On the Scalability of Loop Tiling Techniques

David G. Wonnacott
Haverford College
Haverford, PA, U.S.A. 19041
davew@cs.haverford.edu

Michelle Mills Strout
Colorado State University
Fort Collins, CO, U.S.A. 80523
mstrout@cs.colostate.edu

ABSTRACT

The Polyhedral model has proven to be a valuable tool for improving memory locality and exploiting parallelism for optimizing dense array codes. This model is expressive enough to describe transformations of imperfectly nested loops, and to capture a variety of program transformations, including many approaches to loop tiling. Tools such as the highly successful PLuTo automatic parallelizer have provided empirical confirmation of the success of polyhedral-based optimization, through experiments in which a number of benchmarks have been executed on machines with small- to medium-scale parallelism.

In anticipation of ever higher degrees of parallelism, we have explored the impact of various loop tiling strategies on the asymptotic degree of available parallelism. In our analysis, we consider "weak scaling" as described by Gustafson, i.e., in which the data set size grows linearly with the number of processors available. Some, but not all, of the approaches to tiling provide weak scaling. In particular, the tiling currently performed by PLuTo does not scale in this sense.

In this article, we review approaches to loop tiling in the published literature, focusing on both scalability and implementation status. We find that fully scalable tilings are not available in general-purpose tools, and call upon the polyhedral compilation community to focus on questions of asymptotic scalability. Finally, we identify ongoing work that may resolve this issue.

1. INTRODUCTION

The Polyhedral model has proven to be a valuable tool for improving memory locality and exploiting parallelism for optimizing dense array codes. This model is expressive enough to express a variety of program transformations, including many forms of *loop tiling*, which can improve cache line utilization and avoid false sharing [16, 37, 36], as well as increase the granularity of concurrency.

For many codes, the most dramatic locality improvements occur with *time tiling*, i.e., tiling that spans multiple iterations of an outer time-step loop. In some cases, the degree of locality can increase with the number of time steps in a tile, providing *scalable locality* [39]. For non-trivial examples, time tiling often requires *loop skewing* with respect to the time step loop [27, 39], often referred to as *time skewing* [39, 38]. This transformation typically involves imperfectly nested loops, and was thus not widely implemented before the adoption of the polyhedral approach. However, the PLuTo automatic parallelizer [19, 6] has demonstrated considerable success in obtaining high performance on machines with moderate degrees of parallelism by using this technique to automatically produce OpenMP parallel code.

Unfortunately, the specific tiling transformations that have been implemented and released in tools like PLuTo involve pipelined execution of tiles, which prevents full concurrency from the start. The lack of immediate full concurrency is sometimes dismissed as a start-up cost that will be trivially small for realistic problem sizes. While this may be true for the degrees of parallelism provided by current multicore processors, this choice of tiling can impact the asymptotic degree of concurrency available if we try to scale up data set size and machine size together, as suggested by Gustafson [15]. Furthermore, Van der Wijngaart et al. [35] have modeled and experimentally demonstrated the load imbalance that occurs on distributed memory machines when using the pipelined approach.

In this paper, we review the status of implemented and proposed techniques for tiling dense array codes (including the important sub-case of stencil codes) in an attempt to determine whether or not the techniques that are currently being implemented are well suited to machines with higher demands for parallelism and control of memory traffic and communication. The published literature on tiling for automatic parallelization seems to be divided into two disjoint categories: "practical" papers describing implemented but unscalable techniques for automatic parallelizers for dense array codes, and "theoretical" papers describing techniques that scale well but are either not implemented or not integrated into a general automatic parallelizer.

In Section 2 of this paper, we discuss that the approach currently used by PLuTo does not allow full scaling as described by Gustafson [15]. In Section 4, we survey other tilings that have been suggested in the literature, classify each approach as fully scalable or not, and discuss its implementation status in current automatic parallelization tools. We also address recent work by Bondhugula et al. [3] on a tiling technique that we believe will be scalable, though asymptotic scaling is not addressed in [3]. Section 5 presents our conclusions: we believe the scalable/implemented dichotomy is an artifact of current design choices, not a fundamental limitation of the polyhedral model, and can thus be addressed via a shift in emphasis by the research community.

```
    // update N pseudo-random seeds T times
    // assumes R[ ] is initialized
       for t = 1 to T
          for i = 0 to N-1
S0:          R[i] = (a*R[i]+c) % m
```

Figure 1: "Embarrassingly Parallel" Loop Nest.

2. TILING AND SCALABILITY

In his 1988 article "Reevaluating Amdahl's Law" [15], Gustafson observed that, in actual practice, "One does not take a fixed size problem and run it on various numbers of processors", but rather "expands [the problem] to make use of the increased facilities". In particular, in the successful parallelizations he described, "as a first approximation, the amount of work that can be done in parallel *varies linearly with the number of processors*", and it is "most realistic to assume *run time*, not *problem size*, is constant". This form of scaling is typically referred to as *weak scaling* or *scalable parallelism*, as opposed to the *strong scaling* needed to give speed-up proportional to the number of processors for a fixed-size problem.

Weak scaling can be found in many *data parallel* codes, in which many elements of a large array can be updated simultaneously. Figure 1 shows a trivial example that we will use to introduce our diagrammatic conventions (following [38]). In Figure 2 each statement execution/loop iteration is drawn as an individual node, with sample values given to symbolic parameters. The time axis, or outer loop, moves from left to right across the page. The grouping of nodes into tiles is illustrated with variously shaped boxes around sets of nodes. Arrows in the figure denote direction of flow of information among iterations or tiles. Line-less arrowheads indicate values that are live-in to the space being illustrated. (When comparing our figures to other work, note that presentation style may vary in several ways: some authors use a time axis that moves up or down the page; some draw data dependence arcs from a use to a definition, thus pointing into the data-flow like a weather vane; some illustrate tiles as rectangular grids on a visually transformed iteration space, rather than with varied shapes on the original iteration space.)

Figure 2 makes clear the possibility of both (weak) scalable parallelism and scalable locality. In the execution of a tile of size $(\tau \times \sigma)$, i.e., τ iterations of the t (time) loop and σ iterations of the i (data) loop, data-flow does not prevent P processors from concurrently executing P such tiles. Each tile performs $O(\sigma \times \tau)$ computations and has $O(\sigma)$ live-in and live-out values; if each processor performs all updates of one data element before moving to the next, $O(\tau)$ operations can be performed with $O(1)$ accesses to main memory.

Inter-iteration data-flow can constrain, or even prevent, scalable parallelism or scalable locality. For the code in Figure 1, we can scale up parallelism by increasing N and P, or we can scale up locality by increasing T with the machine balance. However, beyond a certain point (i.e., $\sigma = 1$), we can no longer use additional processors to explore ever increasing values of T for a given N. (An increase in CPU clock speed might help in this situation, though beyond a certain point it would likely not help performance for increasing N for a given T.)

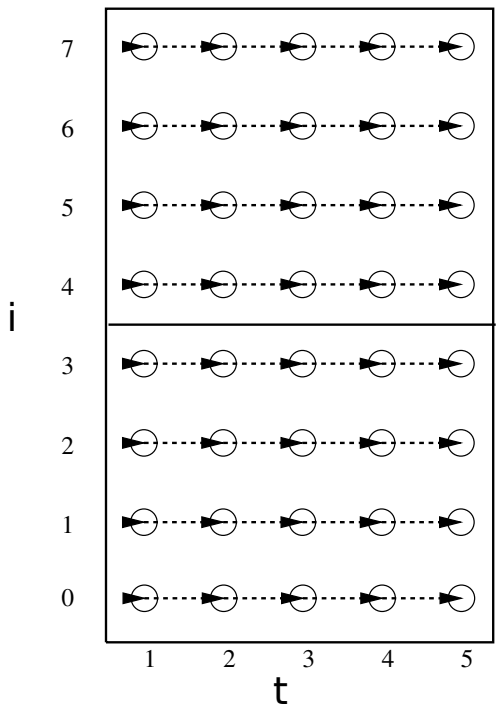

Figure 2: Iteration Space of Figure 1 with `T`=5, `N`=8, tiled with $\tau = 5, \sigma = 4$.

As we show in the next two sections, scalable parallelism may be constrained not only by the fundamental data-flow, but also by the approach to parallelization.

3. PIPELINED PARALLELISM

In a Jacobi stencil computation, each array element is updated as a function of its value and its neighbors' values, as shown (for a one-dimensional array) in Figure 3. Thus, we must perform time skewing to tile the iteration space (except in the degenerate case of $\tau = 1$, which prevents scalable locality). Figure 4 illustrates the usual tiling performed by automatic parallelizers such as PLuTo, though for readability our figure shows far fewer loop iterations per tile. Nodes represent executions of statement `S1`; for simplicity, executions of `S2` are not shown. (The same data-flow also arises from a doubly-nested execution of the single statement `A[t%2,i] = (A[(t-1)%2,i-1]+2*A[(t-1)%2,i]+A[(t-1)%2,i+1])/4`, but some tools may not recognize the program in this form.)

Array data-flow analysis [11, 22, 23] is well understood for programs that fit the polyhedral model, and can be used to deduce the data-flow arcs from the original imperative code. The data-flow arcs crossing a tile boundary describe the communication between tiles; in most approaches to tiling for distributed systems, inter-processor communication is aggregated and takes place between executions of tiles, rather than in the middle of any tile. The topology of the inter-tile data-flow thus gives the constraints on possible concurrent execution of tiles. For Figure 4, concurrent ex-

```
      for t = 1 to T
        for i = 1 to N-2
S1:       new[i] = (A[i-1]+2*A[i]+A[i+1])/4
        for i = 1 to N-2
S2:       A[i] = new[i]
```

Figure 3: Three Point Jacobi Stencil.

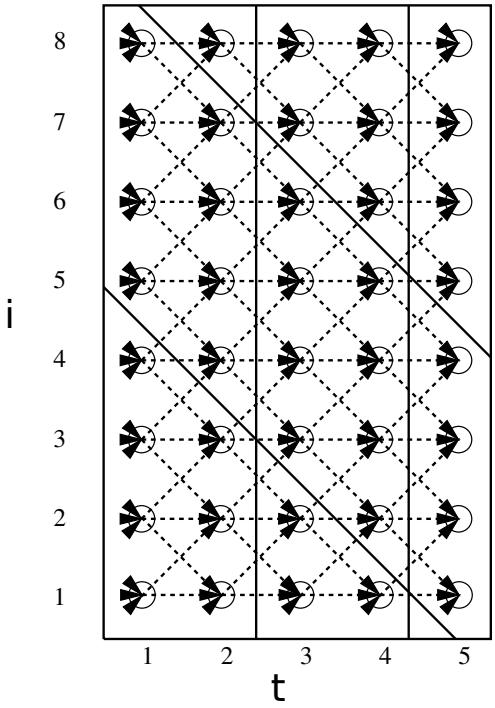

Figure 4: Iteration Space of Figure 3 with T=5, N=10, tiled with $\tau = 2, \sigma = 4$.

ecution is possible in pipelined fashion, in which execution of tiles progresses as a *wavefront* that begins with the lower left tile, then simultaneously executes the two tiles bordering it (above and to the right), and continues to each wave of tiles adjacent to the just-completed wave.

As has been noted in the literature, this is not the only way to tile this set of iterations; however, other tilings are not (currently) selected by fully-automatic loop parallelizers such as PLuTo [19]. Even the semi-automatic AlphaZ system [40], which is designed to allow programmers to experiment with different optimization strategies, cannot express many of these tilings. If such tools are to be considered for extreme scale computing, we must consider whether or not the tiling strategies they support provide the necessary scaling characteristics.

3.1 Scalability

To support our claim that this pipelined tiling does not always provide scalable parallelism, we need only show that it fails to scale on one of the classic examples for which it has shown dramatic success for low-degree parallelism, such as the easily-visualized one-dimensional Jacobi stencil of Figure 3. We will first do so, and then discuss the issue in higher dimensions.

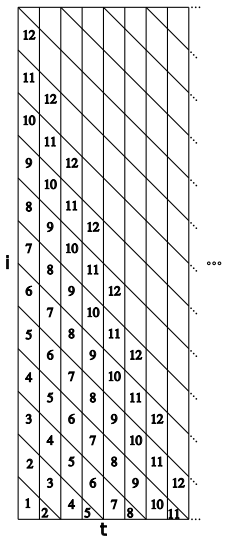

Figure 5: Wavefronts of Pipelined Tile Execution.

For some problem sizes, the tiling of Figure 4 can come close to realizing scalable parallelism: if $P = \frac{N}{\sigma+\tau}$ and $\frac{T}{\tau}$ is much larger than P, most of the execution is done with P processors. Figure 5 illustrates the first 8τ time steps of such an example, with $P = 8$, $\sigma = 2\tau$, and $N = P(\sigma + \tau)$ (the ellipsis on the right indicates a large number of additional time steps). The tiles executed in the first 12 waves are numbered 1 to 12 for reference, and individual iterations and data-flow are omitted for clarity. For all iterations after 11, this tiling provides enough parallelism to keep eight processors busy for this problem size, and for large T the running time approaches $\frac{1}{8}$ of the sequential execution time (plus communication/synchronization time, which we will discuss later). If we double both N and P, a similar argument shows the running time approaches $\frac{1}{16}$ of the now twice-as-large sequential execution time, i.e., the same parallel execution time, as described by Gustafson.

However, as N and P continue to grow, the assumption that $\frac{T}{\tau} \gg P$ eventually fails, and scalability is lost. Consider what happens in Figure 5 if $T = 8\tau$, i.e., the ellipsis corresponds to 0 additional time steps. At this point, doubling N and P produces a figure that is twice as tall, but no wider; parallelism is limited to degree 8, and execution with 16 processors requires 35 steps rather than the 23 needed for Figure 5 when $T = 8\tau$ (note that the upper-right region is symmetric with the lower-left). Thus, communication-free execution time has increased rather than remaining constant. Increasing T (rather than N) with P is no better, and in fact no combination of N and T increase can allow 16 processors to execute twice the work of 8 in the same 23 wavefronts: adding even one full row or one full column of tiles means 24 wavefronts are needed.

Figure 5 was, of course, constructed to illustrate a lack of scalability. But even if we start with a more realistic problem

size, i.e., with $N \gg P$ and $T \gg P$, the pipelined tiling still limits scalability. Consider what happens we apply the tiling of Figure 5 with parameters $N = 10000\sigma, T = 1000\tau, P = 100$, in which over 99% of the tiles can be run with full 100-fold parallelism, and then scale up N and P together by successive factors of ten. Our first jump in size gives $N = 100000\sigma, T = 1000\tau, P = 1000$, which still has full (1,000-fold) parallelism in over 98% of the tiles.

After the next jump, to $N = 1000000\sigma, T = 1000\tau, P = 10000$ there is no 10,000-fold concurrency in the problem. Even if the application programmer is willing to scale up T rather than just N, in an attempt to reach full machine utilization, the execution for $N = 38730\sigma, T = 25820\tau, P = 10000$ still achieves 10,000-fold parallelism in only 85% of the 10^9 tiles. No combination of N and T allows *any* 100,000-fold parallelism on 10^{10} tiles with this tiling... to maintain a given degree of parallelism asymptotically, we must scale *both* N and T with P, contrary to Gustafson's original definition. While this may be acceptable for some application domains, we do not presume it to be universally appropriate, and thus see pipelined tiling as a potential restriction on the applicability of time tiling.

It is not always realistic to scale the number of time steps T. Bassetti et al. [4] introduce an optimization they call sliding block temporal tiling. They indicate that in these relaxation algorithms such as Jacobi "[g]enerally several sweeps are made". In their experiments they use up to 8 sweeps. Zhou et al. [41] use 16,384 in their modified benchmarks. Experiments with the Pochoir compiler [34] used 200 time steps because their cache oblivious performance optimization improves temporal locality. In summary, the number of time iterations in stencil computation performance optimization research varies dramatically. Work from the BeBOP group at Berkeley [8] discusses how often multiple sweeps over the grid within one loop occur and indicate that it may not be as common as those of us working on time skewing imagine. This makes it even more important for tiling strategies to provide scalable parallelism that does not require the number of time steps to be on par with the spatial domain.

3.2 Tile Size Choice and Communication Cost

The above argument presumes a fixed tile size, ignoring the possibility of reducing the tile size to increase the number of tiles. However, communication costs (either among processors or between processors and RAM) dictate a minimal tile size below which performance will be negatively impacted (see [38, 19] for further discussion).

Note that per-processor communication cost is not likely to shrink as the data set size and number of processors is scaled up: Each processor will need to send the same number of tiles per iteration, producing per-processor communication cost that remains roughly constant (e.g., if processors are connected to nearest neighbors via a network of the same dimensionality as the data set space) or rising (e.g., if processors are connected via a shared bus).

Even if we ignore communication costs entirely (i.e. in the notoriously unscalable PRAM abstraction), tile size cannot shrink below a single-iteration (or single-instruction) tile, and eventually our argument of Section 3.1 comes into play in asymptotic analysis.

3.3 Other Factors Influencing Scalability and Performance

Our argument focuses on tile *shape*, but a number of other factors will influence the degree of parallelism actually achieved by a given tiling. As noted above, reductions in tile size could, up to a point, provide additional parallelism (possibly at the cost of performance on the individual nodes).

The use of global barriers or synchronization is common in the original code generators, but note that MPI code generators are under development for both Pluto and AlphaZ. While combining different tilings and code generation schemes raises practical challenges in implementation, we do not see any reason why any of the tilings discussed in the next section could not, in principle, be executed without global barriers within the tiled iteration space.

High performance on each node also requires attention to a number of other code generation issues, such as code complexity and impact on vectorization and prefetching. Furthermore, these issues could be exacerbated by changes in tile shape [3, Section III.B]. Thus, different tiling *shapes* may be optimal for different hardware platforms or even different problem sizes, depending on the relative costs of limiting parallelism vs. per-node performance. Both [3, Section III.B] and [33] discuss these issues and the possibility of choosing a tiling that is scalable in some, but not all, data dimensions.

3.4 Higher-Dimensional Codes

While the two-dimensional iteration space of the three-point Jacobi stencil is easy to visualize on paper, many of the subtleties of tiling techniques are only evident in problems with at least two dimensions of data and one of time. For pipelined tiling, the conflict between scalability is essentially the same in higher dimensions: for a pipelined tiling of a hyper-rectangular iteration space of dimension d, eventually the amount of work must grow by $O(k^d)$ to achieve parallelism $O(k^{d-1})$.

Conversely, in higher dimensions, the existence of a wavefront that is perpendicular to the time dimension (or any other face of a hyper-rectangular iteration space) is frequently the sign of a parallelization that admits some form of weak scalability. However, as we will see, the parallelism of some tilings scales with only some of the spatial dimensions.

4. VARIATIONS ON THE TILING THEME

The published literature describes many approaches to loop tiling. In this section, we survey these approaches, grouping together those that produce similar (or identical) tilings. Our descriptions focus primarily on the tiling that would be used for the code of Figure 3, which is used as an introductory example in many of the descriptions. We illustrate the tilings of this code with figures that are analogous to our Figure 5, with gray shading highlighting a single tile

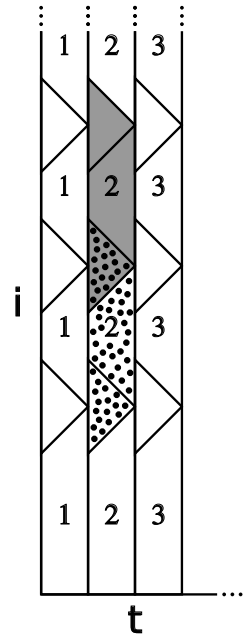

Figure 6: Overlapped Tiling.

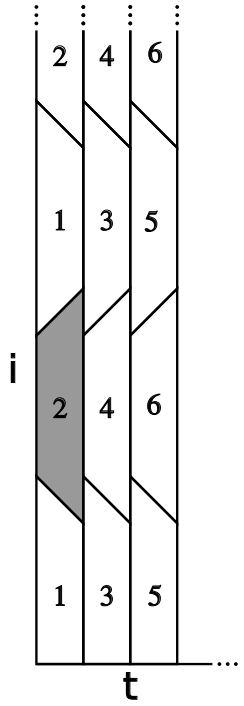

Figure 7: Trapezoidal Tiling.

from time step two. We delve into the complexities of more complex codes only as necessary to make our point.

For iterative codes with intra-time-step data-flow, the tile shapes discussed below are not legal (or cannot be legally executed in an order that permits scalable parallelism). For example, consider an in-place update of a single copy of a data set, e.g. the single statement `A[i] = (A[i-1]+2*A[i]+A[i+1])/4` nested inside `t` and `i` loops. Since each tile must wait for data from tiles of with lower values of `i` and the same value of `t`, pipelined startup is necessary. While such code provides additional asymptotic concurrency when both T and N increase, we see no way to allow concurrency to grow linearly with the total work to be done in parallel. Thus, pipelined tiling does not produce a scalability disadvantage for such codes.

Note that our discussion below focuses on distinct tile shapes, rather than distinctions among algorithms used to deduce tile shape or size or the manner in which individual tiles are scheduled or assigned to processors. For example we do not specifically discuss the "CORALS" approach [32], in which an iteration space is recursively subdivided into parallelograms, avoiding the need to choose a tile size in advance of starting the computation. Regardless of size, variation in size, and algorithmic provenance, the information flow among atomic parallelogram tiles still forces execution to proceed along the diagonal wavefront, and thus still limits asymptotic scalability.

4.1 Overlapped Tiling

A number of projects have experimented with what is commonly called overlapped tiling [26, 4, 2, 25, 10, 24, 19, 7, 20, 41]. In overlapped tiling for stencil computations, a larger halo is maintained so that each processor can execute more than one time step before needing to communicate with other processors. Figure 6 illustrates this tiling. Two individual tiles from the second wavefront have been indicated with shading, one with gray and one with polkadots; the triangular polkadotted and gray region is in both tiles, and thus represents redundant computation. This overlap means that all tiles along each vertical wavefront can be executed in parallel while still improving temporal data locality.

In terms of parallelism scalability, overlapped tiling does scale because all of the tiles can be executed in parallel. If a two-dimensional tiling in a two-dimensional spatial part of a stencil is used as the seed partition, then two dimensions of parallelism will be available with no need to fill a pipeline. This means that as the data scale, so will the parallelism.

The problem with overlapped tiling is that redundant computation is performed. This leads to a trade-off between parallel scalability and execution time. Tile size selection must also consider the effect of the expanded memory footprint caused by overlapped tiling.

Auto-tuning between overlapped sparse tiling and non-overlapped sparse tiling [29, 30] for irregular iteration spaces has also been investigated by Demmel et al. [9] in the context of iterative sparse matrix computations where the tiling is a run-time reordering transformation [28].

4.2 Trapezoidal Tiling

Frigo and Strumpen [12, 13, 14] propose an algorithm for limiting the asymptotic cache miss rate of "an idealized parallel machine" while providing scalable parallelism. Figure 7 illustrates that even a simplified version of their approach

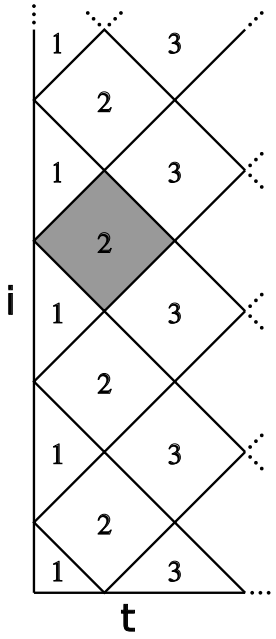

Figure 8: One data dimension and the time dimension in a diamond tiling [33]. The diamond extends into a diamond tube in the second data dimension.

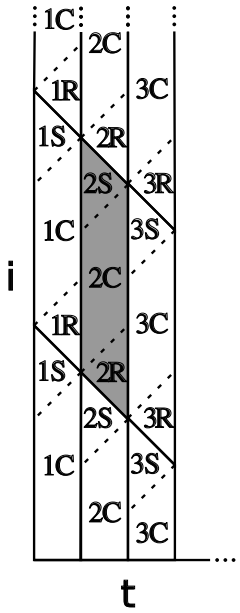

Figure 9: Molecular tiling.

can enable weak scaling for the examples we discuss here (their full algorithm involves a variety of possible decomposition steps; our figure is based on Figure 4 of [14]). In our Figure 7, the collection of trapezoids marked "1" can start simultaneously; after these tiles complete, the mirror-image trapezoids that fill the spaces between them, marked "2", can all be executed; after these steps, a similar pair of sets of tiles "3" and "4" complete another τ time steps of computation, etc. For discussion of the actual transformation used by Frigo and Strumpen, and its asymptotic behavior, see [14].

The limitation of this approach is not its scalability, but rather the challenge of implementing it in a general-purpose compiler. Tang et al. [34] have developed the Pochoir compiler, based on a variant of Frigo and Strumpen's techniques with a higher degree of asymptotic concurrency [34]. However, Pochoir handles a specialized language that allows only stencil computations. Tools like PLuTo handle a larger domain of dense array codes; it may be possible to generalize trapezoidal tiling to PLuTo's domain, but we know of no such work.

4.3 Diamond Tiling

Strzodka et al. [33, 31] present the CATS algorithm for creating diamond "tube" tiles in a 3-d iteration space. The diamonds occur in the time dimension and one data dimension, as in Figure 8. The tube aspect occurs because there is no tiling in the other space dimension. Each diamond tube can be executed in parallel with all other diamond tubes within a temporal row of diamond tubes. For example, in Figure 8 all diamonds labeled "1" can be executed in parallel, after which all diamonds labeled "2" can be executed in parallel, etc. Within each diamond tube, the CATS approach schedules another level of wavefront parallelism at the granularity of iteration points.

Although Strzodka et al. [33] do not use diamond tiles for 1-d data/2-d iteration space, diamond tiles are parallel scalable within that context. They actually focus on the 2-d data/3-d iteration space, where asymptotically, diamond tiling only scales for one data dimension. The outermost level of parallelism over diamond tubes only scales with one dimension of data since the diamond tiling occurs across time and one data dimension. On page 2 of [33], Strzodka et al. explicitly state that their results are somewhat surprising in that asymptotically their behavior should not be as good as previously presented tiling approaches, but the performance they observe is excellent probably due to the concurrent parallel startup that the diamond tiles provide.

Diamond tiling is a practical approach that can perform better than pipelined tiling approaches because it avoids the pipeline fill and drain issue. The diamond tube also has advantages in terms of intra-tile performance: fine-grained wavefront parallelism and leveraging pre-fetchers. The disadvantages of the diamond tiling approach are that it has not been expressed within a framework such as the polyhedral model (although it would be possible, just not with rectangular tiles); that the approach does not cleanly extend to higher dimensions of data (only one dimension of diamond tiles are possible with other dimensions doing some form of pipelined or split tiling); and that the outermost level of parallelism can only scale with one data dimension.

4.4 Molecular Tiling

Wonnacott [38] described a tiling for stencils that allows true weak scaling for higher-dimensional stencils, performs no re-

dundant work, and contains tiles that are all the same shape. However, Wonnacott's *molecular tiles* required mid-tile communication steps, as per Pugh and Rosser's *iteration space slicing* [21] as illustrated in Figure 9. Each tile first executes its *send slice* (labeled "S"), the set of iterations that produce values that will be needed by another currently-executing tile, and then sends those values; it then goes on to execute its *compute slice* ("C"), the set of iterations that require no information from any other currently-executing tile; finally, each tile receives incoming values and executes its *receive slice* ("R"), the set of iterations that require these data. In higher dimensions, Wonnacott discussed the possibility of extending the parallelograms into prisms (as diamonds are extended into diamond tubes in the diamond tiling), but also presented a multi-stage sequence of send and receive slices to provide full scalability.

Once again, a transformation with potential for true weak scaling remains unrealized due to implementation challenges. No implementation was ever released for iteration space slicing [21]. For the restricted case of stencil computations, these molecular tiles can be described without reference to iteration space slicing, but they make extensive use of modulo constraints supported by the Omega Library's code generation algorithms [18], and Omega has no direct facility for generating the required communication primitives.

The developers of PLuTo explored a similar *split tiling* [19] approach, and demonstrated improved performance over the pipelined tiling, but this approach was not used for the released implementation of PLuTo.

4.5 A New Hope

Recent work on the PLuTo system [3] has produced a tiling that we believe will address the issue of true scalability with data set size, though the authors frame their approach primarily in terms of "enabling concurrent start-up" rather than improving asymptotic scalability. For a one-dimensional data set, this approach is essentially the same as the "diamond tiling" of Section 4.3, but for higher-dimensional stencils it allows scalable parallelism in all data dimensions.

Although a Jacobi stencil on a two-dimensional data set has an "obvious" four-sided pyramid of dependences, a collection of four-sided pyramids (or a collection of octahedrons made from pairs of such pyramids) cannot cover all points, and thus does not make a regular tiling. The algorithm of [3] produces, instead, a collection of six-faced tiles that appear to be balanced on one corner (these figures are shown with the time dimension moving up the page). Various sculptures of balanced cubes may be helpful in visualizing this tiling; those with limited travel budgets may want to search the internet for images of the "Zabeel park cube sculpture". The approach of [3] manages to construct these corner-balanced solids in such as way that the three faces at the bottom of the tile enclose the data-flow.

Experiments with this approach [3] demonstrate improved results (vs. pipelined tiling) for current shared-memory systems up to 16 cores. The practicality of this approach on such a low degree of parallelism suggests that the per-node penalties discussed in our Section 3.3 are not prohibitively expensive.

While the algorithm is described in terms of stencils, and the authors only claim concurrent startup for stencils, it is implemented in a general automatic parallelizer (PLuTo). We belive it would be interesting to explore the full domain over which this tiling algorithm provides concurrent startup.

The authors of [3] do not discuss asymptotic complexity, but we hope that future collaborations could lead to a detailed theoretical and larger-scale empirical study of the scalability of this technique, using the distributed tile execution techniques of [1] or [5].

4.6 A Note on Implementation Challenges

The pipelined tile execution shown in Figures 4 and 5 is often chosen for ease of implementation in compilers based on the polyhedral model. Such compilers typically combine all iterations of all statements into one large iteration space; the pipelined tiling can then be seen as a simple linear transformation of this space, followed by a tiling with rectangular solids. This approach works well regardless of choice of software infrastructure within the polyhedral model.

The other transformations may be more sensitive to choice of software infrastructure, or the subtle use thereof. At this time, we do not have an exact list of which transformations can be expressed with which transformation and code generation libraries. We are working with tools that allow the direct control of polyhedral transformations from a text input, such as AlphaZ [40] and the Omega Calculator [17], in hopes of better understanding the expressiveness of these tools and the polyhedral libraries that underlie them.

5. CONCLUSIONS

Current work on general-purpose loop tiling exhibits a dichotomy between largely unimplemented explorations of asymptotically high degrees of parallelism and carefully tuned implementations that restrict or inhibit scalable parallelism. This appears to result from the challenge of general implementation of scalable approaches. The pipelined approach requires only a linear transformation of the iteration space followed by rectangular tiling, but does not provide true scalable parallelism. Diamond tiling scales with only one data dimension. Overlapped, trapezoidal, and molecular tiling each pose implementation challenges (due to redundant work, non-uniform tile shape/orientation, or non-atomic tiles, respectively).

We believe automatic parallelization for extreme scale computing will require a tuned implementation of a general technique that does not inhibit or restrict scalability; thus future work in this area must address scalability, generality, and quality of implementation. We are optimistic that ongoing work by Bondhugula et al. [3] may already provide an answer to this to dilemma.

6. ACKNOWLEDGMENTS

This work was supported by NSF Grant CCF-0943455, by a Department of Energy Early Career Grant DE-SC0003956, and the CACHE Institute grant DE-SC04030.

7. REFERENCES

[1] Abdalkader, M., Burnette, I., Douglas, T., and Wonnacott, D. G. Distributed shared memory and compiler-induced scalable locality for scalable cluster performance. *Cluster Computing and the Grid, IEEE International Symposium on 0* (2012), 688–689.

[2] Allen, G., Dramlitsch, T., Foster, I., Goodale, T., Karonis, N., Ripeanu, M., Seidel, E., and Toonen, B. The cactus code: A problem solving environment for the grid. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC9)* (Pittsburg, PA, USA, 2000).

[3] Bandishti, V., Pananilath, I., and Bondhugula, U. Tiling stencil computations to maximize parallelism. In *Proceedings of 2012 International Conference for High Performance Computing, Networking, Storage and Analysis* (November 2012), SC '12, ACM Press.

[4] Bassetti, F., Davis, K., and Quinlan, D. Optimizing transformations of stencil operations for parallel object-oriented scientific frameworks on cache-based architectures. *Lecture Notes in Computer Science 1505* (1998).

[5] Bondhugula, U. Compiling affine loop nests for distributed-memory parallel architectures. Preprint, 2012.

[6] Bondhugula, U., Hartono, A., and Ramanujam, J. A practical automatic polyhedral parallelizer and locality optimizer. In *In PLDI âĂŹ08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation* (2008).

[7] Christen, M., Schenk, O., Neufeld, E., Paulides, M., and Burkhart, H. Manycore Stencil Computations in Hyperthermia Applications. In *Scientific Computing with Multicore and Accelerators*, J. Dongarra, D. Bader, and J. Kurzak, Eds. CRC Press, 2010, pp. 255–277.

[8] Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., and Yelick, K. Optimizations and performance modeling of stencil computations on modern microprocessors. *SIAM Review 51*, 1 (2009), 129–159.

[9] Demmel, J., Hoemmen, M., Mohiyuddin, M., and Yelick, K. Avoiding communication in sparse matrix computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)* (Los Alamitos, CA, USA, 2008), IEEE Computer Society.

[10] Ding, C., and He, Y. A ghost cell expansion method for reducing communications in solving pde problems. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2001), Supercomputing '01, ACM, pp. 50–50.

[11] Feautrier, P. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming 20*, 1 (Feb. 1991), 23–53.

[12] Frigo, M., and Strumpen, V. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing* (New York, NY, USA, 2005), ICS '05, ACM, pp. 361–366.

[13] Frigo, M., and Strumpen, V. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2006), SPAA '06, ACM, pp. 271–280.

[14] Frigo, M., and Strumpen, V. The cache complexity of multithreaded cache oblivious algorithms. *Theor. Comp. Sys. 45*, 2 (June 2009), 203–233.

[15] Gustfson, J. L. Reevaluating Amdahl's law. *Communications of the ACM 31*, 5 (May 1988), 532–533.

[16] Irigoin, F., and Triolet, R. Supernode partitioning. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (1988), pp. 319–329.

[17] Kelly, W., Maslov, V., Pugh, W., Rosser, E., Shpeisman, T., and Wonnacott, D. The Omega Calculator and Library. Tech. rep., Dept. of Computer Science, University of Maryland, College Park, Apr. 1996.

[18] Kelly, W., Pugh, W., and Rosser, E. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation* (McLean, Virginia, Feb. 1995), pp. 332–341.

[19] Krishnamoorthy, S., Baskaran, M., Bondhugula, U., Ramanujam, J., Rountev, A., and Sadayappan, P. Effective automatic parallelization of stencil computations. In *Proceedings of Programming Languages Design and Implementation (PLDI)* (New York, NY, USA, 2007), vol. 42, ACM, pp. 235–244.

[20] Nguyen, A., Satish, N., Chhugani, J., Kim, C., and Dubey, P. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–13.

[21] Pugh, W., and Rosser, E. Iteration slicing for locality. In *12th International Workshop on Languages and Compilers for Parallel Computing* (Aug. 1999).

[22] Pugh, W., and Wonnacott, D. Eliminating false data dependences using the Omega test. In *SIGPLAN Conference on Programming Language Design and Implementation* (San Francisco, California, June 1992), pp. 140–151.

[23] Pugh, W., and Wonnacott, D. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems 20*, 3 (May 1998), 635–678.

[24] Rastello, F., and Dauxois, T. Efficient tiling for an ode discrete integration program: Redundant tasks instead of trapezoidal shaped-tiles. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2002), IPDPS '02, IEEE Computer Society, pp. 138–.

[25] Ripeanu, M., Iamnitchi, A., and Foster, I. T. Cactus application: Performance predictions in grid environments. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing* (London, UK, UK, 2001), Euro-Par '01, Springer-Verlag, pp. 807–816.

[26] Sawdey, A., and O'Keefe, M. T. Program analysis of overlap area usage in self-similar parallel programs. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, UK, 1998), LCPC '97, Springer-Verlag, pp. 79–93.

[27] Song, Y., and Li, Z. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices (PLDI) 34*, 5 (May 1999), 215–228.

[28] Strout, M. M., Carter, L., and Ferrante, J. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (New York, NY, USA, June 2003), ACM.

[29] Strout, M. M., Carter, L., Ferrante, J., Freeman, J., and Kreaseck, B. Combining performance aspects of irregular Gauss-Seidel via sparse tiling. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)* (Berlin / Heidelberg, July 2002), Springer.

[30] Strout, M. M., Carter, L., Ferrante, J., and Kreaseck, B. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications 18*, 1 (February 2004), 95–114.

[31] Strzodka, R., Shaheen, M., and Pajak, D. Time skewing made simple. In *PPOPP* (2011), C. Cascaval and P.-C. Yew, Eds., ACM, pp. 295–296.

[32] Strzodka, R., Shaheen, M., Pajak, D., and Seidel, H.-P. Cache oblivious parallelograms in iterative stencil computations. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing* (June 2010), ACM, pp. 49–59.

[33] Strzodka, R., Shaheen, M., Pajak, D., and Seidel, H.-P. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the 40th International Conference on Parallel Processing (ICPP)* (Taipei, Taiwan, September 2011), IEEE Computer Society, pp. 517–581.

[34] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and Leiserson, C. E. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures* (New York, NY, USA, 2011), SPAA '11, ACM, pp. 117–128.

[35] Van der Wijngaart, R. F., Sarukkai, S. R., Mehra, and P. The effect of interrupts on software pipeline execution on message-passing architectures. In *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing: Philadelphia, Pennsylvania, USA, May 25–28, 1996* (New York, NY 10036, USA, 1996), ACM, Ed., ACM Press, pp. 189–196.

[36] Wolf, M. E., and Lam, M. S. A data locality optimizing algorithm. In *Programming Language Design and Implementation* (New York, NY, USA, 1991), ACM.

[37] Wolfe, M. J. More iteration space tiling. In *Proceedings, Supercomputing '89, Reno, Nevada* (Reno, Nevada, November 1989), ACM, Ed., ACM Press, pp. 655–664.

[38] Wonnacott, D. Using Time Skewing to eliminate idle time due to memory bandwidth and network limitations. In *International Parallel and Distributed Processing Symposium* (May 2000), IEEE.

[39] Wonnacott, D. Achieving scalable locality with Time Skewing. *Internation Journal of Parallel Programming 30*, 3 (June 2002), 181–221.

[40] Yuki, T., Basupalli, V., Gupta, G., Iooss, G., Kim, D., Pathan, T., Srinivasa, P., Zou, Y., and Rajopadhye, S. Alphaz: A system for analysis, transformation, and code generation in the polyhedral equational model. Tech. rep., Technical Report CS-12-101, Colorado State University, 2012.

[41] Zhou, X., Giacalone, J.-P., Garzarán, M. J., Kuhn, R. H., Ni, Y., and Padua, D. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 207–218.