Languages for High-Performance Computing

CSE 501
Spring 15
Announcements

• Homework 1 due next Monday at 11pm
  – Submit your code on dropbox

• Andre will have office hours today at 2:30 in CSE 615

• Project midpoint report due on May 5
Course Outline

• Static analysis
• Language design
  – High-performance computing
  – Parallel programming
  – Dynamic languages
• Program Verification
• Dynamic analysis
• New compilers
Today

• High-performance computing

• Languages for writing HPC applications
  – What are the design issues?

• Implementations of HPC languages
  – Using stencils as an example
Pick out hardware and software
Find the best vendor to work with
Get your people up to speed on HPC
High Performance Computing

• Application domains
  – Physical simulations
    • Heat equation, geo-modeling, traffic simulations
  – Scientific computations
    • Genomics, physics, astronomy, weather forecast, ...
  – Graphics
    • Rendering scenes from movies
  – Finance
    • High-frequency trading
High Performance Computing

• Hardware characteristics
  – Dedicated clusters of compute and storage nodes
  – Compute nodes:
    • Ultra-fast CPUs
    • Large cache
  – Dedicated interconnect network
    • Nodes arranged in a torus / ring
  – Separated physical storage from compute nodes
Example: Titan

- Built by Cray
- 18688 AMD 16-core CPUs, Tesla GPUs
- 8.2MW
- 4352 Ft²
- 693.5 TB memory
- 40 PB disk storage
- 17.59 P-FLOPS
- $97 million

Not your typical desktop machine
How to program HPC clusters?

• Highly (embarrassingly) parallel programs
  – Fortran, C, C++
  – Now using high performance DSLs
• Utilize both GPU and CPUs
• Batch job submission model

• Goal: utilize as many cores at the same time as possible
Stencil Programs
Stencils Programs

• **Definition**: For a given point, a *stencil* is a fixed subset of nearby neighbors.

• A *stencil code* updates every point in an $d$-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.

• Used in iterative PDE solvers such as Jacobi, multigrid, and adaptive mesh refinement, as well as for image processing and geometric modeling.
Stencil Programs

• Discretize space and time

• Typical program structure:

```c
for (t = 0; t < MAX_TS; ++t) {
    for (x = 0; x < MAX_X; ++x) {
        for (y = 0; y < MAX_Y; ++y) {
            array[t, x, y] =
                f(array[t-1, x, y], array[t-1, x-1, y-1], ...);
        }
    }
}
```
Stencil Programs

• Some terminology:
  – A stencil that updates a given point using $N$ nearby neighbor points is called a $N$-point stencil
  – The computation performed for each stencil is called a kernel
  – Boundary conditions describe what happens at the edge of the grid
    • Periodic means that the edge wraps around in a torus
Example: 2D Heat Diffusion

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

Heat equation

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

$\alpha$ is the thermal diffusivity.

Update rule

$$a[t, x, y] = a[t-1, x, y] \quad \text{+ CX} \cdot (a[t-1, x+1, y] \quad \text{+ 2} \cdot a[t-1, x, y] \quad \text{+ a[t-1, x-1, y])}$$

$$\quad \text{+ CY} \cdot (a[t-1, x, y+1] \quad \text{- 2} \cdot a[t-1, x, y] \quad \text{+ a[t-1, x, y-1])}$$

2D 5-point stencil
More Examples

1D 3-point stencil

3D 19-point stencil
Classical 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 cache optimization: **loop tiling**.
Parallelizing Loops

```c
for (t = 1; t <= T; ++t) {
    cilk_for (x = 0; x < X; ++x) {
        cilk_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]);
        }
    }
}
```

- All the iterations of the spatial loops are independent and can be parallelized straightforwardly.
- Intel Cilk Plus provides a `cilk_for` construct that performs the parallelization automatically.
- OpenMP is another framework for doing this
Issues with Looping

Example: 1D 3-point stencil

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

Divide-and-conquer \textit{cache-oblivious} techniques, based on \textit{trapezoidal decompositions}, are asymptotically efficient, achieving $\Theta(NT/MB)$ cache misses.

```c
void trapezoid(int t0, int t1, int x0, int dx0, int x1, int dx1) {
    lt = t1 - t0;
    if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * lt) {
        int xm = (2 * (x0 + x1) + (2 + dx0 + dx1) * lt) / 4;
        trapezoid(t0, t1, x0, dx0, xm, -1);
        trapezoid(t0, t1, xm, -1, x1, dx1);
    } else if (lt > 1) {
        int halflt = lt / 2;
        trapezoid(t0, t0 + halflt, x0, dx0, x1, dx1);
        trapezoid(t0 + halflt, t1, x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1);
    } else {
        for (int t = t0; t < t1; ++t) {
            for (int x = x0; x < x1; ++x)
                kernel(t, x);
            x0 += dx0;
            x1 += dx1;
        }
    }
}
```

Do you want to write this code?
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.

• Implements stencils using cache-oblivious algorithm that can be parallelized using Cilk.

• Easy to express both periodic and non-periodic boundary conditions.

• There are many DSLs for expressing stencils
  – Pochoir is one of them
Pochoir
(the Language)
2D Heat Equation

```c
int main(void) {
    Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
    return 0;
}
```

```c
Pochoir_Boundary_End
```

```c
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;
}
2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2    return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5    Pochoir_Shape_2D 2D_five_pt[6]
6        = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7    Pochoir_2D heat(2D_five_pt);
8    Pochoir_Array_2D(double) a(X,Y);
9    a.Register_Boundary(zero_bdry);
10   heat.Register_Array(a);
11   Pochoir_Kernel_2D(kern, t, x, y)
12       a(t,x,y) = a(t-1,x,y)
13          + 0.125*a(t-1,x+1,y) -
14          + 0.125*a(t-1,x,y+1) -
15          + 0.125*a(t-1,x,y+1) -
16 Pochoir_Kernel_End
17   for (int x = 0; x < X; ++x)
18       for (int y = 0; y < Y; ++y)
19           a(0,x,y) = rand();
20   heat.Run(T, kern);
21   for (int x = 0; x < X; ++x)
22       for (int y = 0; y < Y; ++y)
23           cout << a(T,x,y);
24   return 0;
25 }
```

Pochoir_Shape_2D name[count] = {cells};

- **dim** is the number of spatial dimensions of the stencil.
- **name** is the name of the declared Pochoir shape.
- **count** is the length of cells.
- **cells** is a list of the cells in the stencil.

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

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5   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}};
6   Pochoir_2D heat(2D_five_pt);
7   Pochoir_Array_2D(double) a(X,Y);
8   a.Register_Boundary(zero_bdry);
9   heat.Register_Array(a);
10  Pochoir_Kernel_2D(kern, t, x, y)
11     a(t,x,y) = a(t-1,x,y)
12       + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x,y+1))
13       + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
14  Pochoir_Kernel_End
15  for (int x = 0; x < X; ++x)
16     for (int y = 0; y < Y; ++y)
17       a(0,x,y) = rand();
18  heat.Run(T, kern);
19  for (int x = 0; x < X; ++x)
20     for (int y = 0; y < Y; ++y)
21       cout << a(T,x,y);  
22  return 0;
23 }
```

Pochoir_Shape_dimD name[count] = {cells};
- `dim` is the number of spatial dimensions of the stencil.
- `name` is the name of the declared Pochoir shape.
- `count` is the length of `cells`.
- `cells` is a list of the cells in the stencil.

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

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2    return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5    Pochoir_Shape_2D 2D_five_pt[6]
6        = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7    Pochoir_2D heat(2D_five_pt);
8    Pochoir_Array_2D(double) a(X,Y);
9    a.Register_Boundary(zero_bdry);
10   heat.Register_Array(a);
11   Pochoir_Kernel_2D(kern, t, x, y)
12      a(t,x,y) = a(t-1,x,y)
13         + 0.125*(a(t-1,x+1,y) -
14                      + 0.125*(a(t-1,x,y+1) -
15                                              + 0.125*(a(t-1,x,y-1));
16 Pochoir_Kernel_End
17   for (int x = 0; x < X; ++x)
18      for (int y = 0; y < Y; ++y)
19          a(0,x,y) = rand();
20   heat.Run(T, kern);
21   for (int x = 0; x < X; ++x)
22      for (int y = 0; y < Y; ++y)
23          cout << a(T,x,y);
24   return 0;
25 }
```

---

Pochoir_Shape_dimD name[count] = {cells};
- **dim** is the number of spatial dimensions of the stencil.
- **name** is the name of the declared Pochoir shape.
- **count** is the length of **cells**.
- **cells** is a list of the cells in the stencil.

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}, {-1,1,0}, {-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) -
    + 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;
}

Pochoir _dimD name (shape);
• dim is the number of spatial dimensions in the stencil computation.
• name is the name of the Pochoir object being declared.
• shape is the name of a Pochoir shape.

Declare a 2-dimensional Pochoir object heat whose kernel function will conform to the Pochoir shape 2D_five_pt. The Pochoir object will contain all the data and operating methods to perform the stencil computation.
# 2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2     return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5     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}};
6     Pochoir_2D heat(2D_five_pt);
7     Pochoir_Array_2D(double) a(X,Y);
8     a.Register_Boundary(zero_bdry);
9     heat.Register_Array(a);
10    Pochoir_Kernel_2D(kern, t, x, y)
11       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,y+1)) + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
12    Pochoir_Kernel_End
13    for (int x = 0; x < X; ++x)
14       for (int y = 0; y < Y; ++y)
15          a(0,x,y) = rand();
16    heat.Run(T, kern);
17    for (int x = 0; x < X; ++x)
18       for (int y = 0; y < Y; ++y)
19          cout << a(T,x,y);
20    return 0;
21}
```

**Pochoir_Array_dimD(type)**

- **type** is the type of the Pochoir array.
- **dim** is the number of dimensions.
- **array** is the name of the declared Pochoir array.
- The `size_{dim-1}, ..., size_1, size_0` are the number of grid points along each spatial dimension, indexed from 0.

**Declare a 2-dimensional Pochoir array** `a` of type `double` with spatial dimensions `X` grid points by `Y` grid points. The Pochoir array contains both underlying storage and requisite operating methods.
2D Heat Equation

```cpp
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2     return 0;
3 Pochoir_Boundary_End

4 int main(void) {
5     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}};
6     Pochoir_2D heat(2D_five_pt);
7     Pochoir_Array_2D(double) a(X,Y);
8     a.Register_Boundary(zero_bdry);
9     heat.Register_Array(a);
10    Pochoir_Kernel_2D(kern, t, x, y)
11       a(t,x,y) = a(t-1,x,y)
12           + 0.125*(a(t-1,x+1,y) - 2*a(t-1,x,y) + a(t-1,x-1,y))
13           + 0.125*(a(t-1,x,y+1) - 2*a(t-1,x,y) + a(t-1,x,y-1)) - 2
14    Pochoir_Kernel_End
15    for (int x = 0; x < X; ++x)
16        for (int y = 0; y < Y; ++y)
17            a(0,x,y) = rand();
18    heat.Run(T, kern);
19    for (int x = 0; x < X; ++x)
20        for (int y = 0; y < Y; ++y)
21            cout << a(T,x,y);
22    return 0;
23 }
```

Pochoir_Boundary_\textit{dim}D(\textit{name}, \\
\textit{array}, \textit{time, }x_{dim-1}, ..., x_1, x_0)  
<\textit{definition}>  
Pochoir_Boundary_end  
\begin{itemize}
  \item \textit{dim} is the number of dimensions.
  \item \textit{name} is a boundary function.
  \item \textit{array} is a Pochoir array.
  \item \textit{time} is the time coordinate.
  \item \textbf{x}_{dim-1}, ..., \textbf{x}_1, \textbf{x}_0 are the coordinates of each spatial dimension.
\end{itemize}  
\textbf{<definition>} is C++ code that returns values for \textit{array} when it is indexed by spatial coordinates that fall outside the declared dimensions.

Declare a \textbf{boundary function} \textit{zero_bdry} on the 2-dimensional Pochoir array \textit{arr} indexed by time coordinate \textbf{t} and spatial coordinates \textbf{x} and \textbf{y}, which always returns 0.
2D Heat Equation

1. Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2. return 0;
3. Pochoir_Boundary_End

4. int main(void) {
5.    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}};
6.    Pochoir_2D heat(2D_five_pt);
7.    Pochoir_Array_2D(double) a(X,Y);
8.    a.Register_Boundary(zero_bdry);
9.    heat.Register_Array(a);
10.   Pochoir_Kernel_2D(kern, t, x, y)
11.      a(t,x,y) = a(t-1,x,y)
12.          + 0.125*(a(t-1,x+1,y) - 2.0*a(t-1,x,y) + a(t-1,x-1,y))
13.          + 0.125*(a(t-1,x,y+1) - 2.0*a(t-1,x,y) + a(t-1,x,y-1));
14. Pochoir_Kernel_End
15.   for (int x = 0; x < X; ++x)
16.      for (int y = 0; y < Y; ++y)
17.        a(0,x,y) = rand();
18.    heat.Run(T, kern);
19.    for (int x = 0; x < X; ++x)
20.       for (int y = 0; y < Y; ++y)
21.         cout << a(T,x,y);
22.   return 0;
23. }

array.Register_Boundary(bdry)
- array is a Pochoir array.
- bdry is the name of a boundary function to return a value when array is indexed by spatial coordinates that fall outside array’s declared bounds.

Register the boundary function zero_bdry with the Pochoir array a.
2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5   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}};
6   Pochoir_2D heat(2D_five_pt);
7   Pochoir_Array_2D(double) a(X,Y);
8   a.Register_Boundary(zero_bdry);
9   heat.Register_Array(a);
10  Pochoir_Kernel_2D(kern, t, x, y)
11     a(t,x,y) = a(t-1,x,y)
12       + 0.125*(a(t-1,x+1,y) -
13       + 0.125*(a(t-1,x,y+1) -
14       + 0.125*(a(t-1,x,y-1))
15  Pochoir_Kernel_End
16  for (int x = 0; x < X; ++x)
17      for (int y = 0; y < Y; ++y)
18        a(0,x,y) = rand();
19  heat.Run(T, kern);
20  for (int x = 0; x < X; ++x)
21      for (int y = 0; y < Y; ++y)
22        cout << a(T,x,y);
23  return 0;
24 }
```

- `name.Register_Array(array)`
  - `name` is a Pochoir object.
  - `array` is a Pochoir array to register with `name`. Several Pochoir arrays can be registered with the same Pochoir object.

Register the Pochoir array `a` with the Pochoir object `heat`. 
2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5   Pochoir_Shape_2D 2D_five_pt[6]  
6     = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7   Pochoir_2D heat(2D_five_pt);
8   Pochoir_Array_2D(double) a(X,Y);
9   a.Register_Boundary(zero_bdry);
10  heat.Register_Array(a);
11  Pochoir_Kernel_2D(kern, t, x, y)
12     a(t,x,y) = a(t-1,x,y)
13       + 0.125*(a(t-1,x+1,y) - 2*
14                      a(t-1,x,y) + a(t-1,x,y+1) - 2*
15                           a(t-1,x,y+1));
16 Pochoir_Kernel_End
17 for (int x = 0; x < X; ++x)
18   for (int y = 0; y < Y; ++y)
19     a(0,x,y) = rand();
20  heat.Run(T, kern);
21 for (int x = 0; x < X; ++x)
22   for (int y = 0; y < Y; ++y)
23     cout << a(T,x,y);
24 return 0;
25}
```

Pochoir_kernel_{\text{dim}}D(func, time, x_{\text{dim-1}}, \ldots, x_1, x_0)

\text{*definition*}

Pochoir_kernel_end

- \textit{dim} is the number of dimensions.
- \textit{func} is the name of the kernel function being declared.
- \textit{time} is the time coordinate.
- \(x_{\text{dim-1}}, \ldots, x_1, x_0\) are the coordinates of the spatial dimension.

\textit{<definition>} is C++ code that defines how each grid point (as represented by Pochoir arrays at a given coordinate) should be updated as a function of neighboring gridpoints earlier in time.

Declare a \textit{kernel function} \text{\textit{kern}} with time parameter \text{\textit{t}} and spatial parameters \text{x} and \text{y}.
2D Heat Equation

```c
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;
}
```

The Pochoir arrays can be initialized in whatever manner the programmer wishes. Time coordinates 0, 1, ..., depth must be initialized, where depth is the shape depth: the zero-based time dimension of the Pochoir shape (usually 1).

Initialize all points of the grid at time 0 to a random value.
2D Heat Equation

```c
1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5    Pochoir_Shape_2D 2D_five_pt[6]
6       = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7    Pochoir_2D heat(2D_five_pt);
8    Pochoir_Array_2D(double) a(X,Y);
9    a.Register_Boundary(zero_bdry);
10   heat.Register_Array(a);
11 Pochoir_Kernel_2D(kern, t, x, y)
12    a(t,x,y) = a(t-1,x,y)
13       + 0.125*(a(t-1,x+1,y) -
14                   + 0.125*(a(t-1,x,y+1) -
15 Pochoir_Kernel_End
16    for (int x = 0; x < X; ++x)
17       for (int y = 0; y < Y; ++y)
18          a(0,x,y) = rand();
19    heat.Run(T, kern);
20    for (int x = 0; x < X; ++x)
21       for (int y = 0; y < Y; ++y)
22          cout << a(T,x,y);
23    return 0;
24 }
```

**name.Run(steps, func)**
- **name** is the name of a Pochoir object.
- **steps** is the number of time steps to run the stencil computation.
- **func** is a defined kernel function. compatible with the Pochoir shape registered with name.

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.
2D Heat Equation

1 Pochoir_Boundary_2D(zero_bdry, arr, t, x, y)
2   return 0;
3 Pochoir_Boundary_End
4 int main(void) {
5   Pochoir_Shape_2D 2D_five_pt[6]
6     = {{0,0,0}, {-1,1,0}, {-1,0,0}, {-1,-1,0}, {-1,0,-1}, {-1,0,1}};
7   Pochoir_2D heat(2D_five_pt);
8   Pochoir_Array_2D(double) a(X,Y);
9   a.Register_Boundary(zero_bdry);
10  heat.Register_Array(a);
11 Pochoir_Kernel_2D(kern, t, x, y)
12   a(t,x,y) = a(t-1,x,y)
13       + 0.125*(a(t-1,x+1,y) -
14           + 0.125*(a(t-1,x,y+1) -
15   Pochoir_Kernel_End
16   for (int x = 0; x < X; ++x)
17     for (int y = 0; y < Y; ++y)
18       a(0,x,y) = rand();
19   heat.Run(T, kern);
20   for (int x = 0; x < X; ++x)
21     for (int y = 0; y < Y; ++y)
22       cout << a(T,x,y);
23 return 0;
24 }

Elements of the Pochoir array can be read out anytime after the computation by indexing elements with time coordinate \textit{time}+\textit{depth}−1, where \textit{time} is the number of steps executed and \textit{depth} is the shape depth. The \texttt{<<} operator is overloaded for Pochoir arrays to pretty-print their contents.

Print the elements of the Pochoir array \texttt{a} to standard out. The statement \texttt{cout \texttt{\textless\textless} a;} would pretty-print the results.
Expressing Boundary Conditions

**Nonperiodic zero boundary**

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

**Periodic (toroidal) boundary**

```c
#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
```
Pochoir
(the Compiler)
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.

Why is this important?
Impact 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 and rely on base C++ compiler
Pochoir
(the Implementation)
Optimizations

• Two code clones
• Unifying the handling of periodic and nonperiodic boundary conditions
• Automatic selection of optimizing strategy
• 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 \textit{slow clone} handles regions that contain boundaries and checks for out-of-range grid points.
- The \textit{fast clone} handles the larger interior regions which require no range checking.

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

• Design specific constructs for domain

• Constructs need to easily map to underlying target language

• Exposing high-level structure allows domain-specific optimizations