Generalizing Run-time Tiling with the Loop Chain Abstraction

Michelle Mills Strout*, Fabio Luporini†, Christopher D. Krieger*, Carlo Bertolli‡,
Gheorghe-Teodor Bercea†, Catherine Olschanowsky*, J. Ramanujam§, and Paul H.J. Kelly†
*Colorado State University, †Imperial College London, ‡IBM T.J. Watson Research, §Louisiana State University
*{mstrout|krieger|cathie}@cs.colostate.edu, †{f.luporini12|gheorghe-teodor.bercea08|p.kelly}@imperial.ac.uk,
‡cbertol@us.ibm.com, §jxr@ece.lsu.edu

***Abstract*—Many scientific applications are organized in a data parallel way: as sequences of parallel and/or reduction loops. This exposes parallelism well, but does not convert data reuse between loops into data locality. This paper focuses on this issue in parallel loops whose loop-to-loop dependence structure is data-dependent due to indirect references such as A[B[i]]. Such references are a common occurrence in sparse matrix computations, molecular dynamics simulations, and unstructured-mesh computational fluid dynamics (CFD). Previously, sparse tiling approaches were developed for individual benchmarks to group iterations across such loops to improve data locality. These approaches were shown to benefit applications such as moldyn, Gauss-Seidel, and the sparse matrix powers kernel, however the run-time routines for performing sparse tiling were hand coded per application. In this paper, we present a *generalized full sparse tiling algorithm* that uses the newly developed loop chain abstraction as input, improves inter-loop data locality, and creates a task graph to expose shared-memory parallelism at runtime. We evaluate the overhead and performance impact of the generalized full sparse tiling algorithm on two codes: a sparse Jacobi iterative solver and the Airfoil CFD benchmark.**

***Keywords*-inspector/executor, run-time reordering transformations, tiling**

I. Introduction

Intranode parallelization is a difficult problem that many libraries and programming models attempt to address [1]. To expose parallelism in these programming models, scientific simulations commonly express the application as a series of data parallel or reduction loops. However, poor and unpredictable data locality often limits performance and data reuse among the loops is not effectively turned into data locality. This is particularly true for irregular applications that access data using an indirection array, such as `A[B[i]]`. These irregular accesses are common in such fields as computational fluid dynamics, molecular dynamics, differential equation solvers on unstructured meshes, and sparse linear algebra.

Sparse tiling techniques were developed to group iterations of irregular applications into atomic tiles at runtime with an inspector [2]–[5]. In general, the inspector iterates over index arrays that do not change during the main computation to determine data reorderings or new schedules, like sparse tiling schedules. The resulting tiles have either an implicit or explicit partial ordering (i.e., a task graph) that exposes asynchronous parallelism [2], [6], [7]. These benchmark-specific, sparse tiling executors exhibited performance improvements for sparse stencil computations [2], Gauss-Seidel [3], [6], moldyn [4], and sparse matrix powers kernel [5]. These approaches were not general because the sparse tiling inspector algorithms were developed by hand, per application.

In this paper, we present a generalized, full sparse tiling algorithm that leverages a common pattern in irregular computations and many other scientific codes: a series of parallel and/or reduction loops that reuse data. Full sparse tiling was called *full* because it segments the whole iteration space into sparse tiles unlike other techniques that grew tiles that could be executed in parallel, but then had a large cleanup tile [3].

Previous work introduces an abstraction called the *loop chain* [8] to represent such loop sequences and the data access information about each of the loops. Fig. 1 illustrates an example extracted from an unstructured mesh, computational fluid dynamics (CFD) program written using the OP2 [9] library, where it is possible to derive a loop chain abstraction. In the example, the first loop iterates over edges in a mesh, reads data associated with each edge that is stored in the `x` array, and then indirectly updates the value of the `vert` data associated with the vertices adjacent to each edge using the `edges2vertices` indirection array. The second loop iterates over cells/triangles and updates all data associated with vertices adjacent to each cell. Finally the third loop visits the edges again. Each of these loops is reusing the data associated with the vertices in the unstructured mesh and data access patterns for each loop can be determined using the OP2 library semantics. Fig. 2 shows a possible loop chain with data access edges that are determined at inspector time.

Generalized full sparse tiling (or gFST) converts data reuse within loop chains into data locality, while exposing task-graph parallelism. Fig. 3 illustrates a full sparse tiling on the example loop chain in Fig. 2. Note that the resulting task graph in Fig. 3 has two tiles/tasks that can be executed in parallel, Tiles 2 and 3. Larger examples result in significant improvements in data locality while still exposing sufficient parallelism.

To prototype usage of the loop chaining abstraction as input for a generalized full sparse tiling algorithm, we developed a library where a programmer can replace a sequence of loops with function calls that specify the computation as a loop chain. In Fig. 1, the calls to `op_par_loop` can be replaced

```
void kernel1 (double * x, double * v1, double * v2) {
  *v1 += *x;  *v2 += *x;
}
// loop over edges
op_par_loop (edges, kernel1,
  op_arg_dat (x, -1, OP_ID, OP_READ),
  op_arg_dat (vert, 0, edges2vertices, OP_INC),
  op_arg_dat (vert, 1, edges2vertices, OP_INC))

// loop over cells
op_par_loop (cells, kernel2,
  op_arg_dat (vert, 0, cells2vertices, OP_INC),
  op_arg_dat (vert, 1, cells2vertices, OP_INC),
  op_arg_dat (vert, 2, cells2vertices, OP_INC),
  op_arg_dat (res, -1, OP_ID, OP_READ))

// loop over edges
op_par_loop (edges, kernel3,
  op_arg_dat (vert, 0, edges2vertices, OP_INC),
  op_arg_dat (vert, 1, edges2vertices, OP_INC))
```

Fig. 1: Section of an OP2 program that is used as a running example to illustrate the loop chain abstraction and show how the sparse tiling algorithm works. The definition of the toy kernel1 shows how all kernels receive their input from the `op_par_loop` implementation.

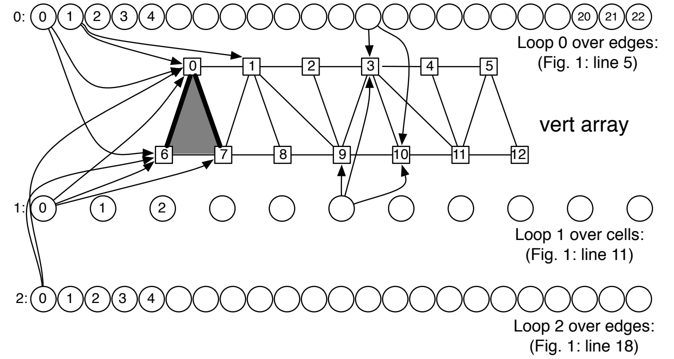

Fig. 2: Visualization of a possible instance of the loop chain for the sequence of loops in Fig. 1. The edge and cell loops use index arrays to indirectly access the data associated with the vertices in the mesh. Squares represent data associated with vertices in a mesh. Circles represent loop iterations. Note that iteration 0 in loop 0 visits the edge connecting vertices 0 and 6 and accesses the data associated with those vertices. Iteration 0 in loop 1 accesses all the vertices associated with the shaded triangular cell. Iteration 0 in loop 2 accesses vertices 0 and 6.

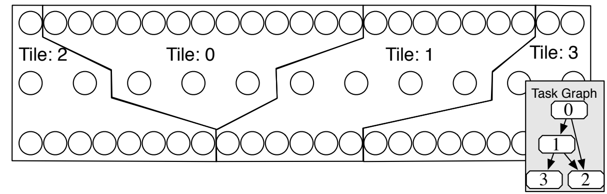

Fig. 3: A sparse tiling for the loop chain in Fig. 2. The iterations in the three loops have been placed into four tiles, which have been partially ordered into a task graph. The partial ordering arises from dependencies between iterations in different tiles.

with calls to routines that indicate which loops are in the loop chain and, for each loop, how each iteration in the loop chain accesses data. At run-time this specification is passed to the gFST inspector, which appends the specification with a task graph and mapping of iterations in the loops to tiles/tasks in that task graph. The executor, which replaces the original computation, then executes that task graph. The ultimate goal is to have a compiler identify loop chains within a program, do a cost-benefit analysis to determine whether sparse tiling would be beneficial, and insert inspectors and executors that perform sparse tiling.

The performance benefits of sparse tiling [2], [6] and specifically full sparse tiling [3]–[5] for irregular applications such as Gauss-Seidel, moldyn, and sparse matrix powers kernel have already been shown. Generalizing the full sparse tiling algorithm can increase the inspector overhead, but the improvements in the executor are similar to specialized full sparse tiling executors. To further explore the performance benefits of full sparse tiling, we applied it to Jacobi on sparse matrices from the Davis Florida collection [10] and an Airfoil simulation written in OP2. For the OP2 code, we adapted the current parallelization algorithm to perform a generalized full sparse tiling parallelization. We compared the performance of these benchmarks when parallelized using OpenMP `parallel for` pragmas on each loop versus the performance when the loops were full sparse tiled. The runtime reduction varied from 7% to 47%.

This paper's major contributions are as follows:

- A general full sparse tiling algorithm that applies to any sequence of loops that can be expressed using the loop chain abstraction.
- Performance evaluations of the full sparse tiling inspector/executor strategy when applied to a Jacobi sparse matrix solver and the Airfoil CFD simulation.

We identify a number of important issues that arise when generalizing full sparse tiling, and we explain how the general algorithm deals with these issues. Additionally, we describe how the key gFST algorithm concepts can be adapted for use in the OP2 implementation context.

In Section II, we describe how data dependence relations can be derived from the loop chain abstraction and indicate issues related to these dependences that a general sparse tiling algorithm must handle. These issues are addressed by the general full sparse tiling algorithm described in Section III. Section IV describes adapting the OP2 parallelization algorithm to implement generalized full sparse tiling. Section V reports performance results, Section VI covers work related to full sparse tiling, and Section VII concludes the paper.

II. Loop Chains for Generality

Sparse tiling techniques have been developed to improve the data locality within groups of iterations from different loops that share data and, therefore, the performance of the overall computation. *Inspector/executor strategies* implement sparse

tiling, where the executor is the transformed code with an added tiling loop and the inspector is a new piece of code that visits index arrays at runtime to determine how iterations can be legally and profitably grouped into tiles.

This section reviews the loop chain abstraction, describes how data dependences can be derived from a loop chain, and presents issues that a general sparse tiling algorithm of any kind must overcome. The issues are that (1) explicit inspection of the data dependencies between loops to perform sparse tiling more generally is too computationally expensive (Section II-B), (2) when one or more of the loops being sparse tiled are performing a reduction then there needs to be a partial ordering between tiles that perform some reduction operation on the same data element (Section II-C), and (3) when growing tiles to some loop l, it is important to consider the dependencies of loop l on all previous or all subsequent loops (Section II-D). Dependencies in these parallel loops are such that they can be from iteration in any loop L_p to iterations in loop L_q where $p < q$ in general.

A. Data Dependence Analysis for Loop Chains

The previous sparse tiling approaches cited in Section I were specialized per benchmark. In this paper we show how the loop chain programming abstraction [8] can be used as a basis for generalized full sparse tiling. As with all loop optimizations that reschedule the iterations in a sequence of loops, any sparse tiling must satisfy the data dependencies. The loop chain abstraction provides enough information to compute all of the dependencies in a computation. When the data accesses in the loop chain involve indirect memory accesses like those that are the focus of this paper, the runtime, or inspector, can determine the data dependencies by querying the data access information contained in the loop chain abstraction.

As described in Krieger et al. [8], a loop chain consists of the following:

- L is a sequence of N loops, $L_0, L_1, ..., L_{N-1}$.
- D is a set of disjoint M data spaces, $D_0, D_1, ..., D_{M-1}$.
- $R_{L_l \to D_d}(\vec{i})$ and $W_{L_l \to D_d}(\vec{i})$, where the R and W access relations are defined over for each data space $D_d \in D$ and indicate which data locations in data space D_d an iteration $i \in L_l$ reads from and writes to respectively.

In our current implementation, iteration space specifications are stored as ranges, data spaces as element size and number of elements, and the read and write access relations are stored implicitly if identity, explicit if through index arrays, and using CSR-like structure if one iteration accesses zero or more locations in an array. The assumption in a loop chain is that each loop is a *fully parallel loop* or a *reduction loop*. Here a reduction loop is more general than a scalar reduction loop. In a reduction loop each iteration of the loop does a read, an associative and commutative operation, and a write to some element(s) in an array, and multiple iterations could read, modify, write the same data element.

The access relations in the loop chain abstraction enable a general derivation of the storage-related dependencies between loops in a loop chain. The storage related dependencies between loops can be described as either flow (read after write), anti (write after read), or output (write after write) dependencies. Loop L_x always comes before loop L_y in the loop chain. The flow dependencies can be enumerated by considering pairs of points ($\vec{i}$ and $\vec{j}$) in the iteration spaces of the two loops L_x and L_y:

$$\{\vec{i} \to \vec{j} \mid \vec{i} \in L_x \wedge \vec{j} \in L_y \wedge W_{L_x \to D_d}(\vec{i}) \cap R_{L_y \to D_d}(\vec{j}) \neq \emptyset\}.$$

Anti and output dependencies are defined as expected.

There are reduction dependencies between two or more iterations of the same loop when those iterations read, modify with a commutative and associative operator, and write to the same location(s). The reduction dependencies in loop L_x are

$$\{\vec{i} \to \vec{j} \mid \vec{i} \in L_x \wedge \vec{j} \in L_x \wedge W_{L_x \to D_d}(\vec{i}) \cap W_{L_x \to D_d}(\vec{j}) \neq \emptyset\}.$$

The reduction dependencies between two iterations within the same loop indicates that those two iterations must be executed atomically with respect to each other.

B. Data Dependence Inspection Issue

With the computations that have been sparse tiled in the past (Jacobi, Gauss-Seidel, moldyn, matrix powers kernel), inspecting the dependencies between loops was implemented by traversing index arrays. For example, in Jacobi, each iteration of a loop accesses a set of neighbors. If the sparse matrix is stored in compressed sparse row format then there is a compact list of neighbor identifiers. By iterating over this list the dependence relation is being inspected. The dependence relation can be $\{[i] \to [j] \mid j \in neighbors(i)\}$. The inspector can traverse the domain for the i iterations and determine data dependencies via the neighbor set.

More generally, traversing data dependencies between loops might require more computation than is preferred in an inspector, which, after all, must be amortized over multiple iterations of the loop chain. For example, the data dependencies between the first edge loop and the cell loop for the running example in Fig. 2 are

$$\{[i] \to [j] \mid edges2vertices(i, *) = cells2vertices(j, *)\},$$

where $*$ indicates any of the index arrays into vertices for edges or cells. Therefore if an edge and a cell share a vertex, then there is a dependence from the edge iteration to the cell iteration. Inspecting this relation requires a doubly-nested loop that iterates over all edges and then for each edge all cells, $O(|E||C|)$, where $|E|$ is the number of iterations in the edge loop and $|C|$ is the number of iterations in the cell loop. In Section III, the generalized full sparse tiling algorithm performs tile growth in $O(|E| + |C|)$ instead of $O(|E||C|)$ by avoiding the explicit traversal of data dependences.

Existing inspector/executor techniques for sparse tiling avoid explicitly enumerating data dependencies between loops by being specialized for specific computations. Wavefront parallelization techniques for do across loops also avoid explicit enumeration of data dependencies by tracking how iterations access data [11]. Sparse tiling techniques are different in that they aggregate iterations into tasks and determine a

task graph instead of determining level sets containing fine grained parallelism. We use a similar approach of tracking data accesses to avoid explicit data dependence enumeration.

C. Handling Reductions

Strout et al. [4] sparse tiled the moldyn computation across three of its loops. The loop over interactions between atoms was a reduction loop. Due to reduction dependencies and the resulting computation being performed serially, there was no need to consider dependencies between tiles.

In general, reduction dependencies require that tiles performing a reduction operation to the same data element are given some arbitrary partial order to avoid concurrent execution, which could lead to data races.

D. Dependencies from Non-Adjacent Loops

Existing sparse tiling techniques only inspect dependencies between adjacent loops. This was because of symmetry between dependencies that caused the dependencies between a loop and much earlier loops to be covered by the transitive closure of dependence steps between intervening loops. In general, a loop could have a dependence, for example an anti dependence, on a loop that is not directly adjacent. The generalized full sparse tiling algorithm handles this case.

III. Generalized Full Sparse Tiling Algorithm

To handle all the issues discussed in Section II, the generalized full sparse tiling algorithm uses the data access relations provided by the loop chain abstraction and a data structure we call Ψ that tracks all tiles that write to and read from each data item in each loop to produce a valid execution schedule. Using the given data access relations and the Ψ data structure, the generalized algorithm is able to satisfy the ordering constraints in the loop chain due to data dependencies.

A. Algorithm Description

The generalized full sparse tiling algorithm takes a loop chain and assigns each iteration of the loops to a tile. Along with the iteration-to-tile mapping, a task graph representing a partial ordering of the tiles is generated. More precisely, the input to the algorithm is a loop chain (LC), the index of the loop chosen for seed partitioning (s), and the number of tiles (T). Note that any loop can be selected for seed partitioning, but heuristically a loop in the middle of the loop chain results in fewer dependencies between tiles [12]. The number of tiles is a tuning parameter used to balance data locality and parallelism. The output is a function θ that maps each iteration of each loop in the chain to a tile and a task graph G that captures the partial ordering among the tiles due to data dependencies.

The general full sparse tiling method for loop chains consists of four phases: *initialization, backward tiling, forward tiling*, and *task graph creation*. Algorithm 1 shows the pseudocode.

Phase 1: Initialize internal data. Data dependency information is required during task graph creation. Rather than tracking data dependencies directly, which may be prohibitively expensive, the algorithm maintains information pertaining to data reads and writes with respect to tiles. The set of tiles in a particular loop (l) that read a particular data item ($\vec{v}$) in data space (d) is denoted as $\Psi_R(d, \vec{v}, l)$. The tiles that write are tracked in a similar set $\Psi_W(d, \vec{v}, l)$.

1 **GeneralizedFullSparseTile**
Input: Loop Chain $LC = (L, D, R, W)$, s, T
Output: θ, G
Data: Ψ_*
2 // Initialize all fields of Ψ to top or empty set
3 // Initialize the tiling function θ values to $\top$
4 $\theta(L_s, *) = PartitionSeedSpace(L_s, R, W, T)$
5 UpdateAccessTable(Ψ_*, s)
6 //Tile the loop chain
7 **foreach** L_l *in* L_{s-1} *to* L_0 **do**
8 | BackwardTile($L, l, R, W, \theta, \Psi_*$))
| UpdateAccessTable(Ψ_*, l)
9 **end foreach**
10 **foreach** L_l *in* L_{s+1} *to* L_{N-1} **do**
11 | ForwardTile($L, l, R, W, \theta, \Psi_*$)
| UpdateAccessTable(Ψ_*, l)
12 **end foreach**
13 // Partial ordering of tiles
14 G = BuildTaskGraph(L,D,Ψ_*,T) **return** θ, G

Algorithm 1: The Generalized Full Sparse Tiling Algorithm

Additionally, tile data access information is used during backward and forward tiling. Associated with each of the Ψ_R and Ψ_W sets are single values that record the first and last tiles that access a specific data element. $\Psi_{LR}(d, \vec{v}, l)$ is the last read performed on the vth data element of the data space D_d from loop L_l. Replacing the subscripts with FR, FW, and LW correspond to the first read, first write and last write respectively. The initialization phase initializes all the tile assignments to top ($\top$) and sets all of the data access information in Ψ to empty set or top as appropriate.

The initialization phase also includes a preliminary partitioning of the seed loop's iteration space, L_s. Partitioning the seed loop involves assigning each iteration of the seed loop to a tile. $UpdateAccessTable$ updates the values in all internal state data sets (Ψ_*) with respect to the seed partitioning. Specifically, since there is a tile assignment for all the iterations in the seed loop, it will be possible to determine the set of tiles that read and write to each data element accessed in that seed loop.

For example, Fig. 4 shows a portion of the Ψ data structure for the loop chain example in Fig. 2. In this example, the seed loop is chosen as loop 1. This is the loop over cells (a cell is a triangle here). In this example we follow the creation of tile 2. After the seed partitioning, iterations 0 and 1 in loop 1 are in tile 2, $\Theta(1, 0) = 2$ and $\Theta(1, 1) = 2$. The Ψ data structure contains tile access information for each data element (i.e., vertices in the mesh) at each loop in the computation (i.e., note three columns for each vertex, one for each of the three loops). The left side of Fig. 4 shows the initial write sets for

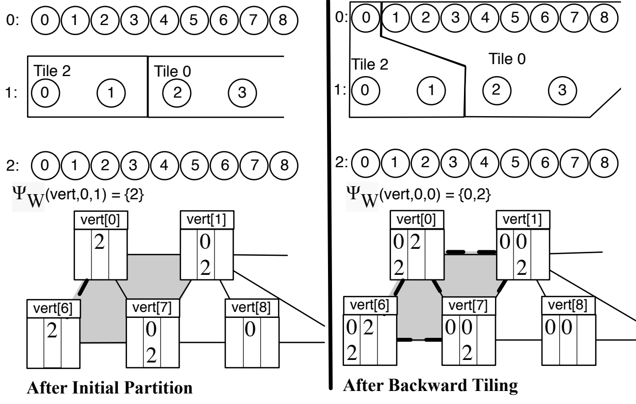

Fig. 4: This figure illustrates part of the evolution of the Ψ_W data structure for the loop chain example in Fig. 2. After the initial seed partitioning of loop 1, which iterates over cells, the middle columns (loop 1) associated with each vertex includes the set of tiles for cells that are adjacent to the vertex. After backward tiling, the Ψ_W data structure is updated to include the tile numbers for all the edges that access a vertex in loop 1. Only one edge is in tile 2 in loop 0, so all vertices shown are written to by edges in Tile 0.

part of the `vert` data structure. The first two cells have been put into tile 2 in the seed partition loop. The vertices adjacent to these two cells have a 2 in the center column to illustrate that the tile 2 is in the sets $\Psi_W(vert, 0, 1)$ and $\Psi_W(vert, 1, 1)$. The zeros in the middle columns of the Ψ table are for vertices adjacent to cells that are in tile 0.

Phase 2: Backward Tiling. The result of tiling is that each iteration in the loop chain is assigned to a tile. Backward tiling (see Algorithm 2) starts with the iterations in the seed partitions and grows tiles to earlier loops ensuring that the data dependencies are satisfied. Each iteration in the loop being tiled is assigned to either the existing tile assignment or $\Psi_{FW}(d, \vec{v}, l)$ or $\Psi_{FR}(d, \vec{v}, l)$, depending on which occurs first (the result of MIN).

Consider the running example in Fig. 4. Backward tiling in this example only occurs on loop 0. Iteration 0 in loop 0 starts with a tile value of top, $\Theta(0, 0) = \top$. Iteration 0 in loop 0 does a reduction operation on two vertices (0 and 6, the edge is represented as a bold, dashed line) and we have $\Psi_{FW}(vert, 0, 1) = 2$ and $\Psi_{FW}(vert, 6, 1) = 2$. Therefore, iteration 0 of loop 0 is assigned to tile 2, $\Theta(0, 0) = 2$. As another example, iteration 1 of loop 0 accesses the vertices 0 and 1, and $\Theta(0, 1) = \top$, $\Psi_{FW}(vert, 0, 1) = 2$, and $\Psi_{FW}(vert, 1, 1) = 0$. $MIN(\Theta(0, 1), \Psi_{FW}(vert, 0, 1), \Psi_{FW}(vert, 1, 1)$ is 0 and therefore iteration 1 in loop 0 is assigned to tile 0, $\Theta(0, 1) = 0$. Intuitively any time an earlier loop iteration will be writing to a data item that an iteration in a later loop accesses, then the earlier loop iteration needs to be in the same or earlier tile.

Once the tile assignments are chosen, Ψ_* is updated to reflect the changes. These values are reflected in the first

1 **BackwardTile**
  Input: LC, l, Ψ_*
  Output: θ
2 Define: MIN($\top$,X) = X.
3 **foreach** $\vec{i} \in L_l$ **do**
4   // all datasets
5   **foreach** $d \in D$ **do**
6   // all data elements read by this iteration
7   **foreach** $\vec{v} \in R_{L_l \to D_d}(\vec{i})$ **do**
8   // each later loop's access tables
9   **foreach** $L_k \in \{L_{l+1}\ to\ L_s\}$ **do**
10   // anti dependence
11         $\theta(l, \vec{i}) = MIN(\theta(l, \vec{i}), \Psi_{FW}(d, \vec{v}, k))$
12   **end foreach**
13   **end foreach**
14   // all data elements written by this iteration
15   **foreach** $\vec{v} \in W_{L_l \to D_d}(\vec{i})$ **do**
16   **foreach** $L_k \in \{L_{l+1}\ to\ L_s\}$ **do**
17   // output dependence
18         $\theta(l, \vec{i}) = MIN(\theta(l, \vec{i}), \Psi_{FW}(d, \vec{v}, k))$
19   // flow dependence
20         $\theta(l, \vec{i}) = MIN(\theta(l, \vec{i}), \Psi_{FR}(d, \vec{v}, k))$
21   **end foreach**
22   **end foreach**
23   **end foreach**
24 **end foreach**

Algorithm 2: The Backward Tiling Algorithm

column of each vertex in Fig. 4.

Phase 3: Forward Tiling. In this phase, loops after the seed space are tiled. This process starts with the iteration space immediately following the seed loop and proceeds forward, loop by loop, until reaching the last loop in the chain. The forward tiling algorithm uses MAX where the backward tiling algorithm uses MIN. It exploits the last read and write information to ensure data dependencies are satisfied.

Phase 4: Task Graph Creation. The edges in the task graph represent the data dependencies that occur between iterations in different tiles (for example see Fig. 3). The four steps of the task graph creation, as shown in Algorithm 3, correspond to the four types of dependencies between tiles. To avoid cycles in the task graph, the source tile of an edge should always be numbered lower than the target tile for the edge.

Reduction dependencies are detected when a single entry in the Ψ_W table includes multiple tiles, thus indicating that multiple tiles are writing to a single vertex in the same reduction loop. A partial ordering from the lower to higher numbered tiles is placed in the task graph to avoid data races. Edges representing flow dependencies are created by connecting all tiles that read a specific data element within a given loop to the tile that has most recently written to that data element in a previous loop. Anti-dependence edges are similar, but connect all of the tiles that read a given element in a given loop to the first tile in a subsequent loop that writes to

```
1  BuildTaskGraph
   Input: L, D, Ψ_*, T
   Output: G = (V, E)
2  E = ∅, V = {0, 1, ··· , T − 1} // Each tile is task
3  foreach d s.t. D_d in D_0 to D_{M−1} do
4      foreach l s.t. L_l in L_0 to L_{N−1} do
5          foreach v⃗ ∈ D_d do
6              // Reductions
7              E = E ∪ {[s] → [t] | s ∈ Ψ_W(d, v⃗, l)∧
8                  t ∈ Ψ_W(d, j⃗, l) ∧ s < t}
9              // Flow dependencies
10             foreach
               k s.t. L_k in L_l to L_0 until Ψ_LW(d, v⃗, k) ≠ ⊤
               do
11                 E = E ∪ {[s] → [t] | s = Ψ_LW(d, v⃗, k)∧
12                     t ∈ Ψ_R(d, j⃗, l) ∧ s < t}
13             end foreach
14             // Anti dependencies
15             foreach
               k s.t. L_k in L_l to L_{N−1} until Ψ_FW(d, j⃗, k) ≠
               ⊤ do
16                 E = E ∪ {[s] → [t] | s ∈ Ψ_R(d, v⃗, l)∧
17                     t = Ψ_FW(d, v⃗, k) ∧ s < t}
18             end foreach
19             // Output dependencies
20             foreach
               k s.t. L_k in L_l to L_{N−1} until Ψ_FW(d, v⃗, k) ≠
               ⊤ do
21                 E = E ∪ {[s] → [t] | s = Ψ_LW(d, v⃗, l)∧
22                     t = Ψ_FW(d, v⃗, k) ∧ s < t}
23             end foreach
24         end foreach
25     end foreach
26 end foreach
```

Algorithm 3: The task graph is determined by inspecting all tiles that read and write to each data element.

the tile. Output dependences are found by connecting the last write of a data element to the first write of the same element in subsequent loops.

The generalized full sparse tiling algorithm maps each iteration in the loop chain to a tile with the tile mapping function θ and generates a partial ordering between the tiles in the form of a task graph. One possible execution model is to then execute each tile/task serially (loops executed in the original loop sequence within each tile) and to execute tiles/tasks that do not share a partial ordering in parallel.

B. Algorithm Complexity and Correctness

The backward (and forward) tiling algorithms traverse all iteration and data index pairs in the read $R_{L\to D}$ and write $W_{L\to D}$ relations. For each such pair, the impact of all previous loops in the loop chain on the data item in question are queried in the Ψ data structure (line 10 of the Backward Tiling algorithm). Therefore the complexity is $O(NMP)$ complexity, where N is the total number of iterations in the loop chain, M is the average number of data accesses per iteration, and P is the number of loops in the loop chain. Note that NM is the number of pairs in the read $R_{L\to D}$ and write $W_{L\to D}$ relations. We avoid $O(N^2)$ behavior such as the $O(|E||C|)$ from Section II-B that would be needed if all data dependencies between iterations were explicitly determined.

We do need to explicitly determine dependencies between tasks to specify the task graph. The task graph construction algorithm has a worst case complexity of $O(T^2)$, where T is the number of tiles, because all of the tiles could read or write to a particular data element. However, if that were the case, then the problem is not sparse enough for full sparse tiling to be effective.

Any schedule for iterations in a loop chain is correct if the schedule satisfies the partial ordering between iterations in the loop chain dictated by the data dependencies detailed in Section II-A. The gFST algorithm satisfies these data dependencies by determining the dependencies between tiles instead of iterations and by ensuring no tile dependence cycles.

IV. OP2 IMPLEMENTATION OF GFST

OP2[1] is a library for implementing applications that solve partial differential equations over unstructured meshes. For example, OP2 is employed in the Hydra CFD application, used at Rolls Royce for the simulation of next-generation components of jet engines, and in Volna [13], a CFD application for the modelling of tsunami waves. These applications are composed of a large number of parallel or reduction loops (hundreds in Hydra, tens in Volna). This section describes how we adapted the current OP2 parallelization algorithm to implement generalized full sparse tiling for OP2 programs. Inclusion of the calls to the OP2 gFST inspector and introduction of tile loops into the executor code is done manually. The goal is to assess the performance of gFST in the context of a mature code base like OP2, before incorporating this optimization into the OP2 compiler.

A. Loop chains in the OP2 Library

OP2 offers abstractions for modeling an unstructured mesh in terms of sets, datasets and mappings between sets. OP2 programs are expressed as sequences of parallel loops, each loop applying a user-programmed function, or "kernel", to every element in the iteration set. For each set element, datasets are accessed through maps. For example, the parallel loops in the program in Fig. 1 iterate over sets of mesh elements (`edges`) and access datasets (`vert`) via mappings, which can be indirect (`edges2vertices`) or direct (`op_id`).

In OP2, access modes are used to indicate how datasets are being accessed: (1) `OP_READ` - the dataset is only read, (2) `OP_WRITE` - the dataset is written to, (3) `OP_INC` - an increment for each value of that dataset is computed without reading the actual value of the dataset. An access descriptor (op_arg_dat) contains: a dataset, a mapping and an access

[1]OP2 denotes that this is the second generation of the OPlus library, or Oxford Parallel Library.

mode. Each parallel loop contains an access descriptor for each dataset used by the kernel. This information captures the loop chain abstraction in OP2.

B. Specializing gFST for OP2 Loop Chains

The standard OP2 OpenMP parallelization is done on a per loop basis and is achieved by block partitioning the iteration set (e.g. cells, edges). In the OP2 gFST inspector, instead, partitioning occurs only once and is performed on the mesh vertices. These partitions are grown to tiles in the backward and forward tiling operations. In both cases, serialization of iterations incrementing the same value (i.e., reduction dependencies) is enforced by giving a color to partitions and allowing only same-colored partitions to execute in parallel, with iterations inside a partition being executed sequentially.

The OP2 gFST inspector extends this technique by coloring partitions to respect tile-to-tile dependencies exposed by the forward and backward tiling operations. The inspector builds a seed partition graph (SPG) with each node of the SPG being a partition of vertices. Edges are inserted in the SPG based on the K-reachability relation between partitions: if two vertices in separate partitions are within K edges or cells of each other then their corresponding partition nodes in the SPG will be connected. Building the SPG costs $O(TN(B^K))$, where T is the number of tiles, N is the number of vertices on the border of a partition, B is the average out degree for vertices in the mesh, and K is how many edges the depth-first search visit traverses from each border vertex. In our examples B is 3 or 4, K is the number of loops in the loop chain and tends to be small, and N is around 200. The SPG is then colored to prevent same-colored tiles from sharing a dependence.

The gFST algorithm tracks all the tiles that read from and write to a particular data item in *each* loop. This information, stored in the Ψ_* data structure, is used to determine a partial order of tile execution due to data dependencies. The OP2 gFST simplifies dependency tracking by exploiting the fact that vertices are accessible by all iteration set via mappings. Instead of storing all the tiles that read and write to a particular vertex per loop, it is only necessary to track the first tile that reads or increments a vertex while doing backward tiling and the last tile that reads or increments a vertex while doing forward tiling. Specifically the complexity of the OP2 gFST tile growth is $O(NM)$ instead of $O(NMP)$, where where N is the total number of iterations in the loop chain, M is the average number of data accesses per iteration, and P is the number of loops in the loop chain; the loop at Line 4 in Algorithm 3 is unnecessary.

Adapting the existing OP2 parallelization algorithm to perform gFST highlights the most crucial aspects of the algorithm while illustrating that these features can be specialized for particular implementation contexts. The most crucial features are (1) there needs to be some mechanism for providing a seed partitioning, whether on an iteration space or a data space that all iteration spaces access directly or indirectly, (2) flow, anti, and output dependencies need to be respected during backward and forward tile growth to maintain tile atomicity, (3) flow, anti, output and reduction dependencies between tiles need to be partially ordered to avoid data races.

V. Evaluation of Generalized Sparse Tiling

To evaluate the performance improvements achieved by full sparse tiling, we conducted several experiments. In all the experiments, the optimal tile size (i.e. the one leading to the best execution time) was determined empirically, for each combination of machine and application. Our objective is to explore the impact of gFST on performance, and to characterize the circumstances where the approach is profitable.

A. The Sparse Jacobi Benchmark

The first experiment was the full sparse tiling of a Jacobi sparse matrix solver. Given a sparse matrix A, and a vector $\vec{f}$, related by $A\vec{u} = \vec{f}$, each iteration of the sparse Jacobi method produces an approximation to the unknown vector $\vec{u}$. In our experiments, the Jacobi convergence iteration loop is unrolled by a factor of two and the resulting two loops are chained together (1000 iterations of the loop chain was executed). Using a ping-pong strategy, each loop reads from one copy of the $\vec{u}$ vector and writes to the other copy. This experiment was run on an Intel Westmere (dual-socket 8-core Intel Xeon E7-4830 2.13 GHz, 24MB shared L3 cache per socket). The code was compiled using `gcc-4.7.0` with options `-O3 -fopenmp` and OpenMP tasks were used to execute the task graph.

The Jacobi recurrence equation includes a sparse matrix vector multiplication and is representative of a broad class of sparse linear algebra applications. It is also an effective testbed because different data dependency patterns can be examined simply by using different input matrices. In these experiments, a set of 6 input matrices, drawn from the University of Florida Sparse Matrix Collection [14], was used. The matrices were selected so that they would vary in overall data footprint, from 45 MB to 892 MB, and in percentage of non-zeros, from very sparse at 0.0006% to much more dense at 0.5539% non-zeros.

Figure 5a shows the execution time reduction achieved by full sparse tiling the Jacobi solver compared with the execution time of a simple blocked parallel version using OpenMP `parallel for` directives on the unrolled loops. The execution time reduction varied from 13% to 47% with the exception of the nd24k matrix, which showed as much as a 1.52x slowdown when full sparse tiled. This matrix is highly connected and yields a task graph that has limited parallelism. The greater parallelism available under a blocked approach provides more benefit in this case than the performance improvements due to improved locality from full sparse tiling.

These execution times do not include the inspection time necessary to full sparse tile the loop chain. To break even when this cost is considered, the inspector time must be amortized over between 1000 and 3000 iterations of the executor, depending on the specific matrix being solved. As the inspector code matures and becomes more efficient, this cost will diminish.

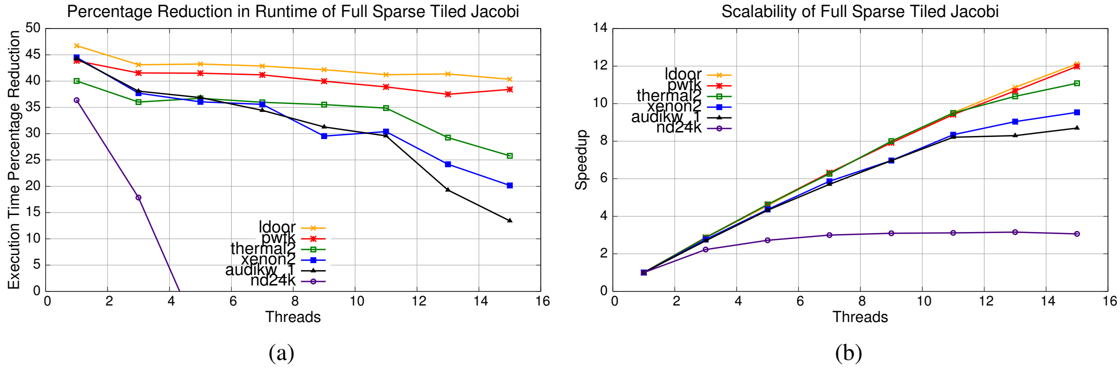

Fig. 5: The Jacobi solver's loop chain performance in terms of percentage reduction over the simple blocked parallel version, and speedup over the sequential full sparse tiled versions. Results for 6 different sparse matrices are shown. The nd24k line goes off the chart, reaching 2.5x slower at 15 cores; full sparse tiling does not always result in better performance.

In Figure 5b, the scalability of the full sparse tiled Jacobi solver is shown. In general, speedups of between 8 and 12 times over the single-threaded performance were observed when using 15 threads. A clear outlier is again the nd24k matrix that did not scale past 3.2 times the single thread performance. The high degree of connectivity present in this matrix limited the parallelism available in the task graph, which in turn limited the scalability.

B. OP2 Airfoil Benchmark

The OP2 adaptation of the generalized sparse tiling technique was evaluated in a representative unstructured mesh application called Airfoil [15]. Three implementations of Airfoil, namely omp, mpi and $tiled$, were compared on two shared-memory machines, an Intel Westmere (dual-socket 6-core Intel Xeon X5650 2.66 GHz, 12MB of shared L3 cache per socket) and a more recent Intel Sandy Bridge (dual-socket 8-core Intel Xeon E5-2680 2.00Ghz, 20MB of shared L3 cache per socket). The code was compiled using the Intel `icc 2013` compiler with optimizations enabled (`-O3`, `-xSSE4.2/-xAVX`).

The OP2 Airfoil application consists of a main time loop with 2000 iterations. This loop contains a sequence of four parallel loops that carry out the computation. In this sequence, the first two loops, called $adt\text{-}calc$ and $res\text{-}calc$, constitute the bulk of the computation. $Adt\text{-}calc$ iterates over cells, reads from adjacent vertices and write to a local dataset, whereas $res\text{-}calc$ iterates over edges and exploits indirect mappings to vertices and cells for incrementing indirect datasets associated to cells. These loops share datasets associated with cells and vertices. Datasets are composed of doubles.

In the omp and mpi implementations of Airfoil, the OpenMP and the MPI back-ends of OP2, were used. The effectiveness of these parallelization schemes has been demonstrated in [9]. The OP2 OpenMP back-end has been intuitively described in Section IV. The $tiled$ implementation exploits the OP2 gFST library for tiling a loop chain composed of 6 loops: the time loop was unrolled by a factor of two so as to tile over $adt\text{-}calc$ and $res\text{-}calc$ twice. The OP2 gFST library uses $METIS$ [16] for computing a seed partitioning of the mesh vertices.

Figure 6 shows the scalability and runtime reduction realized by full sparse tiling the loop chain on the Westmere and Sandy Bridge machines. The input unstructured mesh was composed of 1.5 million edges. It is worth noticing that both the omp and $tiled$ versions suffer from the well-known NUMA effect as threads are always equally spread across the two sockets. It is left as further work extending gFST algorithms to work around this issue. Nevertheless, compared to mpi, the $tiled$ version exhibits a peak runtime reduction of 15% on the Westmere and of 16% on the Sandy Bridge.

Results shown for $tiled$ do not include the overhead of the inspector. By also including the inspector cost, the aforementioned improvements over mpi reduce to roughly 10% on both platforms. However, as the time-marching loop in real-world OP2 applications tends to be larger than in Airfoil, we expect the overhead of the inspector to be, in general, smaller. In addition, we believe the current implementation of the gFST inspector is amenable to several optimisations.

C. Discussion of the Performance Results

The performance results presented here for Jacobi and Airfoil support previous work demonstrating the benefits of full sparse tiling [3]–[5]. Extending this approach by grouping iteration that share data across loops into tiles improves performance due to improved data locality. On multicore machines this avoids memory bandwidth saturation while scaling.

Insufficient parallelism in the task graph can limit the performance improvements. This is observed with the nd24k sparse matrix in the Jacobi Benchmark. A possible future method for increasing parallelism, when it is limited in the task graph, is to take advantage of the parallelism within each loop in each tile.

An additional limiting factor is inspector overhead. This overhead must be amortized of the full execution. The effect of this is limited in irregular scientific applications because they typically require inspector-time partitioning already.

Choosing the correct input parameters to the tiling process is key to achieving performance improvements. The parameters

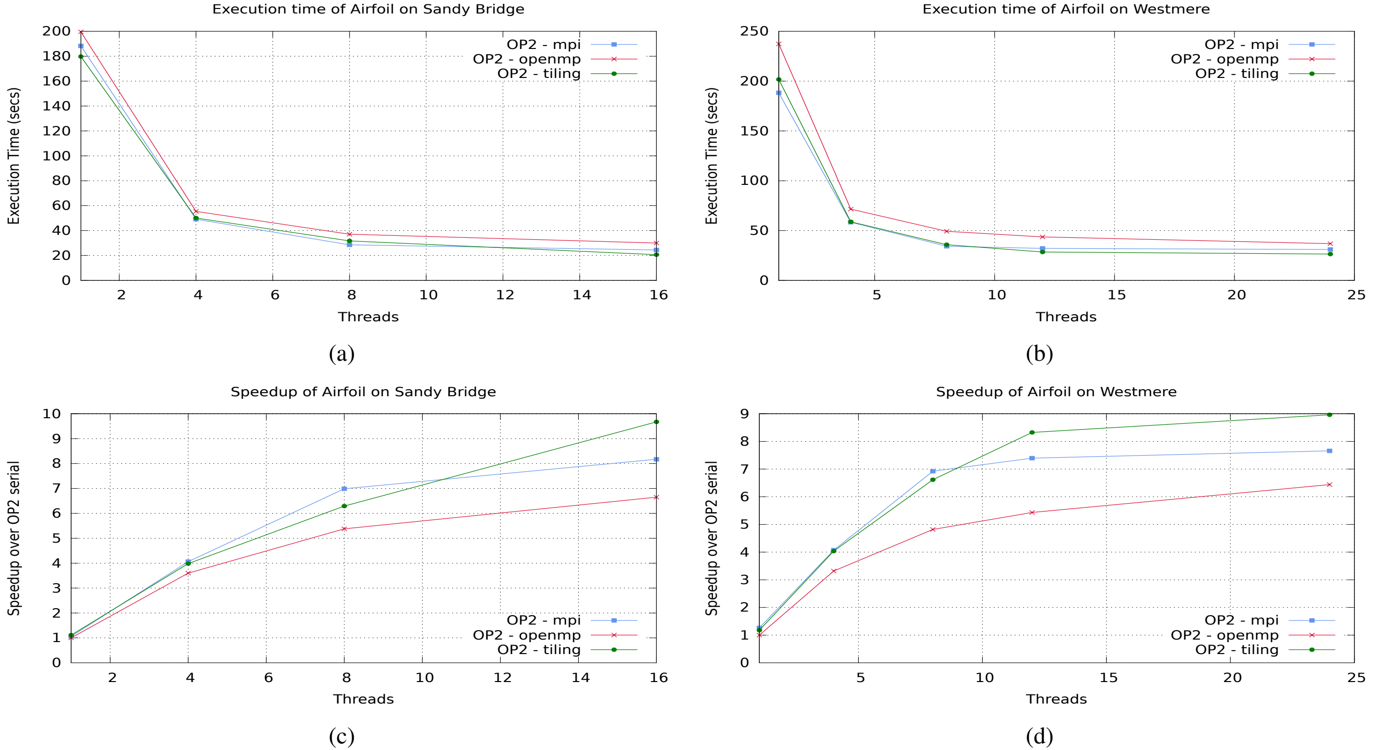

Fig. 6: The Airfoil's loop chain performance in terms of execution time (in seconds) and speedup relative to the best sequential execution time for Sandy Bridge(a,c) and Westmere(b,d). The speedup is evaluated with respect to the *omp* version with one thread (i.e. the slowest sequential back-end).

include, the number of tiles, the iteration space to use as the seed partition, and the numbering of the seed partition. The quality of the seed partition and associated coloring is especially important. Together these determine the degree of parallelism in the task graph.

VI. Related Work

Our definition of a loop chain was presented in [8] along with a discussion of how the loop chain abstraction is complimentary to previous projects that performed task scheduling in order to achieve asynchronous parallelism. In essence, projects that require manual task definition [17]–[20] may benefit from the semantics of a loop chain. Additionally, loop chaining is a general abstraction that allows for broader application than abstractions tailored to specific applications [21] or with more restrictive requirements such as iteration space slicing [22], [23], which is applicable in regular codes.

For unstructured codes, there has been various inspector/executor strategies [24] that reschedule across loops to improve data locality while still providing parallelism [2], [7], [25], [26]. These methods include *communication avoiding* approaches [5] that optimize a series of loops over unstructured meshes. These strategies fall under the broader category of sparse tiling. In this paper we present a generalized sparse tiling algorithm, whereas previous work was specific to particular benchmarks.

Various code transformation have been developed to reschedule computation and reorder data for loop-chain-like code patterns. Many of these techniques also generate parallel execution schedules for the loops. The approach in [27] identifies *partitionable loops*, and schedules these loops for execution on a distributed memory machine. Likewise, there are approaches that take parallel loops identified by OpenMP pragmas and transform them for execution on distributed memory clusters [28].

The approach presented in this paper differs from these techniques in two key ways. First, these approaches generate a schedule in which each partition or processing element executes its assigned iterations of one loop, then communicates a subset of its results to other partitions that are dependent on that data. After executing its iterations of a loop, each processing element potentially waits to receive data from other partitions. The full sparse tiling approach described here does not require any synchronization or communication during the execution of a tile due to the *atomicity* of the tile. Before a tile begins execution, it waits until all necessary data is available and then executes from start to finish without further communication or synchronization. This approach can better exploit the locality available across the sequence of loops.

VII. Conclusions

Full sparse tiling has previously been shown to deliver significant performance gains when applied ad hoc to specific

applications. In this paper, we present a generalized algorithm for correctly sparse tiling any valid loop chain. This algorithm uses the newly developed loop chain abstraction as input, improves inter-loop data locality, and creates a task graph to expose shared-memory parallelism at runtime. For the sparse Jacobi benchmark, we showed that even though the unoptimized, generalized inspector has high overhead, the resulting executor has performance improvements. By adapting the sparse tiling inspector for unstructured mesh applications written using the domain-specific library OP2, we see performance improvements over even MPI on the Airfoil benchmark. These results add to the growing body of evidence that sparse tiling techniques enable communication avoidance and therefore improve parallel performance and scaling on multicore architectures. Future work includes (i) easing loop chain specification possibly through automatic detection, (ii) exploiting parallelism within the sparse tiles, (iii) optimizing the performance of the generalized full sparse tiling algorithm and investigating other ways to specialize it for each application domain, and (iv) automating the process of tuning parameters to full sparse tiling.

VIII. Acknowledgments

This project is supported by the Department of Energy CACHE Institute grant DE-SC04030, DOE grant DE-SC0003956, and NSF grant CCF 0746693. Support was also received from Engineering and Physical Sciences Research Council grants EP/I00677X/1 and EP/I006761/1, and from Rolls Royce and the Technology Strategy Board through the SILOET programme. Mike Giles and Istvan Reguly (Oxford University) are gratefully acknowledged for their contribution to the OP2 project.

References

[1] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balanji, P. C. Diniz, A. Koniges, and M. Snir, "Exascale programming challenges," in *Proc. Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA*. ASCR, DOE, Jul 2011.

[2] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß., "Cache Optimization for Structured and Unstructured Grid Multigrid," *Electronic Transaction on Numerical Analysis*, pp. 21–40, February 2000.

[3] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, "Sparse tiling for stationary iterative methods," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 95–114, February 2004.

[4] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," in *Proc. ACM SIGPLAN Conf. Prog. Lang. Des. & Impl. (PLDI)*. New York, NY, USA: ACM, June 2003.

[5] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. ACM, 2009, pp. 36:1–36:12.

[6] M. F. Adams and J. Demmel, "Parallel multigrid solver algorithms and implementations for 3D unstructured finite element problem," in *Proceedings of SC99*, Portland, Oregon, November 1999.

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

[8] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. Kelly, G. Mudalige, B. Van Straalen, and S. Williams, "Loop chaining: A programming abstraction for balancing locality and parallelism," in *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Boston, Massachusetts, USA, May 2013.

[9] G. Mudalige, M. Giles, J. Thiyagalingam, I. Reguly, C. Bertolli, P. Kelly, and A. Trefethen, "Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems," *Parallel Computing*, vol. 39, no. 11, pp. 669 – 692, 2013.

[10] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[11] L. Rauchwerger, "Run-time parallelization: Its time has come," *Parallel Computing*, vol. 24, no. 3–4, pp. 527–556, 1998.

[12] M. M. Strout, L. Carter, and J. Ferrante, "Managing tile size variance in serial sparse tiling," Poster presented at *Supercomputing*, 2001.

[13] D. Dutykh, R. Poncet, and F. Dias, "The VOLNA code for the numerical modeling of tsunami waves: Generation, propagation and inundation," *European Journal of Mechanics - B/Fluids*, vol. 30, no. 6, pp. 598 – 615, 2011, special Issue: Nearshore Hydrodynamics.

[14] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1:1 – 1:25, 2011.

[15] M. Giles, D. Ghate, and M. Duta, "Using automatic differentiation for adjoint cfd code development," 2005.

[16] G. Karypis and V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 5.0," http://www.cs.umn.edu/~metis, University of Minnesota, Minneapolis, MN, 2011.

[17] P. Cicotti and S. Baden, "Latency hiding and performance tuning with graph-based execution," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*, oct. 2011, pp. 28 –37.

[18] A. Chandramowlishwaran, K. Knobe, and R. W. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[19] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011.

[20] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the openmp tasking model to allow dependent tasks," in *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, ser. IWOMP'08. Springer-Verlag, 2008, pp. 111–122.

[21] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 123–132.

[22] W. Pugh and E. Rosser, "Iteration space slicing and its application to communication optimization," in *Proceedings of the 11th international conference on Supercomputing*. ACM Press, 1997, pp. 221–228.

[23] ——, "Iteration space slicing for locality," in *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, vol. LNCS 1863. London, UK: Springer-Verlag, August 1999, pp. 164–184.

[24] J. H. Salz, R. Mirchandaney, and K. Crowley, "Run-time parallelization and scheduling of loops," *IEEE Transactions on Computers*, vol. 40, no. 5, pp. 603–612, 1991.

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

[26] C. D. Krieger and M. M. Strout, "Executing optimized irregular applications using task graphs within existing parallel models," in *Proceedings of the Second Workshop on Irregular Applications: Architectures and Algorithms (IA^3) held in conjunction with SC12*, November 11, 2012.

[27] M. Ravishankar, J. Eisenlohr, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Code generation for parallel execution of a class of irregular loops on distributed memory systems," in *Proc. Intl. Conf. on High Perf. Comp., Net., Sto. & Anal.*, 2012, pp. 72:1–72:11.

[28] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proc. 11th ACM SIGPLAN Symp. Prin. & Prac. of Par. Prog.* New York, New York, USA: ACM, 2006, pp. 119–128.