Correctness Verification and Boundary Conditions for Chapel Iterator-Based Loop Optimization

Daniel Feshbach Haverford College, Dept. of Computer Science Advisor: David G. Wonnacott

HAVERFORD COLLEGE

Overview

This poster explores two issues of correctness concerning iteration space transformation techniques, and presents:

(1) A data structure for automatically verifying correctness with respect to **data dependencies**

(2) Three approaches to ensuring obedience to **boundary conditions**, and performance results comparing each to ignoring this condition

Background

- Stencil Computations
- Iteration Space Transformations
- Issues:
  - Correctness
  - Performance
  - Flexibility

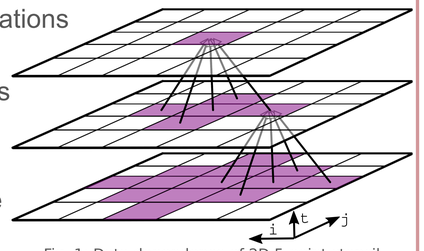
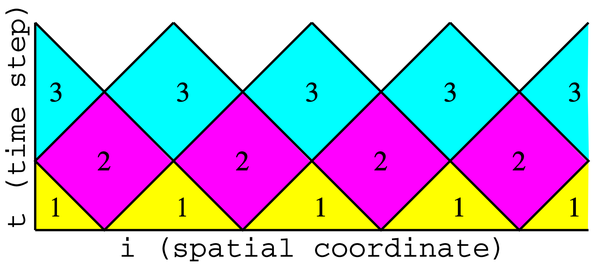


Fig. 1: Data dependence of 2D 5-point stencil

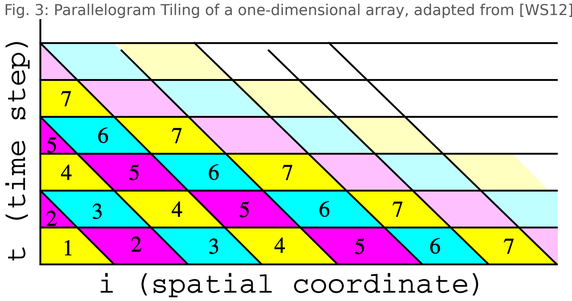
Diamond Tiling 1D

Fig. 2: Diamond Tiling of a one-dimensional array, adapted from [WS12]



Parallelogram Tiling 1D

Fig. 3: Parallelogram Tiling of a one-dimensional array, adapted from [WS12]



Correctness Verification of Data Dependencies

- Previous work [BOHCWS 15]: Chapel iterators can be used to express iteration space transformation techniques
  - Fast when other techniques are fast
  - More flexible for experimentation and maintenance
- Now: Technique to make these less error-prone by allowing for verification of data dependence correctness

CleverArray and CarefulArray

387.5	750	489.0	131.2
192.1	575	500	125
0	0	125	0
0	0	0	0

450 1100 575 87.5 / 125 / 31.2 / 0

Values

3	3	3	3
3	1	1	1
1	1	1	1
1	1	1	1

2 2 2 2 / 2 / 2 / 2

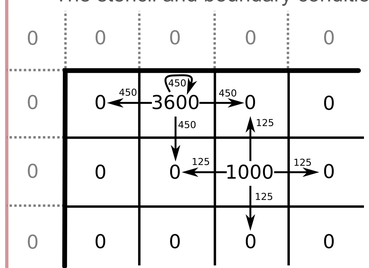
Time Stamps

Fig. 4: CarefulArray Data Structure

- CleverArray is NxNx2
- Keeps track of values at still-relevant time steps only, to reduce memory traffic
- For this size stencil, only need to store most recently calculated odd-time and even-time values at each location
- Fast, but vulnerable to unnoticed data dependence errors
- CarefulArray inherits CleverArray
- Additional NxNx2 array stores time stamps from when each data location was most recently written to
- When reading values, automatically checks that they were written at the expected time step
- Verifies (or detects errors in) data dependencies

Jacobi 2D with edge conditions

The stencil and boundary condition we use for our tests




```
A[t,i,j] =
(k*A[t-1,i,j] +
  A[t-1,i-1,j] +
  A[t-1,i,j-1] +
  A[t-1,i+1,j] +
  A[t-1,i,j+1])/8.0;
```

k = 8 - the number of neighboring cells

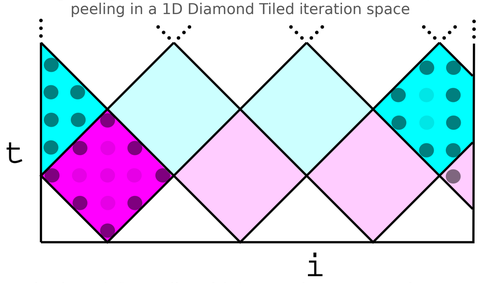
Fig. 5: The boundary condition we apply to our 5-point stencil

Techniques to Correctly Handle Boundary Conditions

- Expands the domain of Iteration Space Transformations to include boundary conditions expressible in terms of the number of neighboring cells
- Our coefficient is *k* = 8 - number of neighbors
- Techniques: If Statements, Loop Peeling, Coefficient Array

Loop Peeling

Fig. 6 Edge tiles and candiate cells identified by loop peeling in a 1D Diamond Tiled iteration space



- Only calculate *k* for cells which may be on an edge or corner: yield *k* = 4 for the rest
- When the iterator implementing the transformation yields the cell coordinates, also yield the appropriate *k*
- Within each wavefront (group that can be executed in parallel) of tiles, peel off the first and last tile in each dimension: these are ‘edge tiles’
- Within each dimension of each edge tile, peel off the first and last cells: these are ‘candidate cells’
- Use if statements to check if each candidate cell is on an edge or corner of the whole array: yield *k* = 8 - number of neighbor

Coefficient Array

387.5	750	489.0	131.2
192.1	575	500	125
0	0	125	0
0	0	0	0

450 1100 575 87.5 / 125 / 31.2 / 0

Values

6	5	5	6
5	4	4	5
5	4	4	5
6	5	5	6

8 - number of neighbors

Fig. 7: Coefficient Array Data Structure

- Stores the appropriate *k* value for each spatial location in an NxN array
- When updating a cell, read to the coefficient array at the spatial coordinates to obtain *k*

Performance Results

Compared to benchmark of ignoring the boundary condition ("Ignoring Edges")
T is the number of time steps, the array used is size NxN

Key Result: the Coefficient Array technique has the fastest performance

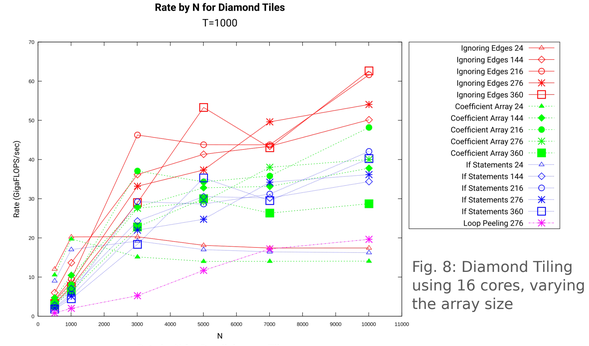


Fig. 8: Diamond Tiling using 16 cores, varying the array size

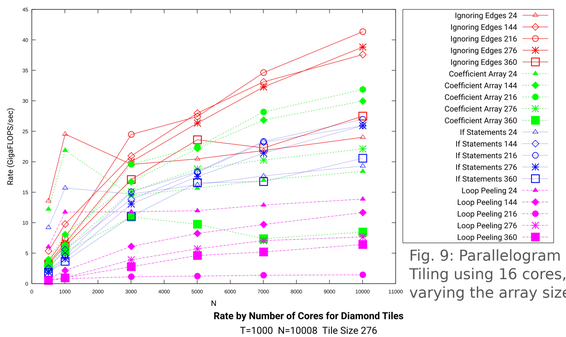


Fig. 9: Parallelogram Tiling using 16 cores, varying the array size

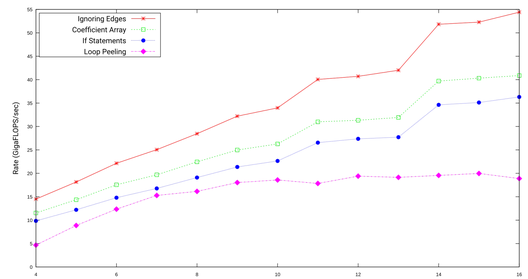


Fig. 10: Diamond Tiling, varying the number of cores

Related Works:

[WS 12] D. G. Wonnacott and M. M. Strout. On the scalability of loop tiling techniques. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, January 2013.

[BOHCWS 15] I. J. Bertolacci, C. Olschanowsky, B. Harshbarger, B. L. Chamberlain, D. G. Wonnacott, and M. M. Strout. Parameterized Diamond Tiling for Stencil Computations with Chapel parallel iterators. In *Proceedings of the International Conference on Supercomputing (ICS)*, October 2015.

Future Work:

- Compare to C/C++ and Pluto
- Explore additional tilings and applications
- Review prior work with ghost cells and other ways of handling edges
- Explore the details of the peeled tiled iterators, to see if they can be improved