The Pochoir Stencil Compiler

Yuan Tang Rezaul Chowdhury Bradley C. Kuszmaul
Chi-Keung Luk Charles E. Leiserson
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139, USA

Chen Wang, 2018/11/28

Motivation

- Stencil computations are easy to implement using nested loops. But looping implementations suffer from poor cache performance.
- Cache oblivious algorithms are more efficient, but they are difficult to write, especially when parallelism is involved.

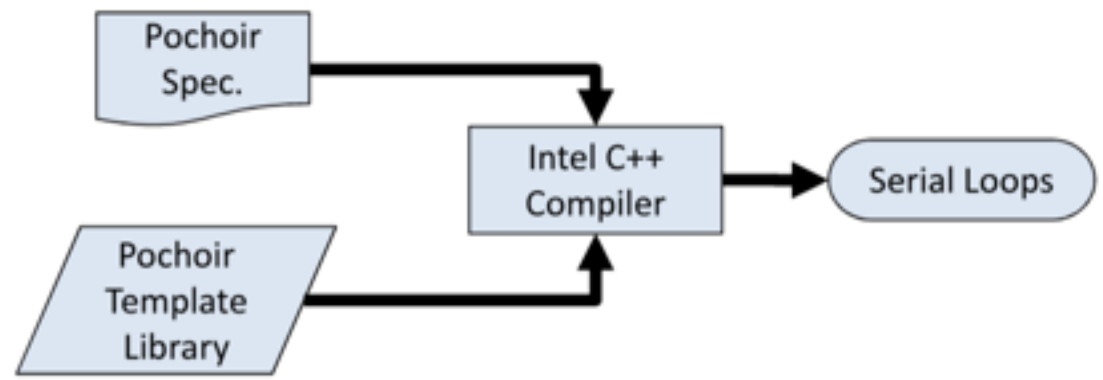
- The Pochoir compiler allows a programmer to write a stencil program in a DSL embedded in C++. The Pochoir compiler then translates it into high-performing **Cilk code** that employs an efficient parallel cache-oblivious algorithm.

What is Pochoir

- Pochoir (pronounced as “PO-shwar”; it means “stencil” in French) is a compiler and runtime system for implementing stencil computations on multicore processors.
  1. Pochoir template library
  2. Pochoir compiler

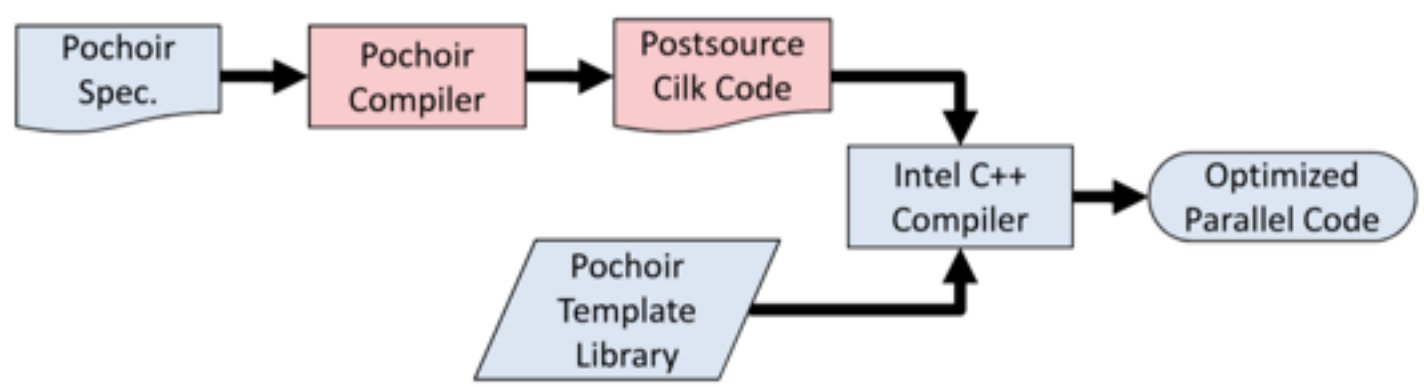
Workflow

- Phase 1:
  the programmer uses the normal Intel C++ compiler to compile his or her code with the Pochoir template library. Phase 1 verifies that the programmer's stencil specification is Pochoir compliant.



(a)

- Phase 2:
  the programmer uses the Pochoir compiler, which acts as a preprocessor to the Intel C++ compiler, to generate optimized multithreaded Cilk code.



(b)

Example: periodic 2D heat equation

- 2d heat equation

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

- Jacobi-style update equation:

$$\begin{aligned} u_{t+1}(x,y) &= u_t(x,y) \\ &+ \frac{\alpha \Delta t}{\Delta x^2} \left(u_t(x-1,y) + u_t(x+1,y) - 2u_t(x,y) \right) \\ &+ \frac{\alpha \Delta t}{\Delta y^2} \left(u_t(x,y-1) + u_t(x,y+1) - 2u_t(x,y) \right) \ . \end{aligned}$$

Example: periodic 2D heat equation

- 2d heat equation

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

- Jacobi-style update equation:

$$\begin{aligned} u_{t+1}(x,y) \quad = \quad & u_t(x,y) \\ & + \frac{\alpha \Delta t}{\Delta x^2} \left(u_t(x-1,y) + u_t(x+1,y) - 2u_t(x,y) \right) \\ & + \frac{\alpha \Delta t}{\Delta y^2} \left(u_t(x,y-1) + u_t(x,y+1) - 2u_t(x,y) \right) \ . \end{aligned}$$

- Simple for loop implementation:

$\textsc{Loops}(u; ta, tb; xa, xb; ya, yb)$

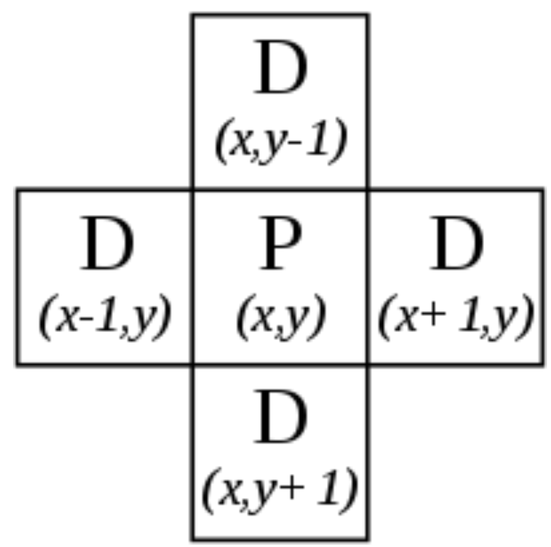
$$\begin{array}{l} \textbf{for } t = ta \textbf{ to } tb - 1 \\ \quad \textbf{parallel for } x = xa \textbf{ to } xb - 1 \\ \quad\quad \textbf{for } y = ya \textbf{ to } ya - 1 \\ \quad\quad\quad u((t+1) \bmod 2, x, y) = u(t \bmod 2, x, y) \\ \quad\quad\quad\quad + CX \cdot (u(t \bmod 2, (x-1) \bmod X, y) \\ \quad\quad\quad\quad + u(t \bmod 2, (x+1) \bmod X, y) - 2u(t \bmod 2, x, y)) \\ \quad\quad\quad\quad + CY \cdot (u(t \bmod 2, x, (y-1) \bmod Y) \\ \quad\quad\quad\quad + u(t \bmod 2, x, (y+1) \bmod Y) - 2u(t \bmod 2, x, y)) \end{array}$$

Example: periodic 2D heat equation

time step

- **Pochoir_Shape_2D** 2d_five_pt[] = {{0, 0, 0}, {−1, 1, 0}, {−1, 0, 0}, {−1, −1, 0}, {−1, 0, 1}, {−1, 0, −1}} ;

x offset y offset



- **Pochoir_2D** heat(2d_five_pt);

Example: periodic 2D heat equation

function name   data array   time step

spatial coordinates

Pochoir_Boundary_2D (heat_bv, array, t, x, y)
    return array.get(t, mod(x, array.size(1)), mod(y, array.size(0)));
Pochoir_Boundary_End

This construct defines a boundary function called **heat_bv** that will be invoked to supply a value when the stencil computation accesses a point outside the domain of the Pochoir array array.

Example: periodic 2D heat equation

function name data array time step spatial coordinates

```
Pochoir_Kernel_2D (heat_fn,   u,   t,   x,   y)
  u(t+1, x, y) = CX * (u(t, x+1, y) - 2 * u(t, x, y) + u(t, x-1, y)) +
                 CY * (u(t, x, y+1) - 2  * u(t, x, y) + u(t, x, y-1)) +
                 u(t, x, y);
Pochoir_Kernel_End
```

This construct defines a kernel function named **heat_fn** for updating a stencil on a spatial grid with dim spatial dimensions.

Example: periodic 2D heat equation

```
#define mod(r,m) ((r)%(m) + ((r)<0)? (m):0)

Pochoir_Boundary_2D(heat_bv, a, t, x, y)
  return a.get(t,mod(x,a.size(1)),mod(y,a.size(0)));
Pochoir_Boundary_End

int main(void) {

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

  Pochoir_Array_2D(double) u(X, Y);
  u.Register_Boundary(heat_bv);
  heat.Register_Array(u);

  Pochoir_Kernel_2D(heat_fn, t, x, y)
    u(t+1, x, y) = CX * (u(t, x+1, y) - 2 * u(t, x,
         y) + u(t, x-1, y)) + CY * (u(t, x, y+1) - 2
         * u(t, x, y) + u(t, x, y-1)) + u(t, x, y);
  Pochoir_Kernel_End

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

  heat.Run(T, heat_fn);

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


  return 0;
}
```

Pochoir_Shape_dimD contains the spatial information. Each of its element has dim+1 integers represent the offset of each memory footprint in the stencil kernel relative to the space-time grid point ⟨t , x, y, · · ·⟩.

Example: periodic 2D heat equation

```
#define mod(r,m) ((r)%(m) + ((r)<0)? (m):0)

Pochoir_Boundary_2D(heat_bv, a, t, x, y)
  return a.get(t,mod(x,a.size(1)),mod(y,a.size(0)));
Pochoir_Boundary_End

int main(void) {

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

  Pochoir_Array_2D(double) u(X, Y);
  u.Register_Boundary(heat_bv);
  heat.Register_Array(u);

  Pochoir_Kernel_2D(heat_fn, t, x, y)
    u(t+1, x, y) = CX * (u(t, x+1, y) - 2 * u(t, x,
         y) + u(t, x-1, y)) + CY * (u(t, x, y+1) - 2
         * u(t, x, y) + u(t, x, y-1)) + u(t, x, y);
  Pochoir_Kernel_End

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

  heat.Run(T, heat_fn);

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


  return 0;
}
```

The static information about a Pochoir stencil computation, such as the computing kernel, the boundary conditions, and the stencil shape, is stored in a **Pochoir_dimD**

Example: periodic 2D heat equation

```
#define mod(r,m) ((r)%(m) + ((r)<0)? (m):0)

Pochoir_Boundary_2D(heat_bv, a, t, x, y)
  return a.get(t,mod(x,a.size(1)),mod(y,a.size(0)));
Pochoir_Boundary_End

int main(void) {

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

  Pochoir_Array_2D(double) u(X, Y);
  u.Register_Boundary(heat_bv);
  heat.Register_Array(u);

  Pochoir_Kernel_2D(heat_fn, t, x, y)
    u(t+1, x, y) = CX * (u(t, x+1, y) - 2 * u(t, x,
         y) + u(t, x-1, y)) + CY * (u(t, x, y+1) - 2
         * u(t, x, y) + u(t, x, y-1)) + u(t, x, y);
  Pochoir_Kernel_End

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

  heat.Run(T, heat_fn);

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


  return 0;
}
```

The boundary function will be invoked to supply a value when the stencil computation accesses a point outside the domain of the Pochoir array array.

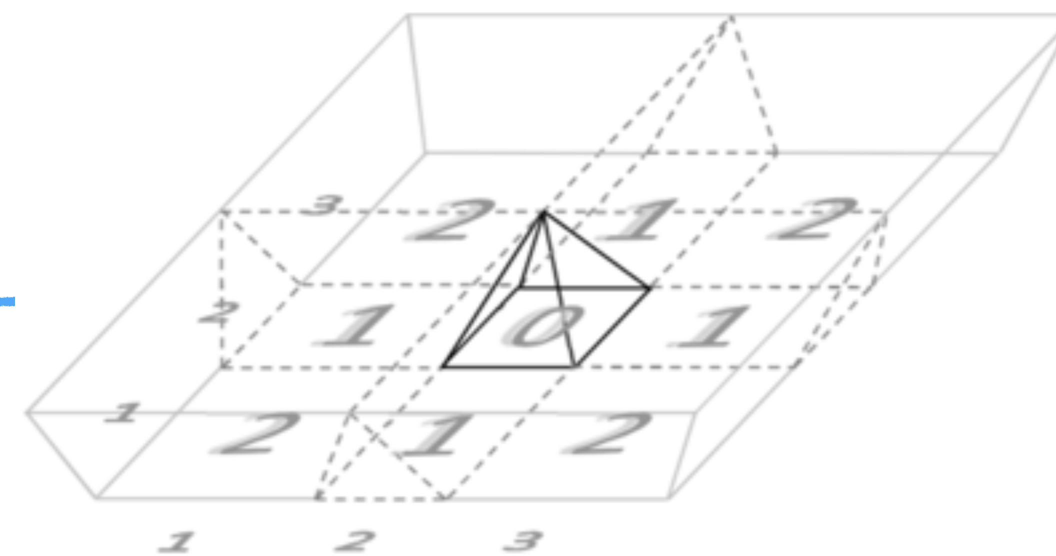
Example: periodic 2D heat equation

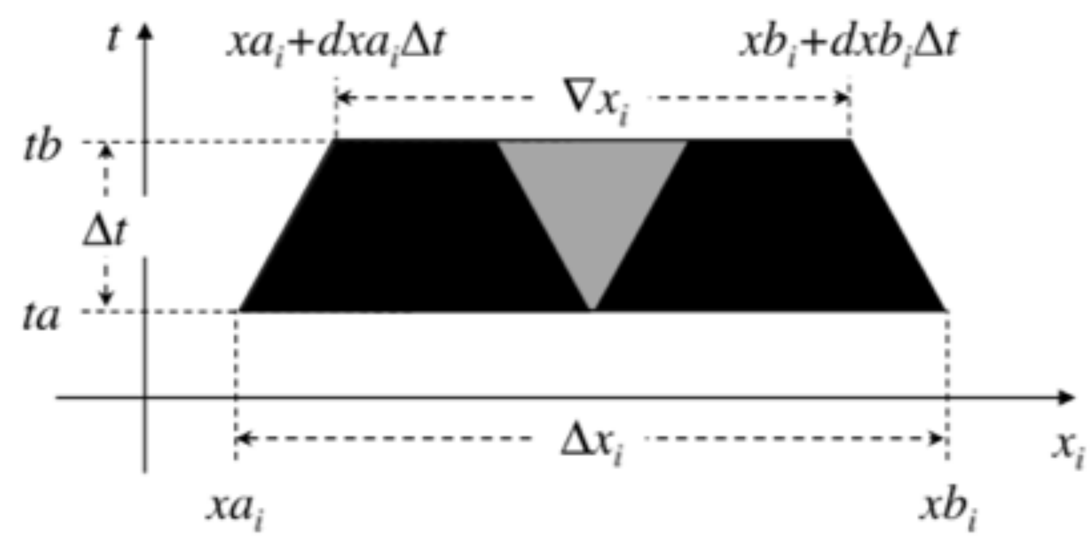
```
#define mod(r,m) ((r)%(m) + ((r)<0)? (m):0)

Pochoir_Boundary_2D(heat_bv, a, t, x, y)
  return a.get(t,mod(x,a.size(1)),mod(y,a.size(0)));
Pochoir_Boundary_End

int main(void) {

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

  Pochoir_Array_2D(double) u(X, Y);
  u.Register_Boundary(heat_bv);
  heat.Register_Array(u);

  Pochoir_Kernel_2D(heat_fn, t, x, y)
    u(t+1, x, y) = CX * (u(t, x+1, y) - 2 * u(t, x,
         y) + u(t, x-1, y)) + CY * (u(t, x, y+1) - 2
         * u(t, x, y) + u(t, x, y-1)) + u(t, x, y);
  Pochoir_Kernel_End

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

  heat.Run(T, heat_fn);

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


  return 0;
}
```

Pochoir_Kernel_dimD arbitrary C++ code for updating a stencil on a spatial grid with dim spatial dimensions .

Trapezoid (zoid) decomposition with hyperspace cut

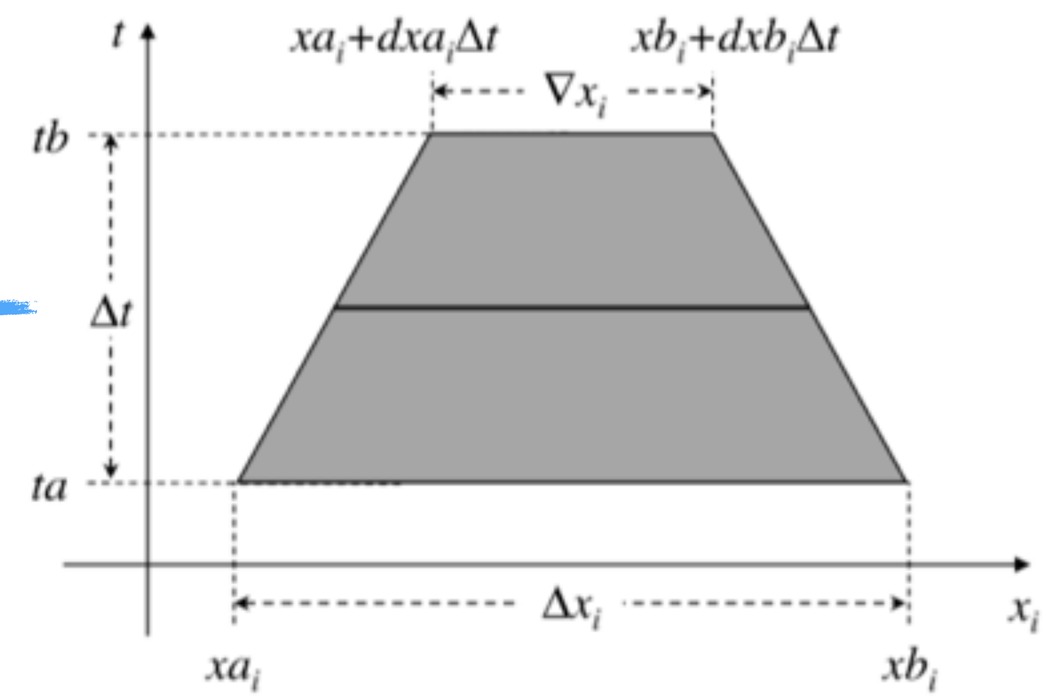
```
TRAP(u; ta, tb; xa, xb, dxa, dxb; ya, yb, dya, dyb)
Δt = tb − ta
Δx = max{xb − xa, (xb + dxbΔt) − (xa + dxaΔt)}  // Longer x-base
Δy = max{yb − ya, (yb + dybΔt) − (ya + dyaΔt)}  // Longer y-base
k = 0  // Try hyperspace cut
if Δx ≥ 2σ_x Δt
    Trisect the zoid with x-cuts
    k += 1
if Δy ≥ 2σ_y Δt
    Trisect the zoid with y-cuts
    k += 1
if k > 0
    Assign dependency levels 0, 1, ..., k to subzoids
    for i = 0 to k  // for each dependency level i
        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 zoid and then the upper
    TRAP(ta, ta + Δt/2; xa, xb, dxa, dxb; ya, yb, dya, dyb)
    TRAP(ta + Δt/2, tb; xa + dxaΔt/2, xb + dxbΔt/2, dxa, dxb;
    ya + dyaΔt/2, yb + dybΔt/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
            ya += dya
            yb += dyb
```





Trapezoid (zoid) decomposition with hyperspace cut

```
TRAP(u; ta, tb; xa, xb, dxa, dxb; ya, yb, dya, dyb)
Δt = tb − ta
Δx = max{xb − xa, (xb + dxbΔt) − (xa + dxaΔt)} // Longer x-base
Δy = max{yb − ya, (yb + dybΔt) − (ya + dyaΔt)} // Longer y-base
k = 0 // Try hyperspace cut
if Δx ≥ 2σ_x Δt
    Trisect the zoid with x-cuts
    k += 1
if Δy ≥ 2σ_y Δt
    Trisect the zoid with y-cuts
    k += 1
if k > 0
    Assign dependency levels 0, 1, ..., k to subzoids
    for i = 0 to k // for each dependency level i
        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 zoid and then the upper
    TRAP(ta, ta + Δt/2; xa, xb, dxa, dxb; ya, yb, dya, dyb)
    TRAP(ta + Δt/2, tb; xa + dxaΔt/2, xb + dxbΔt/2, dxa, dxb;
    ya + dyaΔt/2, yb + dybΔt/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
        ya += dya
        yb += dyb
```



Trapezoid (zoid) decomposition with hyperspace cut

```
TRAP(u; ta, tb; xa, xb, dxa, dxb; ya, yb, dya, dyb)
Δt = tb − ta
Δx = max {xb − xa, (xb + dxbΔt) − (xa + dxaΔt)} // Longer x-base
Δy = max {yb − ya, (yb + dybΔt) − (ya + dyaΔt)} // Longer y-base
k = 0 // Try hyperspace cut
if Δx ≥ 2σ_x Δt
    Trisect the zoid with x-cuts
    k += 1
if Δy ≥ 2σ_y Δt
    Trisect the zoid with y-cuts
    k += 1
if k > 0
    Assign dependency levels 0, 1, ..., k to subzoids
    for i = 0 to k // for each dependency level i
        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 zoid and then the upper
    TRAP(ta, ta + Δt/2; xa, xb, dxa, dxb; ya, yb, dya, dyb)
    TRAP(ta + Δt/2, tb; xa + dxaΔt/2, xb + dxbΔt/2, dxa, dxb;
        ya + dyaΔt/2, yb + dybΔt/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
            ya += dya
            yb += dyb
```

Base case (delta_t = 1)

Coarsening of Base Cases

- Although trapezoidal decomposition reduces cache-miss rates, overall performance can suffer from function-call overhead unless the base case of the recursion is coarsened.
- Solution: reduce the overhead of function(kernel) calls by **coarsening of base cases**.
  - For 2D problems, Pochoir stops the recursion at <u>100×100 space chunks with 5 time steps</u>.
  - For 3D problems, the recursion stops at <u>$1000 \times 3 \times 3$ with 3 time steps</u>.
  - Higher dimensions?

Trapezoid (zoid) decomposition with hyperspace cut

```
TRAP(u; ta, tb; xa, xb, dxa, dxb; ya, yb, dya, dyb)
Δt = tb − ta
Δx = max{xb − xa, (xb + dxbΔt) − (xa + dxaΔt)} // Longer x-base
Δy = max{yb − ya, (yb + dybΔt) − (ya + dyaΔt)} // Longer y-base
k = 0 // Try hyperspace cut
if Δx ≥ 2σ_xΔt
    Trisect the zoid with x-cuts
    k += 1
if Δy ≥ 2σ_yΔt
    Trisect the zoid with y-cuts
    k += 1
if k > 0
    Assign dependency levels 0, 1, ..., k to subzoids
    for i = 0 to k // for each dependency level i
        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 zoid and then the upper
    TRAP(ta, ta + Δt/2; xa, xb, dxa, dxb; ya, yb, dya, dyb)
    TRAP(ta + Δt/2, tb; xa + dxaΔt/2, xb + dxbΔt/2, dxa, dxb;
    ya + dyaΔt/2, yb + dybΔt/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
        ya += dya
        yb += dyb
```

- For 2D problems, Pochoir stops the recursion at 100 × 100 space chunks with 5 time steps.
- For 3D problems, the recursion stops at 1000 × 3 × 3 with 3 time steps.

Handling boundary conditions with code cloning

- Pochoir compiler generates two code clones of the kernel function:
    1. a slower **boundary clone**: the boundary clone is used for boundary zoids: those that contain at least one point whose computation requires an off-grid access.
    2. a faster **interior clone**: the interior clone is used for interior zoids: those all of whose points can be updated without indexing off the edge of the grid.

Loop Indexing

Two ways to generate the interior clone of the kernel function.
-split-pointer by default. User can decide by command-line option.

- -split-macro-shadow

```
Pochoir_Kernel_1D(heat_1D_fn, t, i)
    a(t+1, i) = 0.125 * (a(t, i-1) + 2 * a(t, i) +
          a(t, i+1));
Pochoir_Kernel_End
```

(a)

```
/* a.interior() is a function to dereference the
      value without checking boundary conditions */
#define a(t, i) a.interior(t, i)
Pochoir_Kernel_1D(heat_1D_fn, t, i)
    a(t + 1, i) = 0.125 * (a(t, i - 1) + 2 * a(t, i
          ) + a(t, i + 1));
Pochoir_Kernel_End
#undef a(t, i)
```

(b)

- -split-pointer

```
Pochoir_Kernel_1D(heat_1D_fn, t, i)
/* The base address of the Pochoir array 'a' */
double *a_base = a.data();
/* Pointers to be used in the innermost loop */
double *iter0, *iter1, *iter2, *iter3;
/* Total size of the Pochoir array 'a' */
const int l_a_total_size = a.total_size();
int gap_a_0;
const int l_stride_a_0 = a.stride(0);
for (int t = ta; t < tb; ++t) {
    double * baseIter_1;
    double * baseIter_0;
    baseIter_0 = a_base + ((t + 1) & 0xb) *
          l_a_total_size + (l_grid.xa[0]) *
          l_stride_a_0;
    baseIter_1 = a_base + ((t) & 0xb) *
          l_a_total_size + (l_grid.xa[0]) *
          l_stride_a_0;
    iter0 = baseIter_0 + (0) * l_stride_a_0;
    iter1 = baseIter_1 + (-1) * l_stride_a_0;
    iter2 = baseIter_1 + (0) * l_stride_a_0;
    iter3 = baseIter_1 + (1) * l_stride_a_0;
    for (int i = l_grid.xa[0]; i < l_grid.xb[0];
          ++i, ++iter0, ++iter1, ++iter2, ++iter3) {

    (*iter0) = 0.125 * ((*iter1) + 2 * (*iter2) +
          (*iter3));  }
}
Pochoir_Kernel_End
```

(c)

Loop Indexing

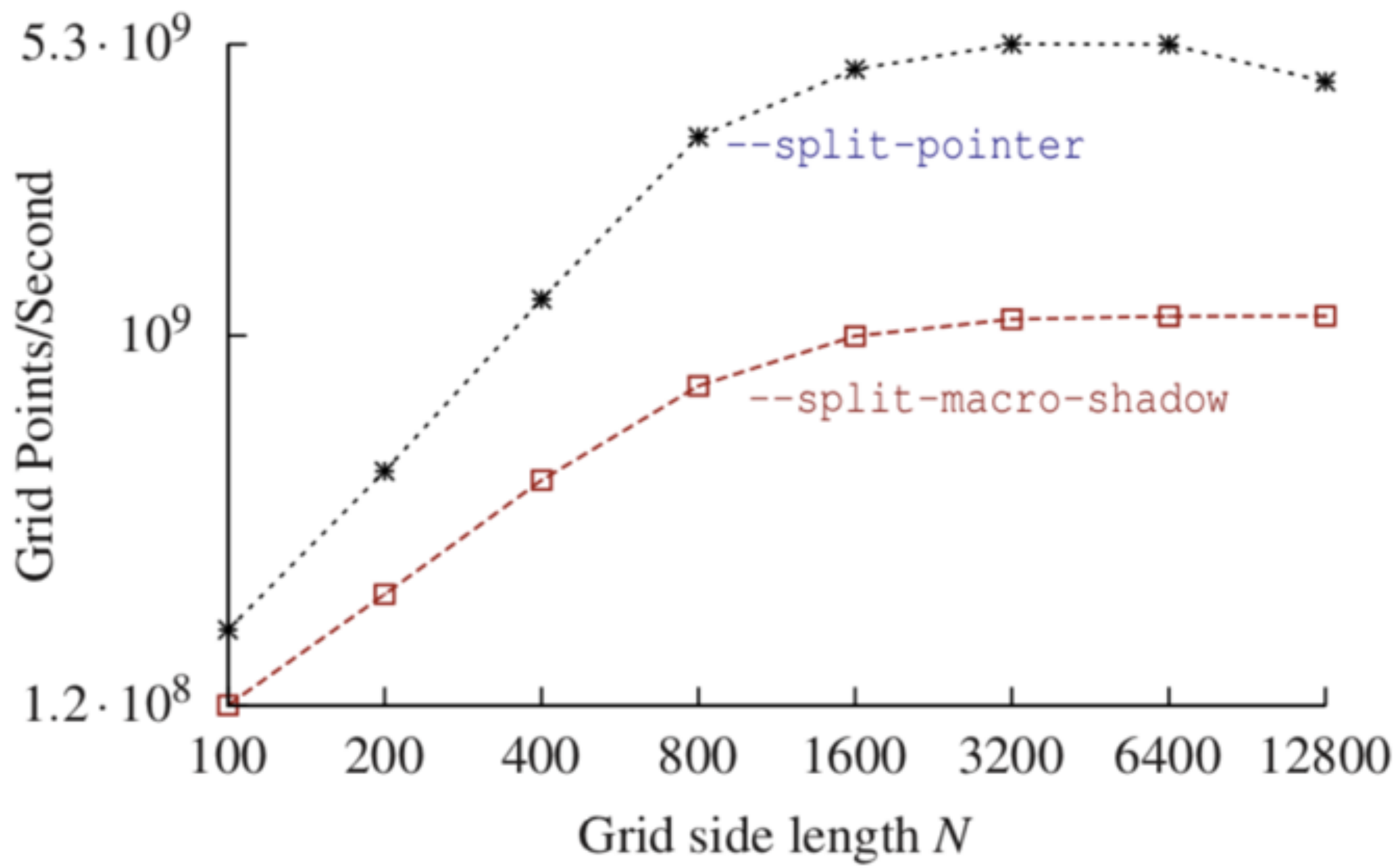


Figure 13: The performance of different loop-index optimizations on a 2D heat equation on torus. The grid is N^2 with 1000 time steps.

Parallelism and cache miss ratio

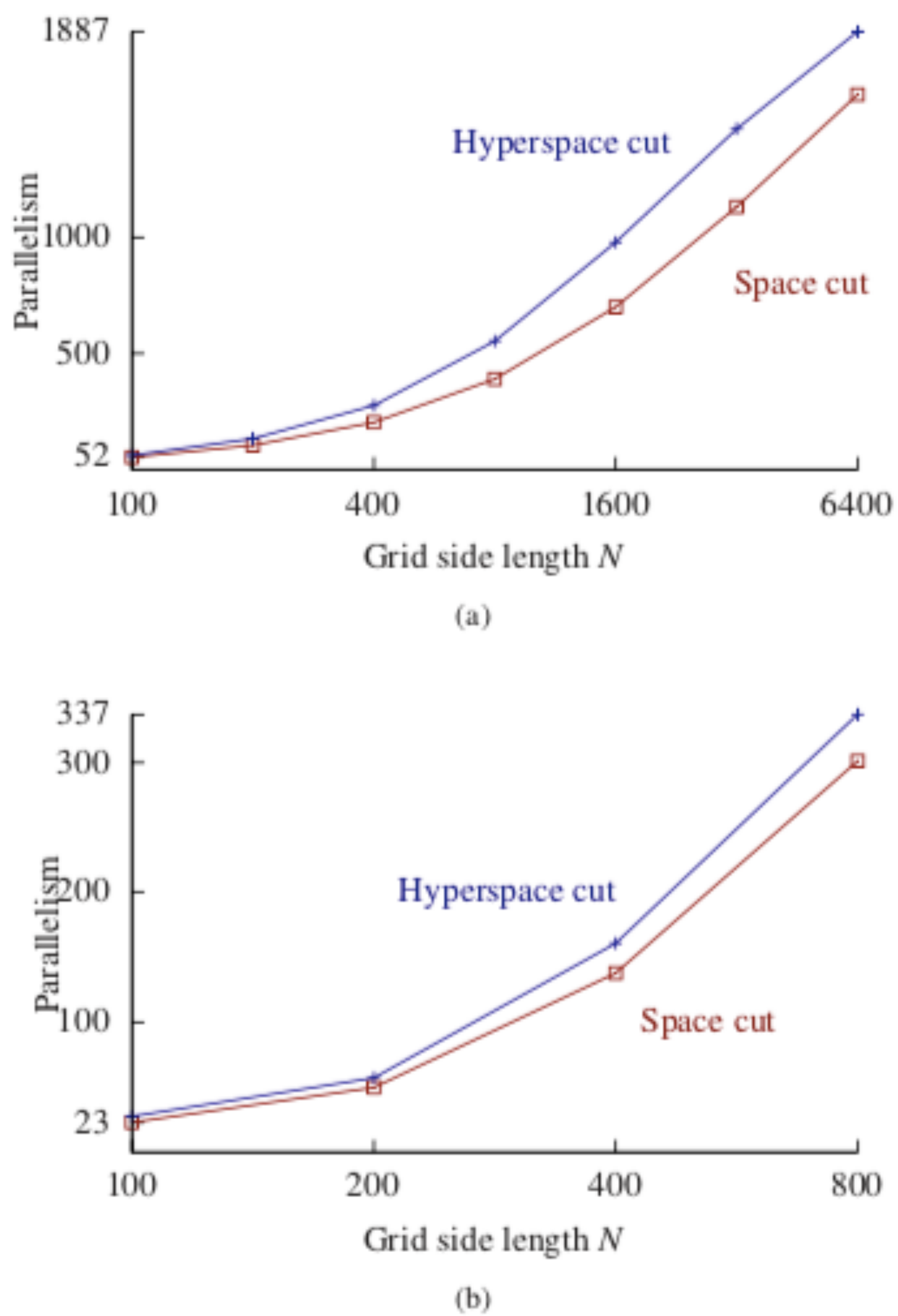


Figure 9: Parallelism comparison on two benchmarks between TRAP, which employs hyperspace cuts, and STRAP, which uses serial space cuts. Measurements are of code without base-case coarsening. (a) 2D nonperiodic heat equation. Space-time size is $1000N^2$. (b) 3D nonperiodic wave equation. Space-time size is $1000N^3$.

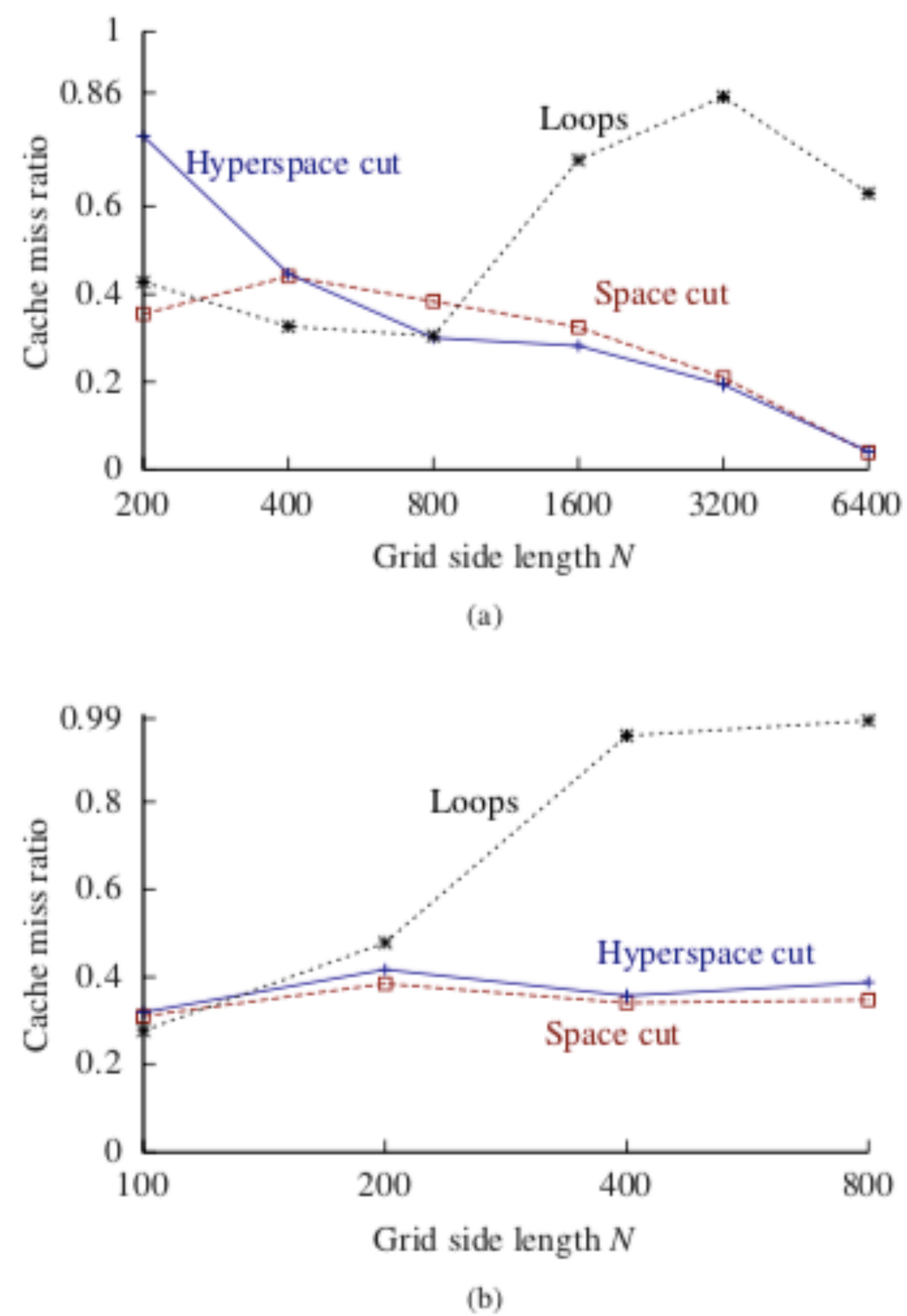


Figure 10: Cache-miss ratios for two benchmarks using TRAP, STRAP, and a parallel-loop algorithm. The cache-miss ratio is the ratio of the cache misses to the number of memory references. Measurements are of code without base-case coarsening. (a) 2D nonperiodic heat equation. Space-time is $1000N^2$. (b) 3D nonperiodic wave equation. Space-time is $1000N^3$.

Benchmark

- Heat: heat equation on a 2D grid, a 2D torus, and a 4D grid;
- Life: Conway's game of Life (Life)
- Wave: 3D finite-difference wave equation
- LBM: lattice Boltzmann method (LBM)
- RNA: RNA secondary structure prediction
- PSA: pairwise sequence alignment
- LCS: longest common subsequence
- APOP: American put stock option pricing (APOP)

Benchmark

Pochoir performance on an Intel Core i7 (Nehalem) machine

Benchmark	*Dims*	*Grid size*	*Time steps*	*Pochoir* 1 *core*	*Pochoir* 12 *cores*	*Pochoir speedup*	*Serial loops time*	*Serial loops ratio*	12-*core loops time*	12-*core loops ratio*
Heat	2	$16,000^2$	500	277s	24s	11.5	612s	25.5	149s	6.2
Heat	2p	$16,000^2$	500	281s	24s	11.7	1,647s	68.6	248s	10.3
Heat	4	150^4	100	154s	54s	2.9	433s	8.0	104s	1.9
Life	2p	$16,000^2$	500	345s	28s	12.3	2,419s	86.4	332s	11.9
Wave	3	$1,000^3$	500	3,082s	447s	6.9	3,170s	7.1	1,071s	2.4
LBM	3	$100^2 \times 130$	3,000	345s	68s	5.1	304s	4.5	220s	3.2
RNA	2	300^2	900	90s	20s	4.5	121s	6.1	26s	1.3
PSA	1	100,000	200,000	105s	18s	5.8	432s	24.0	77s	4.3
LCS	1	100,000	200,000	57s	9s	6.3	105s	11.7	27s	3.0
APOP	1	2,000,000	10,000	43s	4s	10.7	515s	128.8	48s	12.0

- **serial loops:** a serial for loop implementation running on one core
- **12-core loops:** a parallel cilk_for loop implementation running on 12 cores.
- **ratio:** indicates how much slower the looping implementation is than the 12-core Pochoir implementation
- **p** in dims means periodic

Comparison

- The Berkeley **autotuner** focuses on optimizing the performance of stencil kernels by automatically selecting tuning parameters. Their work serves as a good benchmark for the maximum possible speedup one can get on a stencil.
- 7-point stencil and a 27-point stencil on a 2583 grid with "ghost cells"
- "Unfortunately, we were unable to reproduce the reported results from — presumably because there were too many differences in hardware, compilers, and operating system "

Comparison

	Berkeley	*Pochoir*
CPU	Xeon X5550	Xeon X5650
Clock	2.66GHz	2.66 GHz
cores/socket	4	6
Total # cores	8	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
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

Conclusion

- Easier to write parallel cache efficient stencil program.
- Two phases methodology
- Trapezoid decomposition with hyperspace cut

Some questions

- Compare with hand-written parallel cache efficient algorithms?
- Doesn’t support irregularly shaped domains.
- Performance decomposition?
- Performance of dimension > 3?
- Scalability