Mechanisms that Separate Algorithms from Implementations for Parallel Patterns

Christopher D. Krieger
Computer Science Dept.
Colorado State University
1873 Campus Delivery
Fort Collins, CO 80523-1873
krieger@cs.colostate.edu

Andrew Stone
Computer Science Dept.
Colorado State University
1873 Campus Delivery
Fort Collins, CO 80523-1873
stonea@cs.colostate.edu

Michelle Mills Strout
Computer Science Dept.
Colorado State University
1873 Campus Delivery
Fort Collins, CO 80523-1873
mstrout@cs.colostate.edu

ABSTRACT

Parallel programming implementation details often obfuscate the original algorithm and make later algorithm maintenance difficult. Although parallel programming patterns help guide the structured development of parallel programs, they do not necessarily avoid the code obfuscation problem. In this paper, we observe how emerging and existing programming models realized as programming languages, preprocessor directives, and/or libraries are able to support the Implementation Strategy Patterns that were proposed as part of the *Our Pattern Language*. We posit that these emerging programming models are in some cases able to avoid code obfuscation through features that prevent tangling of algorithm and implementation details for parallelization and performance optimizations. We qualitatively evaluate these features in terms of how much they prevent tangling and how well they provide programmer-control over implementation details. We conclude with remarks on potential research directions for studying how to produce efficient and maintainable parallel programs by separating algorithm from implementation.

1. INTRODUCTION

Programming parallel machines is fraught with difficulties. Due to parallelization, programmers have to contemplate details that are absent in serial programming contexts. Such details include communication and synchronization. Since parallelization is motivated by the need for improved performance, programmers must also consider optimization and parallelization details. These details include how to effectively utilize the memory hierarchy, how to exploit various machine-specific features, and how to best strike a balance between communication and computation. Parallelization and optimization details complicate programming and their specification often obfuscates code. To be precise, the term *obfuscation* indicates that the code is difficult to read and maintain.

The formalization of parallel programming patterns provides a rigorous structure to the development of parallel software. However, despite the structure that patterns provide, source code can still be obfuscated when a pattern is realized. As an example consider the loop parallelism pattern [1], which suggests modifying an existing code to improve performance by performing incremental loop transformations[1].

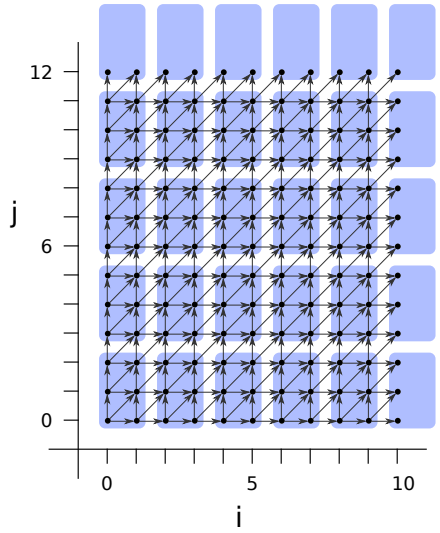
To illustrate the obfuscation that can occur due to loop transformations, Figures 1b and 1c show an example loop that has been modified using the tiling loop transformation. The tiling transformation [17, 27, 34] modifies the schedule of a loop by causing all of the iterations within a tile to be executed before proceeding to the next tile. Figure 1c indicates how such a tiling can be implemented with `for` loops. Note the algorithm code has become tangled with scheduling implementation details.

We use the term *tangling* to indicate that the implementation details are exposed in the algorithm code in an interspersed manner. Due to the tangling in Figure 1c, the code has become obfuscated in that it is no longer easy to see the domain of the computation, which is $(\{[i,j] \mid (0 \leq i < 11) \wedge (0 \leq j < 13)\})$. Parallelization of the code in Figure 1c could be accomplished by applying an additional loop transformation called loop-skewing, which would further obfuscate the code.

Tangling parallelization and program optimization details with algorithm code often leads to obfuscation. Programming models research has produced programming languages and libraries with constructs that prevent tangling by somewhat orthogonalizing implementation details from algorithm code. Examples include parallel programming languages such as Chapel [7, 8], which provide programmer-defined computation and data distributions; OpenMP [9], which provides pragmas for labeling a loop/algorithm as having `forall` parallelism and indicating implementation preferences; and object-oriented parallel libraries such as the Standard Template Adaptive Parallel Library (STAPL) [25, 4], which provides orthogonal scheduling of data structure iterators. There are also more restrictive programming models such as the polyhedral programming model [14, 5] that enable orthogonal specification of scheduling and storage mapping within the compiler, and MapReduce [10], which enables implementation details, such as the runtime environment, to be specified and to evolve during program execution.

In general, it is important that tangling is kept to a minimum while still providing the programmer control over the parallelization and optimization details for performance tuning purposes (i.e., *programmer-control*). In Figure 1c, we have an example where the application of programmer-control causes significant tangling. Thus there is

[1]Loop transformations typically improve performance by increasing data locality and exposing parallelism.



(a) Iteration space with tiling

```
for (i=0; i<11; i++) {
  for (j=0; j<13; j++) {
    A[i,j] = 1/2 * (A[i,j-1] + A[i-1,j-1] + A[i-1,j]);
  }
}
```

(b) Code for iteration space without tiling

```
int Ti, Tj, i, j; /* Tile and point loop iterators. */
int TiLB, TjLB ;  /* Tile loop lower-bound vars. */
TiLB = -1; TiLB = ((int)ceild(tiLB,2) * 2);
for (Ti = TiLB; Ti <= 10; Ti += 2) {
 TjLB = -2; TjLB = LB_SHIFT(TjLB,3);
 for (Tj = TjLB; Tj <= 12; Tj += 3) {
   for (i= max(Ti,0);i<=min(Ti+2-1,10);i++) {
     for (j= max(Tj,0);j<=min(Tj+3-1,12);j++) {
       A[i,j] = 1/2 * (A[i,j-1] + A[i-1,j-1] + A[i-1,j]);
     }
   }
 }
}
```

(c) Code for iteration space with tiling

Figure 1: Iteration space and for loop implementations for a small example loop with dependences much like those in the Smith-Waterman code [28].

definitely a tradeoff between minimal tangling and programmer-control; however, the tradeoff is not always as extreme as it is with the "for loop" programming construct. Other mechanism features provide different points in this tradeoff space. We define a *mechanism* as a particular technology that enables the specification of algorithms and implementation details and aides in realizing their execution.

In this paper, we observe how various implementation strategy patterns [19] might be supported by features in the following mechanisms: the programming language Chapel, the preprocessor/library OpenMP, and the library STAPL. We also discuss the extent to which these features prevent code tangling while still enabling programmer-control, and summarize our results in Tables 1 through 5. To guide this discussion we introduce six classes into which features can be placed based on how they are exposed in algorithm code. The impacts that features have on tangling and programmer-control can be understood through this classification.

Our goal is for programmers using the parallel programming patterns approach to make implementation mechanism decisions based on the tradeoffs between code tangling and programmer-control shown in these tables, the performance requirements, and the context of the computation being implemented.

In Section 2, we describe the six categories for features in terms of their relative positions in the programmer-control and code tangling tradeoff space. In Section 3, we review a set of the implementation strategy patterns and provide an overview of their implementation details. In Sections 4 through 6 we describe the features of Chapel, OpenMP, and STAPL and how they can be used to specify the implementation details of various implementation strategy patterns. We end this paper with concluding remarks.

2. CATEGORIZING ALGORITHM / IMPLEMENTATION SEPARATION

In this paper we aim to compare Chapel, OpenMP, and STAPL. Each of these mechanisms include features that can be used to implement parallel programming patterns. What features programmer's choose to use to implement a pattern will impact how tangled that pattern's implementation code becomes. Feature choice also dictates how much control programmer's have over addressing the implementation details of patterns. These impacts form a tradeoff space of tangling and programmer-control. We illustrate this space in Figure 2.

Within this illustration we identify a goal point of minimal tangling with maximal programmer-control. Generally, less tangling is desirable since it alleviates code obfuscation. However, we argue that some minimal amount of tangling is advantageous since it clarifies where implementation details are relevant. Regardless of the impact a feature has on tangling, maximal programmer-control is desirable since it enables programmers to tune for performance if necessary.

Within Figure 2 we also identify six different feature classes. Features fall into a class based on how they are exposed in algorithm code. The classes are arranged within the figure relative to what impact their features have within the tradeoff space. Thus, by identifying into what class a feature falls, one can gain insight into what impacts that feature has on tangling and programmer-control. We also illustrate several points in the figure that correspond to sample programming language features. In this section we define each of these six classes, namely: interspersed, invisible, hinted, separated, injected, and delegated.

Interspersed – Code for addressing an implementation detail is exposed in the algorithm code. Features in this class have the drawback of a high degree of tangling, but have the benefit of a high degree of programmer-control. The manual application of loop transformations, such as those in Figure 1, fall into this class.

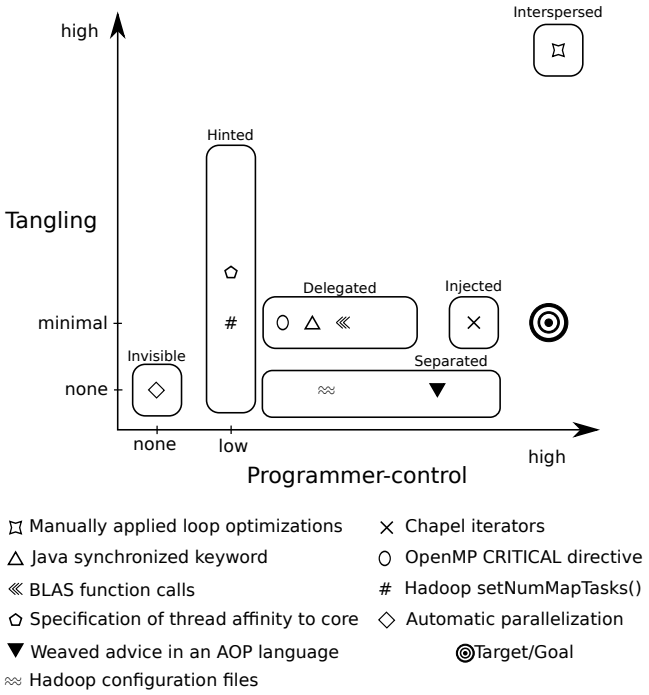


Figure 2: Relative classification of features and their programmer control and tangling impacts.

Invisible – Some overarching system opaquely handles the implementation detail. An example would be a parallel library that determines the number of threads that should be instantiated, but does not provide any mechanism for programmers to override this decision. Another example would be automatic parallelization via compiler. Features in this class are not exposed in algorithm code, and in fact are not exposed to the programmer at all. Thus there is no tangling, but at the cost of no programmer-control.

Hinted – Users supply hints that guide how some overarching system should address the implementation detail. The overarching system is not required to follow these hints. One example would be specifying a processor affinity hint for a task that may or may not be taken into consideration. Another example of a hinted feature is Hadoop's `setNumMapTasks()` function which supplies a suggestion to the Hadoop map/reduce runtime of how many tasks it should spawn. Hinted features can have varying degrees of tangling depending on where they are exposed in code. There is no tangling if the hints are supplied outside of the algorithm code, but there is some tangling if this is not the case. Typically, features of this class have a low degree of tangling, but likewise provide a low level of programmer-control.

Separated – Some external specification defines how to modify algorithm code. The programmer has access to this modification specification, however the fact that these modifications are to be applied is not apparent in the algorithm code. The degree of programmer-control a separated feature has is dependent on how expressible these modifications can be. For example, aspects in Aspect Oriented Programming (AOP) languages are a separating feature [20]. Modification via aspects is limited, however, to adding functionality at expressible joinpoints. External configuration files are also a form of a separated feature, but programmer-control is limited to what is expressible within such files. Features that fall in this class do not tangle algorithm code, but have a variable amount of programmer-control (always more control than invisible features but not necessarily as much as interspersed ones).

Injected – Implementation details are specified via code exposed in some orthogonal data or algorithmic structure. Where to add injected details is explicitly stated in algorithm code, but the injected detail itself is factored out. Iterators and normal function calls would fall into this category. Injected features have low tangling, but programmer-control is limited by the mechanism used to call the injected code.

Delegated – Implementation details are specified behind some non-accessible structure. As in the injected class, delegated features explicitly specify in the algorithm code the location at which to put details. However, unlike injected features, the implementation of delegated features is not specified in code available to the programmer. Delegated features often expose parameters to programmers. The more parameters a delegated feature exposes, the higher degree of programmer control it provides. Delegated features imply a similar level of tangling as injected features, but typically have less programmer-control. OpenMP's pragmas are examples of delegated features, as are BLAS function calls.

3. PARALLEL PROGRAMMING PATTERNS AND IMPLEMENTATION DETAILS

In the previous section we defined six classes for features. In the subsequent sections of this paper, we discuss features as they are used to implement parallel programming patterns. We focus on the patterns defined by the *Our Pattern Language* (OPL) [19]. OPL strives to define a conceptual framework for parallel programming. In OPL the patterns are partitioned into five layers. These layers successively focus on higher to lower level programming concerns. The architectural and computational patterns lie at the top of the hierarchy and focus on high-level design considerations, such as how application components interact. Below these top layers lie the strategy layer patterns, which focus on identifying an algorithm's dependence patterns. Although this focus exposes parallelism, ideally application programmers working in this layer should not worry about the specific mapping of parallelism to hardware. That mapping focus is exposed in the next layer, the implementation strategy patterns.

When realizing these implementation strategy patterns a number of details must be considered. Addressing many of these details entails answering "where and when" questions for computation and data. For instance, "where is the optimal place to store some piece of data?" and "when should some task be executed?" These where and when questions relate to an algorithm's storage-map and schedule. In this section we explicitly list the implementation details of several of the implementation strategy patterns and identify how these details relate to schedule and storage-mapping. The details we list in this section correspond to the rows of Tables 1 through 5. Within these tables we also identify what features mechanisms provide to address these implementation details and into what classes these features fall, so that it can be understood what impact they have on tangling and programmer-control. The patterns and their details are as follows:

Data Structures – As most parallel programs need to share data across execution elements, the data structure patterns are commonly used. The OPL developers have identified three specific abstract data types as meriting specific mention: arrays, hash tables, and queues. When implementing these patterns one must consider the storage mapping detail of where data should be stored and the scheduling detail of when to synchronize data accesses by multiple readers and writers.

Loop-level Parallelism – The Loop-Level Parallelism pattern is relevant when the iterations of a loop body can be executed independently. Since loop nests are common in scientific applications and are often performance-critical, this pattern presents itself frequently when parallelizing existing scientific codes.

The loop-level parallelism pattern suggests increasing the efficiency and exposed parallelism of a loop through the use of several established techniques. Techniques including stripmining, interchange, tiling, fusion, fission, peeling, unrolling, and unrolling and jamming. These same techniques can enable or improve parallelization by improving loop granularity or by decreasing communication or memory bandwidth requirements.

The loop-level parallelism pattern suggests the use of transformations that modify the schedules of loops to expose parallelism. As such, the primary implementation details of this pattern are concerned with scheduling. These details are the distribution of threads to processors (when using a distributed system), the mapping of iterations to threads/processes, and the order in which to execute iterations within a thread.

Single-Program Multiple Data (SPMD) – The SPMD pattern dictates that a single program is used by all processes. Processes are able to distinguish themselves from one another based on an ID. The storage-map for the SPMD pattern has data resident on the node that executes the process responsible for it. Assigning this responsibility of data to process is one of the storage mapping details of SPMD. The scheduling details of this pattern include deciding on the number of threads to instantiate and instantiating these threads. The schedule in the SPMD pattern follows from the storage mapping.

Master-Worker and Task-Queue – In the Master-Worker and Task-Queue patterns, a master generates a set or sets of tasks. One or more workers draw items from these sets, perform the assigned task, and repeat the process until the sets are empty. The storage mapping details entailed by this pattern are concerned with where tasks are stored. Scheduling details are concerned with whether the tasks are assigned in a first-in first-out basis (the semantics of the task queue) and whether there is a notion of task priority.

Bulk Synchronous Parallelism (BSP) – In this pattern, computation is organized as a sequence of super-steps. Within a single super-step, computation occurs on a processor-local view of the data. Communication events are posted within a super-step, but the results are not available until the subsequent super-step. Communication events from a super-step are guaranteed to complete before the subsequent super-step starts.

Scheduling details of the BSP pattern include how many threads should be spawned and how the synchronization (barrier) step is implemented. An additional performance detail is whether communication is aggregated.

There are other parallel patterns in OPL's implementation pattern layer such as strict data parallelism, fork-join, actors, and graph partitioning. Each of these patterns also has its own implementation details, but due to space limitations, we do not consider these patterns in this paper.

4. EVALUATION OF PROGRAMMING LANGUAGE MECHANISMS: CHAPEL

In the previous sections we laid out the criteria of our qualitative evaluation. In this section we use these criteria to study how Chapel's features address the implementation details surveyed in Section 3. We summarize this section in the Chapel columns of Tables 1 through 5. Specifically, we identify what Chapel features can be used to address parallel programming pattern implementation details. We also identify which class these feature fall into. Feature classes are described in Section 2 and the class a feature falls into will impact how much control programmers have over addressing an implementation detail and how tangled code becomes.

4.1 Data Structure Patterns in Chapel

One implementation detail Chapel is particularly adept at addressing is the distribution of arrays. The distribution of arrays is, as the name would suggest, a detail intimately tied with the distributed array pattern. However, the more general implementation detail of distribution of data can also be easily addressed by Chapel when the patterned data structure can be realized using Chapel's more general array data type.

The data structures called *Arrays* in Chapel are more sophisticated than arrays in other imperative languages. Rather than simply being a mapping of indices over a contiguous integral range to values, as in other languages, Chapel's arrays associate a variety of domain entities with values. These entities are represented in the Chapel language through its first-class *domain* data-type. The expressible domains in Chapel include the traditional contiguous integral range, as in a classic two-dimensional array, as well as noncontiguous sets, as in a sparse matrix, or sets of keys as in an associative array.

Regardless of the type of domain used to index an array, a distribution can be applied to it. *Distributions*, like domains, are first-class entities in the Chapel programming language: essentially they are a special kind of class type. Distributions are applied to arrays and define how the array's elements are stored and accessed. The internals of a distribution are defined in a programmer-accessible class that can be modified by programmers. As such, distributions are a quintessential example of an injected feature. They have a high degree of programmer control with little tangling, only interspersed with algorithm code at the point of an array's declaration.

Figure 3 shows an example of such an array declaration. Note that the distribution `blockDist` is a first-class named entity (in this case a 1-dimensional block distribution). The domain `D` is defined as a two-dimensional rectangular domain with rows 1 through m and columns 1 through n. The last line of the example in Figure 3 shows an array A being declared over the distribution. Most of the code to address the implementation detail of distribution of data is

Data structures

	Chapel	OpenMP	STAPL
Distribution of data	Distributions {injected}.	By default data in OpenMP is shared, but the parallel pragma has an option to set private data to threads {delegated}.	Users can create a custom partition_mapper {injected}.
Synchronizing multiple writes	Writes to data-structures can be abstracted behind critical sections in distribution code {injected}. Atomic statements could be used to actually implement the synchronization {delegated}.	Users can mark atomic sections with a pragma {delegated}.	Done implicitly for provided data structures {invisible}.

Table 1: Data structure patterns (and how mechanisms address their implementation concerns)

Loops

	Chapel	OpenMP	STAPL
Distribution of threads to processors	Loops coupled with Chapel Iterators {injected}, or on statements coupled with for/forall loops {delegated}.	Handled by run-time environment or operating system {invisible}. Some implementations have notion of thread affinity {hinted}.	Distribution of computation mirrors distribution of data, specified via a partition mapper {injected}.
Assignment of iterations to threads	Loops coupled with Chapel iterators {injected} or for and forall loop nests not coupled with iterators {delegated}.	Schedule clause in parallel-for directive {delegated}.	pRanges dictate what iterations must be assigned to same thread {injected}. Actual assignment to threads is handled by runtime {invisible}
Order of iterations within thread	Order can be specified via Chapel Iterators {injected} or with for and forall loop nests that are not coupled with iterators {interspersed}.	Handled by OpenMP scheduler {invisible}.	Dependent on implementation of pAlgorithm {delegated}.

Table 2: Loops pattern (and how mechanisms address its implementation concerns)

SPMD

	Chapel	OpenMP	STAPL
Instantiation of threads/processes	Specified through configuration variable maxThreads and Locales array {delegated}.	Specified by omp_set_num_threads() function {delegated}.	Parallel region manager issues calls to awaken/spawn threads as needed {invisible}.
What thread is responsible for what data	Operations manipulating data can be specified to work on nodes directly by "on" statements {interspersed}. Distribution of data is defined by distributions {injected}. Relationship between these two constructs can be asserted with local keyword {delegated}.	No way of explicitly asserting ownership of data to threads, but this is not needed in a shared-memory environment. The domain of data a thread ought to be responsible can be calculated with an equation that includes omp_get_thread_num() as a parameter {interspersed}.	Users can create a custom partition_mapper to distribute data {injected}.

Table 3: SPMD pattern (and how mechanisms address its implementation concerns)

Master-worker

	Chapel	OpenMP	STAPL
Where queue is stored	Pattern can be done implicitly with cobegin {invisible}. When done explicitly can be resolved by distributions {injected}.	Hidden from programmer. {invisible}.	Specified by partition mapper (which can be modified) {injected}.
Instantiation of workers	Set through maxThreads configuration variable {delegated}.	Specified by calling omp_set_num_threads() {delegated}.	Parallel region manager issues calls to awaken/spawn threads as needed {invisible}.
Does queue maintain precise queue semantics?	Not specified for begin/cobegin/coforall {invisible}. If pattern is done explicitly can be hidden using object-oriented patterns {injected}.	Not specified {invisible}.	Depends on data-structure user chooses to use {delegated}.

Table 4: Master-worker pattern (and how mechanisms address its implementation concerns)

BSP

	Chapel	OpenMP	STAPL
Aggregation of communication	No easy way to explicitly do this with PGAS languages.	Remote communication does not happen in a shared-memory environment.	Communication is handled by the underlying ARMI communication library {invisible}.
Instantiation of threads	Set through maxThreads configuration variable {delegated}.	Specified by calling omp_set_num_threads() {delegated}.	Parallel region manager issues calls to awaken/spawn threads as needed {invisible}.
Implementation of barrier	Can be accomplished with sync vars {delegated} or if threads are spawned with cobegin the barrier is implicit {delegated}.	Barrier pragma {delegated}.	Synchronization occurs after calls to pAlgorithms {delegated}.

Table 5: BSP pattern (and how mechanisms address its implementation concerns)

```
// BlockDist is a 1D block distribution
// D is a 2D arithmetic domain with BlockDist
// applied
const blockDist = new dist(Block(...));
const D: domain(2) distributed blockDist;
D = [1..m, 1..n];

// A is an array of ints over the domain D
var A: [D] int;
```

Figure 3: Application of distributions and arrays in Chapel

not seen in this example, but is accessible in the source code for the distribution `Block`.

Another implementation detail for the data-structure patterns is when and how to synchronize multiple writes. There are several ways Chapel enables the programmer to express synchronization details. For instance, the programmer can surround all array writes in algorithm code with critical sections using Chapel's `atomic` statement. This addresses the "where to synchronize" detail in a tangled manner, but the atomic statement is an example of thin tangling that does not result in significant code obfuscation. The logic for how to perform the synchronization itself can be addressed by the implementation of atomic statements in Chapel so this detail is delegated.

A better way of synchronizing multiple writes of data structures is to factor the critical section out of the algorithm code and into the code of the distribution being applied to the structure. Distributions have a method that is called when an array is accessed. The body of this method could be wrapped in a critical section. Using this approach, the implementation detail is addressed in an injected manner.

Chapel's use of distributions eloquently addresses the implementation issues of the data-structure patterns. The main limitation is that distributions are only applicable when the data structure can be expressed as an array using one of Chapel pre-defined domain types. As long as the Chapel compiler is able to generate efficient executables, the first class distribution feature in Chapel provides a high level of programmer control and a minimal level of tangling.

4.2 Program Structure Patterns in Chapel

Chapel includes several features that aid in the realization of program structure patterns, particularly iterators, `forall` statements, `on` statements, and synchronization variables. In this subsection we describe how these language features enable the specification of implementation details needed for various program structure patterns.

Loop-level Parallelism

The loop-level parallelism pattern is most concerned with scheduling details such as the assignment of iterations to threads, the order of iterations within a thread, and the distribution of threads to processors. All of these details can be addressed with Chapel iterators.

Iterators are invoked by `for` and `forall` loops and guide the loop through the values in the domain being iterated. Iterators are syntactically similar to functions; however, rather than returning a value and transferring control back to the call site, they *yield* a value and execution flow back to the call site while maintaining the execution state of the iterator code. Like functions, iterators are an injected feature.

```
def colmajor(d1,d2): 2*int
{
  for j in 1..d2 do
    for i in 1..d1 do
      yield (i,j);
}

def tiledcolmajor(d1,d2): 2*int
{
  var (b1,b2) = computeTileSizes();
    for j in 1..d2 by b2 do
      for i in 1..d1 by b1 do
        for jj in j..min(d2,j+(b2-1)) do
          for ii in i..min(d1,i+(b1-1)) do
            yield (ii,jj);
}

def evolve(d1,d2)
{
  /* can be switched to tiling by replacing
     'colmajor' with 'tiledmajor'.
     Idx is a 2D index (a tuple).
  */
  for idx in colmajor(d1,d2)
  {
    u0[idx] = twiddle[idx, u0[idx]];
  }
}
```

Figure 4: Tiling in Chapel

Figure 4 shows the definition and use of two iterators `colmajor` and `tiledcolmajor` for scheduling within a thread. The function `evolve` can use either of these two iterators to step through a rectangular domain of indices ($\{[i,j] \mid 1 \leq i \leq \texttt{d1} \wedge 1 \leq j \leq \texttt{d2}\}$) in either a column-major or tiled order. As compared to the transformed loop example in Figure 1, it is clear how the use of this injected feature prevents obfuscation of the algorithm in the function `evolve` while still providing complete programmer control over the schedule.

For distributing threads to processors and assigning iterations to threads, a programmer can delegate these details by coupling `on` statements with `forall` loops. The `on` statement is used to specify that code within some block of iterations should be executed on some specific *locale*. The statement is delegated in the sense that the details to spawn new threads to execute the body or to inform existing threads to do so are not exposed to the programmer

Locales are entities in the Chapel language that logically represent a unit of hardware with processing and storage capabilities, e.g. a node in a cluster.

Single-Program Multiple Data (SPMD)

SPMD is a pattern whose realization is more likely to be tangled in Chapel code. Chapel encourages a global-view model that is at odds with the SPMD pattern. Chamberlain et al. [8] cite SPMD as the most prevalent example of a fragmented, that is to say, non-global, programming model. A global view of computation like the one provided in Chapel does not require the implementation detail of explicitly decomposing a computation into per-thread chunks.

```
// Assume blockDist is some previously
// defined distribution that specifies
// how the elements of the domain are
// assigned to locales.
const problemSpace : domain(1,int)
    distributed blockDist = [1..n];

// a, b, and c are arrays over the problem
// space.
var a, b, c : [problemSpace] real;

// iterate through each locale
coforall loc in Locales do on loc {
  // compute the subdomain of the problem
  // space this local is responsible for
  const myProblemSpace : domain(1, int) =
    blockPartition(
      problemSpace, here.id, numLocales);

  local {
    forall idx in myProblemSpace do
      a[idx] = b[idx] + alpha * c[idx];
  }
}
```

Figure 5: SPMD in Chapel

Nevertheless, Chapel does include features that enable a fragmented programming style, and as such Chapel can implement this pattern.

Addressing the implementation detail of assigning data to threads can be done using Chapel's distributions as described previously for the data-structure patterns. Ensuring that threads manipulate the data they own can be done in an interspersed manner with `on` statements. Figure 5 shows an example of specifying SPMD in Chapel and works by coupling the `on` statement with a `coforall` loop. The `local` clause asserts that no remote data-accesses occur in the loop (i.e. no message passing).

Implementation of SPMD must also address the instantiation of concurrently executing threads of control. In Chapel this can be configured by setting the variable `maxThreads`, which can be set through a command-line argument when a Chapel program is invoked.

Master-Worker or Task-Queue

While the SPMD pattern must address the implementation detail of number of threads, the master-worker pattern must address the detail of number of workers. As in SPMD, the number of workers is specified with the `maxThreads` configuration variable.

The master-worker pattern also has a storage-mapping implementation detail, which is to decide where the task pool or task queue should be stored. There are multiple ways of addressing this detail in Chapel, and the one used depends on whether the programmer uses Chapel's task parallel features. These features are the *begin*, *cobegin* and *coforall* statements. These statements spawn a task for the code they surround and place this task in a pool to be picked up by previously instantiated threads. In this instance the scheduling and storage mapping details of the queue, which are where the queue is stored and what are its semantics, are made invisible. This is convenient and has very little tangling, but has an inflexibly low amount of programmer-control.

However, should this low amount of programmer-control be an issue, the pattern can be realized in a more explicit manner. This realization does require the programmer to create his or her own queue data type. Such a data type can be structured using Chapel's object system, which would abstract the implementation details concerning the queue into its class. This delegates these implementation details to the object. However, in this instance the programmer is also responsible for realizing one of the data structure patterns.

Bulk Synchronous Parallelism (BSP)

As with the master-worker pattern, BSP can be realized in both a more implicit and more explicit manner in Chapel. The manner in which the pattern is realized will affect the implementation of the barrier required between each superstep of computation. In either manner, the BSP pattern requires computation be structured as a series of supersteps, where there is no remote write-to communication for local data during a superstep.

Structuring a computation as a series of supersteps can be done implicitly by iterating with a `for` loop and having threads spawned and joined within each superstep via a `coforall` loop. The barrier is thus delegated to the internal `coforall`. This incurs low tangling, but low programmer-control when it comes to the implementation of the barrier. The barrier can also be implemented explicitly using Chapel's *synchronization* variables. In Chapel, synchronization variables have *empty-full* semantics. If empty, attempts to read the variable cause the reader to block until a write moves the state to full. Implementing the barrier with synchronization variables causes more tangling than using cobegins, but it does provide more programmer control over the scheduling of the barrier.

An implementation detail where programmer control is low in Chapel is communication aggregation. Such aggregation is a common performance optimization for the BSP pattern and as such the decision to implement it, and if so, how to implement it, are implementation details of the BSP pattern. There are no easy ways to address this detail in global-view languages such as Chapel, unless assumptions about how the language performs data transfers can be made.

4.3 Related Work

This section described the features Chapel includes to address the implementation details listed in Section 3. There are several other features in Chapel; [3] and [8] provide detailed discussions of these features as well as the features we mentioned above.

Many other languages include features for parallel programming. Such languages include Unified Parallel C (UPC) [13, 12], Co-Array Fortran [24], Titanium [35], and X10 [11]. Many of the features in Chapel have been drawn from other languages. For instance, iterators originally appeared in the language CLU [22], distributions in HPF [15], and Chapel's domain concept is based on regions from ZPL [29].

5. EVALUATION OF PARALLELIZATION VIA PREPROCESSING DIRECTIVES: OPENMP

OpenMP takes a different approach to enabling programmers to specify parallelism than is seen in Chapel. Rather than being a standalone language, with a standalone compiler, OpenMP builds on top of an existing compiler, adding new features to the language through the use of pragmas. The OpenMP compiler performs a

preprocessing pass to address these pragmas through the insertion of calls to a library, and then sends code off to a base compiler to finish the compilation process.

There are, however, also similarities between OpenMP and Chapel, the most noticeable being that both are based on a shared namespace. However, while Chapel operates in both shared and distributed memory contexts, OpenMP is assumed to be running only in a shared-memory environment.

Another difference between Chapel and OpenMP is how features are used to address the implementation details of patterns. In this section we will survey the features OpenMP includes to address these details and categorize them according to the criteria outlined in Section 2.

5.1 Data Structure Patterns in OpenMP

OpenMP's shared memory environment, with its global namespace, makes the implementation of most data-structures fairly straightforward; the distribution of data is typically not a concern when implementing data-structure patterns in a shared memory environment. When distribution of data is not a concern, implementing a data structure is reduced to the somewhat simpler process of constructing it as would be done in a serial context and then accounting for the possibility of multiple processors writing and reading the data structure.

At times, it may be convenient for threads to have local copies of data structures. OpenMP provides for this through its *firstprivate* clause, which specifies that data from the master thread should be copied to each thread, and the *lastprivate* clause, which copies the results from a thread's copy back into the main copy.

Regardless of how data is distributed, most data structures in a parallel context must account for the possibility of multiple writers. OpenMP's `atomic` pragma can address this issue simply by specifying that writes to a data structure fall in critical sections of code. The atomic pragma, as with most OpenMP pragmas, is a delegated feature. It specifies where an atomic section of code lies, but how the atomic section is implemented is not exposed or accessible.

5.2 Program Structure Patterns in OpenMP

The program structure patterns in OpenMP are also commonly addressed using delegated features. In this section we step through the surveyed program structure patterns and describe the features OpenMP includes to address their implementation details.

Loop-level Parallelism

OpenMP is most frequently used to parallelize loops, which is realized using OpenMP's `parallel for` directive. The distribution of threads to processors is handled by OpenMP's runtime environment (or the operating system) and as such this is an invisible feature that is not exposed to the programmer nor something over which the programmer has any control. Some implementations of OpenMP enable the programmer to specify a notion of thread affinity. The mapping of iterations to threads is another invisible feature. A loop in OpenMP marked using a `parallel for` directive is assumed to have all iterations independent of one another and assigning iterations to threads is the responsibility of OpenMP's scheduler.

Although programmers have little control over how threads distribute to processors, they do have more control over the assignment of iterations to threads. OpenMP's *schedule* clause enables programmers to specify that iterations be assigned to free threads in a block cyclic fashion with a specified block size. Scheduling control is limited to being doled out in such a fashion, and since programmers do not have direct access to OpenMP's scheduler this feature is delegated.

```
double data[SIZE];

#pragma omp parallel shared(data)
{
  int chunkSz = SIZE/omp_get_num_threads();
  int lb = omp_get_thread_num() * chunkSz;
  int ub = lb + chunkSz-1;

  for (int i = lb; i < ub; ++i) {
    data[i] = ...
  }
}
```

Figure 6: SPMD in OpenMP

```
#pragma omp parallel
{
  #pragma omp single nowait
  {
    while (work_remains) {
      work = new Work(); // generate work
      #pragma omp task firstprivate(work)
      work.execute();
    }
    #pragma omp taskwait
  }
}
```

Figure 7: Master-worker in OpenMP

Single-Program Multiple Data (SPMD)

Compared to the loop-level parallelism pattern, OpenMP enables a greater degree of control over scheduling in the SPMD pattern. Scheduling in this pattern is dictated through ownership over data. OpenMP does not provide any mechanism to specify such ownership over existing data (although threads can be given private copies), so the logic to do this must be explicitly handled by the programmer. As such the implementation detail of what thread is responsible for what data is handled in a tangled manner. Determination of data ownership is usually done through calculating lower and upper bounds with equations involving the total number of threads and each thread's individual ID. OpenMP's `omp_get_num_threads()` and `omp_get_thread_num()` functions return these data respectively. The code in Figure 6 gives an example of calculating such bounds.

Master-Worker or Task-Queue

Both the SPMD and master-worker patterns have the instantiation of threads as an implementation detail. In both cases this is handled through OpenMP's `omp_set_num_threads()` function, which delegates the process of this instantiation to the OpenMP runtime.

Tasks in the master-worker pattern can be realized in OpenMP 3.0 through its *task* construct [33]. This construct specifies where tasks are generated and is parameterized with a list of data that should be private to each thread. When worker threads are free they begin executing previously generated tasks. The code in Figure 7, adapted from [33], demonstrates this pattern.

OpenMP provides a simple interface for manipulating the task queue. Tasks are enqueued via the `task` pragma and dequeued and assigned to available threads automatically. The `taskwait` pragma is used to synchronize running tasks before continuing computation. This implementation of Master-Worker does not allow the programmer to tie work to specific processors, instead its scheduler, invisible to the programmer, addresses this detail. The number of workers scales to the number of available threads. The number of threads is specified via an environment variable at runtime and can be overwritten by the program by calling a library routine. OpenMP provides no mechanism to change the semantics of the queue; the order tasks are popped off the queue is invisible to the programmer.

Bulk Synchronous Parallelism

Like Chapel, OpenMP is based on a shared namespace. Therefore, it faces many of the same difficulties that were previously discussed with respect to implementing the BSP pattern in Chapel. Addressing the detail of barrier implementation is fairly straightforward in OpenMP, however, by delegating such detail to OpenMP's barrier pragma.

The BSP pattern has been implemented in OpenMP by Marowka [23] using the tool BSP2OMP. This implementation uses four stacks or queues to hold communication requests generated during a superstep. After the barrier, but before the subsequent superstep begins, the queues' requests are completed.

5.3 Related Work

There are several implementations of OpenMP that operate with various C, C++, and Fortran compilers [2]. Additionally, there are projects to extend OpenMP to enable it to work in a cluster environment [30, 16]. Other work has looked at translating OpenMP programs to MPI programs so that existing OpenMP programs can be ported to work in a distributed memory environment [6].

6. EVALUATION OF PARALLELIZATION VIA LIBRARIES: STAPL

Another approach to parallelization is through the use of parallel libraries. There are many different libraries that address parallelism in different ways [18, 26, 21]. For the purposes of this evaluation, we choose STAPL, the Structured Template Adaptive Parallel Library[4], to serve as an exemplar. STAPL strives to be a parallel implementation of the C++ Standard Template Library (STL). The STL implements the idea of generic programming using general algorithms that act on abstract data structures called containers. This idea aims to separate an algorithm from the data structures on which it operates. STAPL mirrors this paradigm, substituting parallel containers (pContainers) and parallel algorithms (pAlgorithms) for their sequential STL counterparts.

In this section, we discuss how the STAPL parallel library can be used to address the implementation decisions of the surveyed implementation strategy patterns.

6.1 Data Structure Patterns in STAPL

STAPL directly supports many of the data structure patterns. It includes support for various types of abstract data type containers including vector, list, and tree; a number of associative containers including parallel, distributed versions of maps, sets, multimaps, multisets, hashmaps, and hashsets [32]; and even some containers not part of the STL standard such as graphs and matrices. Programmers do not have to worry about synchronization of multiple writes as this is handled by the STAPL library in a manner that is invisible.

The distribution of data within these structures is handled via *partition mappers*, which map subsets of data in a pContainer into a *location*. STAPL locations are analogous to Chapel locales in that they represent a component in a parallel machine that has an address space and processing capabilities. The partition mapper is an injected feature as programmers can override its implementation. Such modifications can have a large impact on performance [31].

6.2 Program Structure Patterns in STAPL

The ease seen in implementing the data structure patterns comes from STAPL's data-centric nature. The program structure patterns depend more on the notion of tasks, which are encoded using STAPL's pRanges. In STAPL, the role of pRanges is analogous to the role of STL's iterators. pRanges specify tasks, the data within a pContainer on which these tasks operate, and the dependencies among tasks. In this subsection we examine the program structure patterns and their realization in STAPL.

Loop-level Parallelism

The loop-parallel pattern suggests increasing parallelism and program efficiency by transforming loops in serial programs so as to expose parallelism and improve data locality [1]. In STAPL parallelism can be exposed across elements of data structures. Many common loop transformations focus on changing the order in which iteration points are visited within a loop. In STAPL it is possible to change the order in which data elements are visited by an algorithm. The exact scheduling of an algorithm is handled by STAPL's runtime system but the realized schedule must follow dependences specified by a pContainer's pRange. The distribution of tasks to locations is also handled by STAPL's runtime system, but is guided by the distribution of data, which is specified with a partition mapper. Tasks that operate on a view of data are executed on the location where the data resides.

Single-Program Multiple Data (SPMD)

The SPMD pattern can be implemented in a restricted way in STAPL. In STAPL, the assignment of thread to data works in an owner-computes fashion. Users can assign the ownership of this data with a custom partition mapper. The implementation detail of instantiating threads to conduct this algorithm's work is handled by the STAPL runtime and is made invisible to the programmer. Figure 8 demonstrates an implementation of SPMD in STAPL by applying the function `workfunc` to each element of the data vector.

Master-Worker or Task-Queue

STAPL is built around the concept of algorithms being applied in parallel to data stored in containers. In order to implement the Master-Worker pattern within this limitation, the data stored in a container must represent the actual tasks to be performed. Because the STAPL runtime assigns all the elements of a container to a loca-

```
// place values into data vector ...
pVector<double> data;
p_for_each(
  data.begin(), data.end(), workfunc);
```

Figure 8: SPMD in STAPL

tion when the algorithm starts, all tasks must be determined before any of them begins execution. This is in contrast with OpenMP and Chapel, which allow for a truly dynamic scheme in which tasks themselves can add still more tasks to the queue.

Any STAPL container holding task objects can act as a task queue. The semantic behavior of the queue is delegated to the container the programmer chooses. The key to implementing the master-worker pattern with one of these containers is to place the task object in the memory of the node that will execute the task. This is done using the partition_mapper and partitioner. As mentioned above with respect to loop level parallelism, the partition_mapper is an injected feature that addresses the distribution of data across locations.

Bulk Synchronous Parallelism

The bulk synchronous parallelism pattern (BSP) can be implemented in STAPL in a *ping-pong* manner. This is done using two containers that alternate roles each superstep, one that holds local data and another that contains globally accessible data. Within a superstep only local data may be written. If the algorithm within a superstep is delegated to a pAlgorthim, the STAPL system will do a synchronization after this algorithm – making the implementation detail of the barrier delegated to the pAlgorithm.

7. CONCLUSIONS AND FUTURE WORK

The concept of specifying implementation details orthogonally to the algorithm has been realized to varying degrees in programming language constructs, pragmas, and libraries. Two categories of important implementation details that are specified orthogonally are scheduling and storage mapping. Scheduling considerations focus on which processor should do computations and when; storage map considerations focus on where data should reside. This paper shows how and the extent to which the Chapel programming language, OpenMP preprocessor, and STAPL library support the realization of parallel implementation strategies in the Our Parallel Language (OPL) while avoiding code obfuscation and providing programmer control over implementation details.

Our qualitative evaluation of code tangling and programmer control can be used to help decide what mechanisms and features within those mechanisms to use when realizing implementation strategy patterns. We summarize our results in Tables 1 through 5. To help in comparing the features in these tables we have defined six categories that describe how a feature is used, and evaluate these categories in terms of tangling and programmer-control. The injected and separated categories provide the best tradeoff in terms of low tangling and high programmer control. However, there are other considerations we do not address in this paper, such as performance, ease of adoption, interoperability, maturity of tools, and extent to which programmer control requires programmer responsibility for specifying implementation details.

Future research needed for the orthogonal specification of parallel implementation details from algorithm includes the creation of a computation abstraction hierarchy with a corresponding scheduling and storage mapping abstraction hierarchy. Sample computation abstractions include map-reduce, pipeline, and task graph. These same abstractions are in OPL. Additionally, we need approaches for scheduling and storage mapping across and between computations that do not fit within the same specification abstraction.

As can be seen with the abundance of injected and delegated features, the move to orthogonalize implementation abstractions from algorithm abstractions is gaining momentum in the research areas of programming model and library development. The parallel programming patterns community has provided the start of a computational abstractions hierarchy. By leveraging the work in both communities, we can make significant progress toward easing the development of *efficient* and *maintainable* parallel programs.

Acknowledgements

The authors thank Brad Chamberlain of Cray, Inc., and Sudipto Ghosh of Colorado State University, for their helpful comments and review of this work.

8. REFERENCES

[1] Loop parallelism. the our pattern language wiki. 2009. `http://parlab.eecs.berkeley.edu/wiki/patterns/loopparallelism`. OPL Working Group, Berkeley Parallel Computing Labaratory.

[2] OpenMP website - list of compilers. http://openmp.org/wp/openmp-compilers/.

[3] Chapel language specification 0.785. Technical report, Cray Inc., 2009.

[4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel programming library for c++. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, August 2001.

[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th Interntional Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2004.

[6] A. Basumallik and R. Eigenmann. Towards automatic translation of openmp to mpi. In *Proceedings of the 19th annual International Conference on Supercomputing*, pages 189–198, New York, NY, USA, 2005. ACM Press.

[7] D. Callahan, B. Chamberlain, and H. Zima. The cascade high productivity language. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, 2004.

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

[9] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[11] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *Proceedings of the International Workshop on Language Runtimes, OOPSLA*, 2004.

[12] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. Wiley-Interscience, 2005.

[13] W. C. et al. Introduction to upc and language specification. Technical report, DA Center for Computing Sciences, 1999.

[14] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *In IEEE International Conference on Parallel*

Architecture and Compilation Techniques (PACT), pages 106–111. IEEE Computer Society Press, 1998.

[15] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[16] J. Hoeflinger. Extending OpenMP to clusters. White paper, Intel of Cape Town, 2006.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages*, pages 319–329, 1988.

[18] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 91–108, 1993.

[19] K. Keutzer and T. Mattson. Our Pattern Language (OPL): A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, June 2009.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[21] A. Kukanov and M. J. Voss. The foundation for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 2007.

[22] B. Liskov. A history of CLU. pages 471–510, 1996.

[23] A. Marowka. Bsp2omp: A compiler for translating bsp programs to openmp. *Int. J. Parallel Emerg. Distrib. Syst.*, 24:293–310, August 2009.

[24] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[25] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard templates adaptive parallel library. In *Proceedings of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburg, PA, May 1998.

[26] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, K. Keahey, M. Srikant, and M. Tholburn. Pooma: A framework for scientific simulation on parallel architectures, 1996.

[27] R. Schreiber and J. J. Dongarra. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, 1990.

[28] T. Smith and M. Waterman. Identification of common molecular subsequences, 1981.

[29] L. Snyder. *The ZPL Programmer's Guide*. MIT Press, March 1999.

[30] Y. suk Kee. ParADE: An OpenMP programming environment for SMP cluster systems. In *Proceedings of ACM/IEEE Supercomputing (SC'03)*, pages 12–15, 2003.

[31] G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pArray. In *MEDEA '07: Proceedings of the 2007 workshop on Memory performance*, pages 73–80, New York, NY, USA, 2007. ACM.

[32] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. Associative parallel containers in STAPL. In *Languages and Compilers for Parallel Computing: 20th International Workshop, LCPC 2007, Urbana, IL, USA, October 11-13, 2007, Revised Selected Papers*, pages 156–171, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] R. van der Pas. An overview of OpenMP 3.0. In *International Workshop on OpenMP*, 2009.

[34] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation*, 1991.

[35] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.