Smashing: Folding Space to Tile through Time

Nissa Osheim, Michelle Mills Strout, Dave Rostron, and Sanjay Rajopadhye

Colorado State University, Fort Collins CO 80523, USA

Abstract. Partial differential equation solvers spend most of their computation time performing nearest neighbor (stencil) computations on grids that model spatial domains. Tiling is an effective performance optimization for improving the data locality and enabling course-grain parallelization for such computations. However, when the domains are periodic, tiling through time is not directly applicable due to wrap-around dependencies. It is possible to tile within the spatial domain, but tiling across time (i.e. time skewing) is not legal since no constant skewing can render all loops fully permutable. We introduce a technique called smashing that maps a periodic domain to computer memory without creating any wrap-around dependencies. For a periodic cylinder domain where time skewing improves performance, the performance of smashing is comparable to another method, circular skewing, which also handles the periodicity of a cylinder. Unlike circular skewing, smashing can remove wrap-around dependencies for an icosahedron model of earth's atmosphere.

1 Introduction

Many computational science applications iterate over a discretized spatial domain to model its change over time or to converge to a steady-state solution for unknowns within the discretized space. Regular computations are those where the discretization of the simulation space is done with a one, two, or three-dimensional grid whose values can be stored in an array of the same dimensionality. This paper focuses on the issues that arise when the discretized domain is periodic. The wrap-around dependencies that result from periodicity make it difficult to use a well known program optimization called time skewing. This paper presents a technique called smashing that folds the data space so that all dependencies in the corresponding iteration space are uniform, thereby enabling time skewing. Although other techniques such as circular skewing enable time skewing for some periodic domains, smashing applies more generally and exhibits comparable overhead.

In a periodic domain, some of the points on the grid boundary are simulation space neighbors to points on a different boundary in the discrete grid. Fig. 1 shows the code and the iteration space for a computation that iterates over a ring. The ring has been unrolled and discretized into a one-dimensional array. Notice the long, or wrap-around, dependencies going from iteration point $(1, 1)$

to $(2, 8)$ and $(1, 8)$ to $(2, 1)$. [1] Periodic domains arise from modeling hollow objects or shells. Since they are hollow, shells can be represented in one less dimension. Typically, the object is cut and unrolled to remove one dimension. For example, ring is cut at one point and unrolled to form a line. A line is easy to map into data and iteration domains, but the top and bottom points of the line must still be treated as neighbors in simulation space. A cylinder or torus can be represented with a plane.

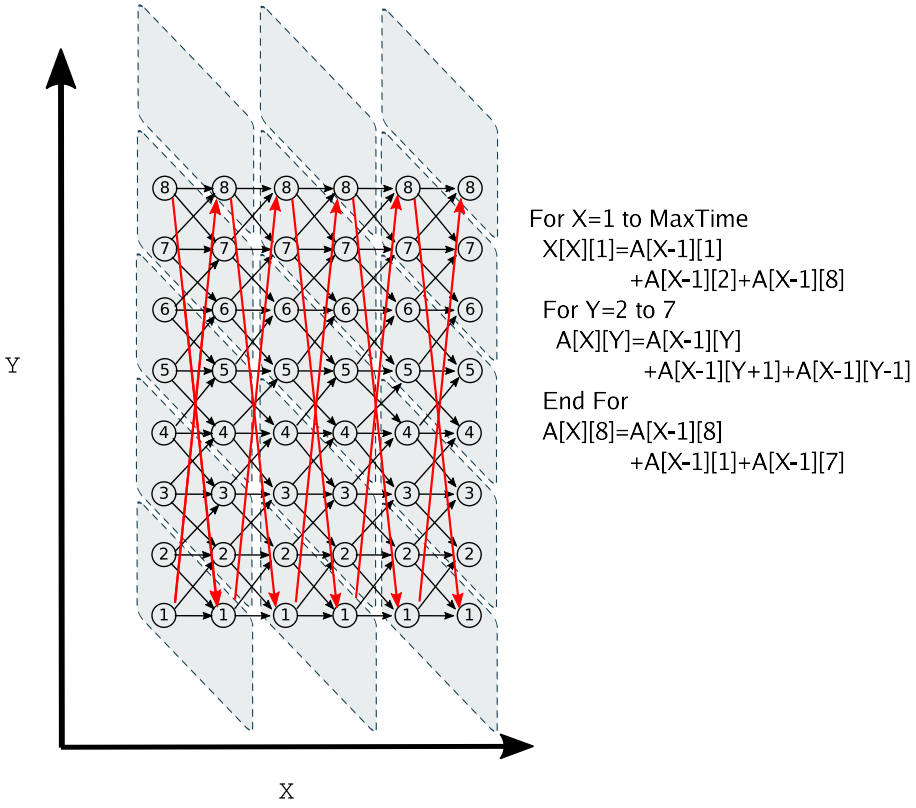


Fig. 1. Iteration space of modeling a ring through time. The ring has been cut and straightened. The vertical axis represents the points on the ring. The horizontal axis represents time. Points are dependent on their neighbors in the previous time step. The dependencies are shown with arrows. Because the top and bottom points are neighbors, there are long dependencies from the top to the bottom and the bottom to the top. The code has not been tiled.

[1] The code in Fig. 1 has been written using single assignment so that the dependencies are clear, but actual implementations only use two one-dimensional arrays.

Shells that result in periodic domains are frequently used to model physical phenomena such as the earth's atmosphere and surface. For example, the earth's atmosphere has been modeled with an icosahedron [11] and a cubed-sphere [1]. Fig. 2 shows how an icosahedron is used to discretize one layer of the atmosphere and how it can be unrolled into a set of two-dimensional grids.

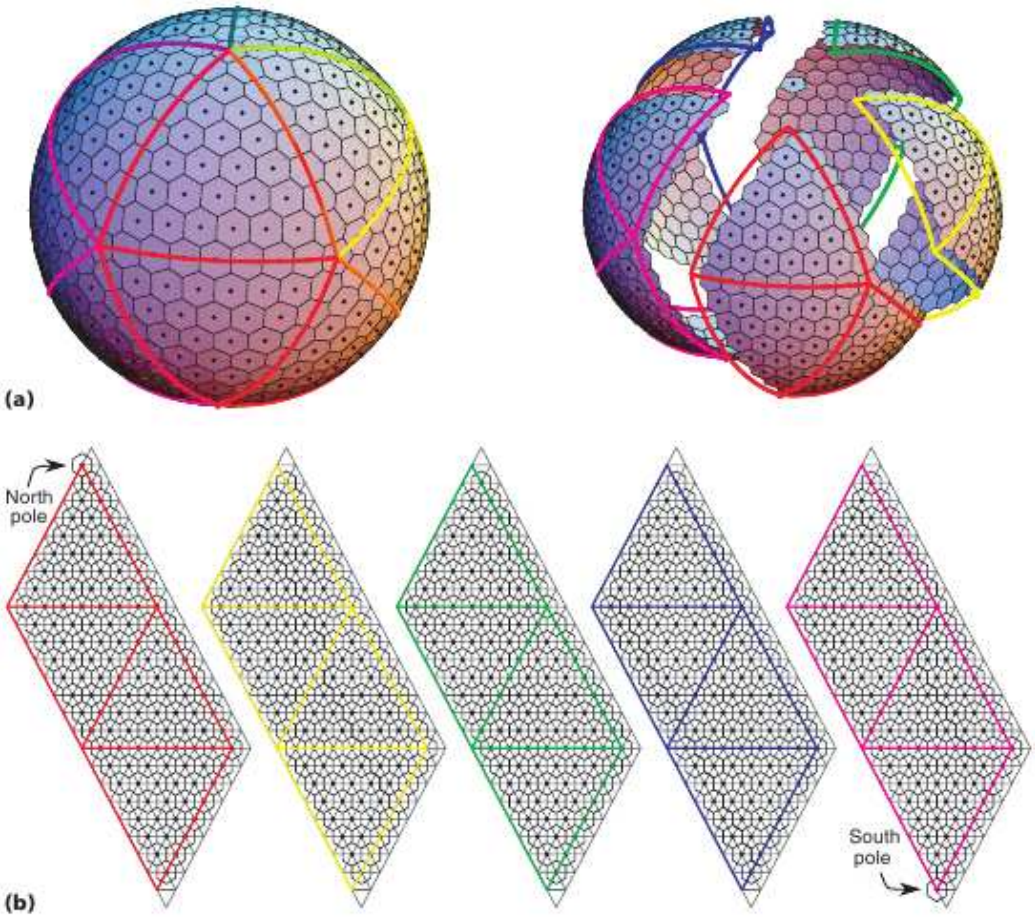
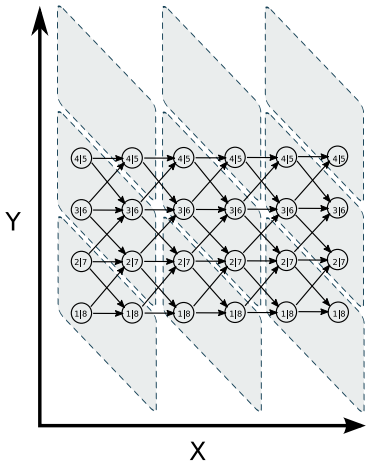


Fig. 2. Icosahedron cut into five parallelograms. Figure courtesy of Randal, Ringer, Heikes, Jones, and Baumgardner [11]

Since stencil computations over periodic domains occur in many important applications, there is significant interest in improving their performance through data locality improvements and parallelization. Stencil computations can be modeled as an iteration space where each point in the space represents one iteration of the loop. This space can then be tiled, breaking up the computations

in an attempt to group those that use the same data together thereby increase data locality. Tiling as a performance optimization has been studied extensively and has been shown to make code more efficient [15, 6, 12, 16].

Tiling these domains through time is difficult because the traditional strategy of performing a uniform time skewing does not remove the cycle of dependencies between tiles. They have non-uniform dependencies in nearly opposite directions. Fig. 1 shows the example of trying to tile a ring as it is modeled through time. The vertical axis represents the points on the ring. The horizontal axis represents time. Because the top and bottom points are neighbors, there are long dependencies from the top to the bottom and the bottom to the top. Any two-dimensional tiling scheme based using only unimodular skewing as a preprocessing step will create tiles with circular dependencies. There is no way to legally schedule tiles with circular dependencies, thus space can not be tiled through time.




```
For X=1 to MaxTime
  A[X][1][0]=A[X-1][1][0]+A[X-1][2][0]+A[X-1][1][1]
  A[X][1][1]=A[X-1][1][1]+A[X-1][2][1]+A[X-1][1][0]
  For Y=2 to 3
    A[X][Y][0]=A[X-1][Y][0]+A[X-1][Y+1][0]+A[X-1][Y-1][0]
    A[X][Y][1]=A[X-1][Y][1]+A[X-1][Y+1][1]+A[X-1][Y-1][1]
  End For
  A[X][4][0]=A[X-1][4][0]+A[X-1][4][1]+A[X-1][3][0]
  A[X][4][1]=A[X-1][4][1]+A[X-1][4][0]+A[X-1][3][1]
End For
```

Fig. 3. Iteration space that results from the smashed ring data space. Each node corresponds to two points from the original computation. Notice that the tiles have no cyclic dependencies. Although tiling is legal, the displayed code has not been tiled.

In this paper, we resolve these difficulties through a new technique called smashing that removes all the non-uniform neighbor relationships from the periodic domains that result from rings, cylinders, tori, and icosahedra. Essentially, it is a "data allocation" technique rather than a loop/iteration transformation. In effect, we allocate the data by defining a map from the simulation space to memory in such a way that all dependencies, including the periodic ones are strictly uniform. Once the neighbor relationships in the data space are all uniform, the dependencies in the resulting iteration space are also uniform and therefore amenable to unimodular skewing to enabling tiling through time, or time skewing. Fig. 3 shows the iteration space that results from the smashed ring data space. Notice that all of the dependencies are uniform.

Section 2 reviews the motivation for time skewing and previous techniques for enabling time skewing in the presence of periodic domains. Section 3 presents the smashing technique and shows how it works for the ring, cylinder, torus, and icosahedron. Section 4 shows that the overhead of circular skewing and smashing on a cylinder are both reasonable and comparable. Section 5 concludes.

2 Related Work

Related work includes research on applying time skewing, or tiling through time, to stencil computations and techniques that enable tiling through time despite wrap-around dependencies due to periodicity.

Tiling through time to improve data locality in non-periodic, stencil computations has been studied and shown to improve performance in numerous contexts. Wolf and Lam [15] showed how the uni-modular transformations skewing, reversal, and permutation could be applied to perfectly nested stencil loops such as SOR to enable tiling over time. Basetti et al. [3] use the term "temporal blocking" for the way they do multiple time steps by having more than one layer of ghostcells in a structured mesh. Douglas et al. [4] modify the smoother in structured and unstructured multigrid implementations so that pieces of multiple time steps that access similar data locations are performed to improve data locality. Ahmed et al. [2] are able to tile across time for computations with imperfectly nested loops such as those that occur in the Jocobi stencil computation. They do this by embedding all of the iterations in the computation into a single product space and performing tiling within that space. Sellappa and Chatterjee [13] perform a temporal blocking across the time steps in Red-Black Gauss-Seidel. Wonnacott [18] shows how to modify the storage mapping and time skew the computation to ensure scalable locality, where scalable locality indicates that a constant tile size may be selected so that the computation experiences data locality no matter how large the problem size parameters grow.

Some previous research has also looked at tiling across time when there were periodic boundaries in the simulation space.

Jin et al. [7] present at technique called recursive prismatic time skewing. They calculate a reverse skewing factor to deal with periodic boundaries. The reverse skewing factor is used to prevent the computation of boundary iteration points at the start of the spatial domain that need computational results from previous iterations of points later in the spatial domain. The code generated computes these boundary iteration points as soon as the dependencies allow. Their technique is applicable to any (hyper) rectangular domain where the periodic boundaries are each pair of parallel sides in the (hyper) rectangle. Therefore, this technique applies to the ring, cylinder, and torus, but not the cubed sphere or icosahedron.

Periodic domains have also been tiled using rhombus shaped tiles [5, 8, 10]. These tiles overlap at their bases, causing two tiles to compute some duplicate results. Each tile computes all the data it will need for subsequent time steps itself. The iteration space does not need to be skewed, and the overlapping rhombus

tiles can start execution at the same time, thus resulting in no start-up cost [10]. These benefits can be enough to overcome the extra time spent recomputing a portion of the iterations. Overlapping tiles can also be used to handle wrap-around dependencies that are introduced due to periodic boundaries [5, 8]. The presented techniques could feasibly be extended to handle the ring, cylinder, and torus domains. More complex periodic boundaries such as the cubed sphere and the icosahedron would require special logic to handle the varying directions of periodicity they exhibit.

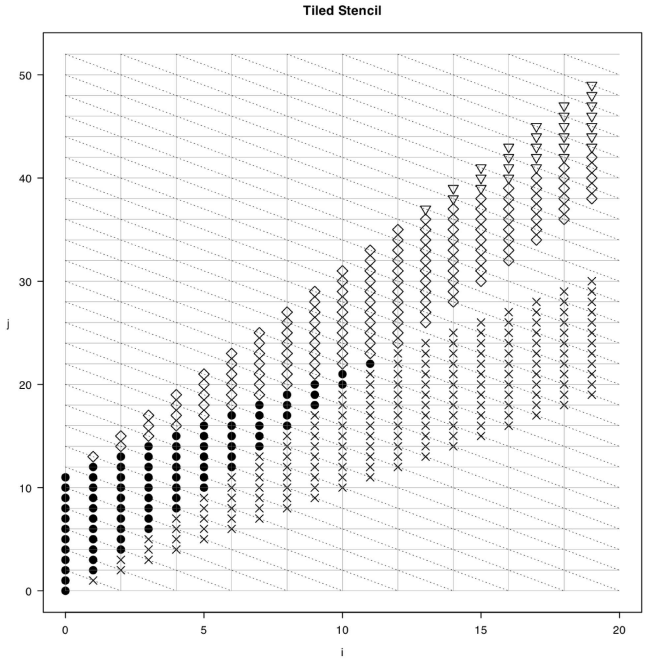


Fig. 4. Example of circular skewing on a ring. The horizontal axis is the time dimension. The vertical axis represents the spatial domain of the ring. Initially, the wrap-around causes non-uniform dependencies between the top and the bottom of this space. The iteration points marked with x are the problem points that do not allow tiling through time. The x points are translated by N (size of the ring) up to the diamonds. At each time step, the number of points that need to be translated increases because more points from the previous time step are translated. The triangles represent the points that need to be translated multiple times. The background grid represents 2x2 tiles.

Song and Li [14] used the circular skewing technique introduced by Wolfe [17] to make all of the long dependencies due to periodic domains point in the same direction and therefore enable tiling through time. They formalized circular skewing only in the case where the spatial domain was one-dimensional. Circular

skewing plus unimodular skewing enables tiling through time. Fig. 4 shows that a skewed iteration space (lower points) can be modified with circular skewing to ensure that the resulting loop is fully permutable and therefore tileable. It removes the negative non-uniform dependencies and creates positive non-uniform dependencies. With all non-uniform dependencies going in the same direction, it is possible to tile legally. It is possible to extend circular skewing to the cylinder and torus domains, but the technique does not extend to the cubed sphere or icosahedron.

Smashing differs from previous work that enables tiling through time despite periodic dependencies due to the fact that it transforms the data space to prevent such periodic dependencies instead of transforming the iteration space to deal with them after the fact. As such, after smashing, techniques such as overlapping tiles [5, 8, 10], which enable load-balanced parallelism, are applicable to the resulting iteration space.

3 Smashing

Smashing is a new storage mapping method that prevents wrap-around dependencies due to periodicity, which occurs when modeling hollow objects. Given a hollow object, one dimension needs to be removed to prevent iterating over the empty space inside the object. Normally this dimension is removed by cutting and unrolling. This creates wrap-around or non-uniform neighbors from one side of the cut to the other. Smashing treats the data domain more like the object it represents and smashes or flattens it to remove the extra dimension. This can also be done by proceeding with the unrolling as is typical and then folding the unrolled data space. For example, the unrolled ring creates a line, this line is folded in half to create the smashed ring. Whether the data transformation is done by smashing or folding, the end result is the same data space. It has multiple layers but no wrap-around. In the ring example, the single fold causes two layers. These layers can be stored as separate arrays or as a single two dimensional array where the inner dimension has a size of two.

To explain how this removes the wrap-around dependencies in the iteration space, we use the concept of neighbors in the data space. Neighbors are points that are near each other in simulation space. The stencil defines the neighbors of interest. In the data space a neighboring relationship is uniform if the neighbors are a constant distance from one another. Non-uniform neighbors in the data space cause the non-uniform dependencies in the iteration space. Smashing creates a data space with only uniform neighboring relationships. A data space with uniform neighbors can be mapped to an iteration space, skewed, and tiled using the same methods that work on non-periodic domains. The remaining subsections show the details of how smashing creates a uniform neighbor relation for a ring, torus, and icosahedron.

3.1 Ring

Here we smash the data space of the ring and show how it removes the non-uniform neighbor relations. The data space of the unrolled ring is a one dimensional array ranging from 0 to N-1, where N is the number of discrete points in the ring. We specify the neighbors in this data space as a piecewise affine function. For this example, we assume a stencil that uses the left and right adjacent points.

$$\text{Right}(i) \;=\; \begin{cases} i = N-1 : (0) \\ i < N-1 : (i+1) \end{cases}$$

$$\text{Left}(i) \;=\; \begin{cases} i = 0 : (N-1) \\ i > 0 : (i-1) \end{cases}$$

The above neighbor functions contain non-uniformity. For the $Right()$ neighbor function, when $i = N-1$, its right neighbor is 0. The distance between them is $N-1$ and is thus dependent on the number of points in the ring and non-uniform. Likewise, when $i = 0$ its left neighbor is $N-1$. Smashing folds this data space into a two-dimensional data space with the following transformation:

$$Smash(i) \;=\; \begin{cases} i < N/2 : (i, 0) \\ i \geq N/2 : (N-1-i, 1) \end{cases}$$

We assume that N is even.

This creates a two dimensional array where the second dimension has two possible values. After this transformation, the neighbor function in the transformed data space is as follows:

$$Right(i,k) \;=\; \begin{cases} i = N-1,\ k = 0 : (i, k+1) \\ i < N-1,\ k = 0 : (i+1, k) \\ i = 0,\ k = 1 \quad\ : (i, k-1) \\ i > 0,\ k = 1 \quad\ : (i-1, k) \end{cases}$$

$$Left(i,k) \;=\; \begin{cases} i = 0,\ k = 0 \quad\ : (i, k+1) \\ i > 0,\ k = 0 \quad\ : (i-1, k) \\ i = N-1,\ k = 1 : (i, k-1) \\ i < N-1,\ k = 1 : (i+1, k) \end{cases}$$

Notice that all the neighbors are now constant offsets from the data space point (i, k).

3.2 Torus

A torus consists of a cylinder that has been rolled into a three-dimensional ring. The top and bottom of the cylinder are connected, creating an object that looks like a donut. To smash this we set the torus on edge and flatten it resulting in four rectangles set on top of each other. Fig. 5 visualizes this another way. Cut the torus vertically into two even halves. Straighten the two halves into cylinders. Cut each cylinder along the long edge into two even halves, and straighten these four halves into rectangles. The folding method results in the same four-layered rectangle.

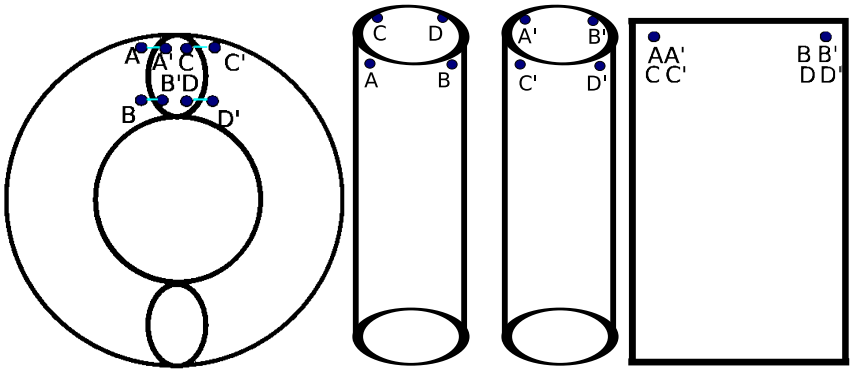


Fig. 5. Smashing a torus by cutting it into two cylinders

The unrolled torus is a single rectangle with wrap-around from the north to south and from the east to west. The folding method folds this plane in half horizontally and again vertically resulting in four layered rectangles of one-fourth the size of the original. The points that were non-uniform neighbors on the edge of the unrolled rectangle before folding are now within three layers of each other on the edge of the folded rectangle.

The neighborhood of the unrolled torus is described with the following piecewise affine functions:

$$North(i,j) \;=\; \begin{cases} j = M-1 : (i,0) \\ j < M-1 : (i,j+1) \end{cases}$$

$$South(i,j) \;=\; \begin{cases} j = 0 : (i,M-1) \\ j > 0 : (i,j-1) \end{cases}$$

$$East(i,j) \;=\; \begin{cases} i = N-1 : (0,j) \\ i < N-1 : (i+1,j) \end{cases}$$

$$West(i,j) \;=\; \begin{cases} i = 0 : (N-1, j) \\ i > 0 : (i-1, j) \end{cases}$$

where N is the size of the domain in the i dimension, and M is the size in the j dimension.

Unrolling introduces non-uniform neighbors as it did in the ring example. The north neighbors of the points (i,M-1) are the points (i,0). These points are a distance of M-1 from each other. The points along the other edges also have non-uniform neighbors. Smashing folds the two-dimensional, unrolled data space into a three-dimensional data space with the following transformation:

$$Smash(i,j) \;=\; \begin{cases} i < N/2,\ j < M/2 : (i, j, 0) \\ i \geq N/2,\ j < M/2 : (N-1-i, j, 1) \\ i < N/2,\ j \geq M/2 : (i, M-1-j, 2) \\ i \geq N/2,\ j \geq M/2 : (N-1-i, M-1-j, 3) \end{cases}$$

N and M must be even.

The new neighbor functions are:

$$North(i,j,k) \;=\; \begin{cases} k < 2,\ j = M/2-1 : (i, j, k+2) \\ k \geq 2,\ j = 0 \quad\quad\ : (i, j, k-2) \\ k < 2,\ j < M/2-1 : (i, j+1, k) \\ k \geq 2,\ J > 0 \quad\quad\ : (i, j-1, k) \end{cases}$$

$$South(i,j,k) \;=\; \begin{cases} k < 2,\ j = 0 \quad\quad\ : (i, j, k+2) \\ k \geq 2,\ j = M/2-1 : (i, j, k-2) \\ k < 2,\ j > 0) \quad\quad\ : (i, j-1, k) \\ k \geq 2,\ J < M/2-1 : (i, j+1, k) \end{cases}$$

$$East(i,j,k) \;=\; \begin{cases} k = 0 \text{ or } 2,\ i = N/2-1 : (i, j, k+1) \\ k = 1 \text{ or } 3,\ i = 0 \quad\quad\ : (i, j, k-1) \\ k = 0 \text{ or } 2,\ i < N/2-1 : (i, j+1, k) \\ k = 1 \text{ or } 3,\ i > 0 \quad\quad\ : (i, j-1, k) \end{cases}$$

$$West(i,j,k) \;=\; \begin{cases} k = 0 \text{ or } 2,\ i = 0 \quad\quad\ : (i, j, k+1) \\ k = 1 \text{ or } 3,\ i = N/2-1 : (i, j, k-1) \\ k = 0 \text{ or } 2,\ i > 0 \quad\quad\ : (i, j-1, k) \\ k = 1 \text{ or } 3,\ i < N/2-1 : (i, j+1, k) \end{cases}$$

The smashed torus has uniform neighbor relations, and the resulting iteration space can be tiled without dealing with wrap-around dependencies.

3.3 Icosahedron representation of the earth

Modeling a sphere is difficult since a curve is hard to discretize. Randall et al. [11] use a geodesic grid to model the earth's atmosphere. They start with an icosahedron and repeatedly bisect the faces to create increasingly finer grids. We smash the icosahedron and remove all non-uniform neighbors. To do this, we start with the unrolled icosahedron specified by Randall et al. [11], which is made up of 20 triangles, or alternatively, of 10 rhombi (diamonds) of two triangles each, or even five parallelograms of four triangles each (see Fig. 2). The first and second parallelograms are reconnected where their third and second triangles, respectively, were connected in the original icosahedron (see Figures 7 and 8). The third, fourth, and fifth parallelograms are similarly reconnected. This creates a single polygon, but with non-uniform neighbors.

The non-uniform neighbors in the unrolled icosahedron can be made uniform by folding (see Figures in Appendix A). We make ten vertical folds along the ten main (i.e., longer) diagonals of the ten rhombi. The folds are made "accordion style" so that we end up with ten long parallelograms laid out on top of each other in ten layers (actually there are eleven layers – the first triangle, nine parallelograms, and a last triangle, but the triangles can be "joined" together to form a single parallelogram). The points that were non-uniform neighbors are now at the same point of the new equilateral, but on a different layer from their neighbor. To help visualize this, the reader is encouraged to cut out the Figures in Appendix A and do some origami. The new data space can now be skewed and tiled through time because all neighbors are an uniform distance from each other.

4 Experimental Results

To test the viability of smashing, we use a cylindrical domain and perform a nearest neighbor stencil computation over time. Smashing enables tiling through time, but introduces overhead in terms of accessing the data space. Our experiments show that the overhead for smashing on a cylinder is comparable with the overhead of circular skewing and that both result in an overall performance benefit because they enable tiling over time.

In the stencil computation test, the neighbors are the north, south, east, and west points, and the calculations (averaging the neighbors) are done using the values of the neighbors from the previous time step. We iterate over this domain with four separate methods, no tiling, tiling in two dimensions, tiling through time using circular loop skewing, tiling through time using smashing. The results for no tiling, circular loop skewing, and smashing are shown in fig. 6. Tiling in two dimensions did not improve the results over not tiling for this domain, therefore we do not include them in fig. 6.

We tried various tile sizes to find a reasonable tile size for each method. We then compare the methods using the tile size that is the best we found for each method. Our goal was not do determine what the optimal tile size is for

the selected problem sizes, but instead to show that with a good tiling, the overhead of smashing competes with the less general circular skewing. In the time dimension the tiles sizes ranged from 2 to 64. In the space dimension we have square tiles ranging from 2 to 1024 on a side. We also tried not tiling the inner loop. Kamil et al. [9] recommend not tiling the inner loop because of the prefetcher. We tried multiple tile sizes and not tiling the inner loop was the best strategy for all the methods except smashing on the 4000 size domain.

For our final results we ran the tile sizes that performed well for each method and domain twenty times and took an average to produce Fig. 6. With this wide variety of tile sizes, we can be reasonably sure that our choice of tile size is not distorting the results. Table 1 shows the tiles size used for each method and domain. The first dimension is time and the last is the inner loop. When the size of the last dimension is zero, it means we did not tile the inner loop. The columns show the problem size. We used the same tile size for problem the 1000, 1500, and 2000 problem sizes. They are in one column (1000-2000).

Table 1. Tile Sizes

Method	**1000-2000**	**4000**
Circular Skew	**16x2x0**	**8x1x0**
Smashing	**8x2x0**	**64x64x64**

Tiling through time with both circular loop skewing and smashing shows a speedup over not tiling through time. The times are comparable between circular loop skewing and smashing. These results show that smashing does not introduce more overhead than less general techniques for enabling time skewing on computations with periodic domains.

5 Conclusion

We have proposed, analyzed, and implemented in a number of examples a data mapping technique called smashing. Smashing is an enabling transformation that allows a well known and useful tiling transformation—time skewing—to be applied to stencil computations for domains with periodic boundary conditions. Unlike previous techniques that address such periodic domains, smashing is not an iteration space transformation of a given piece of code. Rather, it can be viewed as a piecewise affine memory map—allocation of discretization points of the physical domain being modeled to memory locations that are viewed as a contiguous multidimensional array. Our main result is to show that for many practical cases smashing preserves the property that the neighbors of any point in the physical domain remain neighbors in the multidimensional memory. As a

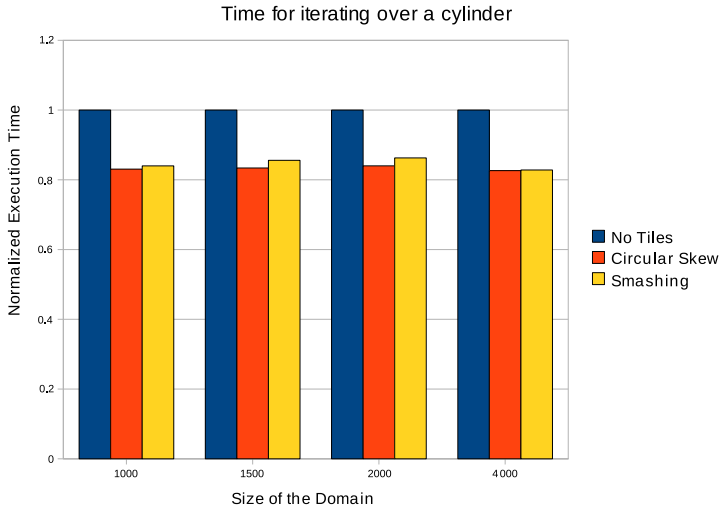


Fig. 6. Graph of the execution times for the cylinder using no tiling, circular skewing, and smashing. The iteration spaces are all three-dimensional. The graph has been normalized to the execution time of the no tiling method.

result, any iterative stencil computation on this data space automatically enjoys the uniform dependence properties that allow direct implementation of time skewing.

Smashing is easy to describe and implement, and as we have shown, provides performance comparable to circular skewing. It removes the wrap-around from the data space before it is converted to the iteration space and works for domains such as the cube and icosahedron that earlier techniques like circular skewing cannot handle.

An open question that we are investigating is a proof of generality: under what conditions can an arbitrary "hollow" physical object be smashed down to a contiguous multidimensional array while retaining the neighborhood property. In addition we are developing tools to automatically generate code that incorporates the smashing transformation from high-level stencil specifications for hollow objects.

References

1. A. Adcroft, J.-M. Campin, C. Hill, and J. Marshall. Implementation of an atmosphere-ocean general circulation model on the expanded spherical cube. *Monthly Weather Review*, pages 2845–2863, 2004.
2. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Conference Proceedings of the 2000 International Conference on Supercomputing*, pages 141–152, Santa Fe, New Mexico, May 2000.

3. Frederico Bassetti, Kei Davis, and Dan Quinlan. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science*, 1505, 1998.
4. Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rüde, and Christian Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transaction on Numerical Analysis*, pages 21–40, February 2000.
5. Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS)*, pages 361–366, New York, NY, USA, 2005. ACM.
6. F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages*, pages 319–329, 1988.
7. Guohua Jin, John Mellor-Crummey, and Robert Fowler. Increasing temporal locality with skewing and recursive blocking. In *High Performance Networking and Computing (SC)*, Denver, Colorodo, November 2001. ACM Press and IEEE Computer Society Press.
8. Shoaib Kamil, Kaushik Datta, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Implicit and explict optimizations for stencil computations. In *Memory Systems Performance and Correctness*, 2006.
9. Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the Workshop on Memory System Performance*, pages 36–43, New York, NY, USA, 2005. ACM Press.
10. Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of Programming Languages Design and Implementation (PLDI)*, pages 235–244, New York, NY, USA, 2007. ACM.
11. David A. Randall, Todd D. Ringler, Ross P. Heikes, Phil Jones, and John Baumgardner. Climate modeling with spherical geodesic grids. *Computing in Science and Engineering*, 4(5):32–41, 2002.
12. Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, 1990.
13. Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. In *Proceedings of the 2001 International Conference on Computational Science*, Lecture Notes in Computer Science, San Francisco, CA, USA, May 28-30, 2001. Springer.
14. Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices (PLDI)*, 34(5):215–228, May 1999.
15. Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.
16. Michael J. Wolfe. Iteration space tiling for memory hierachies. *Proc. of the 3rd SIAM Conf. on Parallel Processing for Scientific Computing*, pages 357–361, 1987.
17. Michael J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
18. David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002.

Appendix A

Models of the icosahedron and the smashed icosahedron

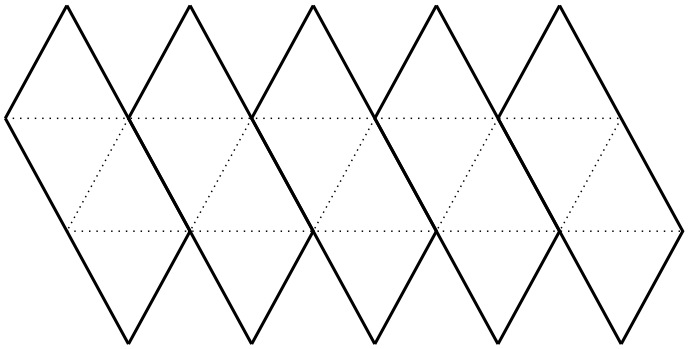


Fig. 7. Model of the icosahedron. Cut out the figure and fold in along the dotted lines. The edges will meet to form a icosahedron. Where the edges meet is where the non-uniform neighbors are when it is flattened out.

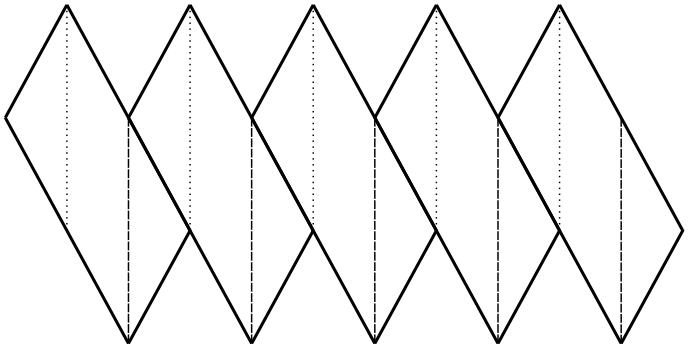


Fig. 8. Model of the smashed icosahedron. Cut out the figure, fold in along the dotted lines and out along the dashed lines. The result are ten layered parallelograms. The points that were non-uniform neighbors lie on top of each other.