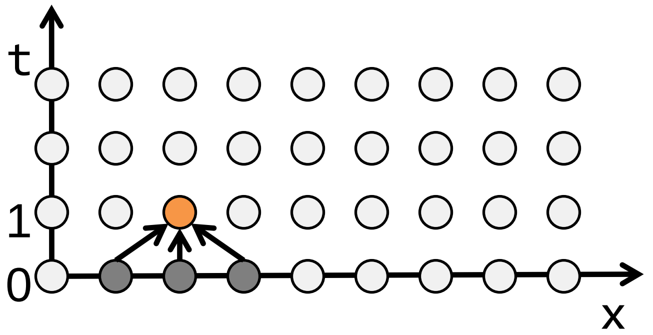
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t–1`, `t–2`, …, `t–k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

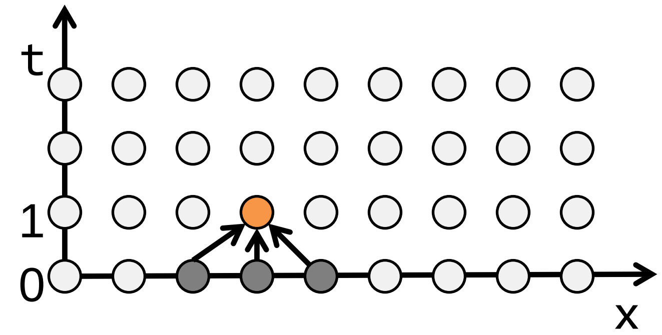
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t–1`, `t–2`, …, `t–k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

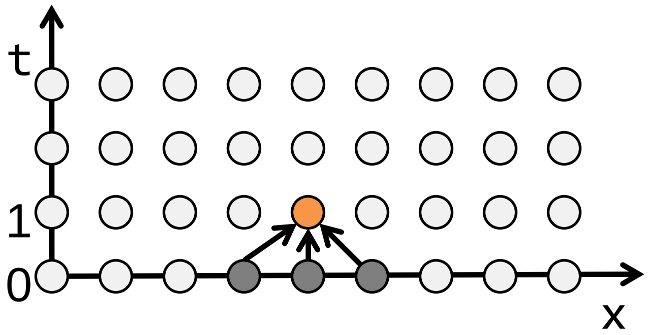
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t-1`, `t-2`, …, `t-k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

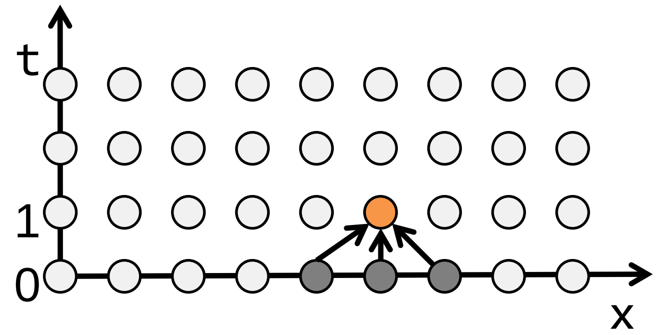
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t−1`, `t−2`, …, `t−k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

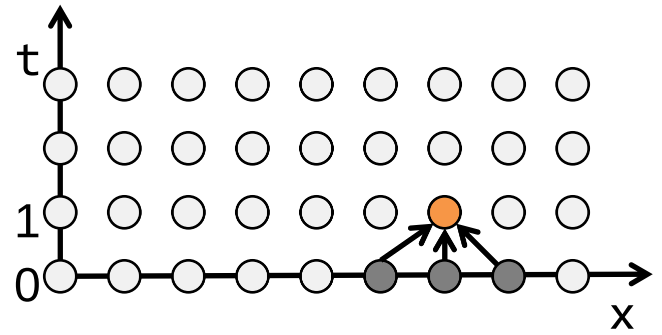
1D 3-point stencil



Background

- A ***stencil code***updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t–1`, `t–2`, …, `t–k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

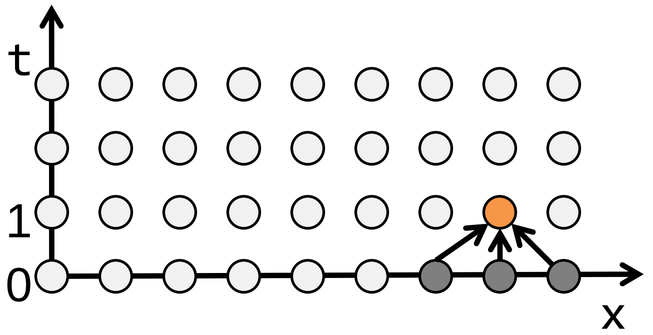
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t−1`, `t−2`, …, `t−k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

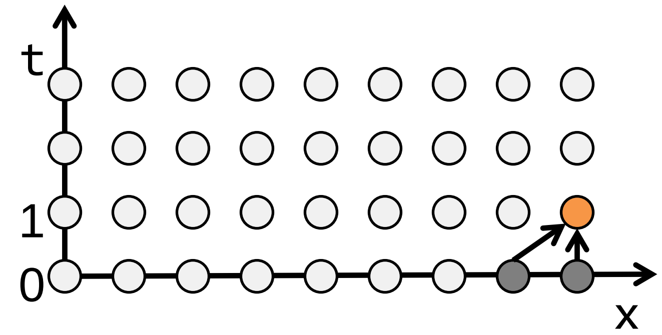
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t−1`, `t−2`, …, `t−k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

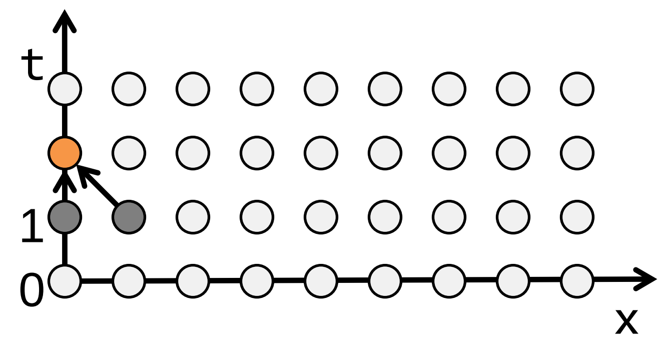
1D 3-point stencil



Background

- A ***stencil code*** updates every point in a `d`-dimensional spatial grid at time t as a function of nearby grid points at times `t–1`, `t–2`, …, `t–k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

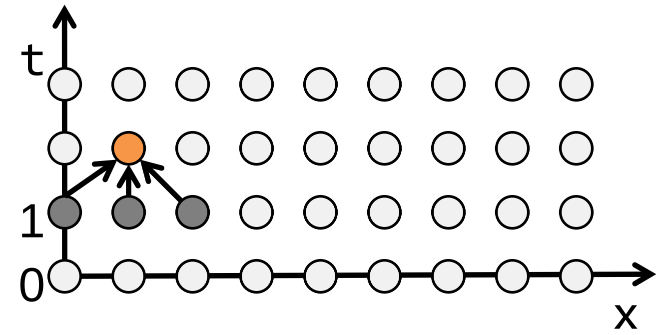
1D 3-point stencil



Background

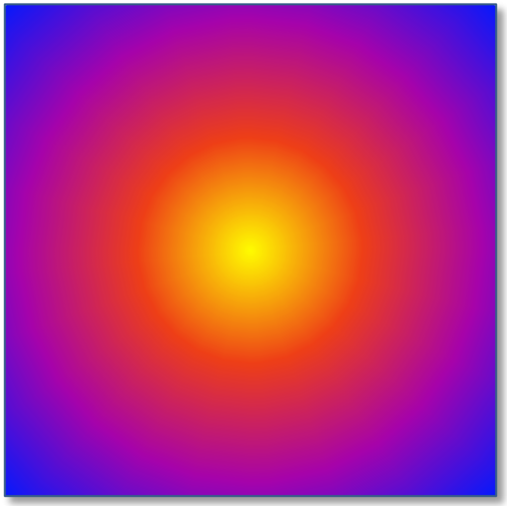
- A ***stencil code*** updates every point in a d-dimensional spatial grid at time t as a function of nearby grid points at times `t–1`, `t–2`, …, `t–k`, for T time steps.
- Stencils are used in iterative PDE solvers such as Jacobi, multigrid, and AMR, as well as for image processing and geometric modeling.

1D 3-point stencil



Example: 2D Heat Diffusion

Let a[t,x,y] be the temperature at time t at point (x,y).



Heat equation

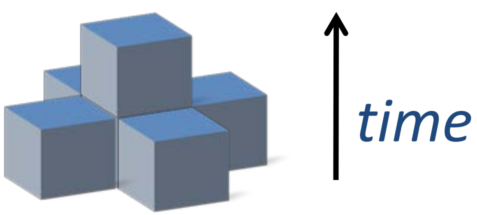
$$\frac{\partial a}{\partial t} = \alpha\left(\frac{\partial^2 a}{\partial x^2} + \frac{\partial^2 a}{\partial y^2}\right)$$

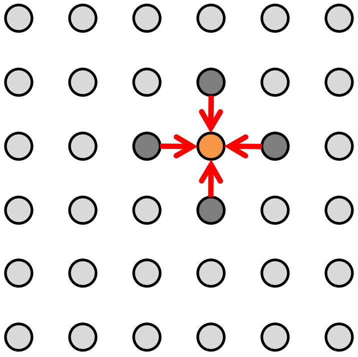
α is the ***thermal diffusivity***.

Update rule

```
a[t,x,y] = a[t–1,x,y]
           + CX·(a[t–1,x+1,y] - 2·a[t–1,x,y] + a[t–1,x–1,y])
           + CY·(a[t–1,x,y+1] - 2·a[t–1,x,y] + a[t–1,x,y–1])
```

2D 5-point stencil

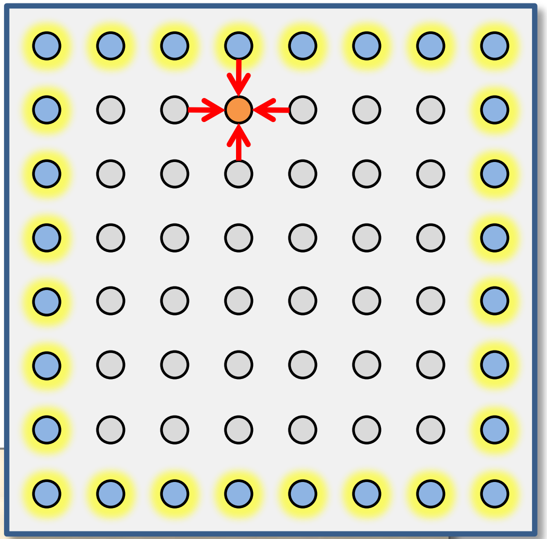




CLASSIC LOOPING IMPLEMENTATION

Implementation tricks

- Reuse storage for even and odd time steps.
- Keep a ***halo*** of ***ghost cells*** around the array with boundary values.

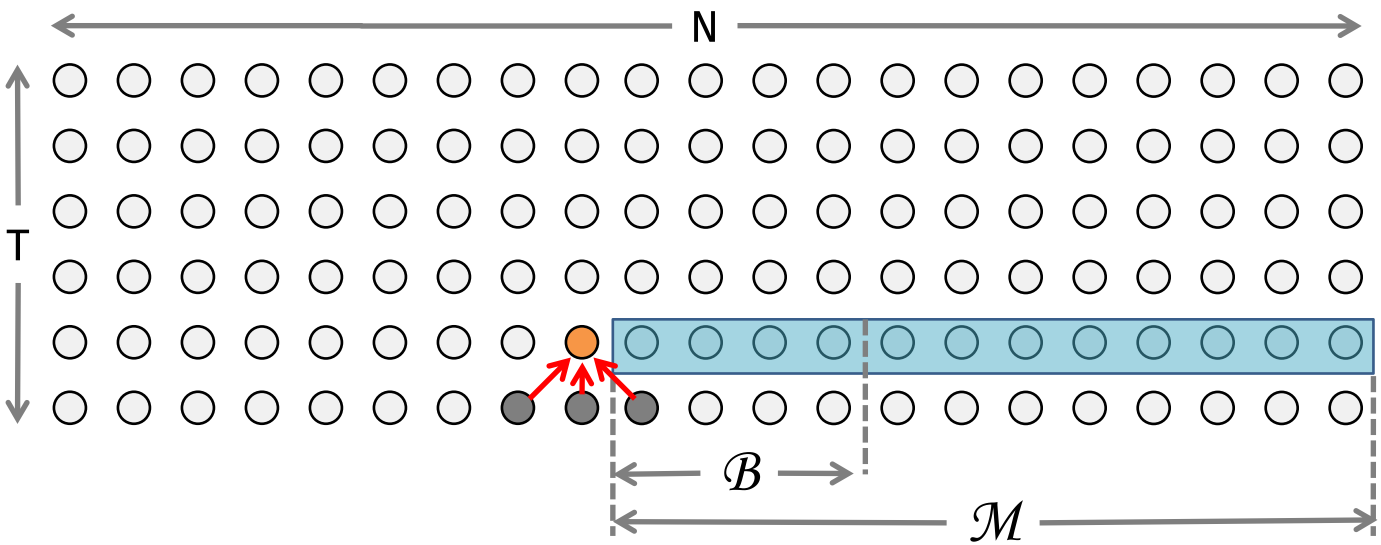


```
for (t = 1; t <= T; ++t) {
  for (x = 0; x < X; ++x) {
    for (y = 0; y < Y; ++y) { // do stencil kernel
      a[t%2, x, y]
          = a[(t-1)%2, x, y]
            + CX*(a[(t-1)%2, x+1, y] - 2.0*a[(t-1)%2, x, y]
                  + a[(t-1)%2, x-1, y])
            + CY*(a[(t-1)%2, x, y+1] - 2.0*a[(t-1)%2, x, y]
                  + a[(t-1)%2, x, y-1]);
} } } }
```

Conventional optimization: *loop tiling*.

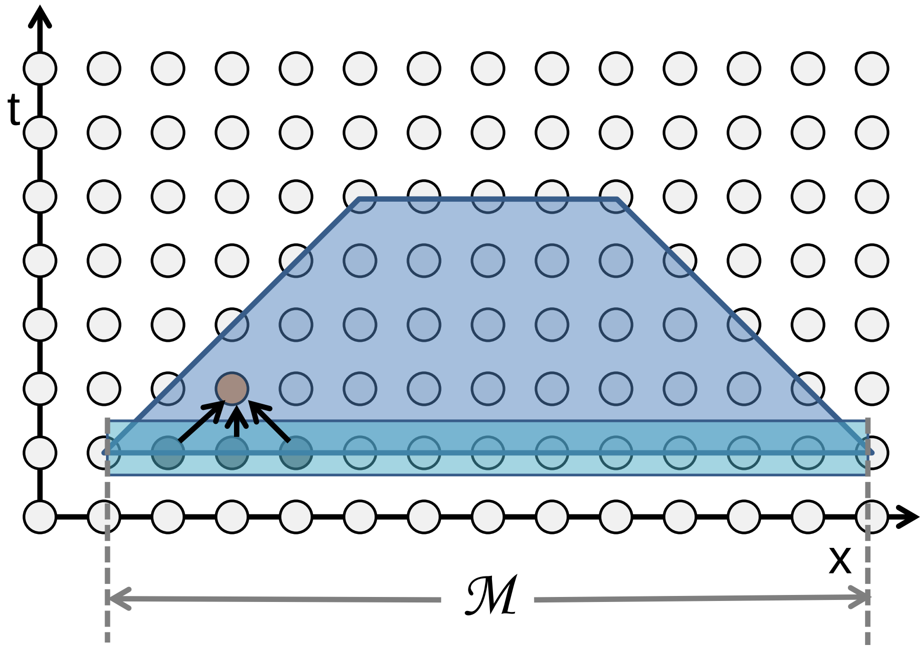
Cache inefficiency in Looping

Example: 1D 3-point stencil



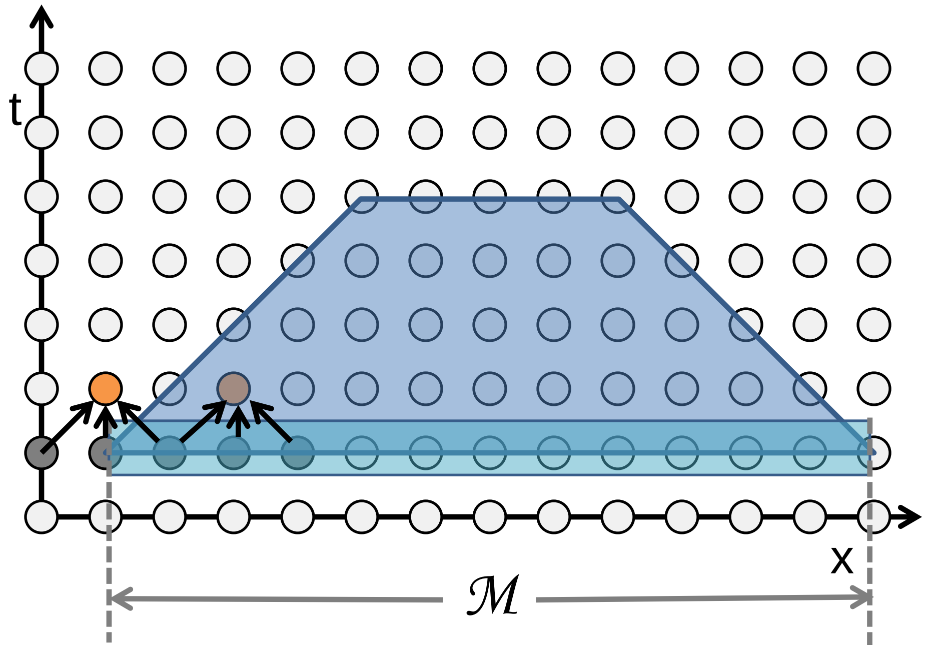
Issue: Looping is memory intensive and uses caches poorly. Assuming data-set size N, cache-block size $\mathcal{B}$, and cache size $\mathcal{M}$<N, the number of cache misses for T time steps is $\Theta(NT/\mathcal{B})$.

Cache-Oblivious Stencils



Based on divide-and-conquer ***cache-oblivious***[FLPR99] techniques, ***trapezoidal decompositions***[FS05], are asymptotically efficient, achieving $\Theta(NT/\mathcal{M}\mathcal{B})$ cache misses.

Cache-Oblivious Stencils



Based on divide-and-conquer ***cache-oblivious***[FLPR99] techniques, ***trapezoidal decompositions***[FS05], are asymptotically efficient, achieving $\Theta(NT/\mathcal{M}\mathcal{B})$ cache misses.

Algorithm Behind Pochoir

```
void TRAP(u, ta,tb, xa,xb, dxa,dxb, ya,yb, dya,dyb) {
    Δt = tb – ta
    Δx = max{xb-xa, (xb + dxb Δt) – (xa + dxa Δt)}
    Δy = max{yb-ya, (yb + dyb Δt) – (ya + dya Δt)}
    k = 0 // Try hyper space cut
    if (Δx ≥ 2σΔt) // cut x dimension
        Trisect the zoid with x-cuts, k += 1
    if (Δy ≥ 2σΔt) // cut y dimension
        Trisect the zoid with y-cuts, k += 1
    if (k > 0)
        Assign dependency levels 0, 1, …, k to subzoids
        for i = 0 to k
            parallel for all subzoids (ta,tb, xa’,xb’, dxa’,dxb’, ya’,yb’, dya’, dyb’) with
    dependency level i
                TRAP(ta,tb, xa’,xb’, dxa’,dxb’, ya’,yb’, dya’,dyb’)
    elseif Δt > 1 //time cut: recursively walk the lower and then upper zoids
           TRAP(ta,ta+Δt/2, xa,xb, dxa,dxb, ya,yb, dya,dyb)
           TRAP(ta+ Δt/2,tb, xa+dxa Δt/2,xb+dxb Δt/2, dxa,dxb, ya+dya Δt/2,yb+dyb Δt/2,
    dya,dyb)
    else // base case
        for t = ta to tb-1
            for x = xa to xb-1
                for y = ya to yb-1
                    u((t+1) mod 2, x, y) = u(t mod 2, x, y) + CX * (u(t mod 2, (x-1) mod X,
    y) + u(t mod 2, (x+1) mod X, y) – 2u(t mod 2, x, y)) + CY * (u(t mod 2, x, (y-1) mod Y)
    + u(t mod 2, x, (y+1) mod Y) – 2u(t mod 2, x, y))
            xa += dxa
            xb += dxb
            yb += dya
            yb += dyb
}
```

Algorithm Behind Pochoir

```
void TRAP(u, ta,tb, xa,xb, dxa,dxb, ya,yb, dya,dyb) {
    Δt = tb – ta
    Δx = max{xb-xa, (xb + dxb Δt) – (xa + dxa Δt)}
    Δy = max{yb-ya, (yb + dyb Δt) – (ya + dya Δt)}
    k = 0 // Try hyper space cut
    if (Δx ≥ 2σΔt) // cut x dimension
        Trisect the zoid with x-cuts, k += 1
    if (Δy ≥ 2σΔt) // cut y dimension
        Trisect the zoid with y-cuts, k += 1
    if (k > 0)
        Assign dependency levels 0, 1, …, k to subzoids
        for i = 0 to k
            parallel for all subzoids (ta,tb, xa’,xb’, dxa’,dxb’, ya’,yb’, dya’, dyb’) with
    dependency level i
                TRAP(ta,tb, xa’,xb’, dxa’,dxb’, ya’,yb’, dya’,dyb’)
    elseif Δt > 1 //time cut: recursively walk the lower and then upper zoids
           TRAP(ta,ta+Δt/2, xa,xb, dxa,dxb, ya,yb, dya,dyb)
           TRAP(ta+ Δt/2,tb, xa+dxa Δt/2,xb+dxb Δt/2, dxa,dxb, ya+dya Δt/2,yb+dyb Δt/2,
    dya,dyb)
    else // base case
        for t = ta to tb-1
            for x = xa to xb-1
                for y = ya to yb-1
                    u((t+1) mod 2, x, y) = u(t mod 2, x, y) + CX * (u(t mod 2, (x-1) mod X,
    y) + u(t mod 2, (x+1) mod X, y) – 2u(t mod 2, x, y)) + CY * (u(t mod 2, x, (y-1) mod Y)
    + u(t mod 2, x, (y+1) mod Y) – 2u(t mod 2, x, y))
            xa += dxa
            xb += dxb
            yb += dya
            yb += dyb
}
```

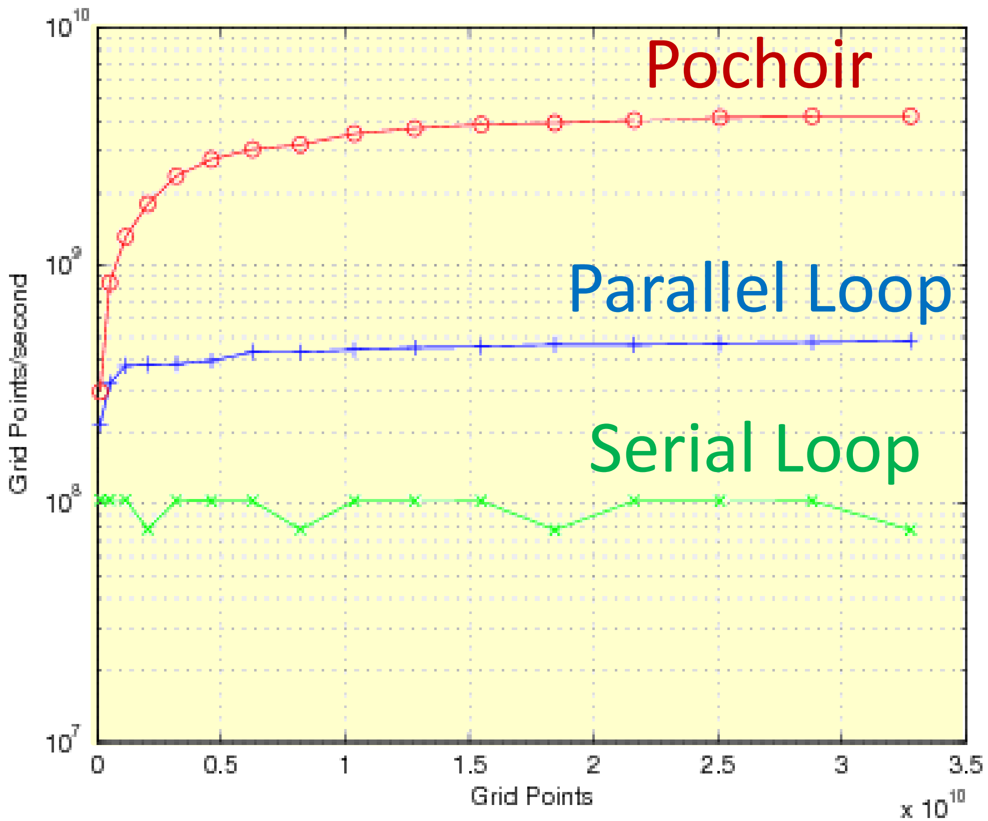
Difficult to write!!

Pochoir Stencil Compiler

- Domain-specific compiler programmed in Haskell that compiles a stencil language embedded in C++, a traditionally difficult language in which to embed a separately compiled domain-specific language.
- Employs a novel cache-oblivious algorithm for arbitrary d-dimensional grids which is parallelized using Intel Cilk Plus.
- Makes it straightforward to code arbitrary periodic and nonperiodic boundary conditions, including Neumann and Dirichlet conditions.
- Implements a variety of stencil-specific optimizations.
- The stencil specification can be executed and debugged without the Pochoir compiler.

2D Heat Equation

5-point stencil on a torus

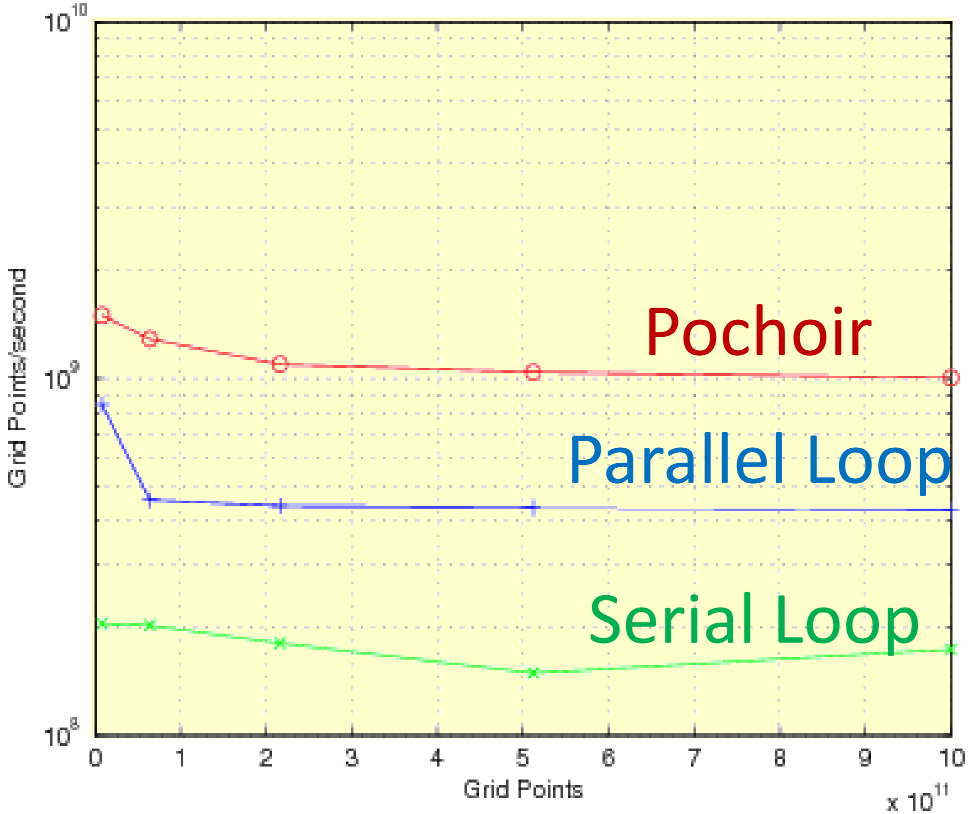


Intel C++ compiler 12.0.0 with Cilk Plus on
12 core Intel core i7 (Nehalem)

pochoir@csail.mit.edu

3D Wave Equation

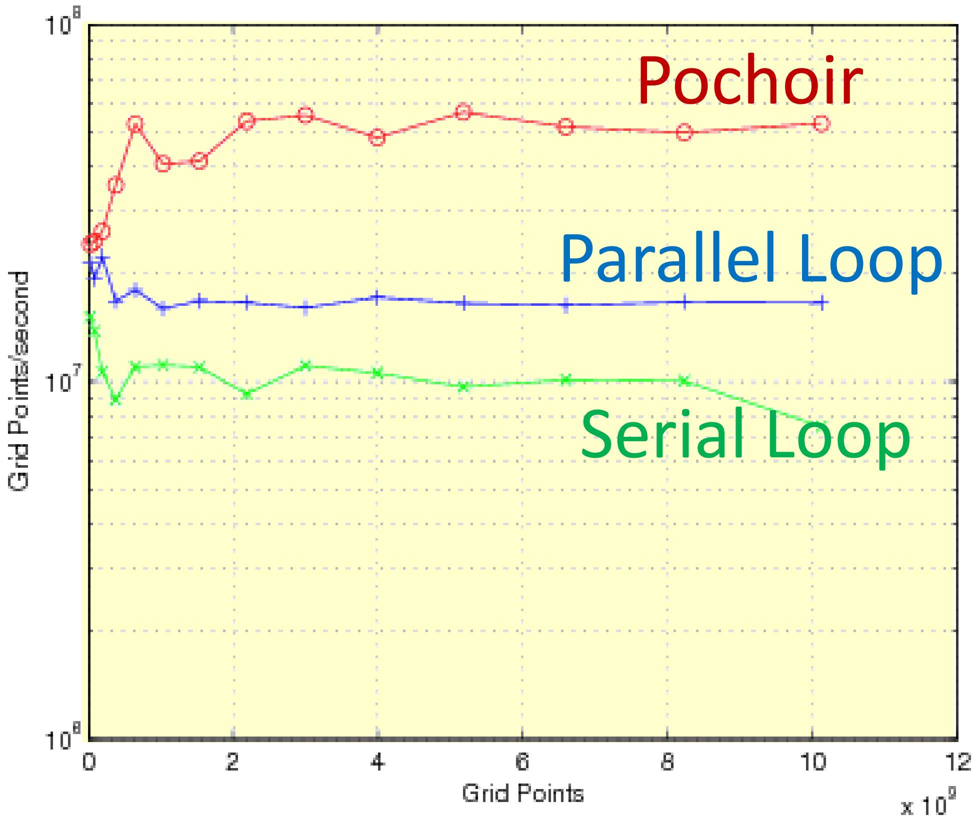
25-point stencil on a nonperiodic domain



Intel C++ compiler 12.0.0 with Cilk Plus on
12 core Intel core i7 (Nehalem)

3D Lattice Boltzmann Method

19-point stencil on a nonperiodic domain



Intel C++ compiler 12.0.0 with Cilk Plus on
12 core Intel core i7 (Nehalem)

pochoir@csail.mit.edu

Auto-tuned vs Pochoir

	Berkeley Autotuner	*Pochoir*
CPU	Xeon X5550	Xeon X5650
Clock	2.66GHz	2.66 GHz
cores/socket, total	4, 8	6, 12
Hyperthreading	Enabled	Disabled
L1 data cache/core	32KB	32KB
L2 cache/core	256KB	256KB
L3 cache/socket	8MB	12 MB
Peak computation	85 GFLOPS	120 GFLOPS
Compiler	icc 10.0.0	icc 12.0.0
Linux kernel		2.6.32
Threading model	Pthreads	Cilk Plus
Problem Size	$\mathbf{258^3 * 1}$	$\mathbf{258^3 * 200}$
3D 7-point 8 cores	2.0 GStencil/s 15.8 GFLOPS	***2.49 GStencil/s*** ***19.92 GFLOPS***
3D 27-point 8 cores	***0.95 GStencil/s*** ***28.5 GFLOPS***	0.88 GStencil/s 26.4 GFLOPS

Publications

- The Pochoir Stencil Compiler, Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson, SPAA'11: The 23rd ACM Symposium on Parallelism in Algorithms and Architectures, June 4-6, San Jose, CA, USA.
- Coding Stencil Computation using the Pochoir Stencil-Specification language, Yuan Tang, Rezaul Alam Chowdhury, Chi-Keung Luk, and Charles E. Leiserson, HotPar'11: 3rd USENIX Workshop on Hot Topics in Parallelism, May 26-27, Berkeley, CA, USA.

Thank You!

Email pochoir@csail.mit.edu to request a copy of Pochoir compiler

Outline

- Functional Specification
- How the Pochoir System Works
- Algorithms
- Optimizing Strategies
- Conclusion

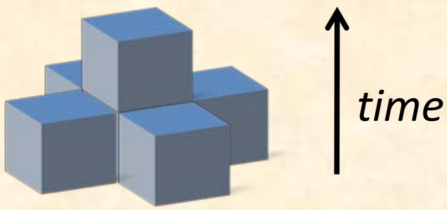
Outline

- Functional Specification
- How the Pochoir System Works
- Algorithms
- Optimizing Strategies
- Conclusion

Functional Specification

- Embedded in C++.
- Directly executable and debuggable via any native C++ tool chain.
- Supports arbitrary `d`-dimensional rectangular grids.
- The stencil shape can be arbitrary.
- A point at time `t` can depend on points at time `t-1`, `t-2`, …, `t-k`. ---- arbitrary depth
- Both periodic and nonperiodic boundary conditions can be programmed.

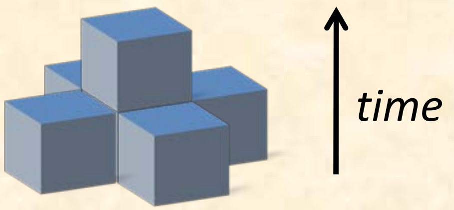
2D Heat Equation




```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,0,0}, {-1,1,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);

  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);

  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
              + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
              + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();

  heat.Run(T, kern);

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);

  return 0;
}
```

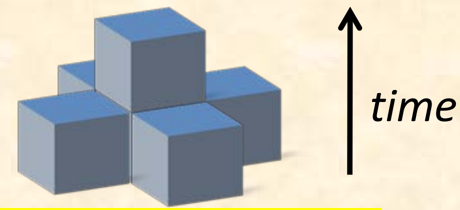
2D Heat Equation

```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,0,0}, {-1,1,0}, {-1,-
  Pochoir_2D heat(2D_five_pt);

  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);

  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
               + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
               + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();

  heat.Run(T, kern);

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);

  return 0;
}
```



Declare a ***kernel function*** `kern` with time parameter `t` and spatial parameters `x` and `y`.

2D Heat Equation

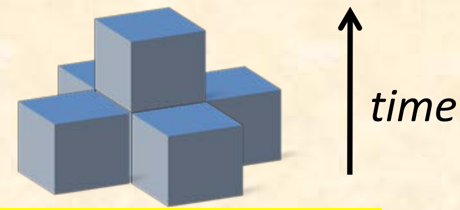



```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,0,0}, {-1,1,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);
  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);
  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
              + 0.125*(a(t-1,x+1,y) -
              + 0.125*(a(t-1,x,y+1) -
  Pochoir_Kernel_End
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();
  heat.Run(T, kern);
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);
  return 0;
}
```

Declare the 2-dimensional ***Pochoir shape*** `2D_five_pt` as a list of 6 cells. Each cell specifies the relative offset of indices used in the kernel function, *e.g.*, for `a(t,x,y)`, we specify the corresponding cell `{0,0,0}`, for `a(t−1,x+1,y)`, we specify `{−1,1,0}`, and so on.

2D Heat Equation

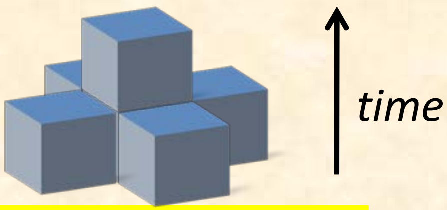
```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End
int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,0,0}, {-1,1,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);
  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);
  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
             + 0.125*(a(t-1,x+1,y) -
             + 0.125*(a(t-1,x,y+1) -
  Pochoir_Kernel_End
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();
  heat.Run(T, kern);
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);
  return 0;
}
```



Declare the 2-dimensional ***Pochoir shape*** `2D_five_pt` as a list of 6 cells. Each cell specifies the relative offset of indices used in the kernel function, *e.g.*, for `a(t,x,y)`, we specify the corresponding cell `{0,0,0}`, for `a(t−1,x+1,y)`, we specify `{-1,1,0}`, and so on.

2D Heat Equation

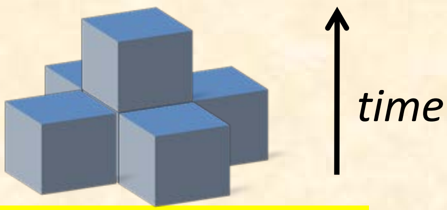
```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End
int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,0,0}, {-1,1,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);
  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);
  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) = a(t-1,x,y)
             + 0.125*(a(t-1,x+1,y) -
             + 0.125*(a(t-1,x,y+1) -
  Pochoir_Kernel_End
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();
  heat.Run(T, kern);
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);
  return 0;
}
```



Declare the 2-dimensional ***Pochoir shape*** `2D_five_pt` as a list of 6 cells. Each cell specifies the relative offset of indices used in the kernel function, *e.g.*, for `a(t,x,y)`, we specify the corresponding cell `{0,0,0}`, for `a(t-1,x+1,y)`, we specify `{-1,1,0}`, and so on.

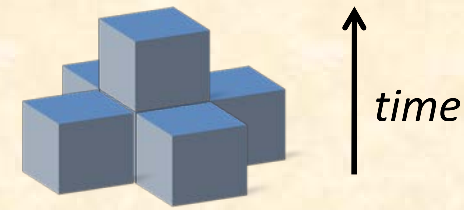
2D Heat Equation

```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,0,0}, {-1,1,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);

  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);

  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
              + 0.125*(a(t-1,x+1,y) -
              + 0.125*(a(t-1,x,y+1) -
  Pochoir_Kernel_End

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();

  heat.Run(T, kern);

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);

  return 0;
}
```



Declare the 2-dimensional ***Pochoir shape*** `2D_five_pt` as a list of 6 cells. Each cell specifies the relative offset of indices used in the kernel function, *e.g.*, for `a(t,x,y)`, we specify the corresponding cell `{0,0,0}`, for `a(t−1,x+1,y)`, we specify `{-1,1,0}`, and so on.

2D Heat Equation




```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,0,0}, {-1,1,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);
  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);
  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
              + 0.125*(a(t-1,x+1,y) - 2
              + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();
  heat.Run(T, kern);
  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);
  return 0;
}
```

Declare a ***boundary function*** `zero_bdry` on the 2-dimensional Pochoir array `arr` indexed by time coordinate `t` and spatial coordinates `x` and `y`, which always returns `0`.

Boundary Conditions

Nonperiodic zero boundary

```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
   return 0;
Pochoir_Boundary_End
```

Periodic (toroidal) boundary

```
#define mod(r,m) (((r) % (m)) + ((r)<0)?(m):0)
Pochoir_Boundary_2D(periodic, arr, t, x, y)
   return arr.get( t,
                   mod(x, arr.size(1)),
                   mod(y, arr.size(0)) );
Pochoir_Boundary_End
```

Cylindrical boundary

```
#define mod(r,m) (((r) % (m)) + ((r)<0)?(m):0)
Pochoir_Boundary_2D(cylinder, arr, t, x, y)
   if (x < 0) || (x >= arr.size(1))
     return 0;
   return arr.get( t, x, mod(y, arr.size(0)) );
Pochoir_Boundary_End
```

Boundary Conditions (Cont.)

Dirichlet boundary

```
Pochoir_Boundary_2D(dirichlet, arr, t, x, y)
   return 100+0.2*t;
Pochoir_Boundary_End
```

Neumann boundary

```
Pochoir_Boundary_2D(neumann, arr, t, x, y)
int xx(x), yy(y);
   if (x<0) xx = 0;
   if (x>=arr.size(1)) xx = arr.size(1);
   if (y<0) yy = 0;
   if (y>=arr.size(0)) yy = arr.size(0);
   return arr.get(t, xx, yy);
Pochoir_Boundary_End
```

2D Heat Equation

```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);

  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);

  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
              + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
              + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();

  heat.Run(T, kern);

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);

  return 0;
}
```

Initialize all points of the grid at time `0` to a random value.

2D Heat Equation

```
Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
  return 0;
Pochoir_Boundary_End

int main(void) {
  Pochoir_Shape_2D 2D_five_pt[6]
    = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
  Pochoir_2D heat(2D_five_pt);

  Pochoir_Array_2D(double) a(X,Y);
  a.Register_Boundary(zero_bdry);
  heat.Register_Array(a);

  Pochoir_Kernel_2D(kern, t, x, y)
    a(t,x,y) =  a(t-1,x,y)
              + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
              + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
  Pochoir_Kernel_End

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      a(0,x,y) = rand();

  heat.Run(T, kern);

  for (int x = 0; x < X; ++x)
    for (int y = 0; y < Y; ++y)
      cout << a(T,x,y);

  return 0;
}
```

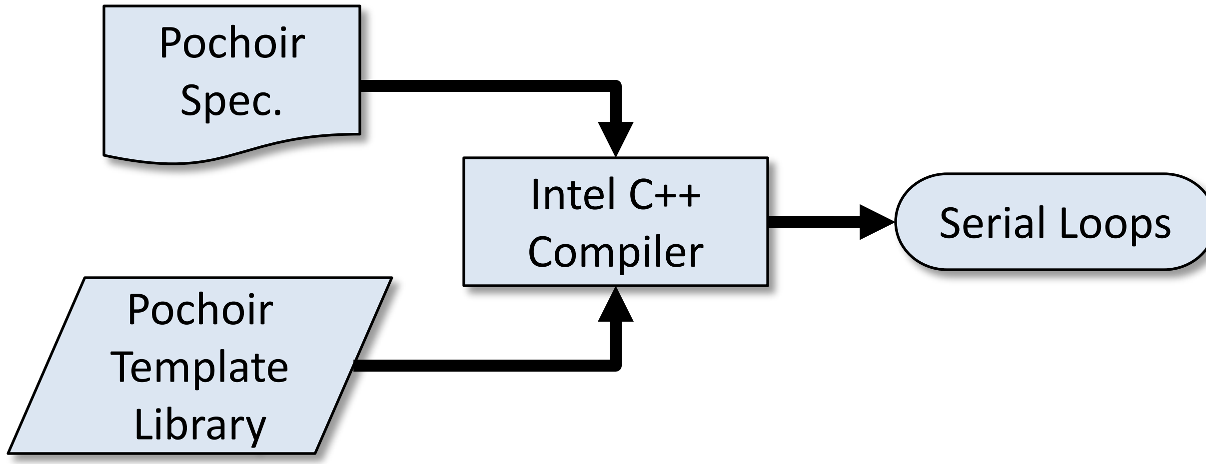
Run a stencil computation on the Pochoir object heat for T time steps using kernel function kern. The Run method can be called multiple times.

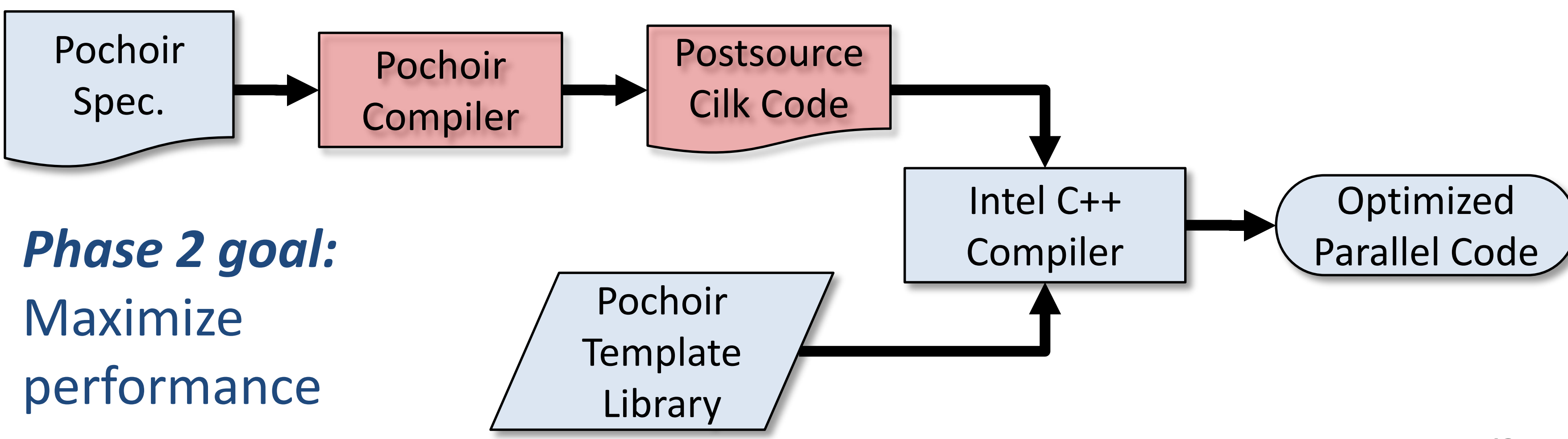
Outline

- Functional Specification
- How the Pochoir System Works
- Algorithms
- Optimizing Strategies
- Conclusion

Two-Phase Compilation Strategy

Phase 1 goal:
Check functional correctness

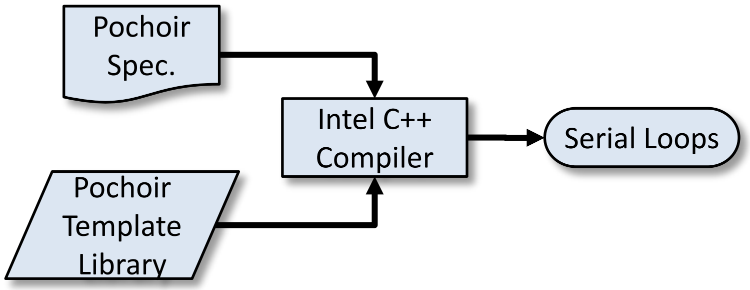




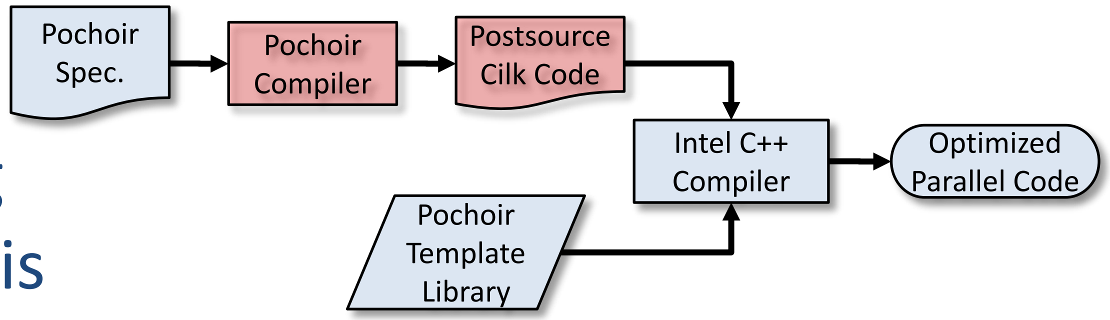
Phase 2 goal:
Maximize performance

Pochoir Guarantee

If a stencil program compiles and runs with the Pochoir template library during Phase 1,



then no errors will occur during Phase 2 when it is compiled with the Pochoir compiler or during the subsequent running of the optimized binary.



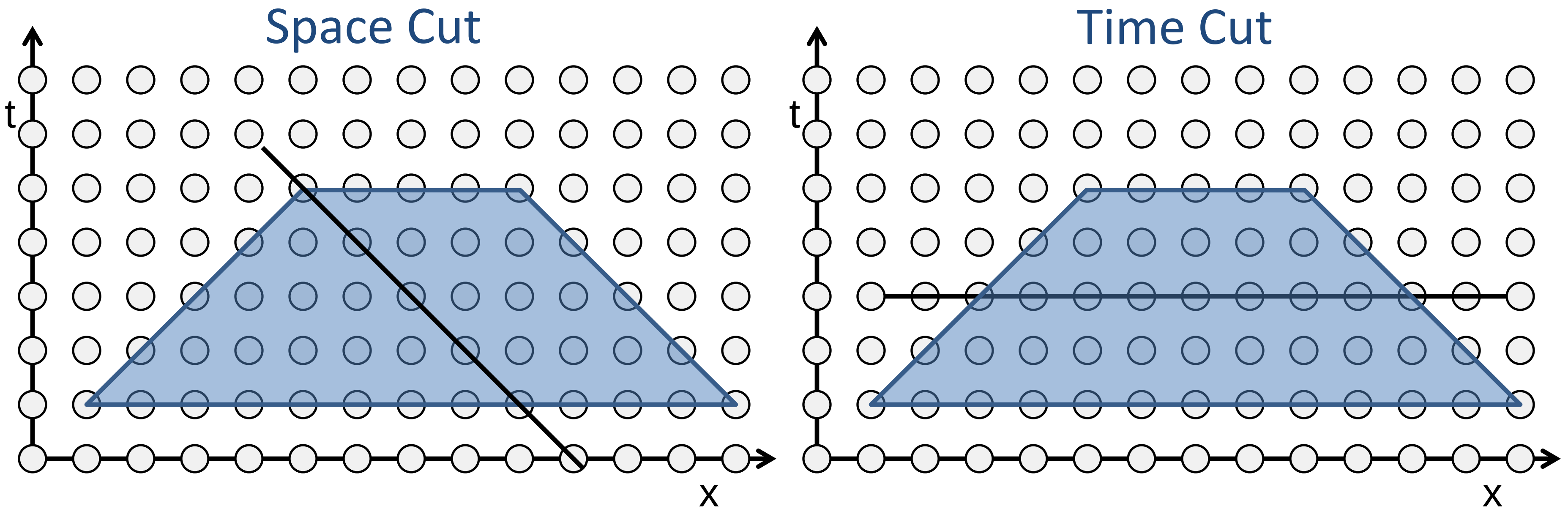
Benefits of the Pochoir Guarantee

- The Pochoir compiler can parse as much of the programmer's C++ code as it is able without worrying about parsing it all.
- If the Pochoir compiler can "understand" the code, which it can in the common case, it can perform strong optimizations.
- If the Pochoir compiler cannot "understand" the code, it can treat the code as correct uninterpreted C++ text, confident that all the syntax- and type-checking was performed during Phase 1.

Outline

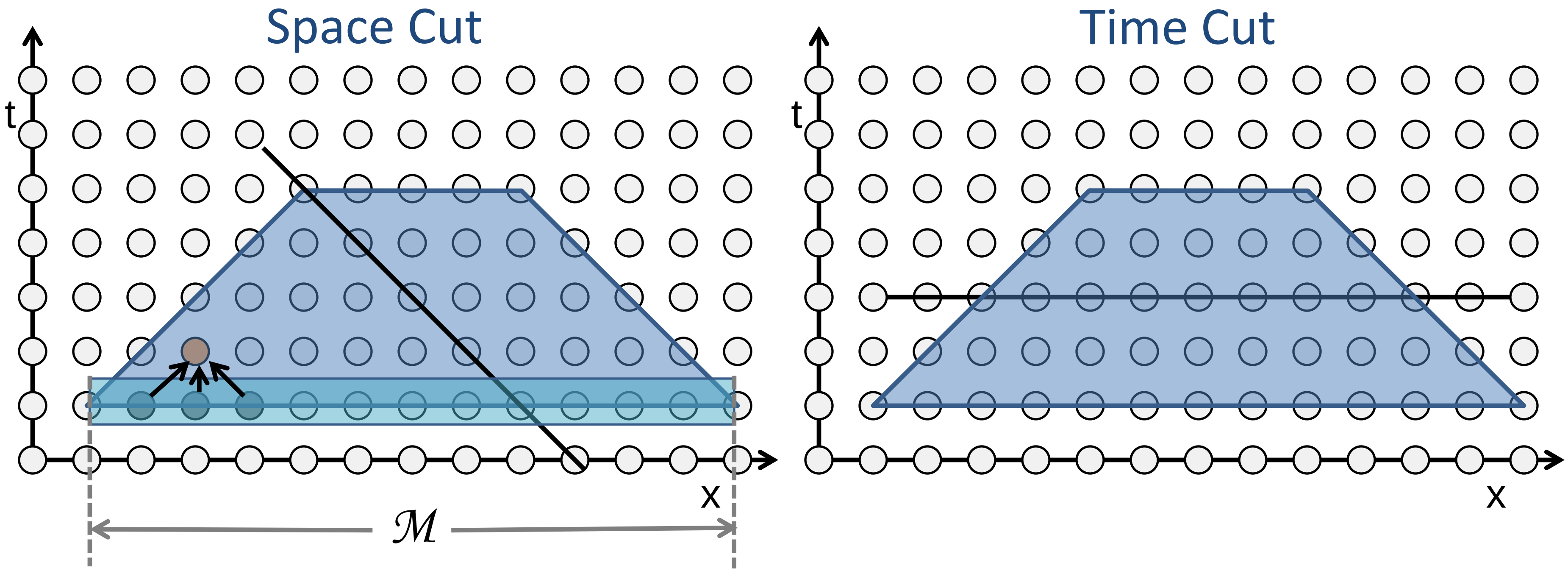
- Functional Specification
- How the Pochoir System Works
- Algorithms
- Optimizing Strategies
- Conclusion

CACHE-OBLIVIOUS STENCILS



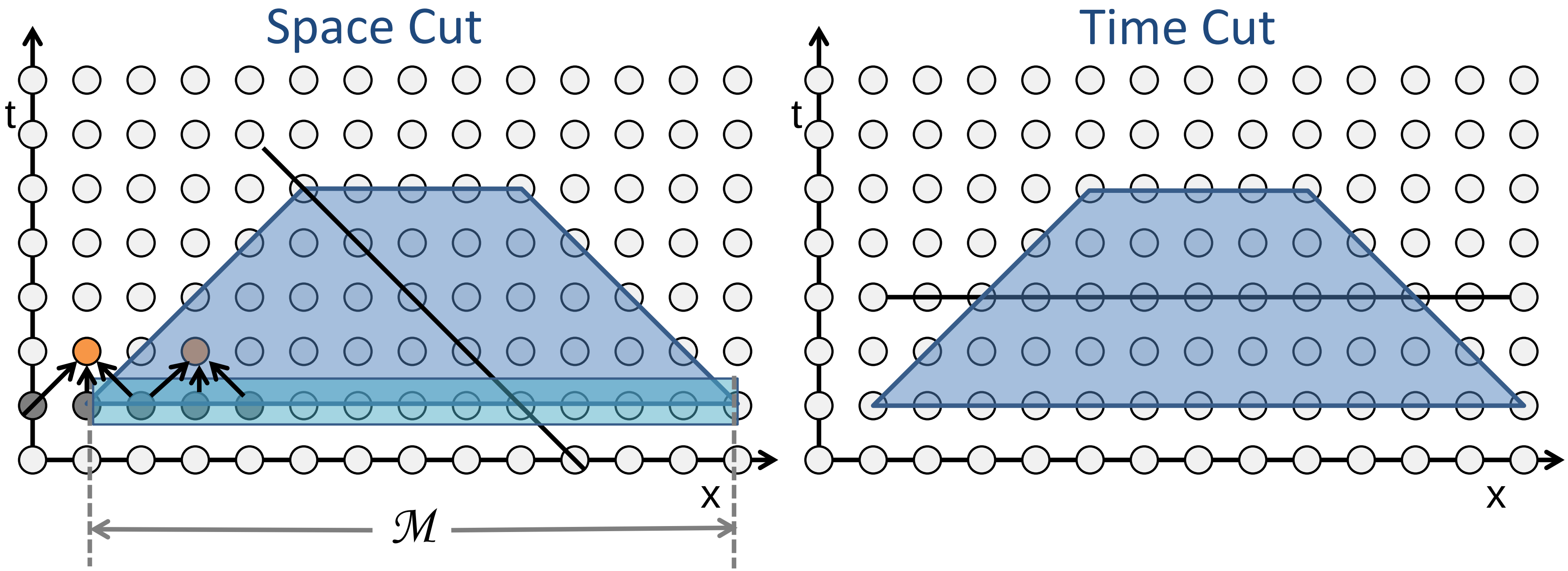
Based on divide-and-conquer ***cache-oblivious***[FLPR99] techniques, ***trapezoidal decompositions***[FS05], are asymptotically efficient, achieving $\Theta(NT/\mathcal{MB})$ cache misses.

Cache-Oblivious Stencils



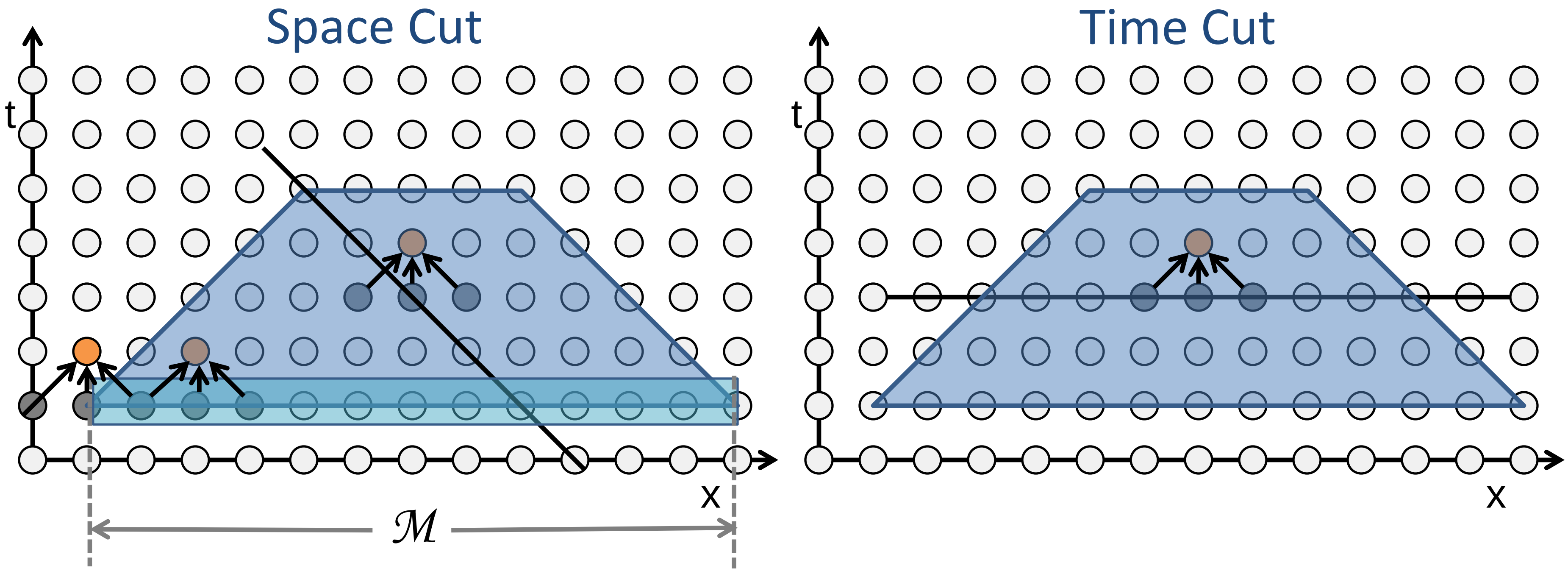
Based on divide-and-conquer ***cache-oblivious***[FLPR99] techniques, ***trapezoidal decompositions***[FS05], are asymptotically efficient, achieving $\Theta(NT/\mathcal{M}\mathcal{B})$ cache misses.

Cache-Oblivious Stencils



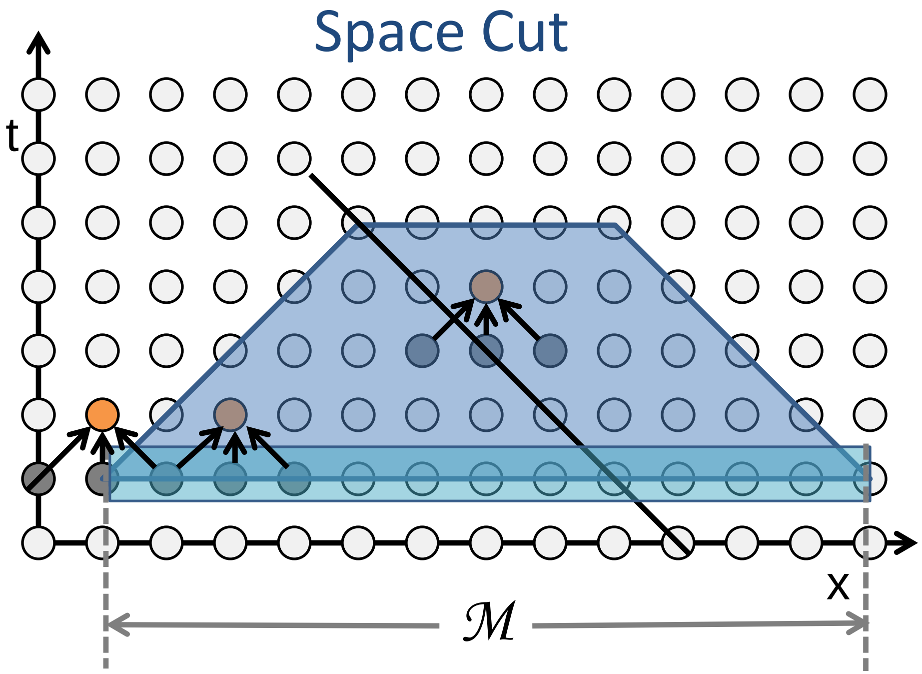
Based on divide-and-conquer ***cache-oblivious***[FLPR99] techniques, ***trapezoidal decompositions***[FS05], are asymptotically efficient, achieving $\Theta(NT/\mathcal{MB})$ cache misses.

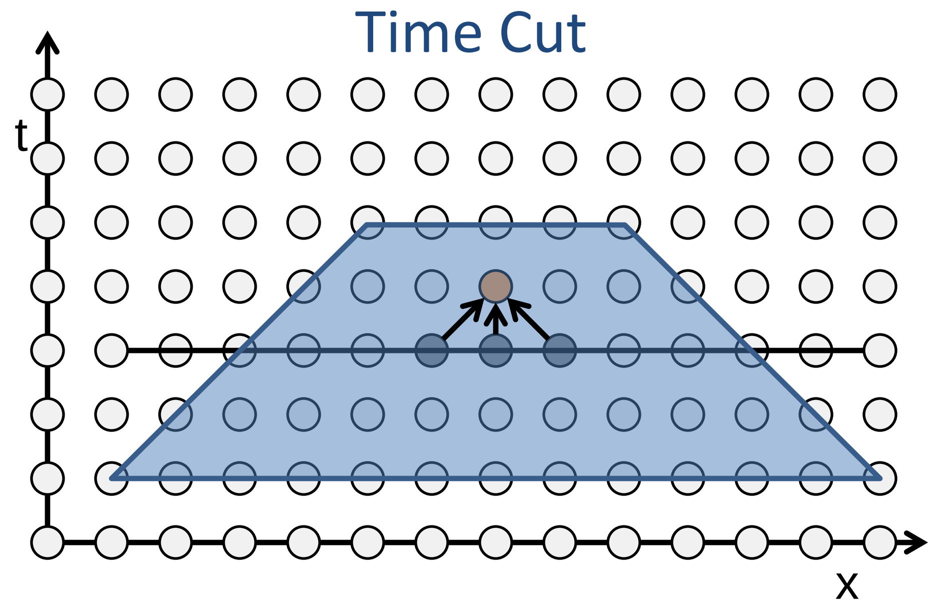
Cache-Oblivious Stencils



Based on divide-and-conquer ***cache-oblivious***[FLPR99] techniques, ***trapezoidal decompositions***[FS05], are asymptotically efficient, achieving $\Theta(NT/\mathcal{MB})$ cache misses.

CACHE-OBLIVIOUS STENCILS

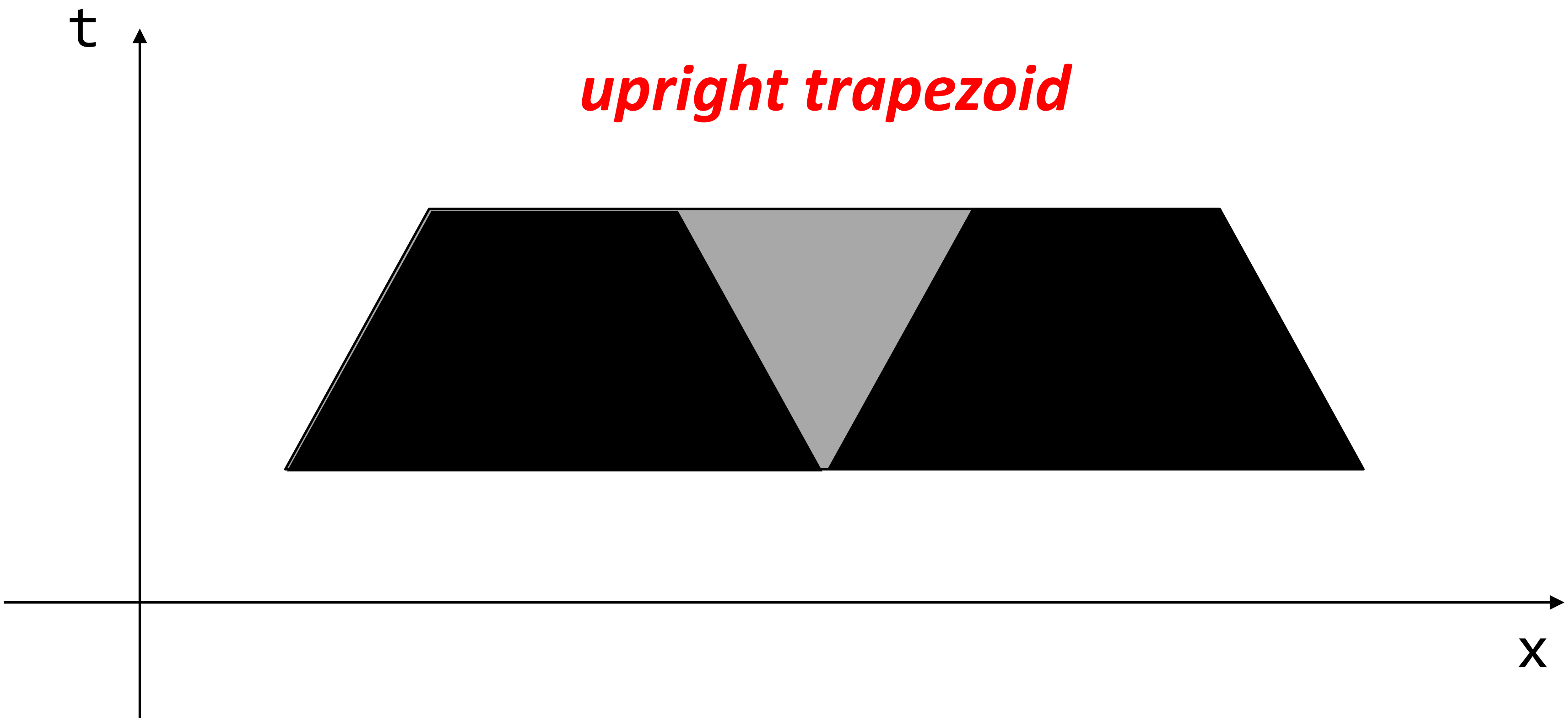




Based on divide-and-conquer ***cache-oblivious***[FLPR99] techniques, ***trapezoidal decompositions***[FS05], are asymptotically efficient, achieving $\Theta(NT/\mathcal{MB})$ cache misses.

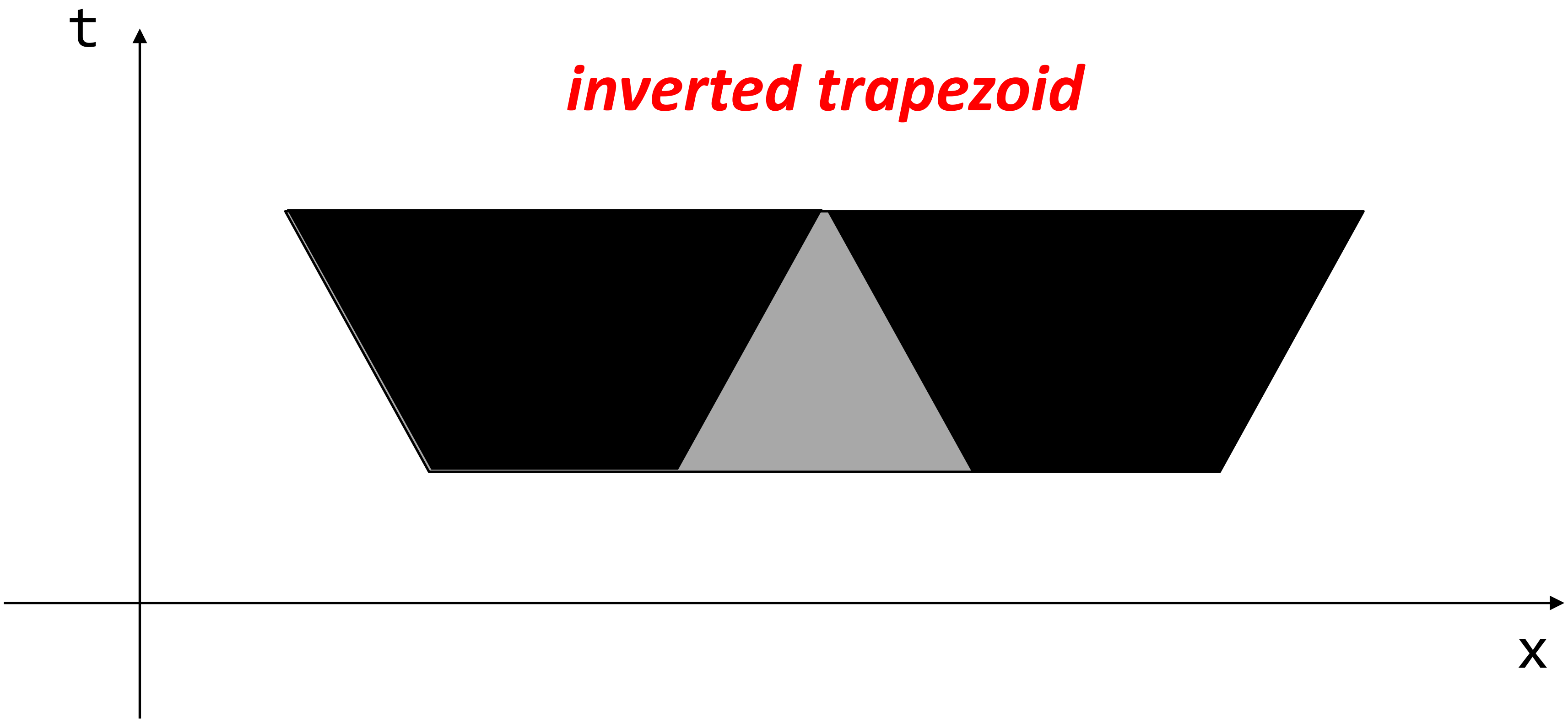
How to parallelize the algorithm?

Parallel Space Cuts



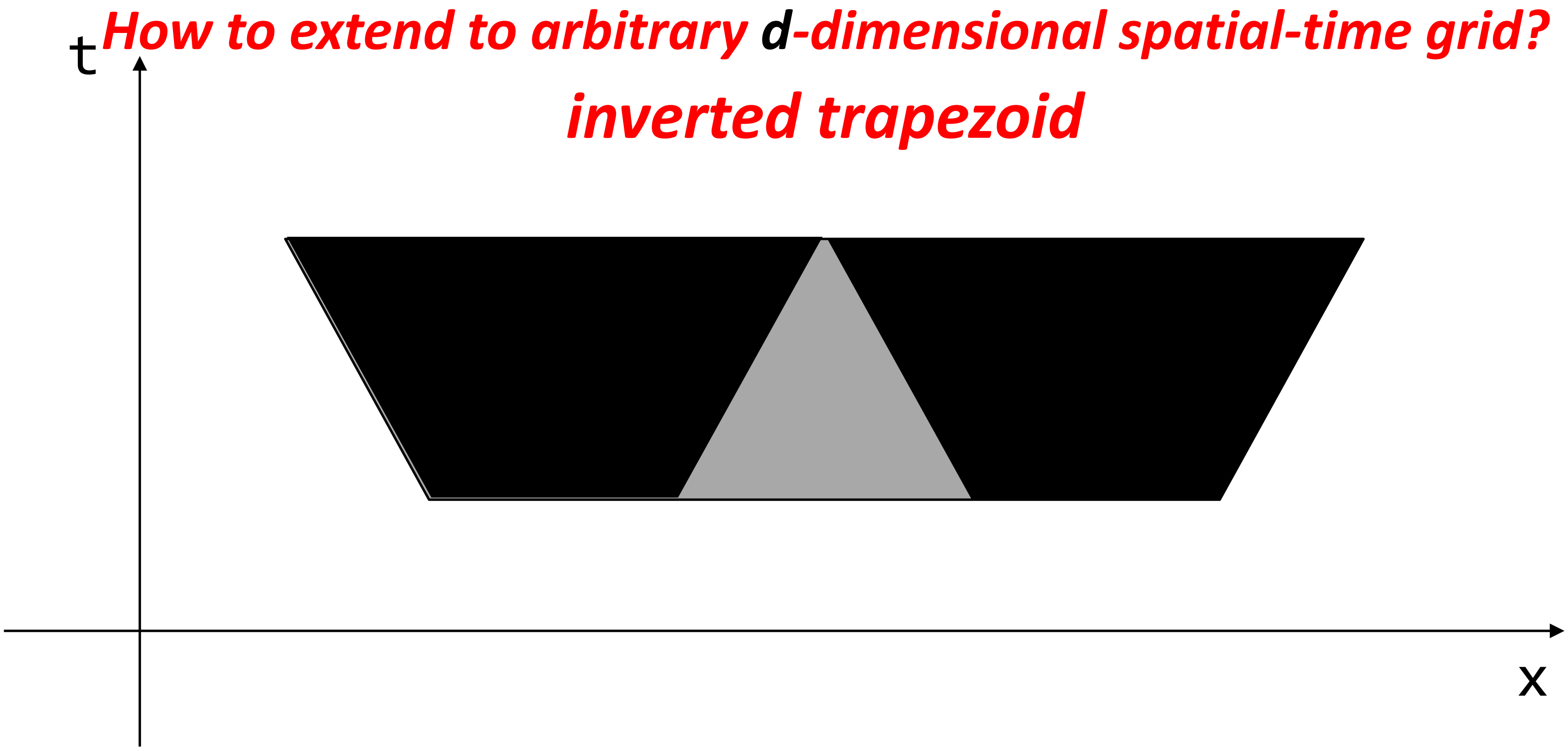
A ***parallel space cut*** [FS07] produces two ***black*** trapezoids that can be executed in parallel and one ***gray*** trapezoid that must execute in series with the black trapezoids.

Parallel Space Cuts



A ***parallel space cut*** [FS07] produces two ***black*** trapezoids that can be executed in parallel and one ***gray*** trapezoid that must execute in series with the black trapezoids.

PARALLEL SPACE CUTS



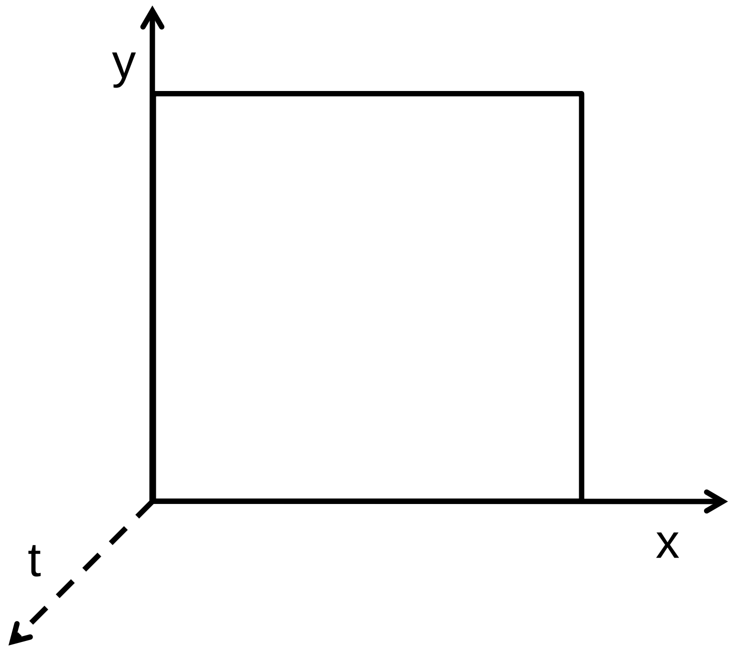
A ***parallel space cut*** [FS07] produces two ***black*** trapezoids that can be executed in parallel and one ***gray*** trapezoid that must execute in series with the black trapezoids.

Serial Space Cutting

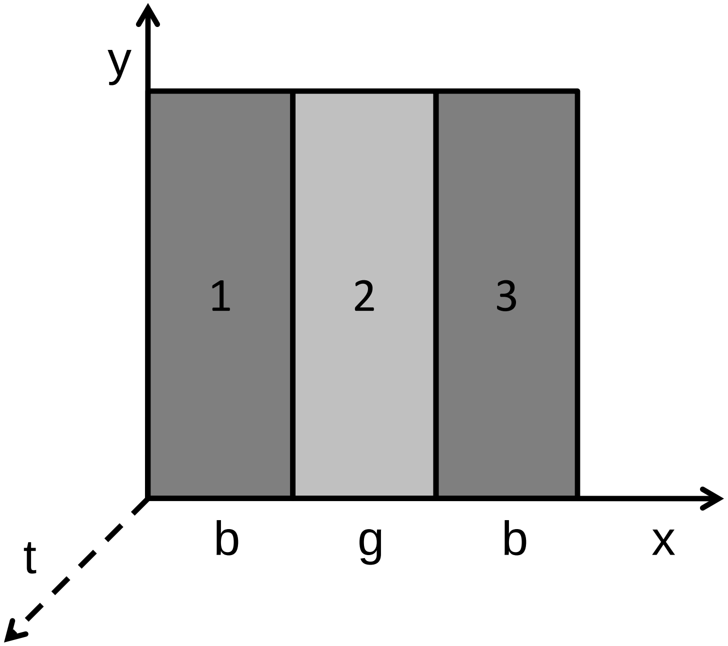
```
void walk(u, t0,t1, x0,x1,dx0,dx1, y0,y1,dy0,dy1, z0,z1,dz0,dz1) {
    int dt = t1 – t0, dx = max((x1-x0), ((x1+dx1*dt)-(x0+dx0*dt)),
        dy = max((y1-y0), ((y1+dy1*dt)-(y0+dy0*dt)),
        dz = max((z1-z0), ((z1+dz1*dt)-(z0+dz0*dt));
    if (dx>=DX_THRESH &&dx>=4*sigma_x*dt) /* cut x dimension */
       if (x1-x0 == dx) { /* cut an upright trapezoid */
          /* spawn black trapezoids */
              cilk_spawn walk(u, t0,t1, x0,x0+dx/2,dx0,–sigma_x,
                                    y0,y1,dy0,dy1, z0,z1,dz0,dz1);
              walk(u, t0,t1, x0+dx/2,x1,sigma_x,dx1,
                   y0,y1,dy0,dy1, z0,z1,dz0,dz1);
              cilk_sync;
              /* spawn gray trapezoid */
              walk(u, t0,t1, x0+dx/2,x0+dx/2,–sigma_x,sigma_x,
                   y0,y1,dy0,dy1, z0,z1,dz0,dz1);
              } else { /* cut an inverted trapezoid */
                   ...
              }
          }
     } else if (.../* cut y dimension */) {
         ...
     } else if (.../* cut z dimension */) {
         ...
     } else if (.../* cut t dimension */) {
         ...
     } else { /* call the base case */
         base_case(u, t0,t1, x0,x1,dx0,dx1, y0,y1,dy0,dy1, z0,z1,dz0,dz1);
     }
}
```

Spatial dimensions are cut one at a time.

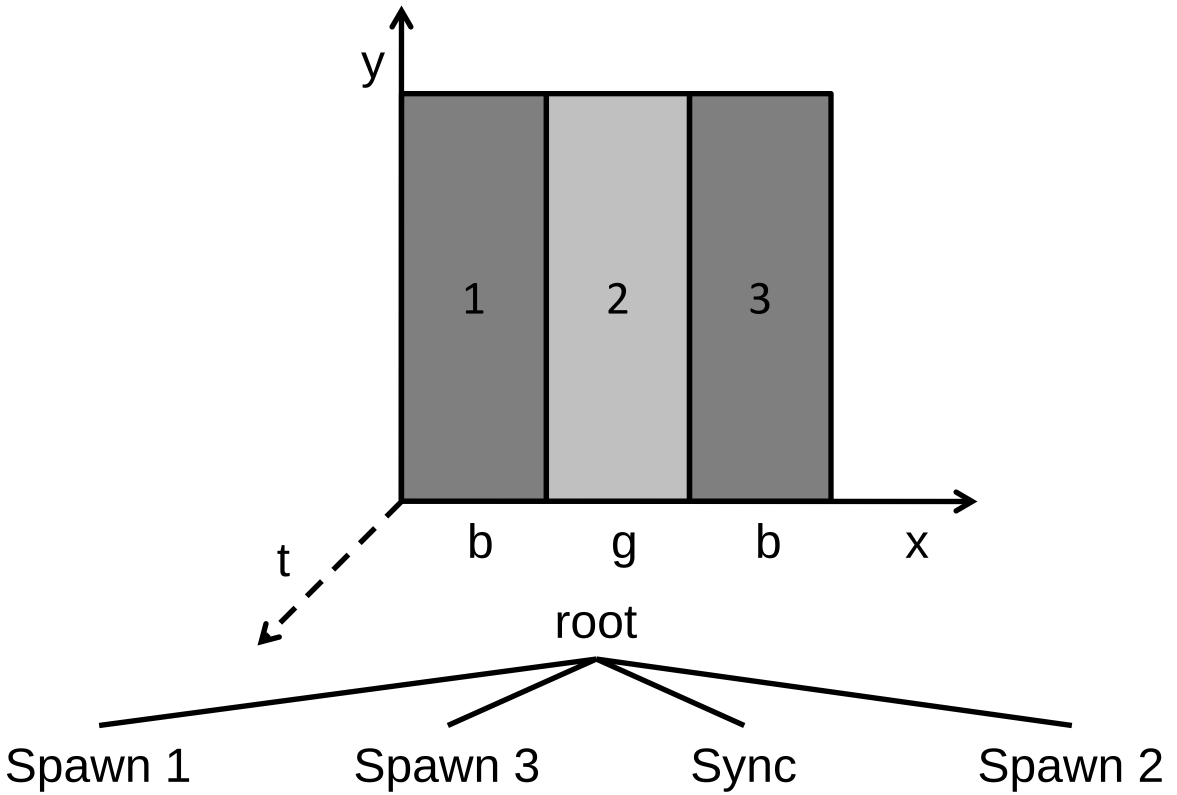
Serial Space Cut — 2D



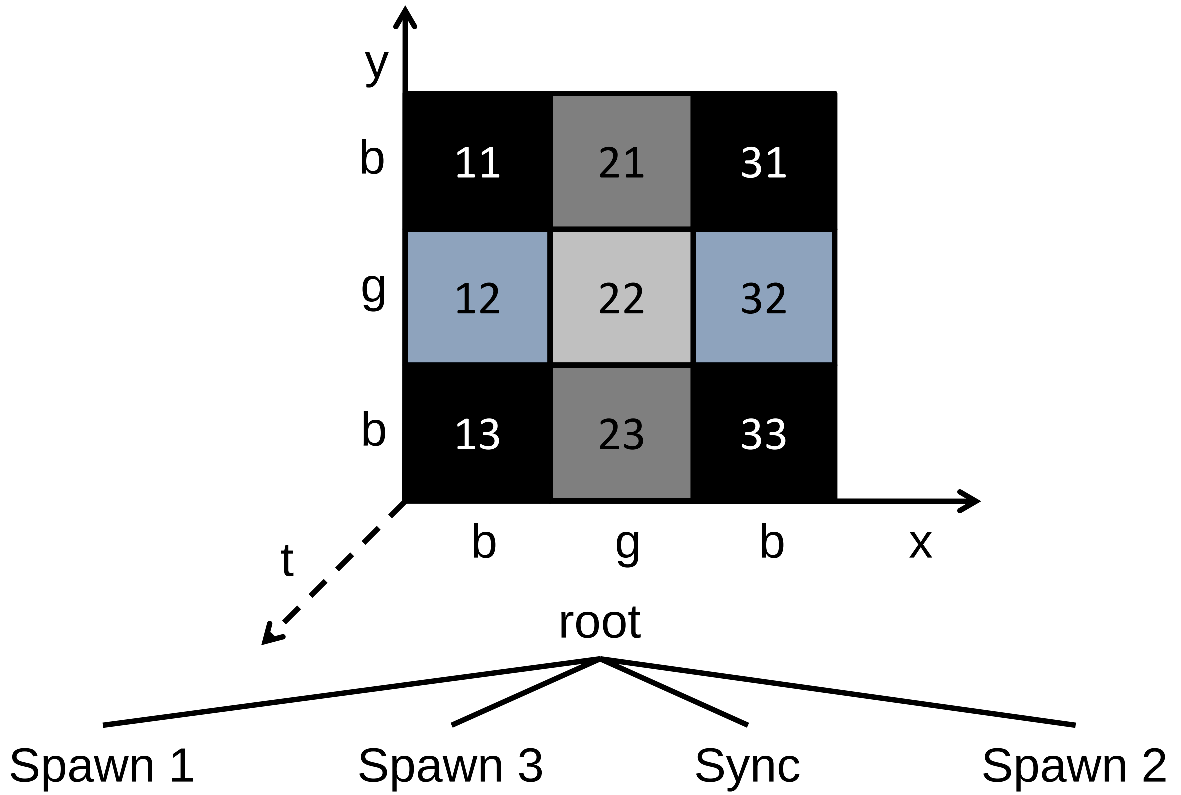
Serial Space Cut — 2D



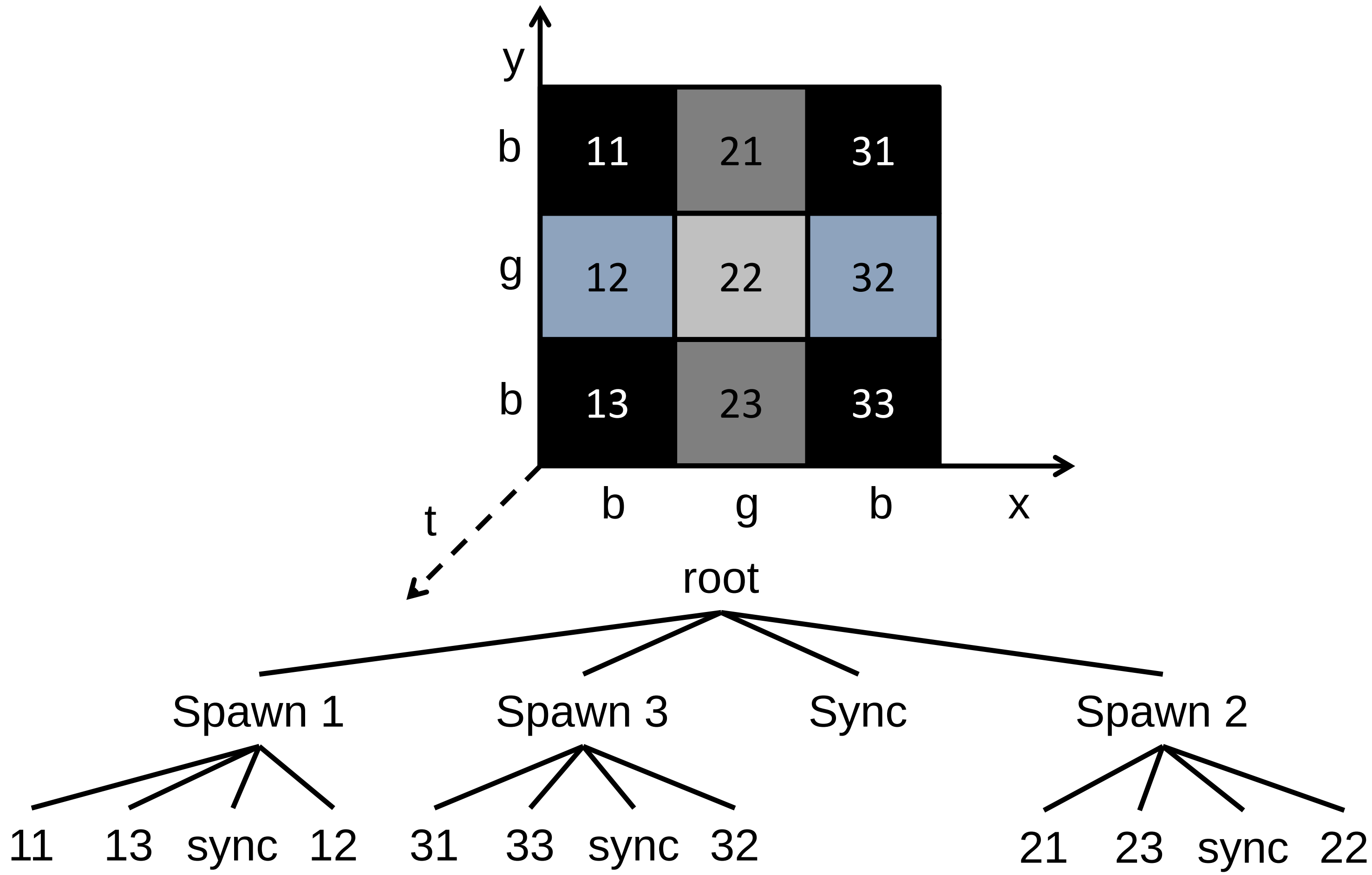
Serial Space Cut — 2D



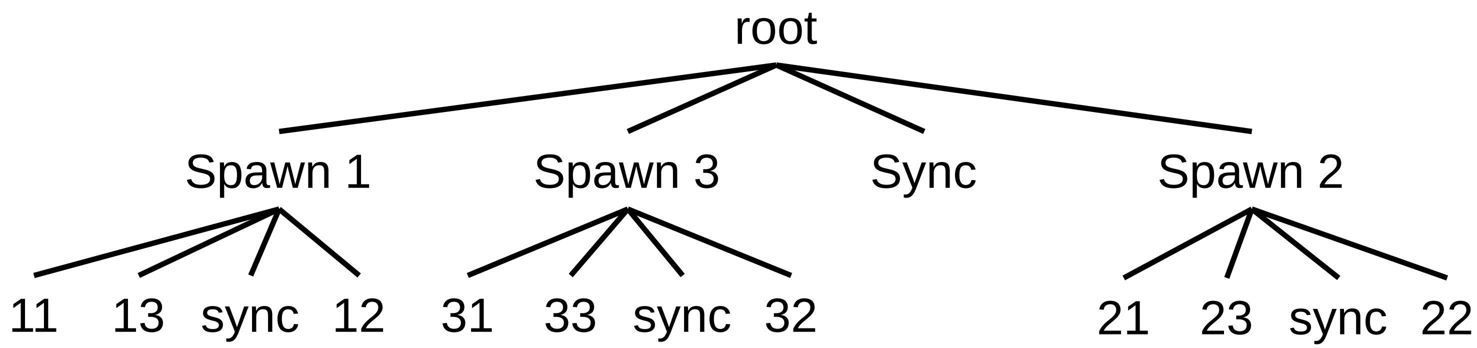
Serial Space Cut — 2D



Serial Space Cut — 2D

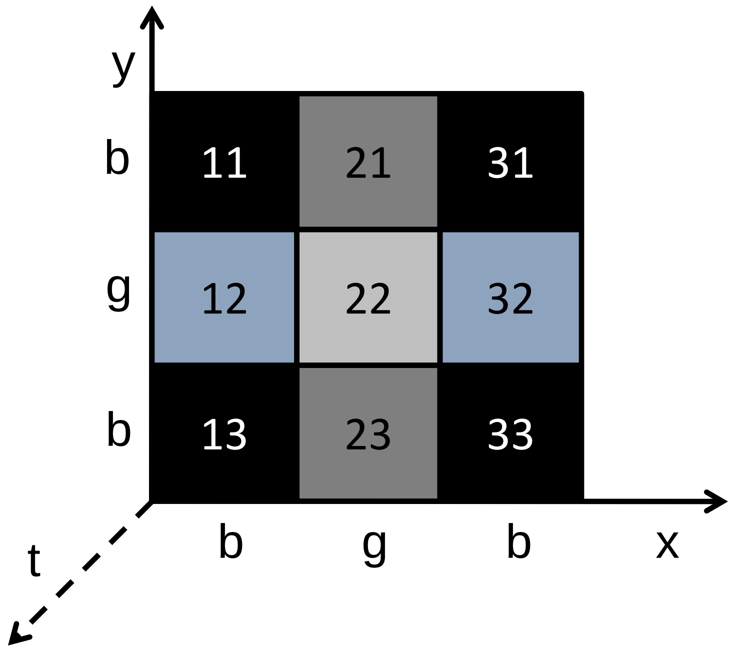


Serial Space Cut — 2D



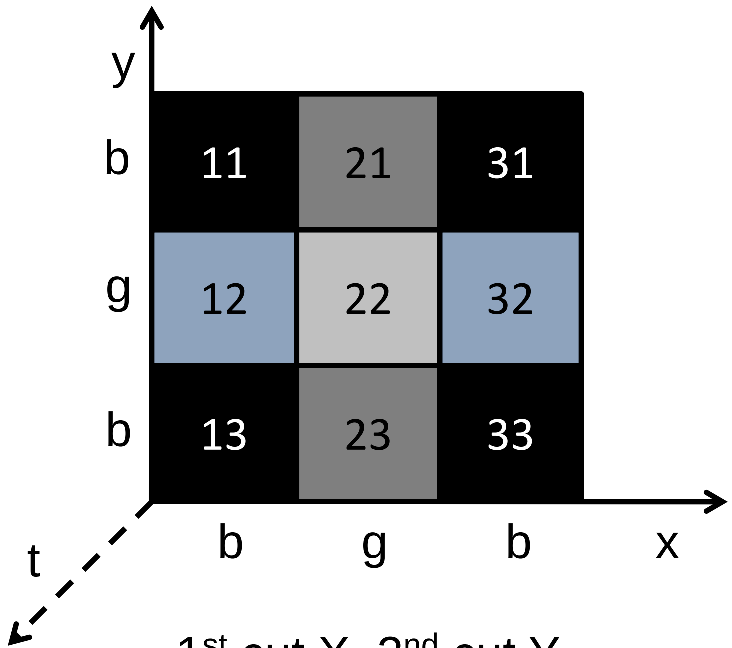
Spawn 11, 13, 31, 33
Sync
Spawn 12, 32
Sync
Spawn 21, 23
Sync
Spawn 22

Serial Space Cut — 2D



Spawn 11, 13, 31, 33
Sync
Spawn 12, 32
Sync
Spawn 21, 23
Sync
Spawn 22

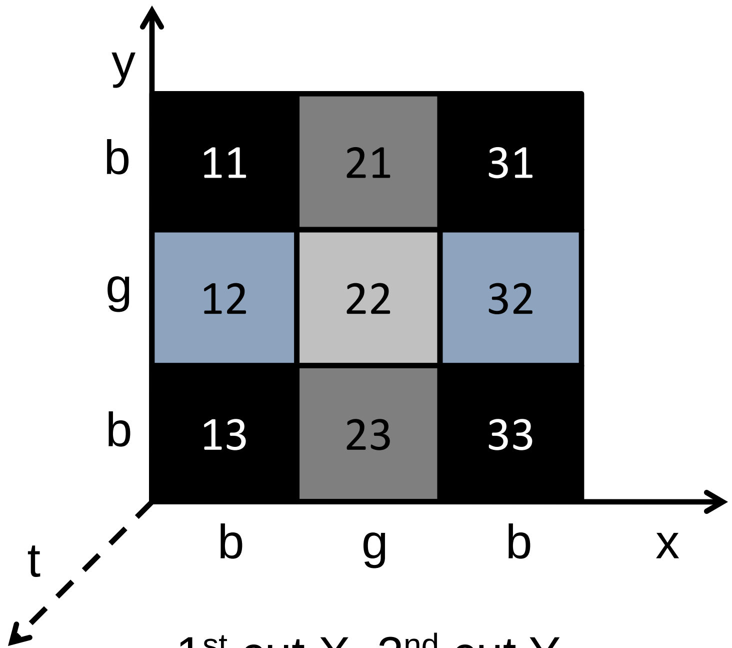
Serial Space Cut — 2D



1^{st} cut X, 2^{nd} cut Y

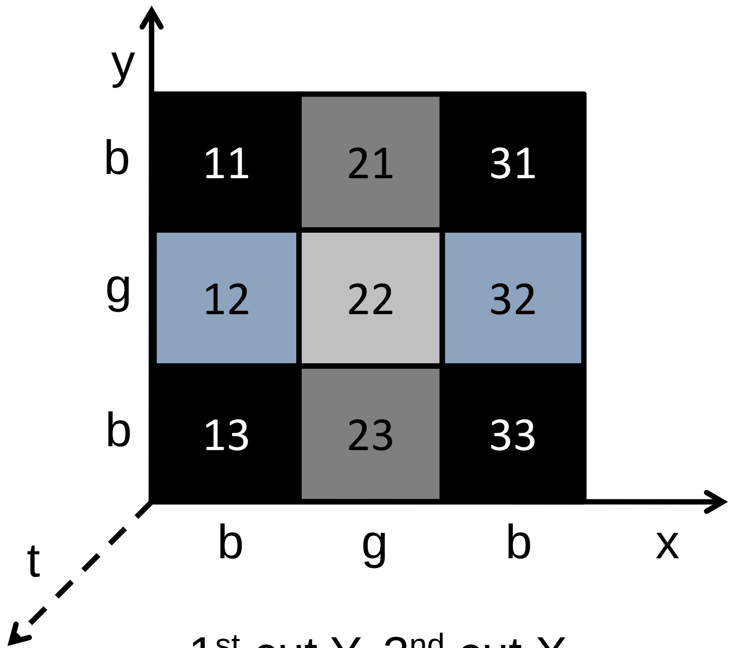
Spawn 11, 13, 31, 33
Sync
Spawn 12, 32
Sync
Spawn 21, 23
Sync
Spawn 22

Serial Space Cut — 2D



1st cut X, 2nd cut Y

Spawn 11, 13, 31, 33
Sync
Spawn 12, 32
Sync
Spawn 21, 23
Sync
Spawn 22

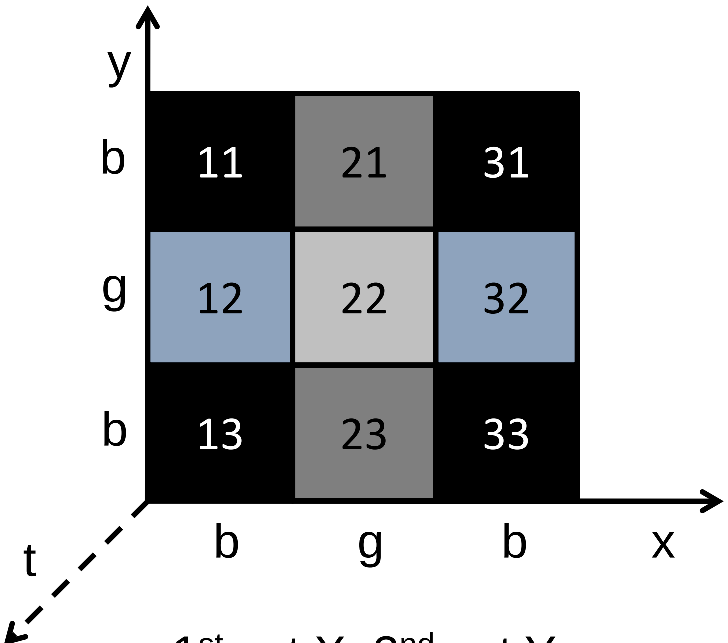


1st cut Y, 2nd cut X

Spawn 11, 13, 31, 33
Sync
Spawn 21, 23
Sync
Spawn 12, 32
Sync
Spawn 22

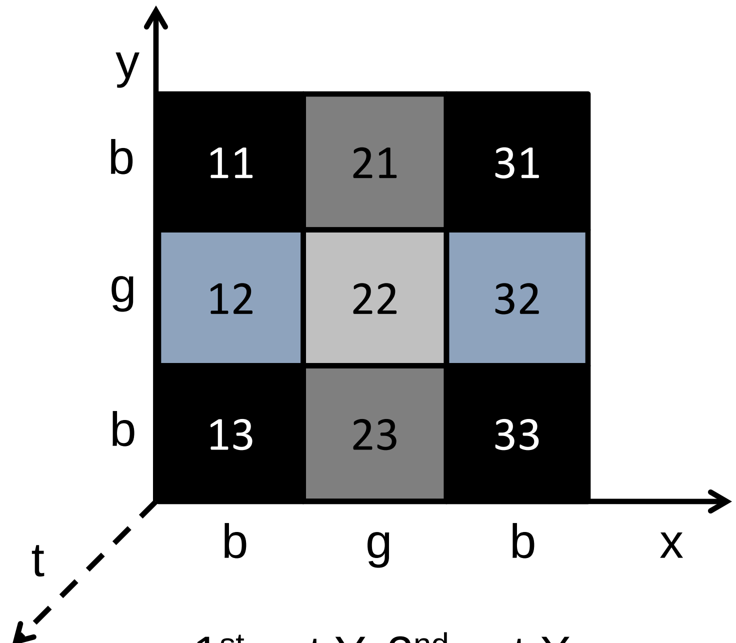
pochoir@csail.mit.edu

Serial Space Cut — 2D



1st cut X, 2nd cut Y

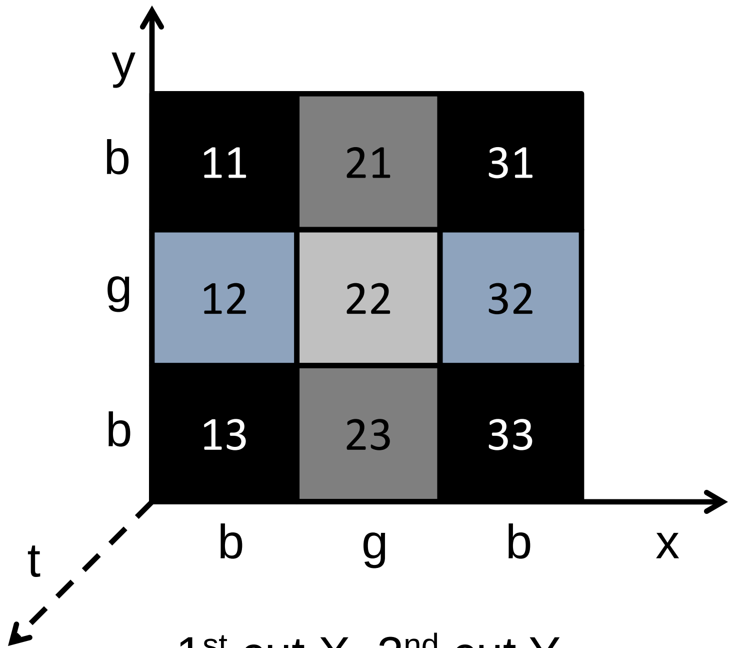
Spawn 11, 13, 31, 33
Sync
Spawn 12, 32
Sync
Spawn 21, 23
Sync
Spawn 22



1st cut Y, 2nd cut X

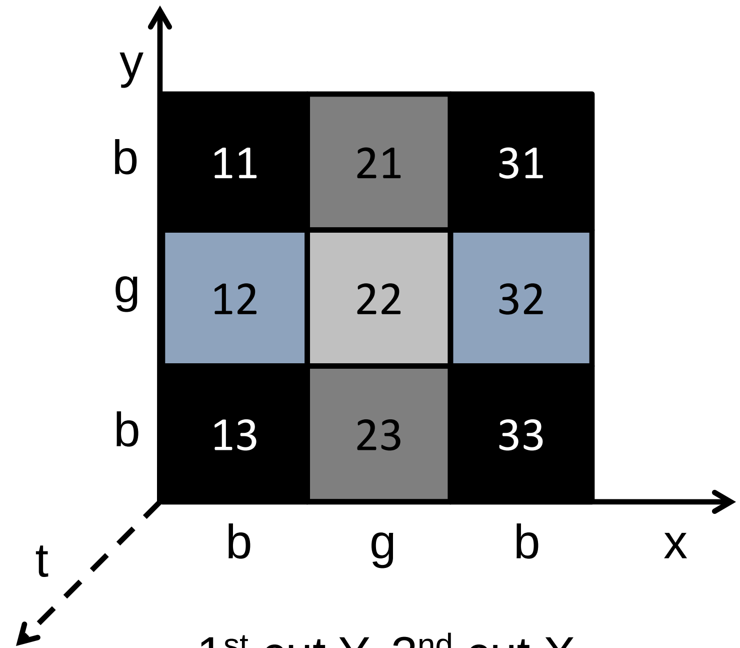
Spawn 11, 13, 31, 33
Sync
Spawn 21, 23
Sync
Spawn 12, 32
Sync
Spawn 22

Serial Space Cut — 2D



1st cut X, 2nd cut Y

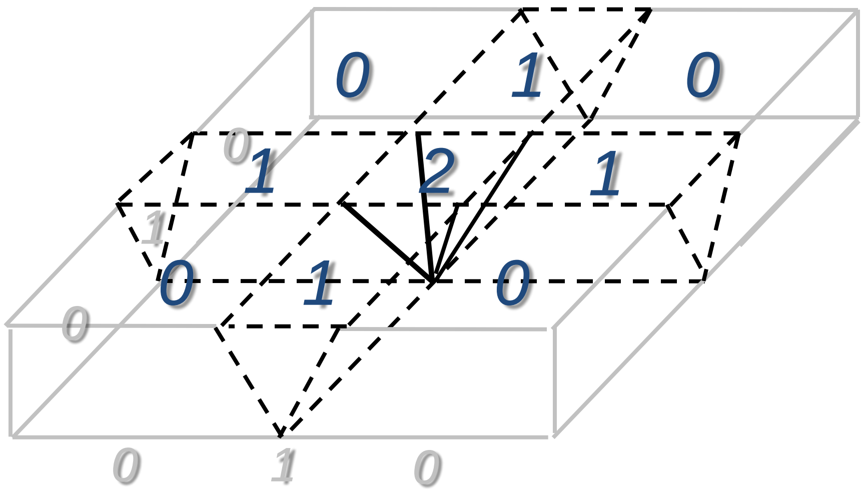
Spawn 11, 13, 31, 33
Sync
Spawn 12, 32
Sync
Spawn 21, 23
Sync
Spawn 22



1st cut Y, 2nd cut X

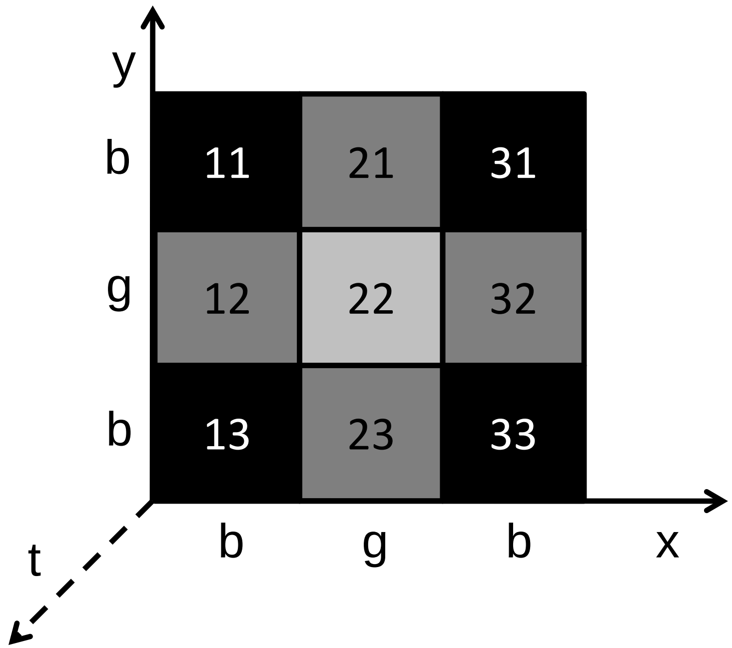
Spawn 11, 13, 31, 33
Sync
Spawn 21, 23
~~Sync~~
Spawn 12, 32
Sync
Spawn 22

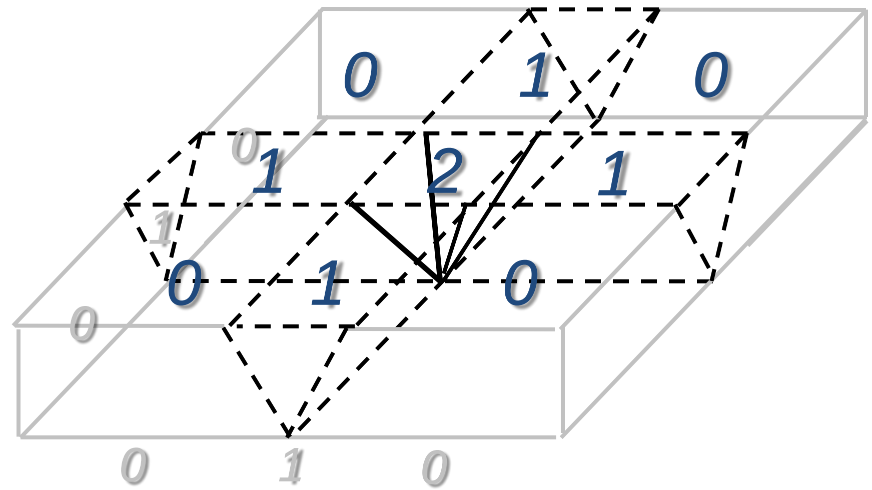
Hyper Space Cut

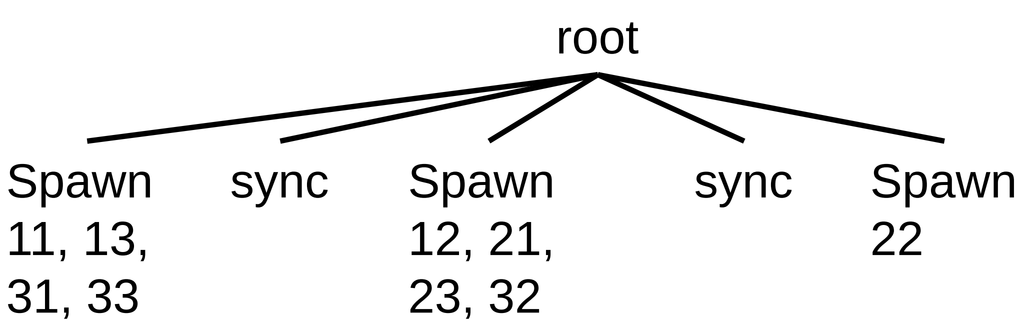


- ***Space Cut***: Evaluate and assign dependency levels to as many spatial dimensions as possible. Spawn and sync subtrapezoids according to dependency levels.
- ***Time Cut***: The same as Sequential Space Cut
- ***Base Case***: The same as Sequential Space Cut

Hyper Space Cut

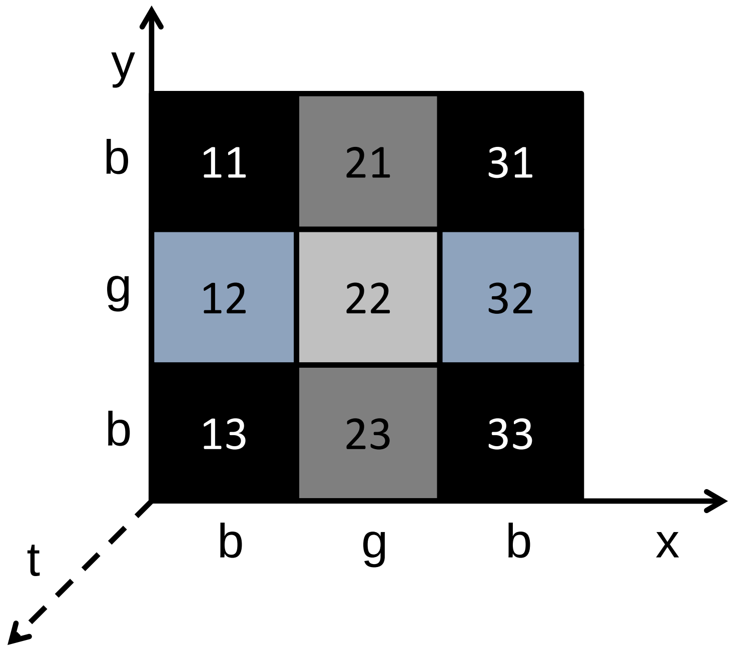


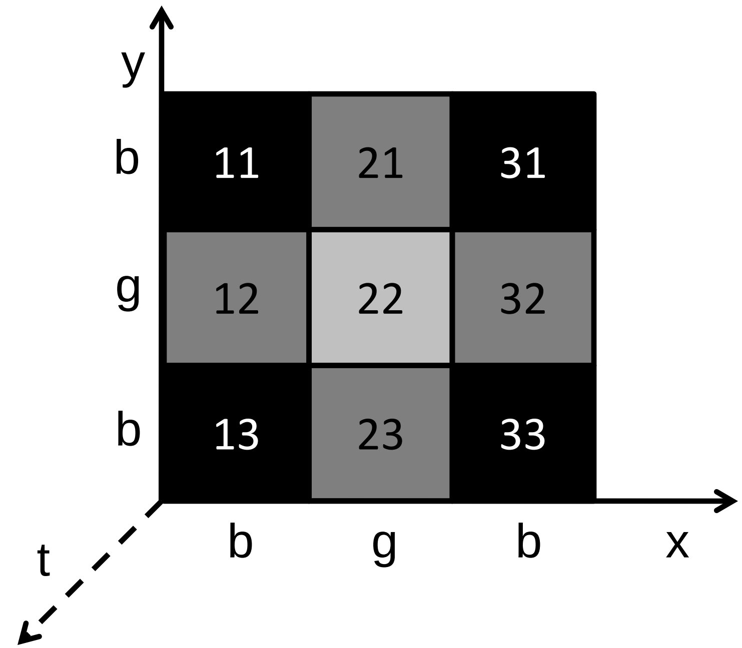




Spawn 11, 13, 31, 33
Sync
Spawn 21, 23, 12, 32
Sync
Spawn 22

Serial space cut vs Hyper space cut





1st cut X, 2nd cut Y

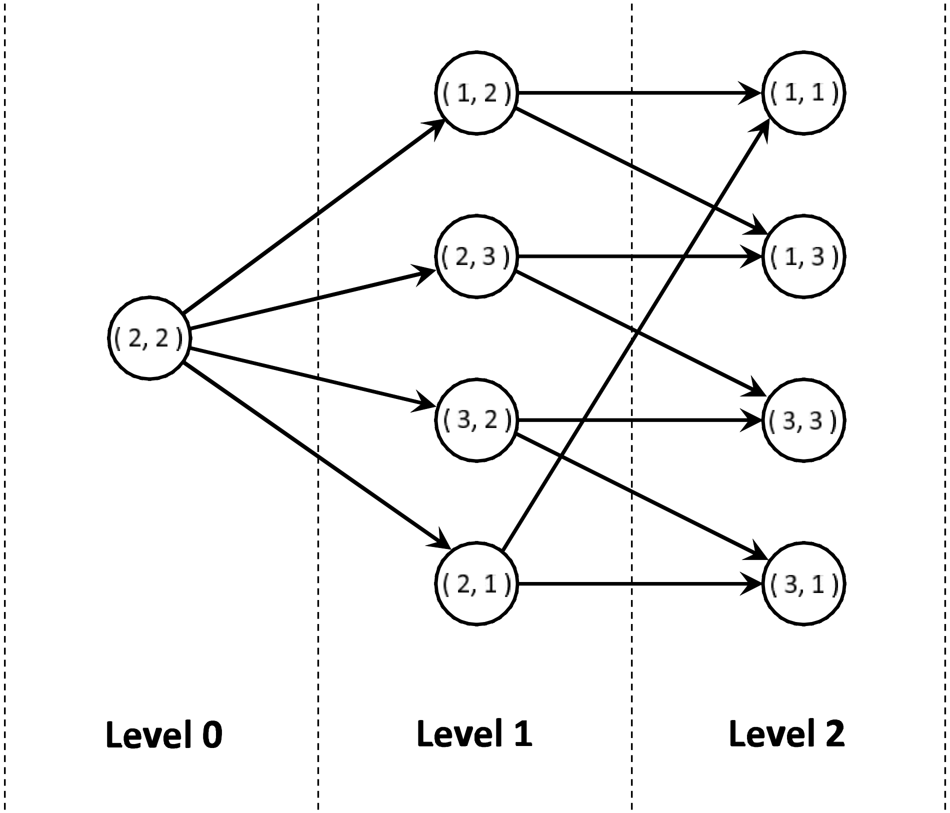
Spawn 's
Sync
Spawn 's
Sync
Spawn 's
Sync
Spawn 's

1st cut Y, 2nd cut X

Spawn 's
Sync
Spawn 's
Sync
Spawn 's
Sync
Spawn 's

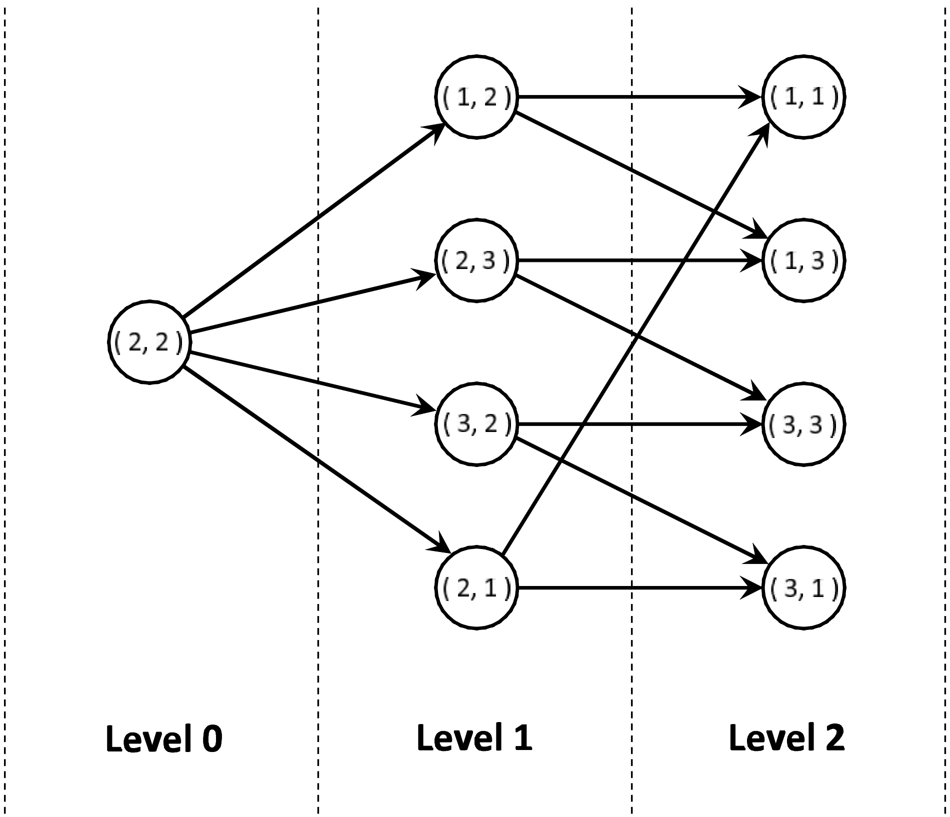
Spawn 's
Sync
Spawn 's
Sync
Spawn 's

Hyper Space Cut



Lemma 3: All (2r+1)k subtrapezoids created by a hyper space cut on $k \geq 1$ of the $d \geq k$ spatial dimensions of a (d+1)-dimensional trapezoid can be evaluated in k+1 parallel steps

Hyper Space Cut



Lemma 3: All (2r+1)k subtrapezoids created by a hyper space cut on $k \geq 1$ of the $d \geq k$ spatial dimensions of a (d+1)-dimensional trapezoid can be evaluated in k+1 parallel steps

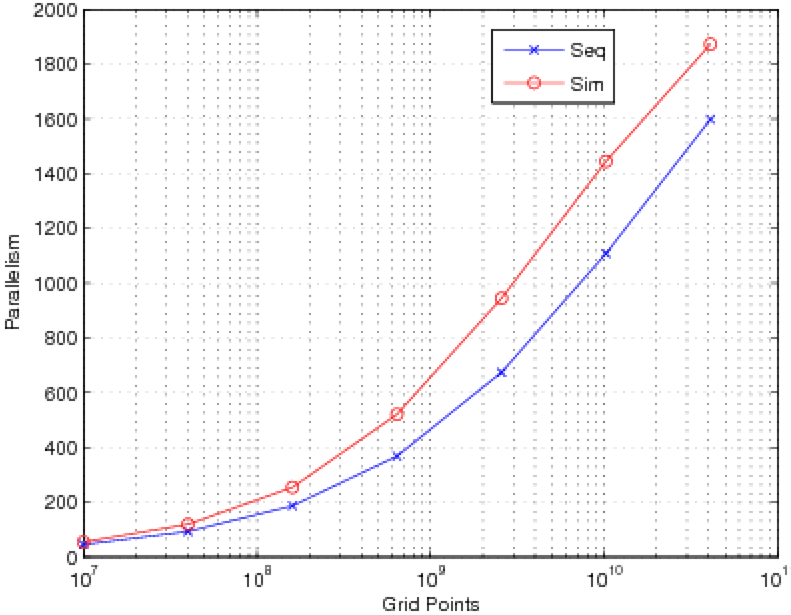
Lemma 2: All (2r+1)k subtrapezoids created by a serial space cut on $k \geq 1$ of the $d \geq k$ spatial dimensions of a (d+1)-dimensional trapezoid can be evaluated in 2^k parallel steps.

pochoir@csail.mit.edu

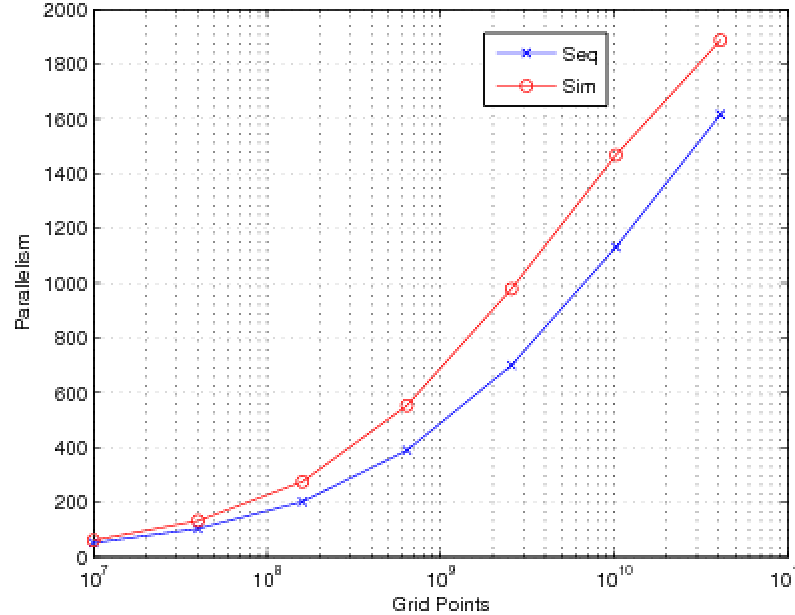
Superiority of Hyper space Cuts

Theorem. On a d-dimensional spatial grid with all spatial dimensions roughly equal to the time dimension h, Pochoir's hyperspace-cutting algorithm achieves $\Theta(h^{d+1-\lg(d+2)}/d)$ parallelism, while Frigo and Strumpen's original serial space-cutting algorithm achieves only $\Theta(h^{d+1-\lg(2^d+1)}) = O(h)$ parallelism.

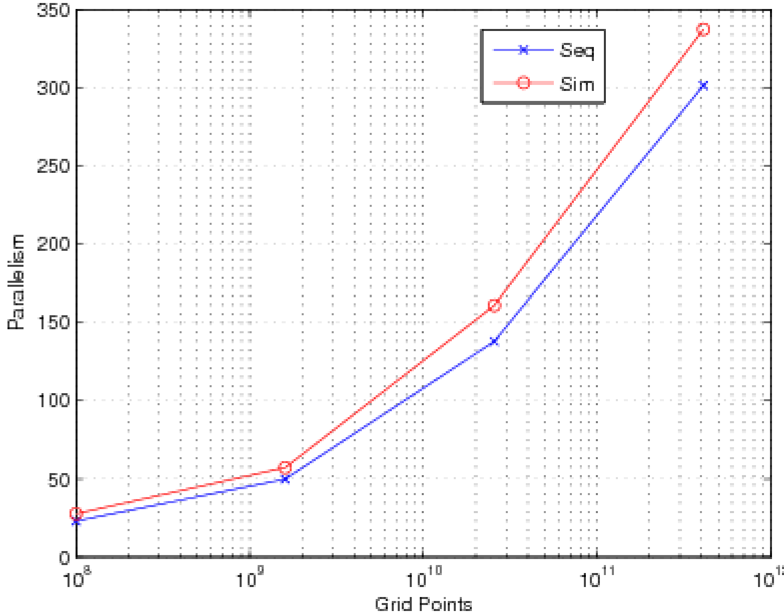
Both algorithms have the same asymptotic cache complexity.



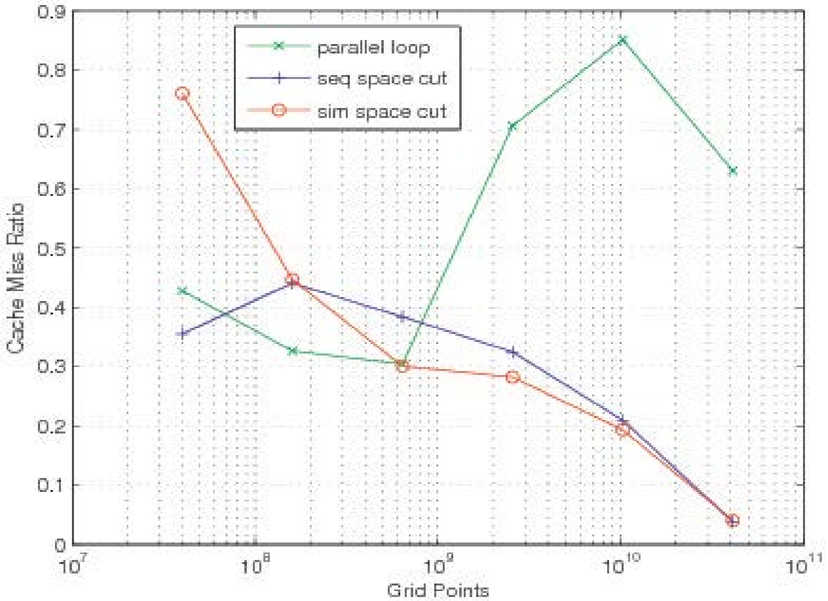
Parallelism of heat_2D_P



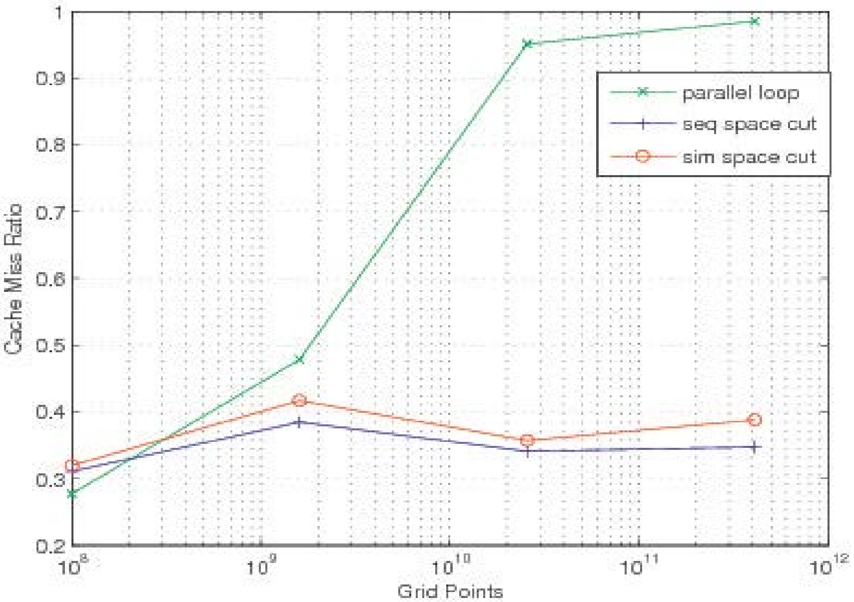
Parallelism of heat_2D_NP



Parallelism of 3dfd



Cache Miss Ratio of heat_2D_NP



Cache Miss Ratio of 3dfd

Outline

- Functional Specification
- How the Pochoir System Works
- Algorithms
- Optimizing Strategies
- Results
- Conclusion

Optimizations

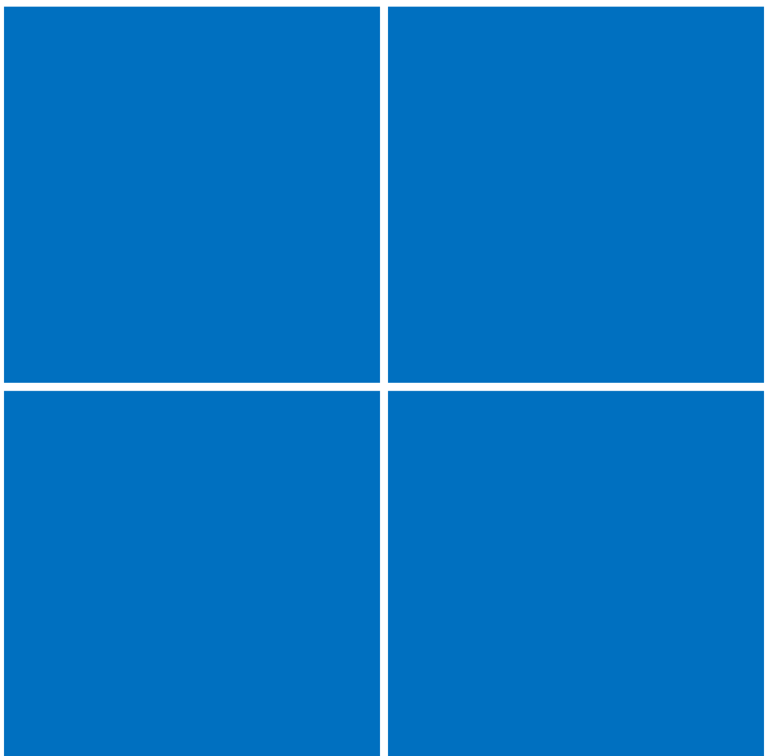
- Two code clones
- Unifying the handling of periodic and nonperiodic boundary conditions
- Automatic selection of traversal strategy
  - `-split-macro-shadow`
  - `-split-opt-pointer`
- Coarsening of base cases

Two Code Clones

- The ***slow clone*** handles regions that contain boundaries and checks for out-of-range grid points.
- The ***fast clone*** handles the larger interior regions which require no range checking.

Two Code Clones

- The ***slow clone*** handles regions that contain boundaries and checks for out-of-range grid points.
- The ***fast clone*** handles the larger interior regions which require no range checking.

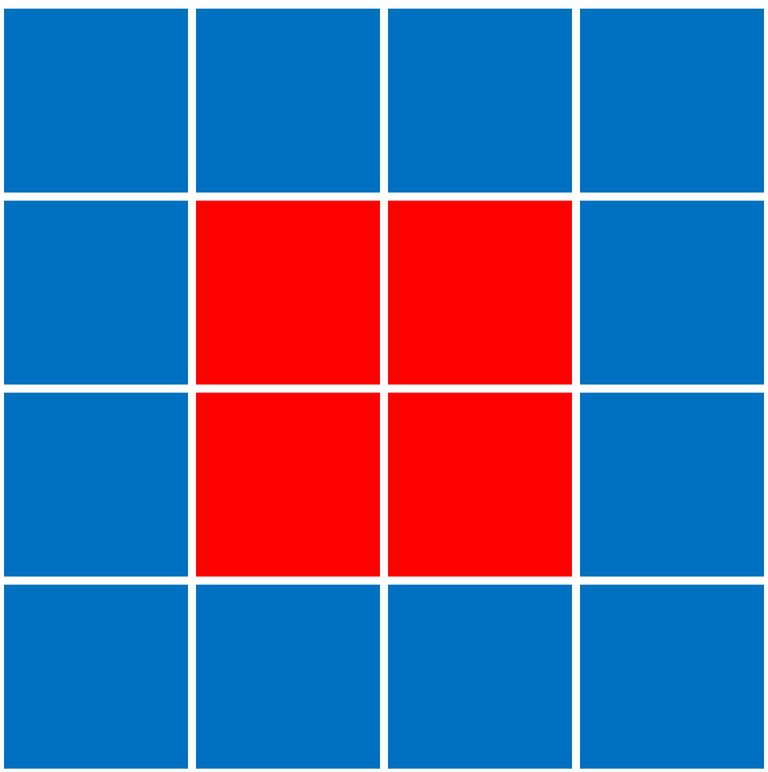


During the recursive algorithm*, the fast clone is used whenever possible.

*The actual algorithm decomposes the grid into trapezoids, not rectangles.

Two Code Clones

- The ***slow clone*** handles regions that contain boundaries and checks for out-of-range grid points.
- The ***fast clone*** handles the larger interior regions which require no range checking.

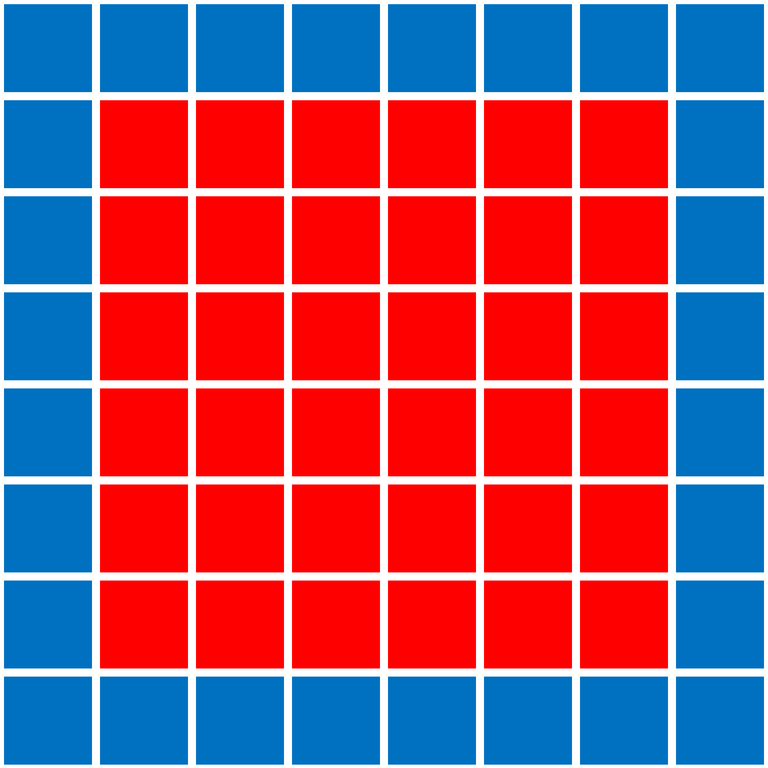


During the recursive algorithm*, the fast clone is used whenever possible. Once the fast clone is used for a region, the fast clone is always used for its subregions.

*The actual algorithm decomposes the grid into trapezoids, not rectangles.

Two Code Clones

- The ***slow clone*** handles regions that contain boundaries and checks for out-of-range grid points.
- The ***fast clone*** handles the larger interior regions which require no range checking.

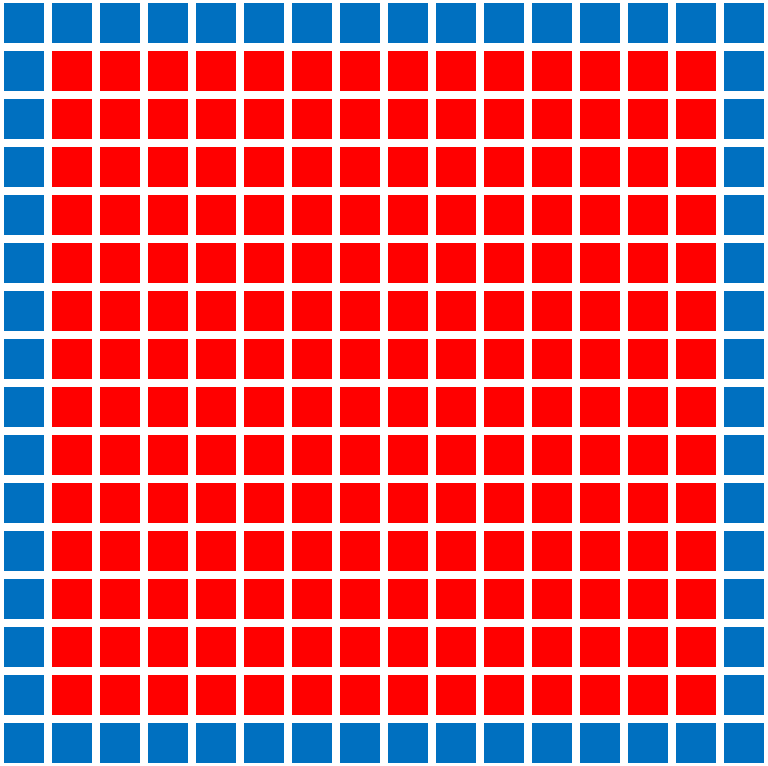


During the recursive algorithm*, the fast clone is used whenever possible. Once the fast clone is used for a region, the fast clone is always used for its subregions.

*The actual algorithm decomposes the grid into trapezoids, not rectangles.

Two Code Clones

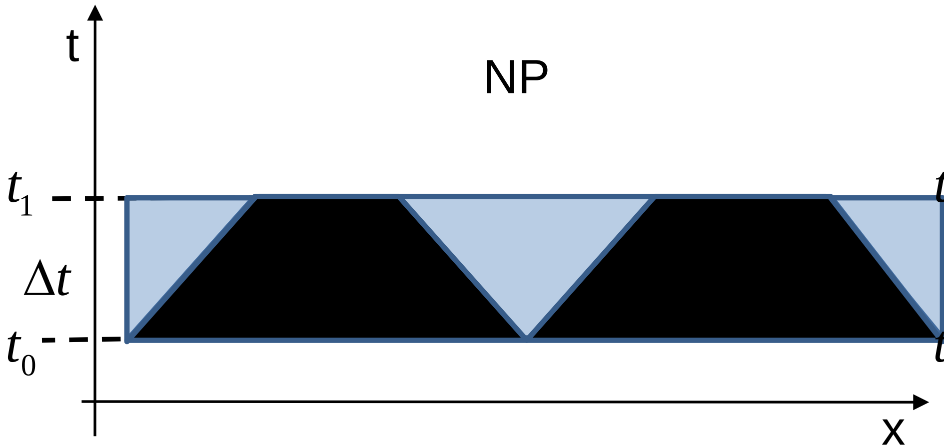
- The ***slow clone*** handles regions that contain boundaries and checks for out-of-range grid points.
- The ***fast clone*** handles the larger interior regions which require no range checking.

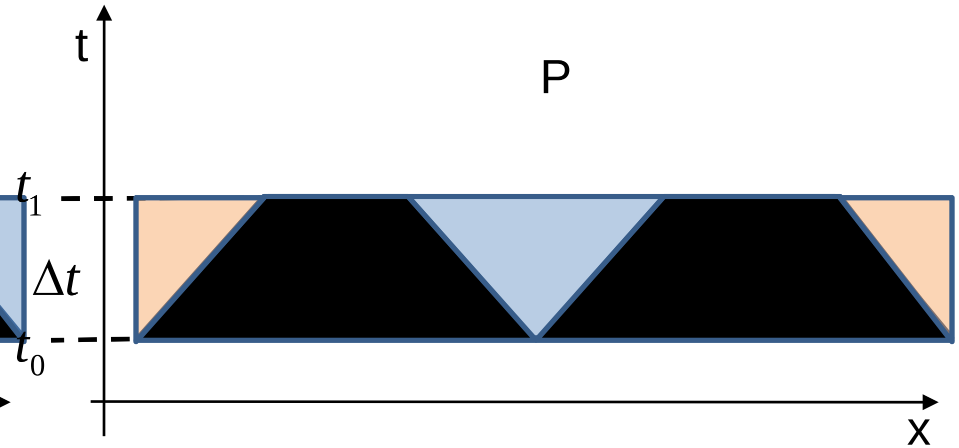


During the recursive algorithm*, the fast clone is used whenever possible. Once the fast clone is used for a region, the fast clone is always used for its subregions.

*The actual algorithm decomposes the grid into trapezoids, not rectangles.

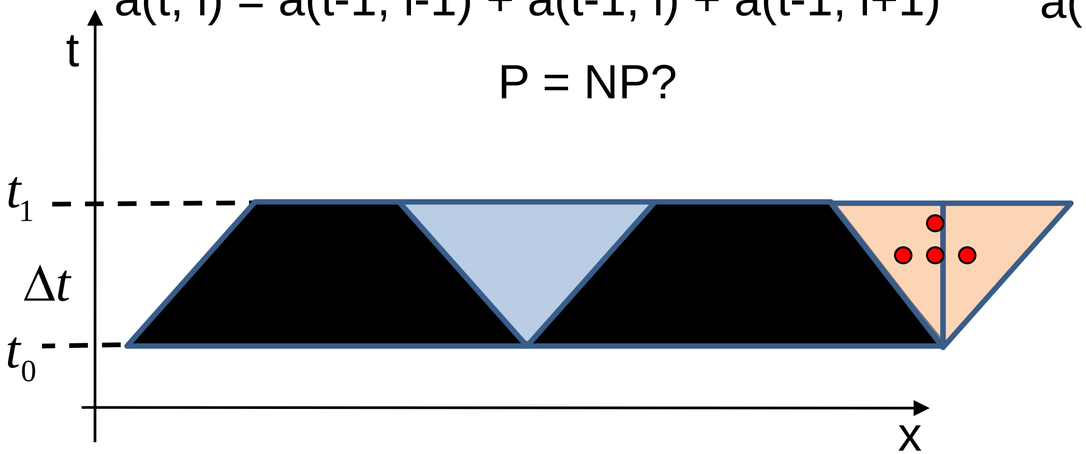
A unified algorithmic framework for both periodic and nonperiodic boundary conditions



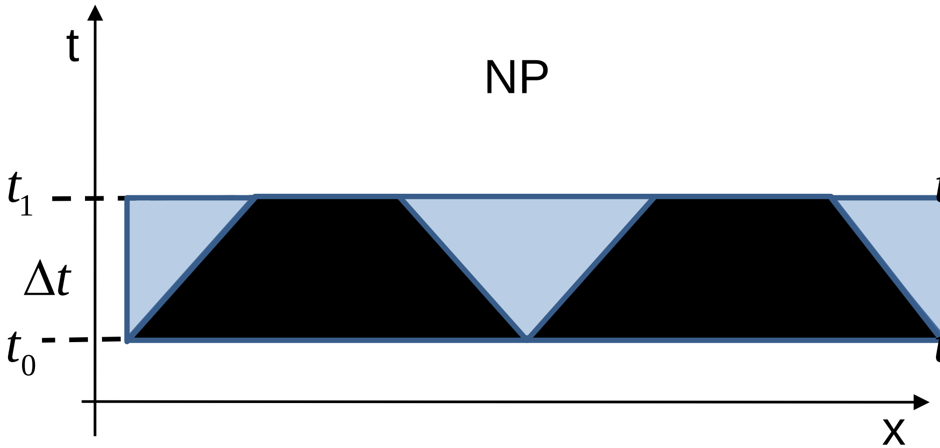


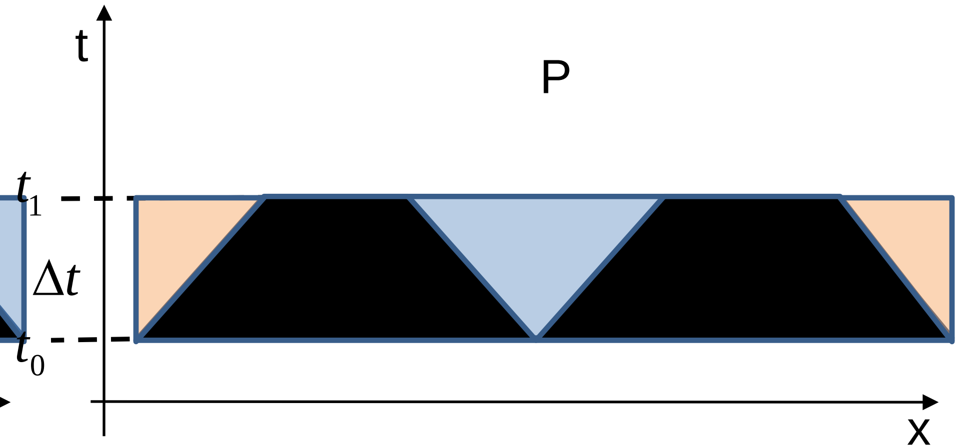
a(t, i) = a(t-1, i-1) + a(t-1, i) + a(t-1, i+1)

a(t, i%N) = a(t-1, (i-1)%N) + a(t-1, i%N) + a(t-1, (i+1)%N)



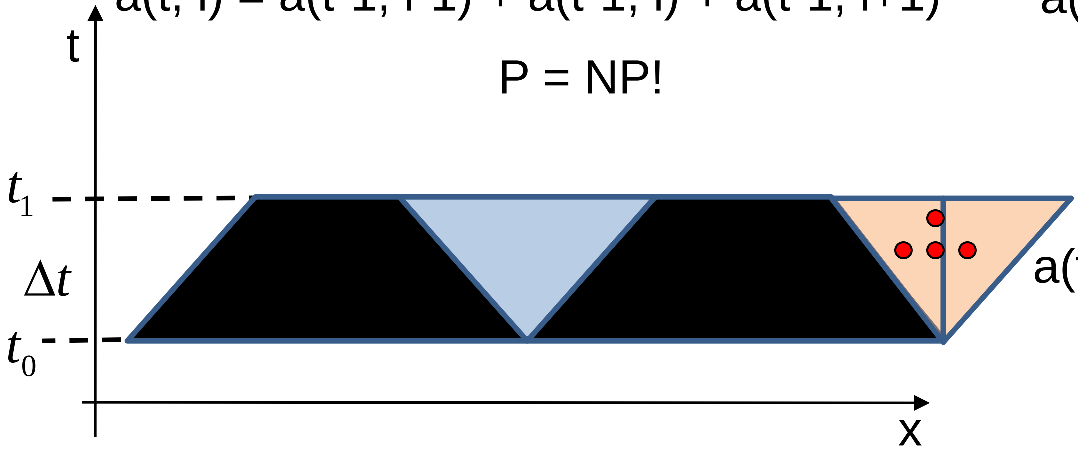
A unified algorithmic framework for both periodic and nonperiodic boundary conditions





a(t, i) = a(t-1, i-1) + a(t-1, i) + a(t-1, i+1)

a(t, i%N) = a(t-1, (i-1)%N) + a(t-1, i%N)
+ a(t-1, (i+1)%N)



a(t, i%N) = a(t-1, i-1) + a(t-1, i)
+ a(t-1, i+1)

Outline

- Functional Specification
- How the Pochoir System Works
- Algorithms
- Optimizing Strategies
- Conclusion

The Pochoir Project

- Pochoir version 1.0 has been released under GNU GPL v3.0. Please email your name and affiliation to pochoir@csail.mit.edu to request a copy of Pochoir compiler.
- We will be interested in user feedback on usability, the Pochoir language, performance issues, feature requests, and anything else.
- We would like to collect more examples and benchmarks of stencil computations.

Future Work

- Irregular computing domains
- Extending to Cluster (MPI)
- And much more…

THANK YOU!

EMAIL POCHOIR@CSAIL.MIT.EDU TO REQUEST A COPY OF POCHOIR COMPILER