Haverford College Computer Science Tech Report 2012-01
Archived 06 August 2012

On the Scalability of Loop Tiling Techniques

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

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

Abstract

The challenge of extreme scale computing will test the limits of our ability to scale computational infrastructure. While much attention has been given to the scalability of hardware designs and of the novel algorithms to be run thereon, and significant practical success has been achieved with data-flow-based automatic parallelization of dense array codes, current automatic parallelizers focus almost exclusively on transformations that are inherently not fully scalable. We discuss the limitations on asymptotic scalability of the transformations applied by successful automatic parallelizers like PLuTo, and review the literature of other approaches to this problem. As part of this survey, we discuss both the scalability and implementation status of each current technique. Finally, we identify ongoing work that may resolve this issue.

On the Scalability of Loop Tiling Techniques

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

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

***Abstract*—The challenge of extreme scale computing will test the limits of our ability to scale computational infrastructure. While much attention has been given to the scalability of hardware designs and of the novel algorithms to be run thereon, and significant practical success has been achieved with data-flow-based automatic parallelization of dense array codes, current automatic parallelizers focus almost exclusively on transformations that are inherently not fully scalable. We discuss the limitations on asymptotic scalability of the transformations applied by successful automatic parallelizers like PLuTo, and review the literature of other approaches to this problem. As part of this survey, we discuss both the scalability and implementation status of each current technique. Finally, we identify ongoing work that may resolve this issue.**

I. Introduction

As extreme scale computing is likely to require extremely fast processors in great number, peak performance will require attention to both memory system performance and parallelism. Both of these questions have been the subject of significant prior work in a number of contexts, including the development of novel algorithms, the introduction of new languages or frameworks, and the development of automatic optimizers for various classes of codes.

In efficient algorithms, libraries, and automatic optimizers, some form of aggregation (e.g., *tiling*) is often used to improve cache line utilization and avoid false sharing [Wol89], [IT88], [WL91]. The general approach is to group parts of the computation into *tiles* and create a data-flow graph where each tile is a task. The execution of tiles generally is ordered so that no tile begins execution until after the completion of all tiles from which it receives data-flow; thus each tile can be executed without internal communication delays or synchronization (i.e. atomically). Since there are sets of tiles that can be executed in parallel, tiling can be used to address issues of parallelization and memory traffic control simultaneously.

The construction of the task/data-flow graph varies with the nature of the problem being tiled. For automatic parallelization of dense array codes, tiles contain sets of loop iterations. The *polyhedral model* provides a powerful framework for specifying tiling transformations and determining when tiling is legal through the use of array data-flow analysis [Fea91], [PW92], [PW98]. (See [Col02] for a more complete treatment of the polyhedral framework.)

After tiling, the task/data-flow graph can be scheduled dynamically [QOIQOvdG09], [BVB$^+$09], [BLKD09], [SYD09], [HRS09], [WHW09], [CNvdGZ10], [CKV10], [LZDK11], or it can be scheduled statically in a bulk synchronous fashion, as is the default in the widely-used OpenMP [Ope] system. However, no (correct) algorithm for scheduling a tile/data-flow graph will produce more parallelism than is implicit in the graph itself, and thus good performance requires that the graph contain parallelism of sufficient scale for the target architecture.

For non-trivial examples, tiling often requires *loop skewing* with respect to the time step loop [SL99], [Won02], which is often referred to as *time skewing* [Won02], [Won00]. The PLuTo automatic parallelizer [KBB$^+$07], [BHR08] 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 actually been implemented and released in tools like PLuTo involve pipelined execution of tiles, which prevents full concurrency from the start. The lack of full concurrency at the start 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 multi-core processors, this choice of tiling can impact the asymptotic degree of parallelism available if we try to scale up data set size and machine size together, as suggested by Gustafson [Gus88]. Furthermore, Van der Wijngaart et al. [VSMP96] have modeled and experimentally demonstrated the load imbalance that occurs on distributed memory machines when using pipelined parallelism.

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. Unfortunately, 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 II of this paper, we demonstrate that the approach currently used by PLuTo does not allow full scaling as described by Gustafson [Gus88]. In Section III, we survey other tilings that have been suggested in the published literature classify each approach as fully scalable or not, and discuss its implementation status in current automatic parallelization tools. We have recently learned of ongoing work by Bondhugula et al. [BPB12] of a tiling technique that we believe will be scalable, and we briefly mention it in Section III. We believe the proposed approach will address our concerns about asymptotic complexity. Section V presents our conclusions: we believe the scalable/implemented dichotomy is an artifact of current design choices, not a fundamental limitation of the underlying software infrastructure, and thus is something that can be addressed via a shift in emphasis by the research community.

II. Tiling and Scalability

In his 1988 article "Reevaluating Amdahl's Law" [Gus88], 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; the flow of information among the computations constrains the ways in which we can organize computations for concurrent execution. To illustrate this principle graphically, we will adopt most of the conventions of [Won00], in which each execution of a repeatedly-executed statement is drawn as an individual node, with arcs denoting flow of information (or other dependences, in a non-data-flow context). The time axis, or outer loop, moves from left to right across the page, and the grouping of nodes into tiles is illustrated by drawing variously shaped boxes around sets of nodes (rather than by repositioning the nodes on the page and using only rectangular boxes to indicate tiles, as in some other work).

Figures 1 and 2 illustrate this graphical style, and the possibility of weak scaling, for a simple loop nest that updates performs T updates to N pseudo-random values in an array R (with $T = 5$ and $N = 8$ in Fig. 2; line-less arrowheads in Fig. 2 indicate values that are live-in in Fig. 1). Such a loop nest might be used as part of a Monte Carlo simulation, but for simplicity we show only the updates to the random seeds here. Since each value of R[i] depends on the value of R[i] in the previous time step, there is no flow of information between the different random sequences, and we can enable concurrent execution by dividing the operations into groups or *tiles* in which all references to any given array element are in the same tile (as outlined in the figure). Weak scaling occurs if P processors can execute P such tiles in the time needed for one processor to execute one such tile.

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

Fig. 1. "Embarrassingly Parallel" Loop Nest.

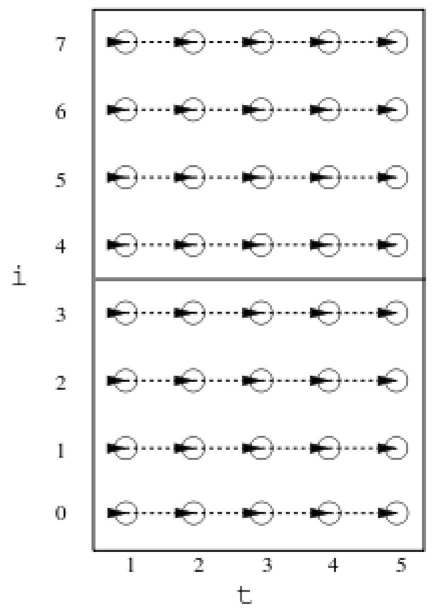

Fig. 2. Iteration Space, Data-flow, and Tiling, of Fig. 1.

The example of Figures 1 and 2 is often referred to as *embarrassingly parallel*, since weak scaling can be achieved in a relatively straightforward manner. This example illustrates both scalable parallelism and what Wonnacott called *scalable locality* [Won02]: the ability to scale up, with some problem size parameter, the amount of processing done per reference to main memory. Consider the execution of a tile of size $(\tau \times \sigma)$, i.e., τ iterations of the t (time) loop σ iterations of the i (data) loop. If we assume that the values of R reside in main memory prior to the execution of the loops of Fig. 1, we cannot avoid doing σ loads from main memory for one tile. However, if we perform all τ updates to one element before moving on to the next, we can increase the ratio of computation to memory traffic by increasing τ.

Thus for this embarrassingly parallel example, scalable parallelism and scalable locality are both straight forward. When the flow of information among computations is more complex, exposing scalable parallelism and scalable locality is less straightforward. This section details the parallel scalability issues that occur with pipelined tiles.

A. Dependences, Loop Skewing, and Pipelined Parallelism

In a Jacobi stencil computation, each array element is updated with an average of its value and its neighbor's values, as shown (for a one-dimensional array) in Fig. 3. If we were to

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

Fig. 3. Three Point Jacobi Stencil.

try to group loop iterations of this code as we did in Fig. 2, the iterations at the top and bottom edges of each tile would refer to elements from an neighboring tile. Such tilings are (appropriately) avoided by automatic parallelizers, as they produce high communication/synchronization costs. (The degenerate case of $\tau = 1$ avoids intra-tile synchronization, but prevents scalable locality.)

Fig. 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 averaging operations; copies from `new` into `A` are omitted for simplicity, as they have no impact on data-flow. For the sake of readability, the tiles shown in Fig. 4 contain far fewer iterations then would be typical. Tile sizes depend on a number of hardware performance characteristics; the PLuTo default is to create tiles of size 32 in every dimension [Bon12a], though this can be overridden at compile time. For some conditions, such as when processors dramatically outpace their memory system or nearest-neighbor communication infrastructure, much larger sizes are appropriate: [Won02] used tiles of size 1200 by 500 iterations, and [ABDW12] used size 1250 by 1250.

Array data flow analysis [Fea91], [PW92], [PW98] 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 number of data-flow arcs crossing a tile boundary describes the volume of 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 Fig. 4, concurrent execution 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 [KBB$^+$07]. Even the semi-automatic AlphaZ system [YBG$^+$12], 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.

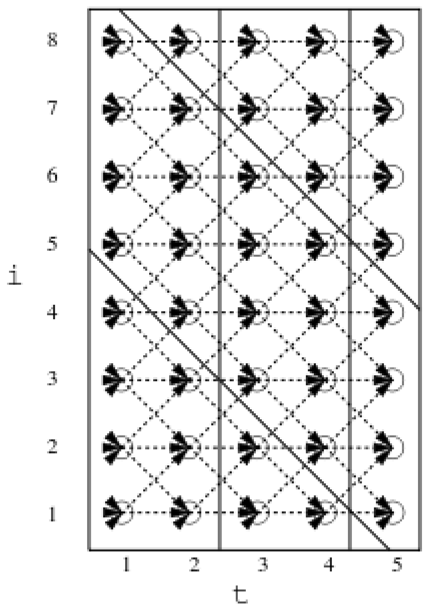

Fig. 4. Iteration Space, Data-flow and Tiling of Fig. 3.

B. Scalability

To support our claim that this pipelined tiling does not always provide scalable parallelism, it is sufficient to 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 Fig. 3. We will first do so, and then discuss the issue in higher dimensions.

For some problem sizes, the tiling of Fig. 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. Fig. 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 Fig. 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 Fig. 5 when $T = 8\tau$ (note that the upper-right region is symmetric with

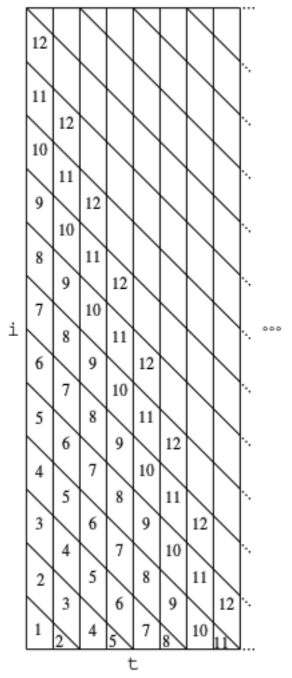

Fig. 5. Wavefronts of Pipelined Tile Execution.

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.

Even if we start with a more realistic problem size, e.g., one for which $N \gg T \gg P$, the pipelined tiling still limits scalability. Consider what happens we apply the tiling of Fig. 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 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.

C. A Note on Communication Cost

The above analysis ignores time for synchronization and communication, as if the computation were to be executed on the notoriously unscalable PRAM abstraction. Note that this is not an appallingly unrealistic assumption: the costs of communication between processors, or between processors and RAM, can be controlled by proper setting of tile sizes, as discussed in [Won00], [KBB+07] and other work.

D. 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, it may be the case that scaling of parallelism is restricted to only some of the spatial dimensions.

III. 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 Fig. 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 Fig. 5, with gray shading highlighting one tile from time step two in each figure. We delve into the complexities of more complex codes only as necessary to make our point.

Note that we focus 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 [SSPS10], 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 provenence, the information flow among atomic parallelogram tiles still forces execution to proceed along the diagonal wavefront, and thus still limits asymptotic scalability.

A. Overlapped Tiling

A number of projects have experimented with what is commonly called overlapped tiling [SO98], [BDQ98], [ADF+00], [RIF01], [DH01], [RD02], [KBB+07], [CSN+10], [NSC+10], [ZGG+12]. 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. Fig. 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

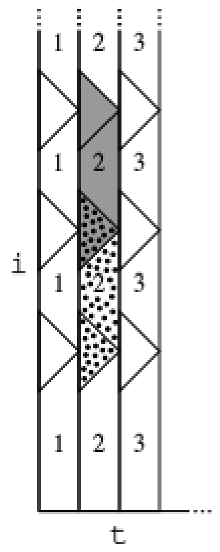

Fig. 6. Overlapped Tiling.

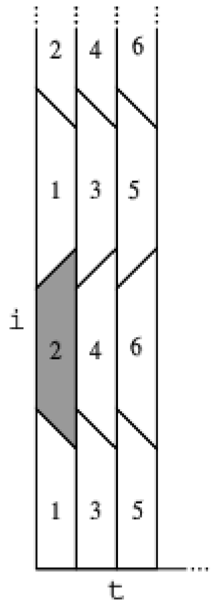

Fig. 7. Trapezoidal Tiling.

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 [SCF$^+$02], [SCFK04] for irregular iteration spaces has also been investigated by Demmel et al. [DHMY08] in the context of iterative sparse matrix computations where the tiling is a run-time reordering transformation [SCF03].

B. Trapezoidal Tiling

Frigo and Strumpen [FS05], [FS06], [FS09] propose an algorithm for limiting the asymptotic cache miss rate of "an idealized parallel machine" while providing scalable parallelism. Fig. 7 illustrates that even a simplified version of their approach 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 [FS09]). In our Fig. 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 [FS09].

The limitation of this approach is not its scalability, but rather the challenge of implementing it in a general-purpose compiler. Tang et al. [TCK$^+$11] have developed the Pochoir compiler, based on a variant of Frigo and Strumpen's techniques with a higher degree of asymptotic concurrency [TCK$^+$11]. 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.

C. Diamond Tiling

Strzodka et al. [SSPS11], [SSP11] 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 Fig. 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 Fig. 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. [SSPS11] 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 [SSPS11], Strzodka et al. explicitly state that their results are somewhat surprising in

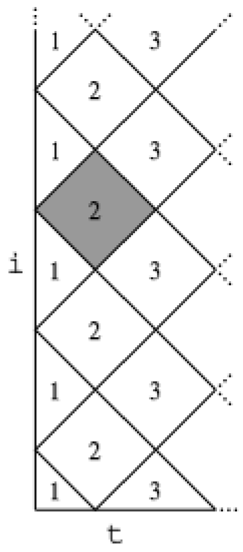

Fig. 8. One data dimension and the time dimension in a diamond tiling [SSPS11]. The diamond extends into a diamond tube in the second data dimension.

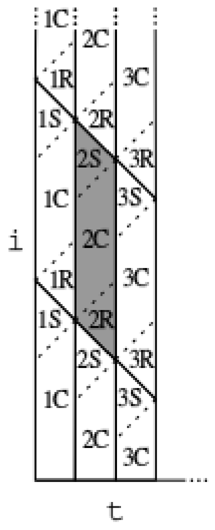

Fig. 9. Molecular tiling.

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.

D. Molecular Tiling

Wonnacott [Won00] described a tiling for stencils that allows true weak scaling for higher-dimensional stencils, performs no redundant 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* [PR99] and illustrated in Fig. 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 extension of 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 [PR99]. 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 [KPR95], and Omega has no direct facility for generating the required communication primitives.

The developers of PLuTo explored a similar *split tiling* [KBB$^+$07] approach, and demonstrated improved performance over the pipelined tiling, but this approach was not used for the released implementation of PLuTo.

E. A New Hope

We have recently learned of ongoing work on the PLuTo system [BPB12] that we believe will address the issue of true scalability without placing unrealizable demands on the code generator. The authors frame their proposal in terms of "enabling concurrent start-up" rather than improving asymptotic scalability. However, our hand-constructed examples encourage us to suspect that this tiling provides true weak scaling (though we have not yet constructed a proof).

The authors of [BPB12] discuss scalability and show improved results (vs. pipelined tiling) for current shared-memory systems up to 16 cores. However, they do not discuss asymptotic complexity. 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 [ABDW12] or [Bon12b].

F. 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 hope to better understand the expressiveness of various polyhedral libraries than can be used within compilers, as well as tools that allow the direct control of these libraries from a text input, such as AlphaZ [YBG+12] and the Omega Calculator [KMP+96].

IV. 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.

V. Conclusions

Current work on loop tiling appears to exhibit 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 may already provide an answer to this to dilemma, but further study is required for confirmation.

References

[ABDW12] Mohamed Abdalkader, Ian Burnette, Tim Douglas, and David G. Wonnacott. Distributed shared memory and compiler-induced scalable locality for scalable cluster performance. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:688–689, 2012.

[ADF+00] Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Tom Goodale, Nick Karonis, Matei Ripeanu, Ed Seidel, and Brian Toonen. 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.

[BDQ98] 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.

[BHR08] Uday Bondhugula, Albert Hartono, and J. Ramanujam. A practical automatic polyhedral parallelizer and locality optimizer. In *In PLDI 08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation*, 2008.

[BLKD09] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.*, 35(1):38–53, 2009.

[Bon12a] Uday Bondhugula. Personal communication, July 2012.

[Bon12b] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. Preprint, 2012.

[BPB12] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. Preprint, to appear in SuperComputing '12, 2012.

[BVB+09] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *PPOPP*, 44(4):219–228, 2009.

[CKV10] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard W. Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[CNvdGZ10] Ernie Chan, Jim Nagle, Robert van de Geijn, and Field Van Zee. Transforming linear algebra libraries: From abstraction to parallelism. In *Proceedings of the 15th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, 2010.

[Col02] Jean-Francois Collard. *Reasoning about Program Transformations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[CSN+10] M. Christen, Olaf Schenk, E. Neufeld, M. Paulides, and H. Burkhart. Manycore Stencil Computations in Hyperthermia Applications. In J. Dongarra, D. Bader, and J. Kurzak, editors, *Scientific Computing with Multicore and Accelerators*, pages 255–277. CRC Press, 2010.

[DH01] Chris Ding and Yun He. A ghost cell expansion method for reducing communications in solving pde problems. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Supercomputing '01, pages 50–50, New York, NY, USA, 2001. ACM.

[DHMY08] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[FS05] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[FS06] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '06, pages 271–280, New York, NY, USA, 2006. ACM.

[FS09] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theor. Comp. Sys.*, 45(2):203–233, June 2009.

[Gus88] John L. Gustfson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[HRS09] J. D. Hogg, J. K. Reid, and J. A. Scott. A dag-based sparse cholesky solver for multicore architectures. Technical report, Science and Technology Facilities Council, April 27, 2009.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 319–329, 1988.

[KBB+07] 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)*, volume 42, pages 235–244, New York, NY, USA, 2007. ACM.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Calculator and Library. Technical report, Dept. of Computer Science, University of Maryland, College Park, April 1996.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[LZDK11] Jun Liu, Yuanrui Zhang, Wei Ding, and Mahmut T. Kandemir. On-chip cache hierarchy-aware tile scheduling for multicore machines. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization*, pages 161–170, 2011.

[NSC+10] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 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*, SC '10, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society.

[Ope] OpenMP. http://openmp.org/wp/.

[PR99] William Pugh and Evan Rosser. Iteration slicing for locality. In *12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[PW92] William Pugh and David Wonnacott. Eliminating false data dependences using the Omega test. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 140–151, San Francisco, California, June 1992.

[PW98] William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Programming Languages and Systems*, 20(3):635–678, May 1998.

[QOIQOvdG09] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 121–130, New York, NY, USA, 2009. ACM.

[RD02] Fabrice Rastello and Thierry Dauxois. 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*, IPDPS '02, pages 138–, Washington, DC, USA, 2002. IEEE Computer Society.

[RIF01] Matei Ripeanu, Adriana Iamnitchi, and Ian T. Foster. Cactus application: Performance predictions in grid environments. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 807–816, London, UK, UK, 2001. Springer-Verlag.

[SCF+02] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, Jonathan Freeman, and Barbara Kreaseck. 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.

[SCF03] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. 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.

[SCFK04] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–114, February 2004.

[SL99] Yonghong Song and Zhiyuan Li. New tiling techniques to improve cache temporal locality. *ACM SIGPLAN Notices (PLDI)*, 34(5):215–228, May 1999.

[SO98] Aaron Sawdey and Matthew T. O'Keefe. 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*, LCPC '97, pages 79–93, London, UK, UK, 1998. Springer-Verlag.

[SSP11] Robert Strzodka, Mohammed Shaheen, and Dawid Pajak. Time skewing made simple. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP*, pages 295–296. ACM, 2011.

[SSPS10] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*, pages 49–59. ACM, June 2010.

[SSPS11] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the 40th International Conference on Parallel Processing (ICPP)*, pages 517–581, Taipei, Taiwan, September 2011. IEEE Computer Society.

[SYD09] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–11, New York, NY, USA, 2009. ACM.

[TCK+11] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[VSMP96] R. F. Van der Wijngaart, S. R. Sarukkai, Mehra, and P. The effect of interrupts on software pipeline execution on message-passing architectures. In ACM, editor, *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing: Philadelphia, Pennsylvania, USA, May 25–28, 1996*, pages 189–196, New York, NY 10036, USA, 1996. ACM Press.

[WHW09] Markus Wittmann, Georg Hager, and Gerhard Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. *CoRR*, abs/0912.4506, 2009.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, New York, NY, USA, 1991. ACM.

[Wol89] Michael J. Wolfe. More iteration space tiling. In ACM, editor, *Proceedings, Supercomputing '89, Reno, Nevada*, pages 655–664, Reno, Nevada, November 1989. ACM Press.

[Won00] David Wonnacott. Using Time Skewing to eliminate idle time due to memory bandwidth and network limitations. In *International Parallel and Distributed Processing Symposium*. IEEE, May 2000.

[Won02] David Wonnacott. Achieving scalable locality with Time Skewing. *Internation Journal of Parallel Programming*, 30(3):181–221, June 2002.

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

[ZGG+12] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.