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

Abstract and Summary

Daniel Feshbach
Haverford College
Haverford, PA
dfeshbach@haverford.edu

David G. Wonnacott (Advisor)
Haverford College
Haverford, PA
davew@cs.haverford.edu

ABSTRACT

We explore two issues of correctness concerning iteration space transformation techniques: data dependencies and boundary conditions. First, we present a data structure which automatically verifies correctness of data dependencies for stencil computations with transformed iteration spaces. This further confirms the viability of Chapel iterators for defining iteration space transformations, by demonstrating that simple tool support can verify data dependencies and assist debugging. Second, we explore the performance and simplicity of three strategies for implementing boundary conditions in transformed iteration spaces: if statements, loop peeling, and an array of coefficients. Testing their performance against the benchmark of ignoring the boundary condition, we find that the coefficient array technique performs the best, often at 70 to 80 percent speed, with if statements not far behind and loop peeling much worse. The coefficient array and if statements are indifferent to the transformation technique applied, while loop peeling must be implemented within the transformation.

KEYWORDS

Performance Measurement, Modeling, and Tools: Analysis, modeling or simulation methods

Programming Systems: Programming language techniques for reducing energy and data movement; Tools for parallel program development; Program analysis, synthesis, and verification to enhance cross-platform portability, maintainability, result reproducibility, resilience

ACM Reference format:
Daniel Feshbach and David G. Wonnacott (Advisor). 2017. Correctness Verification and Boundary Conditions for Chapel Iterator-Based Loop Optimization. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, November 2017 (SC'17),* 2 pages.
https://doi.org/10.475/123_4

SUMMARY

This summary presumes familiarity with BOHCWS 15 [1].

SC'17, November 2017, Denver, CO, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.475/123_4

Correctness Verification of Data Dependencies

We present further support for the viability of Chapel iterators as a means to implement iteration space transformations, by showing how simple tool support can verify correctness and assist debugging. We show this by presenting a data structure that keeps track of data dependencies and automatically verifies the correctness of each read to the array.

For performance reasons, we parameterize the number of time-steps S to keep track of in the time dimension of the array Vals of values, and read time-coordinates into Vals modulo S. We store Vals in a class so that neither this performance-tuning nor adjusting S based on context sacrifices clarity or modularity. For correctness verification, we make an additional array, Times, of the same size as Vals, which stores stamps of the time steps at which the corresponding positions in Vals were written. This allows us to check at run-time if data dependence is as expected. It takes negligible effort to switch between one class, which includes Times and verifies correctness, and another with only Vals which runs as fast as built-in Chapel arrays.

This tool was invaluable for debugging in our following work, and would presumably help other projects implementing iteration space transformations. It also furthers the viability of iterators to express these transformations, because the ability to easily verify correctness without engaging with the internal details of the transformation is at its most useful in many of the same situations when iterators are the best way to organize code (i.e, when reusing code or experimentally swapping between transformation techniques).

Techniques for Handling Boundary Conditions

We used a modified Jacobi 2D 5-point stencil, thinking about it as heat flow. Half of the heat stays put while the rest splits in equal parts to move in each direction, 'bouncing back' inwards at boundaries. To avoid reading illegal indices, we surround the array with 'ghost cells' set to 0 (the additive identity), updating within 1 through N in each spatial dimension while the array is 0 to N+1. We update each cell with the formula

$$\begin{aligned} A[t,i,j] = (k * A[t-1,i,j] \\ + A[t-1,i-1,j] \\ + A[t-1,i,j-1] \\ + A[t-1,i+1,j] \\ + A[t-1,i,j+1])/8 \end{aligned}$$

where k is 8 minus the number of (non-ghost) adjacent cells: 6 for corners, 5 for edges, 4 elsewhere. The idea is that for each adjacent

boundary we add one additional multiple of the cellâĂŹs own value to stay put. This lets us use one computation to explore the different approaches.

Our approaches to finding k should be generalizable to any stencil computation for which there exists an identity value of the operation being applied directly to the neighbor cells, and for which the boundary condition can be parameterized so the same formula can update each cell. We have three such approaches:

(1) Use if statements in the main code (where the cells are updated, rather than in the iterator) on the spatial coordinates to set k. If both i and j are on edges (equal to 1 or N) then k is 6; else if either of them is then k is 5; else k is 4.
(2) Apply loop peeling in the iterator to identify 'candidate' tiles and cells which could be on edges, calculate k for each candidate cell and yield it along with the coordinates, and yield k=4 for the rest of the cells. While the other two approaches may be applied with no additional effort to any iteration space transformation technique, this requires rewriting each new loop nest.
(3) Store the k for each spatial coordinate in a 'coefficient array,' and access it inline when we use k. For our application this is a 2D array from 1 to N in each dimension with 6 in the corners, 5 on the edges, and 4 in the center.

We tested these approaches against the common benchmark approach of ignoring the boundary condition entirely. Our experiments varied tile sizes, array sizes, and number of cores, and involved both diamond and parallelogram tiling patterns. We found that the coefficient array technique performs the best, at around 75 to 80 percent of the speed of ignoring the edges for the largest examples. The if statement approach is not far behind this performance, often at 65 to 70 percent of the ignoring edges speed, while our implementation of loop peeling is substantially slower, at less than half the speed of ignoring the edges.

Future Work

Areas for future work include comparing the performance to C/C++ and Pluto, exploring additional tilings and other applications, reviewing prior work about ghost cells and other ways of handling edges, and seeing if the peeling technique can be improved.

REFERENCES

I. J. Bertolacci, C. Olschanowsky, B. L. Chamberlain B. Harshbarger, D. G. Wonnacott, and M. M. Strout. 2015. Parameterized Diamond Tiling for Stencil Computations with Chapel parallel iterators. *Proceedings of the 29th ACM International Conference on Supercomputing* (2015), 179–206. https://doi.org/10.1145/2751205.2751226