Parameterized Diamond Tiling for Stencil Computations with Chapel parallel iterators

Ian J. Bertolacci
Colorado State University
Ft. Collins, Colorado
ibertola@cs.colostate.edu

Catherine Olschanowsky
Colorado State University
Ft. Collins, Colorado
cathie@cs.colostate.edu

Ben Harshbarger
Cray Inc.
Seattle, Washington
bharshbarg@cray.com

Bradford L. Chamberlain
Cray Inc.
Seattle, Washington
bradc@cray.com

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

Michelle Mills Strout
Colorado State University
Ft. Collins, Colorado
mstrout@cs.colostate.edu

ABSTRACT

Stencil computations figure prominently in the core kernels of many scientific computations, such as partial differential equation solvers. Parallel scaling of stencil computations can be significantly improved on multicore processors using advanced tiling techniques that include the time dimension, such as diamond tiling. Such techniques are difficult to include in general purpose optimizing compilers because of the need for inter-procedural pointer and array data-flow analysis, plus the need to tune scheduling strategies and tile size parameters for each pairing of stencil computation and machine.

Since a fully automatic solution is problematic, we propose to provide parameterized space and time tiling iterators through libraries. Ideally, the execution schedule or tiling code will be expressed orthogonally to the computation. This supports code reuse, easier tuning, and improved programmer productivity. Chapel iterators provide this capability implicitly. We present an advanced, parameterized tiling approach that we have implemented using Chapel parallel iterators. We show how such iterators can be used by programmers in stencil computations with multiple spatial dimensions. We also demonstrate that these new iterators provide better scaling than a traditional data parallel schedule.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Control structures; D.3.4 [**Processors**]: Code generation, Optimization

Keywords

stencil computations, diamond tiling, Chapel, parallel iterators, separation of concerns

1. INTRODUCTION

Stencil computations are ubiquitous in scientific simulation applications, often composing the most time-intensive kernels of the application. Advanced optimization techniques, such as *diamond tiling* [1] reduce memory bandwidth pressure, which is crucial to efficient scaling and is becoming more important as memory hierarchies deepen and the bandwidth-to-flops ratio decreases [18]. However, applying complex tiling techniques manually to application code obfuscates the core computational kernels and causes code maintenance issues by complicating the control flow. Additionally, tiling techniques require architecture-specific tile size selection, and the inclusion of such tuning parameters within the primary specification of a computational kernel thwarts performance portability. This work provides solutions to the challenges associated with diamond tiling using a framework that can be extended to other advanced tiling techniques.

Loop tiling [17] changes the execution order of the iterations of a loop nest to improve data locality, which in turn reduces memory bandwidth pressure. Tiling has been shown to improve both single-processor performance and the scalability of performance on parallel systems. Including the time dimension in the tiling takes further advantage of the available locality in stencil operations [14,24,25,30,31]. Diamond tiling further improves performance by allowing work on tiles to begin simultaneously, thus increasing available parallelism.

Most optimizing compilers do not include diamond tiling techniques. Even when a compiler can perform this optimization on a small benchmark, it may fail to do so on a larger code due to program analysis limitations. This leaves application developers no choice but to include such tiling transformations in the primary specification of the computation. This is undesirable for two reasons: (1) The tiling loop structures obfuscate the original stencil computation to the point of making them virtually unrecognizable [21]. (2) The size and shape of the tiles required varies with architecture; including this diminishes portability.

The goal of this work is to enable the quick adoption of novel tiling techniques without obfuscating application code. This is achieved by removing the execution schedule from the primary specification of the computation using Chapel

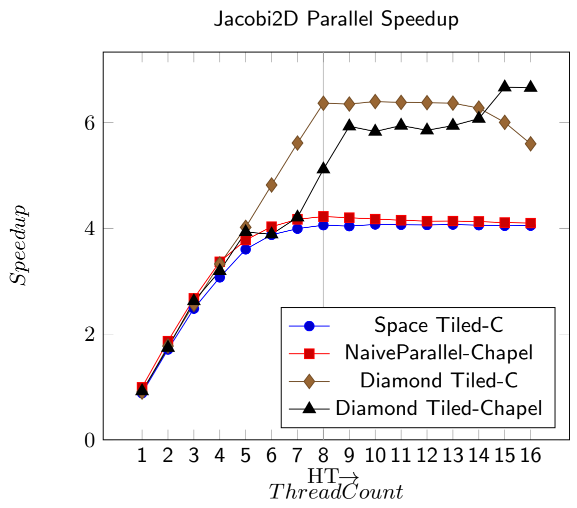

Figure 1: Speedup of the Jacobi2D benchmark with respect to a serial version of space tiling implemented in C+OpenMP. Diamond tiling clearly outperforms the parallel space tiling version, and the version implemented with a parameterized Chapel parallel iterator is competitive with the comparable C+OpenMP version when the number of threads is a multiple of the number of cores.

parallel iterators. The iterator implementation of diamond tiling is parameterized to enable performance tuning.

1.1 Motivation for Parameterizing Tiling

Currently, applying diamond tiling requires manually including the tiling loops in the primary expression of the computation. The source-to-source compiler Pluto [7] can automate diamond tiling in some cases and code generation tools can be used to aid with this process; however, they require tile sizes be specified for non-rectangular tiles [3,23].

Given a fixed tile size, the iteration space that results from diamond tiling is affine. However, adding the parameter size as a variable causes the iteration space to no longer be affine. This is because the tile size is involved in a multiplication determining the bounds of the generated loops. Current code generation tools cannot handle non-affine iteration spaces. Parameterizing the diamond tiling overcomes this challenge, thereby simplifying the development and tuning workflow.

The workflow for applying diamond tiling within a stencil computation currently involves the following steps:

1. develop the code using a standard execution schedule,
2. generate the tiling loops for a best-guess tile size,
3. manually incorporate the generated code,
4. recompile and run a performance test, and
5. improve the tile size guess and return to step 2 until satisfactory performance has been achieved.

By parameterizing the generated code, we remove the repeated manual step, prevent the need for code regeneration and recompilation, and enable auto-tuning using run-time parameters.

1.2 Motivation for Using Iterators

The loop nests necessary for a tiled execution schedule are complex and their direct use within application code leads to code duplication, challenging and error-prone incorporation, and obfuscated application code. Iterators are an increasingly common programming construct that solve these challenges by providing code modularity for the loop nests themselves. Modular code lends itself to reuse, testing, and maintenance.

An additional benefit to this approach is that parameterized iterators provide a level of performance portability. Performance portability means that the code can be optimized for various architectures without changing the primary specification of the calculations. Given that the iterator can be parameterized for tuning parameters, in this case tile size, the parameters can be changed to accommodate varying target architectures.

1.3 Technical Contributions

The technical contributions of this paper include:

- **Parameterizing diamond tiling.** We present a systematic method for creating diamond tiling code that is parameterized for tile size and specific to the tiling hyperplanes.
- **Implementing the parameterized diamond tiling using Chapel parallel iterators.** This separates the expression of the tiling schedule (the loop nest) from the expression of the computation itself (the loop body). We created a proof of concept that shows we can develop libraries of schedules that could be adopted by end-users.

We demonstrate that the parameterized diamond tiling variants perform as well as fixed-size diamond tiling variants, and reproduce results showing that diamond tiling is faster than rectangular space tiling. The Chapel parallel iterators demonstrate competitive performance while providing significant programmability benefits over the C+OpenMP implementations (see Figure 1). The results are discussed in detail in Section 5. Section 7 concludes.

2. BACKGROUND

This work makes practical the use of advanced tiling techniques, specifically diamond tiling, within stencil-based applications. This section provides background about stencil applications in general, the stencil benchmarks used in this work, standard space and time tiling techniques, and diamond tiling.

2.1 Stencil Computations/Benchmarks

A stencil computation is one in which the result, or update to each cell, depends on the values of some set of neighbors. In this paper, we use the Jacobi 1D and 2D benchmarks. These benchmarks are general representations of other Jacobi-like stencils with dependencies through time that are common in applications, such as heat and discrete wave equations. The quintessential Jacobi stencil is the average of a Von Neumann neighborhood of depth 1 over a series of time-steps.

A one-dimensional implementation of Jacobi can be set up to simulate heat dissipation through a theoretical one dimensional rod. In our implementation we set the initial conditions to 0.0 at each end and use a random number generator to assign values at each discreet point within the rod. The “dissipation” is calculated using an average of each point and its direct neighbors. It is not likely that a scientific application will use a Jacobi1D calculation on its own;

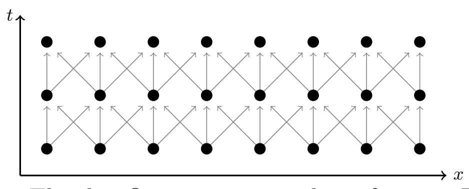

Figure 2: The dataflow pattern resulting from a 1D stencil of depth 1. The vertical axis is time.

however, this is a common case study for temporal tiling approaches.

Figure 2 illustrates the data dependency pattern in the Jacobi1D benchmark. Each cell is updated using the following statement:

```
  A[t,i] = ( A[t-1, i-1]
           + A[t-1, i  ]
           + A[t-1, i+1] ) / 3 ;
```

This operation is fully parallel within a time step. In fact a naive parallelization of Jacobi1D surrounds this statement with two loops, an outer time loop, and an inner space loop. An OpenMP statement, or other programming construct for thread-level parallelism, can be used to indicate that the inner loop should be executed in parallel.

The 2D case can be thought of as simulating heat dissipation through a theoretical surface. Each cell in the surface has 4 neighbors (not using diagonal neighbors). The cell update statement is the following:

```
 A[t,i,j] = ( A[t-1,i,j-1]
            + A[t-1,i,j+1]
            + A[t-1,i,j  ]
            + A[t-1,i-1,j]
            + A[t-1,i+1,j] ) / 5 ;
```

As with the Jacobi1D operation, within a single time step the loops executing this statement are fully parallel. A naive implementation executes one of the two space dimension loops in parallel. One obvious improvement to this is to collapse the space loops into a single loop and parallelize that. We refer to this as naive parallelization. Given that the problem size falls outside of last-level cache, it benefits performance to apply a simple space-only tiling using rectangular tiles laid over the plane. This implementation is referred to as a naive space tiling variant of the benchmark.

2.2 Rectangular Tiling in Space

Rectangular tiling breaks a large iteration space into a set of smaller iteration spaces [17]. The goal in doing so is to improve spatial and temporal locality. The advantages of this for 2D data are very clear. When iterating over a large 2D dataset applying a 5 point stencil (north, south, east, west, self), it is likely that the north neighbor will be pushed out of cache before the iteration comes around to the point again (this time as self). Breaking the larger iteration space into smaller tiles can prevent that cache miss.

2.3 Diamond Tiling

Diamond tiling [1] is a method of space-time tiling that provides parallelism, including concurrent startup, while maintaining data locality. The original presentation of diamond tiling demonstrated excellent performance and scaling, beating the previous state of the art [6] that required a wavefront startup, and, therefore, had less available parallelism.

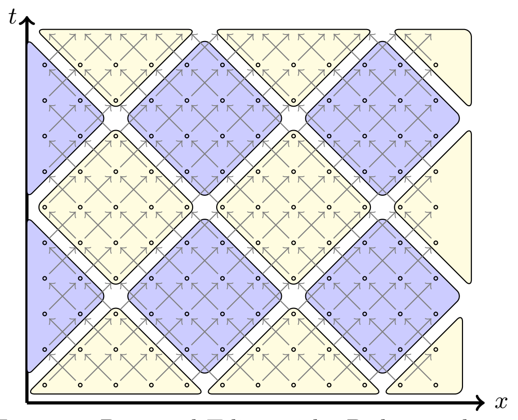

Figure 3: Diamond Tiling with 1D data and time.

Figure 3 shows a possible diamond tiling for a 1D stencil operation across many time steps. The available parallelism comes in wavefronts. The first wavefront of tiles that can be executed concurrently is the set of lower yellow tiles. This is then followed by the first row of blue tiles, the second row of yellow tiles, and so on. The number of tiles that compose the width of the problem domain determines the amount of parallelism available.

The downside of applying diamond tiling to an application is the difficulty of implementation. Diamond tiling code — the loop structures and bounds necessary to execute the schedule — is determined by the slopes and the spacing of the hyperplanes. Without a fixed procedure for creating the code it is time consuming and error-prone (see Figure 4 for a preview of the code complexity).

3. PARAMETERIZED DIAMOND TILING

In this section we describe a method to parameterize any tiling for a given set of tiling hyperplanes assuming a hyperrectangular iteration space. First, we review how tiling hyperplanes are used to specify a tiling schedule and why, if the tile sizes in those schedules are left as parameters, the schedule is non-affine. Non-affine schedules cannot be handled by existing code generators. Then, we describe a tractable process for deriving the loops over the tiles and the loops over iterations within such tiles while keeping the tile size a parameter.

3.1 Tiling Hyperplanes

Tiling hyperplanes determine the shape and, therefore, the iteration space within tiles. Figure 3 illustrates the tiles that result from two sets of hyperplanes: one set of lines with slope -1 and the other set with slope 1. Each set of tiling hyperplanes can be specified by a vector normal to the hyperplanes and the distance between hyperplanes in that set. The normal for the -1 slope hyperplanes in Figure 3 is $(1, 1)$ and for the 1 slope hyperplane is $(-1, 1)$.

Given tiling hyperplanes, it is possible to specify a tiling to a polyhedral code generator with a scattering function. A *scattering function* maps the original iteration space for a loop into a new iteration space. The code generator creates loop code that executes the computations in the new iteration space in lexicographical order.

Assume a d-dimensional rectangular iteration space $\{[i_0, i_1, ..., i_{d-1}] \mid (l_0 \le i_0 < u_0) \wedge ... \wedge (l_{d-1} \le i_{d-1} < u_{d-1})\}$.

The scattering function for a d-dimensional tiling with hyperplane normals $\vec{v}_0$, $\vec{v}_1$, ..., $\vec{v}_{d-1}$ is the following:

$$\{[i_0, i_1, ..., i_{d-1}] \rightarrow [k_0, k_1, ..., k_{d-1}, i_0, i_1, ..., i_{d-1}] \mid (k_0 = (\vec{v_0} \cdot \vec{i})/\tau) \wedge ... \wedge (k_{d-1} = (\vec{v_{d-1}} \cdot \vec{i})/\tau)\},$$

where k_0, k_1, ... k_{d-1} are the iterators for the new tile loops and τ is the spacing between hyperplanes for all d sets of tiling hyperplanes[1]. Using the Chinese remainder theorem, we can remove the division by introducing remainders r_0, r_1, ..., r_{d-1} to obtain the following:

$$\{[i_0, i_1, ..., i_{d-1}] \rightarrow [k_0, k_1, ..., k_{d-1}, i_0, i_1, ..., i_{d-1}] \mid \exists r_0, r_1, ..., r_{d-1}, (0 \leq r_0 < \tau) \wedge (\tau k_0 + r_0 = \vec{v_0} \cdot \vec{i}) \wedge ... \wedge (0 \leq r_{d-1} < \tau) \wedge (\tau k_{d-1} + r_{d-1} = \vec{v_{d-1}} \cdot \vec{i})\}.$$

The multiplication of the tile size τ by each of the tile iterators causes the scattering function to be non-affine.

3.2 Plan for Tile Size Parameterization

We could do a symbolic Fourier Motzkin elimination process to obtain the new loop bounds, but that results in an exponential blow-up in loop bounds. All of the extra loop bounds result in more loop overhead.

Approaches already exist to do parametric tilings when the tiles are rectangular. With rectangular tiling, developing a parameterized tiling can be split into two problems: (1) expanding the iteration space so it includes all possible tile origins for a parameterized-sized tile and having the tile loops iterate over these origins, and (2) generating parameterized tile loops that iterate over the points within a tile [16, 22, 23]. This approach of splitting the code generation problem into loops over tiles and then loops within tiles was also used by Goumas et al. [13] to generate code for fixed-sized, non-rectangular tiles. The parametric tiling techniques work with rectangular tiles and any polyhedral iteration space, but leverage the fact that with rectangular tiles the tile boundaries align with the iteration axes (although possibly in the skewed iteration space).

Diamond tilings have hyperplanes that do *not* align with the iteration space axes. It is not possible to skew the iteration space to compensate without introducing holes in the iteration space. However, since we are focused on diamond tilings in the context of stencil computations, we can leverage the fact that the iteration space is rectangular to split the problem into two similar pieces: (1) finding bounds on the tile space (i.e., bounds on the k iterators) and (2) generating parameterized tile loops that iterate over the points within a tile. Our solution to each of the sub-problems is different because the tile iterators k_j are iterating within a separate tile space instead of over the tile origins in the original iteration space.

3.3 Iterating Over Tiles

In the context of diamond tiling, the loops over tiles do not iterate over tile origins as they do in rectangular tiling. Rather, they loop over a separate, parameterized tile space. The parameterized tile loops are found by characterizing a single tile as a parameterized set and then projecting that set on the axes of the original iteration space. A single, representative tile can be characterized by the following set:

$$\{[k_0, k_1, ..., k_{d-1}, i_0, i_1, ..., i_{d-1}] \mid \exists r_0, r_1, ..., r_{d-1}, (0 \leq r_0 < \tau) \wedge (\tau k_0 + r_0 = \vec{v_0} \cdot \vec{i}) \wedge ... \wedge (0 \leq r_{d-1} < \tau) \wedge (\tau k_{d-1} + r_{d-1} = \vec{v_{d-1}} \cdot \vec{i})\}.$$

To determine the bounds on the tile iterators k_0, k_1, ..., k_{d-1}, we project the single tile, which is parameterized by tile size, on each of the canonical axes of the rectangular iteration space, i_0, i_1, ..., i_{d-1}. With the projection, we will have the min and max values for each tile on each axis. Then we can make sure that if even one point of the tile is in the iteration space, the tile is visited. The tile's lower bounds are inclusive, and therefore anytime the projected tile lower bound is less than or equal to the iteration space upper bound, the tile contains at least one iteration point that is in the iteration space.

It is possible to compute the lower and upper bounds for a single tile (i.e., setting all k_j values to 1) projected on each axis with a code generator as was done in ISCC [27], but the loop bounds generated are inclusive. We did the projection by hand to determine the real number of intersection points on the axes. We did the projection for stencils with depth 1 in two-dimensional and three-dimensional iteration spaces for use with Jacobi1D and Jacobi2D, but here we show the results for the three-dimensional iteration spaces.

For the three-dimensional diamond tiling for Jacobi2D, the hyperplanes are $v_0 = (1, 1, 0)$, $v_1 = (1, 0, 1)$, and $v_2 = (1, -1, -1)$. Given this set of hyperplanes the iteration space within a tile is defined as shown in Equation 1.

$$\begin{aligned} \{[t, i, j] | \exists r_0, r_1, r_2, \\ & (0 \leq r_0 < \tau) \wedge (0 \leq r_1 < \tau) \wedge (0 \leq r_2 < \tau) \\ & \wedge\ k_0\tau + r_0 = t + i \\ & \wedge\ k_1\tau + r_1 = t + j \\ & \wedge\ k_2\tau + r_2 = t - i - j\} \end{aligned} \tag{1}$$

Here are the projections of the parameterized tile on the axes:

$$\begin{array}{rcl} (k_0 + k_1 + k_2)\tau/3 \leq & t & < (3 + k_0 + k_1 + k_2)\tau/3 \\ (2k_0 - k_1 - k_2 - 2)\tau/3 < & i & < (2 + 2k_0 - k_1 - k_2)\tau/3 \\ (2k_1 - k_0 - k_2 - 2)\tau/3 < & j & < (2 + 2k_1 - k_0 - k_2)\tau/3 \end{array}$$

To find the bounds for the tile-iterator loops k_0, k_1, and k_2, we substitute the (unique) upper (or lower) bound on each dimension of the iteration space for the corresponding iteration space variable. For example, to be sure to visit every tile whose minimum t value is below the upper bound of the iteration space (T), we substitute T for t in the above lower bound on t, producing the upper bound on k_0 of $(k_0 + k_1 + k_2)\tau/3 \leq T$. Substituting 1 for t in the upper bound on t gives $(3 + k_0 + k_1 + k_2)\tau/3 > 1$. Continuing this process for i and j, we produce the following set of constraints for the tile iterators:

$$\begin{aligned} &\{[k_0, k_1, k_2]| \\ &(3 + k_0 + k_1 + k_2)\tau/3 > 1 \wedge (k_0 + k_1 + k_2)\tau/3 \leq T \\ &\wedge(2 + 2k_0 - k_1 - k_2)\tau/3 > L_i \\ &\wedge(2k_0 - k_1 - k_2 - 2)\tau/3 < U_i \\ &\wedge(2 + 2k_1 - k_0 - k_2)\tau/3 > L_j \\ &\wedge(2k_1 - k_0 - k_2 - 2)\tau/3 < U_j\} \end{aligned}$$

[1]Using different spacing per set of hyperplanes would be more general, but results in less regular parallel wavefronts of diamond tiles and causes the parameterization to be more complex.

```
// Loop over tile wavefronts.
for (kt=ceild(3,tau)-3; kt<=floord(3*T,tau); kt++) {

  // The next two loops iterate within a tile wavefront.
  int k1_lb = ceild(3*Lj+2+(kt-2)*tau,tau*3);
  int k1_ub = floord(3*Uj+(kt+2)*tau,tau*3);
  int k2_lb = floord((2*kt-2)*tau-3*Ui+2,tau*3);
  int k2_ub = floord((2+2*kt)*tau-3*Li-2,tau*3);

  //Loops over tile coordinates within a parallel wavefront of tiles.
  #pragma omp parallel for ...
  for (k1 = k1_lb; k1 <= k1_ub; k1++) {
    for (x = k2_lb; x <= k2_ub; x++) {
      k2 = x - k1;  // Removing k1 term from k2 upper and lower bounds enables collapse(2).

      // Loop over time within a tile.
      for (t =  max(1, floord(kt*tau-1, 3)); t < min(T+1, tau + floord(kt*tau, 3)); t++) {
          write = t & 1; // equivalent to t mod 2
          read = 1 - write;

        // Loops over the spatial dimensions within each tile.
        for (i =  max(Li,max((kt-k1-k2)*tau-t, 2*t-(2+k1+k2)*tau+2));
             i <= min(Ui,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau)); i++) {
          for (j =  max(Lj,max(tau*k1-t, t-i-(1+k2)*tau+1));
               j <= min(Uj,min((1+k1)*tau-t-1, t-i-k2*tau)); j++) {
                 A[write][x][y] = (A[read][x-1][y] + A[read][x][y-1] + ... ;
} } } } } }
```

Figure 4: Parameterized diamond tiled code for any 2D stencil of depth 1 that uses values from the previous time step only (Jacobi-like data dependencies). Note that the `floord` function computes the integer quotient of its operands, like C's `%` operation, but unlike C's `%` rounds any fractional part toward negative infinity rather than zero [28].

To create wavefronts of tiles that can be executed in parallel, we introduce a new tile space iterator $k_t = k_0 + k_1 + k_2$, by solving for k_0 and replacing all of the k_0s in the above bounds with $k_t - k_1 - k_2$. We then determine the bounds for the k_t, k_1, and k_2 loops to be the following:

$$\begin{aligned}
&\{[k_t, k_1, k_2] \mid k_t > 3/\tau - 3 \wedge k_t \leq 3T/\tau \\
&\wedge\ 3k_1 > (3L_j/\tau) + k_t - 2 \\
&\wedge\ 3k_1 < (3U_j/\tau) + k_t + 2 \\
&\wedge\ 3k_2 > 2k_t - 3k_1 - 2 - (3U_i/\tau)) \\
&\wedge\ 3k_2 < 2k_t - 3k_1 + 2 - (3L_i/\tau)\},
\end{aligned}$$

which can be implemented with the outer three loops shown in Figure 4.

3.4 Iterating within a Diamond Tile

After determining the loop bounds for the tile loops, the next step is to determine the bounds for the loops that iterate over points within each tile. We start with the set that specifies a single tile (see Equation 1). Then we do the wavefront substitution $k_0 = k_t - k_1 - k_2$.

The resulting loop bounds are not affine because of the multiplication of two parameters in three of the constraints: $k_0\tau$, $k_1\tau$, and $k_2\tau$. To enable code generation with ISCC, we perform a substitution. Let $x_t = k_t\tau$, $x_1 = k_1\tau$, and $x_2 = k_2\tau$. This is legal because the tuple (k_t, k_1, k_2) is constant. The resulting loop bounds over points within a tile for the three-dimensional diamond tiling of Jacobi2D are shown in Figure 4.

In summary, we can apply the tile size parameterization process to any set of tiling hyperplanes (diamond or otherwise). Therefore, other stencils that require different diamond tiling hyperplanes can also benefit from this process.

4. PROVIDING TILING IN A LIBRARY

Due to the complexity of the diamond tiling loop structure it is desirable to abstract it away from the primary specification of the algorithm. We do this by implementing the diamond tiling code within a Chapel parallel iterator. This section provides a brief overview of the Chapel language and information about our implementation.

4.1 Chapel Parallel Iterators

We use Chapel parallel iterators to explore the viability of providing advanced tiling schedules as a module. Chapel is an open-source language currently in development at Cray Inc.[2] which is designed to simplify parallel programming for the desktop and at scale via various language features [10, 11]. It represents a growing movement to design languages and compilers that abstract away the complexity of developing highly parallel programs. Sample features include: *forall loops* and a rich set of domains and arrays for data parallel computations; support for task-based concurrent programming including data-centric coordination between tasks via atomic variables and *sync variables* with full-empty semantics; and *locales* as a first-class language concept for reasoning about architectural locality. Chapel follows a *multiresolution philosophy* in which users can control low-level details like parallel loop schedules or array layout and distribution while making them available to others via high-level abstractions.

As part of its productivity-oriented features, Chapel sup-

[2]http://chapel.cray.com

```
iter DiamondTileIterator( L: int, U: int, T: int, tau: int,
                          param tag: iterKind): 4*int
                          where tag == iterKind.standalone {
  // Loop over tile wavefronts.
  for kt in ceild(3,tau) .. floord(3*T,tau) {

    // The next two loops iterate within a tile wavefront.  Assumes a square iteration space.
    var k1_lb: int = floord(3*L+2+(kt-2)*tau, tau*3);
    var k1_ub: int = floord(3*U+(kt+2)*tau-2, tau*3);
    var k2_lb: int = floord((2*kt-2)*tau-3*U+2, tau*3);
    var k2_ub: int = floord((2+2*kt)*tau-3*L-2, tau*3);

    // Loops over tile coordinates within a parallel wavefront of tiles.
    forall k1 in k1_lb .. k1_ub {
      for x in k2_lb .. k2_ub {
        var k2 = x-k1;

        // Loop over time within a tile.
        for t in max(1,floord(kt*tau,3)) .. min(T,floord((3+kt)*tau-3,3)){
          write = t & 1; // equivalent to t mod 2
          read = 1 - write;

          // Loops over the spatial dimensions within each tile.
          for i in max(L,max((kt-k1-k2)*tau-t, 2*t-(2+k1+k2)*tau+2))
                .. min(U,min((1+kt-k1-k2)*tau-t-1, 2*t-(k1+k2)*tau)) {
            for j in max(L,max(tau*k1-t,t-i-(1+k2)*tau+1))
                  .. min(U,min((1+k1)*tau-t-1,t-i-k2*tau)){
              yield (read, write, i, j);
  } } } } } } }
```

Figure 5: 2D Diamond Tile Parallel Iterator in Chapel

ports *iterators* [9], inspired by those of the CLU language [19]. An iterator is a function that *yields* values back to its callsite and then continues executing rather than simply returning a single value per call like traditional functions. In this way, iterators can be used to drive loops, or to specify the elements defining a collection. Just as traditional functions can be used to parameterize common idioms and factor them away from straight-line code, iterators can provide similar software engineering benefits for loops and loop nests, improving code reuse and readability.

This effort makes use of Chapel's multiresolution philosophy by specifying complex loop schedules using the language's features for parallel iterators and then invoking them via high-level data parallel forall loops. In this way, the details of tile specification and scheduling can be factored away from the parallel stencil computation loops themselves, making it easier for application programmers to focus on their computations without tripping over the details of tiling specification and programming.

```
forall (read, write, x ,y)
       in DiamondTileIterator(L, U, T, tau) {

  A[write, x, y] =
      (A[read,x-1,y] + A[read,x,y-1] +
       A[read,x  ,y] + A[read,x,y+1] +
       A[read,x+1,y]) / 5;
}
```

Figure 6: Use of the diamond tiling iterator. Contrast the programmabilty of this with that of Figure 4.

4.2 Iterator Implementation

Figure 5 shows the iterator code for the 2D diamond tile iterator. Inside the body of the iterator, the code is much like the diamond-tiled code in C shown in Figure 4. Translation simply consists of the modification of variable declarations, the transformation from C loop bounds to range iterations, and the replacement of stencil calls with the *yield* statement. Parallel $forall$ loops take the place of loops with OpenMP pragmas.

Figure 6 illustrates an invocation of the iterator. Using the diamond tile iterator written in Chapel code is simple. Most importantly, the computation specification is much cleaner than if the diamond tiling iterator code were inlined in the loop nest itself.

Consider a program that has a dozen invocations of one of these stencil loops with slight variations in the stencil computation. In the Chapel program, this amounts to 12 of these clean loop nests, potentially passing in different parameters such as tile size at each callsite. But in a language like C, C++, or Fortran, there is no good option for abstracting such parallel loop nests away from their uses. While one can create serial iterators via classes/structs and support begin()/next()/end()-style methods/functions on them, there is no clean way to abstract the OpenMP-parallelized loops into such methods and functions. One could create a helper function representing the loop nest and pass a function pointer representing the loop body into it, but this would result in a very expensive loop body without optimizations to inline the function pointer and specialize the loop nest function for each callsite. Alternatively, one could use something outside the language like pre-processor macro expansion to create the loop nests, but this is less robust than supporting parallel iterators in the language directly.

5. EXPERIMENTAL RESULTS

Our experimental results demonstrate that the parameterized, diamond-tiled code has *better* performance than code generated for specific tile sizes, and that using Chapel parallel iterators does not result in any appreciable overhead when compared with C+OpenMP code. Additionally, we reproduce results demonstrating the positive performance impact of diamond tiling on Jacobi 1D and 2D benchmarks. We focus on intranode parallelism, specifically leveraging the on-chip cache hierarchy, because stencil computations do not scale well at small core counts due to their memory bandwidth demands.

In this section we also present our methods for selecting tile sizes and a comparison of diamond tiling to spatial tiling in the context of Jacobi2D.

5.1 Experimental Setup

Hardware: Experiments were run on a 2.60GHz Intel Xeon E5-2650 v2 workstation. This machine has a single NUMA domain, one socket containing 8 cores (16 Hyper-Threads). Each core has its own 32K L1 data cache and 256K L2 cache. The L3 cache is 20Mb and shared among all cores. The memory is 32Gb.

Problem Size: We tested all benchmarks with problem sizes that exceeded L3 cache. The Jacobi1D problem size (N=5242880, footprint=2*N*sizeof(double)) is 2 times the size of L3. The Jacobi2D problem size (N=4096 X 4096) is 12.8 times the size of L3. Each benchmark ran 100 time steps.

Compilers: GCC (version 4.8.3) was used to compile all the C+OpenMP benchmarks, as well as the back-end for compiling the Chapel compiler. The Chapel compiler itself (version 1.11) was retrieved from the public download page maintained by Cray and the Chapel team[3].

Compiler Options: When building the serial C and C+OpenMP benchmarks the `-O3` flag was used to optimize. Chapel builds used the flags `-sassertNoSlicing`, which disables unnecessary array striding calculations, and `--fast`, which removes run-time safety checks and compiles generated C code with the `-O3` flag.

Baseline for Speedup: All speedup measurements are relative to a serial C version of the same stencil.

5.2 Tiling Methods Comparison

The diamond tiling variants of the code outperformed the naive parallel for both Jacobi1D and Jacobi2D. The naive parallel variant for Jacobi1D was parallelized with an OpenMP statement around the spatial loop using the default static schedule. Figure 7 shows that the naive parallel implementation stops effectively scaling at 4 threads even though this version tiles the spatial loops. The diamond tiling variant uses a dynamic schedule with the default chunk size being one row of tiles. Jacobi2D when diamond tiled maintains nearly linear speedup out to 8 cores.

5.3 Fixed Verses Parameterized Tile Size Performance

A key requirement for success with parameterized diamond tiling is that the parameterization does not result in a negative performance impact. Our experiments confirm that it does not. To measure this impact we generated a

[3]`https://github.com/chapel-lang/chapel`

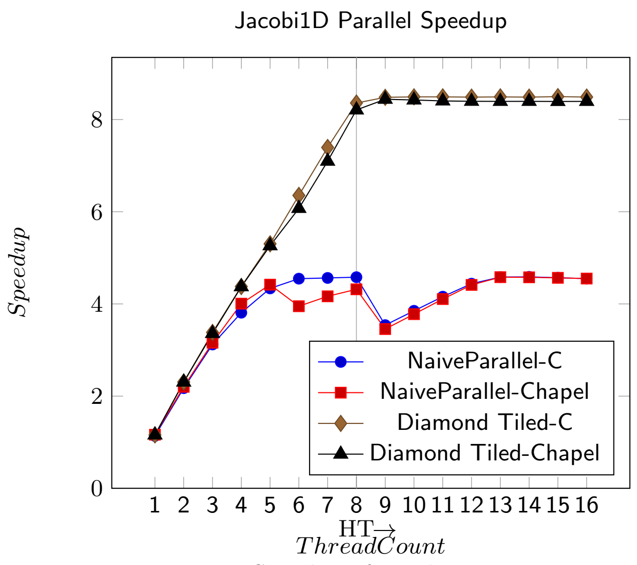

Figure 7: Speedup of Jacobi1D.

sampling of tile-size-specific variants of the code for a selection of tile sizes for Jacobi2D. The code was generated using the ISCC code generator [27], which is used as the code generator for Pluto [7], PoCC[4], and Polly [15]. Figure 8 shows a comparison of the execution times for the equivalent parameterized tiling runs. We use a tile size of $\tau = 276$ for both of the Jacobi2D diamond tiling variants in Figure 1.

5.4 Chapel versus C Performance

There are two primary comparisons to make with respect to Chapel performance. First, the performance differences when using naive parallel scheduling techniques, and, second, the differences when using diamond tiling. In both cases Chapel's performance is competitive with C+OpenMP's.

A consistent performance variation was observed in the Chapel executions. For specific thread counts, 5-12 for diamond tiling and 5-8 for naive parallel, repeated execution revealed a multimodal distribution. This variation occurred only within the Chapel executions and for both the dynamic and static parallel schedules. The execution times in the fastest mode of the distribution were on-par or faster than the execution times of C+OpenMP. The Chapel team believes this to be due to a race in how Chapel tasks are assigned to cores resulting in an occasional load imbalance. The Chapel development team plans to address this in a future release. All times reported here are the average of all execution times, each test was repeated 32 times.

The lower two lines of data in Figure 7 show that for the Jacobi1D benchmark Chapel matches the performances of C+OpenMP except in the 5-8 thread region. Both of these implementations use a naive parallelization. Figure 1 shows the same for the Jacobi2D benchmark. In this case it is notable that the naive implementation of Chapel is outperforming the space-tiled variant of C+OpenMP. This can be attributed to the parallel schedule used by Chapel. The performance of the Chapel variants using diamond tiling is also competitive with that of the C+OpenMP variants except for the aforementioned issue in the 5-12 thread range.

[4]`http://www.cs.ucla.edu/~pouchet/software/pocc/`

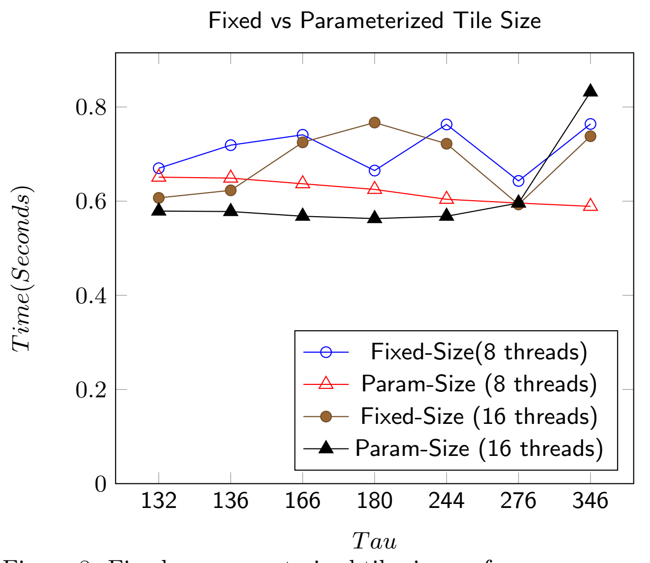

Figure 8: Fixed vs parameterized tile size performance comparison in C+OpenMP

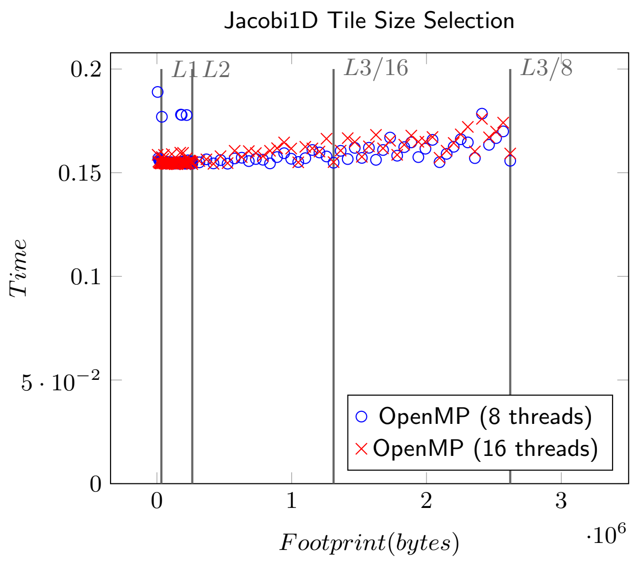

Figure 9: Tile size selection done with a sweep of tile sizes Jacobi1D.

5.5 Tile Size Selection

Tile size selection is an important factor in the performance of all tiling methods. If a tile size is too big it will fall out of cache or push another thread's tile out of cache, and if a tile is too small it will create unnecessary loop overhead and will exhibit poor cache reuse. The optimal tile size was found using an exhaustive search over the span of reasonable tile widths. The tile width is expressed as τ. The reasonable range of τ was determined by using a footprint calculation and targeting each level of cache.

The memory footprints were determined by using the single tile description for two- and three-dimensional parameterized diamond tiles and projecting them onto the spatial dimensions of the computation and using ISCC [27] to compute the cardinality of the set as a function of the tile size τ. The memory footprint in bytes of a diamond tile in 1D is

$$footprint = (\tau \cdot 2 - 1) \cdot (2) \cdot \texttt{sizeof(type)} \tag{2}$$

In 2D, the footprint is

$$footprint = (\tau^2 - \tau - 1) \cdot (2) \cdot \texttt{sizeof(type)} \tag{3}$$

The Jacobi benchmarks store only the previous and current time steps. Thus the multiplication by 2.

We expect that under normal operating conditions (i.e. one task/thread per processing unit with no thread migration) that the entire memory projection of a tile (maximum spatial bounds on the tile) will not be evicted from last-level cache until that tile is completely finished.

Figures 9 and 10 show execution time versus tile footprint for the Jacobi 1D and 2D benchmarks respectively. The vertical lines represent cache occupancy for L1 and L2 with no hyper-threading and then for L3 being shared among 8 threads and 16 threads. For Jacobi1D the optimal point lies just beyond L1 occupancy with a τ value of 4259. The optimal point for Jacobi2D fell within L3 (τ=276).

6. RELATED WORK

Our work extends current practice with both run-time parametrization of diamond tiling [1], and a novel approach to expressing this tiling with a Chapel parallel iterator. Related work includes various approaches to tiling through time, other work that uses iterators/generators, and previous work that has hand-tuned stencil computations in Chapel.

6.1 Tiling through Time

Loop tiling and more specifically tiling through time is a long-standing technique [17]. As it has increased in importance due to recent architecture trends, a number of variants have been explored; Wonnacott and Strout provide a description of many [31].

The recent development of diamond tiling for multi-dimensional data sets by Bandishti et al. [1] combines (we believe, for the first time) asymptotic scalability, high performance even on relatively small numbers of processors, and full implementation in the polyhedral infrastructure that underlies many automatic optimization tools such as Pluto [31]. However, the work of Bandishti et al. requires the tile size to be known at compile time. Prior work on parameterized tiling [23] has been limited to semi-oblique parallelepiped tilings, and thus is not applicable to multi-dimensional diamond tiles. We have developed a parameterized version of this tiling for the 2D Jacobi stencil, and expressed this tiling in both OpenMP and as an iterator for the Chapel language. The former provides a basis for performance comparison with other OpenMP tilings, and the latter provides a clean mechanism to give programmers control of iteration-space tiling.

6.2 Iterator Programming Constructs

Chapel's iterators are quite similar to both Python's *generator functions* [26] and the *iterator functions* of C# and Visual Basic [20, 29]. Being a parallel language, Chapel expands upon conventional approaches by adding the ability to specify iterators that can drive data parallel computations such as *forall loops*. This is done by specifying *standalone* or *leader-follower iterators* [12] which support parallel loops over multiple iterators in a simultaneous—or *zippered*—manner, similar to the NESL language [4, 5].

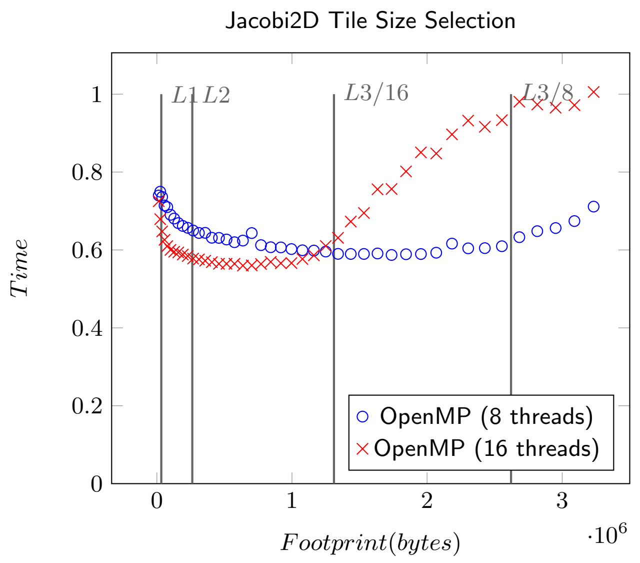

Figure 10: Tile size selection done with a sweep of tile sizes Jacobi2D.

However, in NESL zippered iterations were only supported by the compiler for a small number of built-in data types, whereas in Chapel users have the ability to author their own parallel zippered iterators.

6.3 Stencil Computations in Chapel

In Chapel's formative years, Barrett *et al.* studied different ways of expressing stencil computations in Chapel, with the eventual goal of being able to express the *sweep3d* benchmark in Chapel [2]. At this point in Chapel's history, its compiler was only just starting to generate parallel code, preventing the study from performing compelling performance studies. More recently, in [8], variants of a stencil computation were implemented in Chapel to determine which idioms generated the best performance compared to conventional programming models. Our work differs from this effort in that it studies tiling across time and space and considers the programmability benefits of using parallel iterators to factor complex loop structures away from the stencil computations themselves.

7. CONCLUSIONS

Diamond tiling is an advanced scheduling method that results in improved cache reuse and a high degree of concurrency. However, its adoption is hindered by its complexity and fixed tile-size code generation constraints. We have presented a method of generating parameterized diamond tiling and implemented it in benchmarks, showing that diamond tiling outperforms naive schedules that are simpler to develop and maintain. Additionally, we have shown that these schedules can be implemented in the Chapel language to take advantage of the parallel iterator construct and that performance is competitive with that of C+OpenMP.

We have also demonstrated that Chapel parallel iterators are an effective way to simplify the development process of stencil computations and increase the adoption of better schedules. Typically, the development of the stencil computation and the development of the iteration schedule are intertwined, although they have little to do with each other. This requires that both stencil computation and iteration schedule be modified together through the application's lifetime, impeding improvements to performance. The use of Chapel parallel iterators allows for separate development of each, creating a more agile development process. The addition of a new tiling schedule only requires that the schedule yield the appropriate values, and that new stencil codes can reuse existing high performance schedules.

The growth of high-performance computing into the Exascale era, and the creation of scientific applications requiring such computing power, poses many challenges. Among them, Exascale applications may require further advances in scheduling methods, and developers will benefit from ways to explore new schedules, or adopt those of other codes, without interfering with other aspects of their application. We have demonstrated that this separation of application code and high-performance scheduling can be achieved via Chapel iterators.

8. ACKNOWLEDGMENTS

This project is supported by a Department of Energy Early Career Grant DE-SC0003956 and a National Science Foundation Grant CCF-1422725. We thank our anonymous reviewers and Andrew Lumsdaine for their many useful suggestions for improving the paper, and Wayne Trzyna for providing us dedicated computation time.

9. REFERENCES

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

[2] R. Barrett, P. Roth, and S. Poole. Finite difference stencils implemented using chapel. Technical Report TM-2007/119, Oak Ridge National Laboratory, 2007.

[3] M. M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan. Parameterized Tiling Revisited. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM.

[4] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):102–111, April 1994.

[5] G. E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon, Pittsburgh, PA, September 1995.

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In L. Hendren, editor, *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin Heidelberg, 2008.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI), New York, NY, USA, June 2008. ACM.

[8] H. Burkhart, M. Sathe, M. Christen, O. Schenk, and M. Rietmann. Run, stencil, run! HPC productivity studies in the classroom. In *PGAS*, 2012.

[9] B. Chamberlain. Chapel parallel iterators: Giving programmers productivity and control. Cray Inc. blog, September 2013.

[10] B. L. Chamberlain. Chapel. In P. Balaji, editor, *A Brief Overview of Parallel Programming Models*. MIT Press, 2015 (expected).

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applicati ons*, 21(3):291–312, August 2007.

[12] B. L. Chamberlain, S.-E. Choi, S. J. Deitzand, and A. Navarro. User-defined parallel zippered iterators in chapel. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS 2011)*, Galveston Island, TX, USA, October 2011.

[13] G. Goumas, M. Athanasaki, and N. Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10), October 2003.

[14] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[15] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[16] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorth, B. Norris, J. Ramanujam, and P. Sadayappan. PrimeTile: A parametric multi-level tiler for imperfect loop nests. In *Prioceedings of the 23rd International Conference on Supercomputing, June 8-12, 2009, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA*, 2009.

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

[18] P. Kogge and D. Resnick. Yearly update: Exascale projections for 2013. Technical Report SAND2013-9229, Sandia National Laboratories, 2013.

[19] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[20] Microsoft Corporation. *C# Language Specification*, version 5.0 edition, June 2013.

[21] C. Olschanowsky, S. Guzik, J. Loffeld, J. Hittinger, and M. M. Strout. A study on balancing parallelism, data locality, and recomputation in existing PDE solvers. In *The IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2014.

[22] L. Renganarayana, D. Kim, M. M. Strout, and S. Rajopadhye. Parameterized loop tiling. *Transactions on Programming Languages and Systems (TOPLAS)*, 34(1), May 2012.

[23] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. M. Strout. Parameterized tiled loops for free. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[24] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 215–228, May 1999.

[25] R. Strzodka, M. Shaheen, and D. Pajak. Time skewing made simple. In *PPOPP*, pages 295–296, 2011.

[26] G. van Rossum and P. J. Eby. Coroutines via enhanced generators, May 2005.

[27] S. Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[28] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT12), Paris, France*, 2012.

[29] P. Vick and L. Wischik. *The Microsoft® Visual Basic®Language Specification*. Microsoft Corporation, version 11.0 edition, June 2013.

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

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