Loop Chaining: A Programming Abstraction For Balancing Locality and Parallelism

Christopher D. Krieger
Michelle Mills Strout
Catherine Olschanowsky
Andrew Stone
Computer Science Department
Colorado State University
Fort Collins, CO 80526
{krieger,mstrout,cathie,stonea}@cs.colostate.edu

Stephen Guzik and Xinfeng Gao
Mechanical Engineering Department
Colorado State University
Fort Collins, CO 80526
{stephen.guzik, xinfeng.gao}@colostate.edu

Carlo Bertolli and Paul H.J. Kelly
Imperial College London
London, UK
{cbertolli, p.kelly}@imperial.ac.uk

Gihan Mudalige
Oxford e-Research Centre
University of Oxford, Oxford, UK
gihan.mudalige@oerc.ox.ac.uk

Brian Van Straalen and Sam Williams
Lawrence Berkeley National Lab
Berkeley, CA 94720
{BVStraalen, swwilliams}@lbl.gov

***Abstract*—There is a significant, established code base in the scientific computing community. Some of these codes have been parallelized already but are now encountering scalability issues due to poor data locality, inefficient data distributions, or load imbalance. In this work, we introduce a new abstraction called *loop chaining* in which a sequence of parallel and/or reduction loops that explicitly share data are grouped together into a chain. Once specified, a chain of loops can be viewed as a set of iterations under a partial ordering. This partial ordering is dictated by data dependencies that, as part of the abstraction, are exposed, thereby avoiding inter-procedural program analysis. Thus a loop chain is a partially ordered set of iterations that makes scheduling and determining data distributions across loops possible for a compiler and/or run-time system. The flexibility of being able to schedule across loops enables better management of the data locality and parallelism tradeoff. In this paper, we define the loop chaining concept and present three case studies using loop chains in scientific codes: the sparse matrix Jacobi benchmark, a domain-specific library, OP2, used in full applications with unstructured grids, and a domain-specific library, Chombo, used in full applications with structured grids. Preliminary results for the Jacobi benchmark show that a loop chain enabled optimization, full sparse tiling, results in a speedup of as much as 2.68x over a parallelized, blocked implementation on a multicore system with 40 cores.**

I. Introduction

The scientific computing community has an enormous investment in existing, mature codes. Any change to these codes must be accompanied by an extensive and costly test and validation effort. However, continuous changes are necessary in order to effectively utilize new computational resources, many of which require ever increasing levels of parallelism for efficient execution. Existing solutions for achieving parallelism include the use of OpenMP pragmas [1], reimplementation using MPI [2], reimplementation in task graph based programming models, emerging programming languages, and automated parallelization techniques. Due to shortcomings in each of these approaches we propose a new programming abstraction, *loop chaining*, as a solution.

One current approach is the incremental addition of parallelism using OpenMP parallel loop pragmas. In this approach, the compiler adds the necessary code for parallel execution and synchronization. This method enables the unobtrusive addition of parallelism. Unfortunately, this approach typically provides the programmer with little control over how data and computation are grouped or distributed. For example, OpenMP only allows statically or dynamically sized blocks of contiguous iterations to be assigned to threads. As is, it provides no way to group iterations together based on the data they access. It also does not provide a way to group iterations of different loops together. These limitations make it difficult to improve data locality during parallel execution.

An earlier and still quite popular approach to parallelization that is significantly less incremental is to use explicit message passing libraries such as MPI. MPI has been used to parallelize most existing, mature codes. MPI development is disruptive and requires a considerable coding effort. However, the difficult aspects of the MPI programming model also lead to advantages in terms of data locality. Specifically, the programmer must manage the distribution for both the data and the computation and specify communication between nodes with explicit sends and receives. The programmer can therefore put computation together that accesses similar data.

Both OpenMP and MPI programming models have led to codes with one parallel loop after another in sequence. Communication typically occurs between loops. However, in order to achieve the high levels of parallelism required to efficiently execute on large parallel machines without being hindered by communication, it is necessary to move from bulk parallelism to asynchronous parallelism [3], [4], [5],

[6], [7], [8], [9]. Many new programming models provide a task graph abstraction and suggest that programmers rewrite existing code with sequences of parallel loops in the form of task graphs instead. Task graphs and other ways of expressing asynchronous parallelism are good targets, but require that programmers do a significant amount of code rewriting.

Other approaches to managing the parallelism and data locality tradeoff include Partitioned Global Address Space (PGAS) languages such as Unified Parallel C [10] or Co-Array Fortran [11]. These languages ameliorate the complexity and maintenance situation by abstracting away much of the necessary communication code behind a one-sided communication model and by accessing remote data in a way that is syntactically similar to local accesses. However, the programmer is still required to make computation and data distribution decisions. These languages also provide no way to improve data locality across loops within a node.

Relatively new parallel programming languages such as Chapel [12] or X10 [13] provide mechanisms for specifying data locality such as places and locales, but these specifications are done on a loop by loop or computation by computation basis. The parallel constructs are considered in isolation with respect to other adjacent or surrounding constructs. There is no way to aggregate parallel loops. There is also a prohibitive cost in porting existing code to an entirely new language.

At the opposite end of the spectrum lie autoparallelization [14], [15] approaches. Under these schemes, a compiler or other tool determines the data and computation distribution automatically. Examples such as the Build To Order BLAS compiler do data locality and parallelization optimizations across function calls [16]. An autoparallelization approach can achieve good performance portably and frees the programmer from having to make fixed choices, but also means the programmer has little ability to alter or further optimize the distributions. More significantly, automatic approaches typically rely on extensive data dependence analysis. Precise data dependence analysis is an open research challenge, fraught with difficulties such as pointer aliasing and interprocedural analysis that historically have forced these algorithms to make overly conservative assumptions on real-world codes.

Given these extremes of user-laborious, non-portable parallelization approaches or limited automatic methods, it seems that the best answer lies between the two. Ideally, the computation scheduling and distribution and the data distribution would be handled automatically by a compiler and/or runtime that is aware of the target system and can optimize for it. More realistically, we want an approach that still enables the programmer to inject scheduling and distribution decisions in some orthogonal way. However, it is important for a system to not rely heavily on the automatic detection of data dependences especially across the many procedure calls that occur in modular scientific computing software.

For this reason, we propose a new programming abstraction, the *loop chain*. A loop chain is a sequence of parallel or reduction loops that share data along with a specification that exposes how loop iterations access data. A parallel loop is one in which each iteration of the loop is independent of the others, therefore having no loop carried dependencies or only reduction dependencies. A loop chain is a kind of component composition graph as proposed in the Themis project [17], where each loop is a component, the loops are composed in a sequence, and the access functions for each loop are the component dependence summaries.

Requiring the programmer to express the data access functions circumvents the need for data dependence analysis. Given these access functions, a compiler and/or runtime system can determine the partial ordering on the iterations and then group iterations, possibly across loops, into tasks in a task graph. This scheduling can be done in such a way as to improve data locality while maintaining adequate parallelism.

In this paper, we make the following contributions:

- Define the loop chaining abstraction.
- Provide an example of the partially ordered set of iterations that would be exposed to the compiler and/or runtime system due to loop chaining.
- Present possible mechanisms for realizing the loop chaining abstraction using libraries and pragmas.
- Show preliminary results for a sparse Jacobi benchmark that exhibits the performance improvements made possible by scheduling across loops.
- Provide two case studies that describe how loop chaining could be incrementally provided in existing domain-specific libraries and would enable optimizations that currently are not feasible.
- Review a number of existing performance optimizations that could leverage the loop chaining abstraction.

Section II defines the loop chaining abstraction and also includes details of possible optimizations facilitated by loop chains and possible mechanisms for expressing loop chains. We then present three cases studies that use loop chains in scientific codes. In Section III, we evaluate the performance gains seen by applying one loop chain enabled optimization, full sparse tiling, to a sparse Jacobi solver. In Section IV, we show that the existing function interfaces plus a loop grouping construct enables the representation of loop chains in the context of the OP2 unstructured mesh library. In Section V, we propose to use domain-specific source-to-source translation to detect and use loop chains in the Chombo Fortran adaptive mesh refinement library for solving partial differential equations on structured meshes. Section VI discusses prior work related to loop chains and Section VII concludes the paper.

II. The Loop Chain Programming Abstraction

This section defines the loop chaining program abstraction and its requirements. We also show an example of a partially ordered iteration set that is the goal of the loop chaining construct. Using this example, we illustrate possible task graph schedules for the partially ordered iteration set and indicate how individual schedules have been used successfully by others in the past. The section concludes with some possible mechanisms for realizing the loop chaining abstraction.

A. Definition of a Loop Chain

A loop chain associates together a *sequence of loops*, $L_0, L_1, ..., L_{N-1}$, where N is the number of loops. Consecutive loops in the sequence should have no intervening code. Each L_i is a domain of iterations. One could define the domain as a contiguous range, a polyhedral set, or a collection of items. The loop chain abstraction does not require that a particular definition of the domain of iterations be used, just that there is some definition of the domain.

Each *data space* being accessed by the loops should also have a well-defined domain, $D_0, D_1, ..., D_m$. For the data spaces / arrays, the numbering does not necessarily indicate a sequence.

Each loop L_i in the sequence must incorporate some mechanism for exposing *access functions* $R_{ij} : L_i \rightarrow D_j$ or $W_{ij} : L_i \rightarrow D_j$, where R access functions indicate which data location in data space D_j that an iteration $k \in L_i$ reads and W access functions indicate writes. Therefore, it should be possible to determine a function that maps each iteration point to a constant number of memory locations, without the need to do complex program analysis. The function could be uninterpreted at compile-time but should be known at runtime before the execution of the loop (i.e. at inspector time). These access functions provide a mechanism for determining which iterations of one loop in the sequence must be computed before the next loop in the chain; the access functions provide a way to compute dependencies between loops.

Finally, each loop must be either fully parallel and/or a reduction. This needs to be the case, as otherwise it would be necessary to analyze the loop to determine if loop-carried dependences were due to reductions or not. If each loop is fully parallel and/or a reduction, then any scheduling of the iterations can permute the iterations in any loop.

B. Other Uses of the Term Loop Chain

Loop chaining is conceptually related to *vector chaining*. Vector chaining [18][19] is an optimization used in early vector units to enable the overlapping of vector operations. Rather than waiting for the entire vector output of one operation to be available before starting the following dependent operation, instead, as soon as each element of a vector becomes available, it is sent immediately to the subsequent operation. Our use of the term is similar, especially if one views a vector as a limited form of loop. In loop chaining, once an iteration of a loop completes, dependent iterations of subsequent loops are enabled to execute, rather than waiting for the entire preceding loop to complete first.

The term *loop chaining* was also used by the Polaris compiler project at the University of Illinois Urbana-Champaign. In [20], they use the term *loop chaining* to refer to a schedule between loops where one process would execute part of the first loop in a sequence and then post the data for another process that would use that data. Their usage is limited to a compiler loop optimization and is not exposed to the programmer. It can actually be thought of as a scheduling approach made possible by the more general loop chaining abstraction we propose here.

```
1  for (int t=0; t < numIters; t += 2) {
2    for (int i=0; i<numrows; i++) {
3      double diag = 1.0;
4      Uprime[i] = 0.0;
5      for (int p=IA[i]; p < IA[i+1]; p++) {
6        int j = JA[p];
7        if (j==i) { diag = A[p]; }
8        else { Uprime[i] += A[p] * U[j]; }
9      }
10     Uprime[i] = (F[i] - Uprime[i]) / diag;
11   }
12
13   for (int i=0; i<numrows; i++) {
14     double diag = 1.0;
15     U[i] = 0.0;
16     for (int p=IA[i]; p < IA[i+1]; p++) {
17       int j = JA[p];
18       if (j==i) { diag = A[p]; }
19       else { U[i] += A[p] * Uprime[j]; }
20     }
21     U[i] = (F[i] - U[i]) / diag;
22   }
23 }
```

Fig. 1: The kernel of the sparse Jacobi solver for CSR sparse matrices. Output vectors U and Uprime are used in an alternating, ping-pong, fashion. The loops beginning on lines 2 and 13 can be chained.

C. Example of a Conceptual Loop Chain and Resulting POSet

Given a loop chain specification, a compiler and/or runtime system can represent it internally as a chained iteration space. We demonstrate this process using a Jacobi solver as an example. A Jacobi solver is an iterative algorithm for determining the solution to a system of linear equations. Given a sparse matrix A, and two vectors $\vec{u}$ and $\vec{f}$, related by $A\vec{u} = \vec{f}$, one iteration of the sparse Jacobi method produces an approximation to the unknown vector $\vec{u}$. This method is known to converge if A is strictly diagonally dominant. The recurrence equation that describes Jacobi is as follows: $u_i^{(k)} = \frac{1}{A_{ii}} \left(f_i - \sum_{j \neq i} A_{ij} * u_j^{(k-1)} \right)$. Note that part of Jacobi is a sparse matrix vector product, which is a common performance bottleneck in many codes.

Figure 1 shows the C code for a sparse Jacobi solver. It uses a ping-pong approach to the output vector, switching between `U` and `Uprime`. The outermost loop repeats the kernel for a fixed number of repetitions. If the two loops shown on lines 2 and 13 are combined into a loop chain, they form the chained iteration space shown in Figure 2. The boxes represent data spaces/arrays $D_0 = U$ and $D_1 = Uprime$, while the circles represent iterations of the loops beginning on lines 2 and 13. The iteration spaces shown for the two loops are $L_0 = \{1, 2, \ldots, 7\}$ and $L_1 = \{1, 2, \ldots, 7\}$, which would be the case for a sparse matrix with 7 rows. Arrows represent the data access relations, with upward pointing arrows indicating the data reads R_{00} and R_{11} and downward pointing arrows indicating the identity relation for data writes W_{01} and W_{10}.

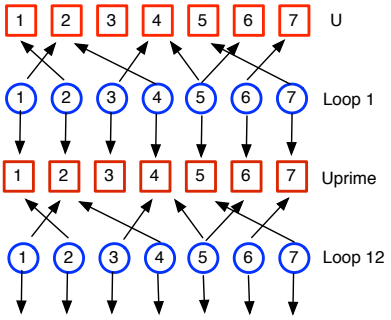


Fig. 2: The data and iteration space view of the Jacobi code in Figure 1. The boxes represent the U and Uprime vectors, the circles represent iterations of the two loops. The data access functions are represented by arrows. The upward pointing read arrows are irregular and dictated by the particular sparse matrix, while the downward pointing write accesses are an identity relation. Chain invariant data, such as F, IA, JA, and A, are present in the loop chain data structures but are omitted from this diagram.

```
1  #pragma chain \\
2  domain( range(0,numrows-1) ),
3  access( read, U, range(IA[i]<=p<IA[i+1], JA[p]))
4  access(write, Uprime, single(i))
5  for (int i=0; i<numrows; i++) {
6    double diag = 1.0;
7    Uprime[i] = 0.0;
8    for (int p=IA[i]; p < IA[i+1]; p++) {
9      int j = JA[p];
10     if (j==i) { diag = A[p]; }
11     else { Uprime[i] += A[p] * U[j]; }
12   }
13   Uprime[i] = (F[i] - Uprime[i]) / diag;
14 }
15   ...
```

Fig. 3: The kernel of the sparse Jacobi solver loop chained using hypothetical OpenMP style pragmas.

D. Possible Optimizations and Schedules for Example

Once we have an internal representation of the loop chains, there are numerous optimizations to be considered. Each of these optimizations largely consists of grouping together elements in the chained iteration space into tasks and then scheduling between and within tasks. Some simple examples serve to illustrate the tradeoffs between parallelism and data locality. For example, it can be seen from Figure 2 that if the left two columns, 1 and 2, are grouped together, they can be executed without any external synchronization or communication. These columns have no dependences on data from outside the group. If columns 3 and 4 are grouped together, they depend only on column 2 outside their group. Columns 5, 6 and 7 likewise depend only on column 4. This schedule ensure a high amount of locality and data reuse between the grouped iterations of loop 2 and loop 13 and also ensures little communication between work groups. Furthermore, it represents a well balanced amount of work per grouping. Unfortunately, under this schedule, the three groups are serialized, allowing only internal parallelization and disallowing inter-group parallel execution. Depending on the targeted hardware, this may or may not be acceptable.

Another possible schedule is to execute all iterations of loop 2 in parallel, followed by a barrier, then followed by all iterations of loop 13 executed in parallel. This is the direct parallelization provided by a simple OpenMP `parallel for` pragma. While this schedule allows for all iterations in each loop to execute in parallel, it incurs the cost of a barrier and has poor data locality between loops. On some hardware, e.g. an 8 core system with a shared last level cache, this schedule might be efficient if the working set size is small enough to fit in cache.

These trivial examples serve to illustrate the tradeoffs between parallelism and locality that are exposed to the compiler or runtime system by loop chaining. It then is incumbent upon this optimizer, with knowledge of the target hardware, to select the optimal iteration scheduling and data distribution for the loop chain to minimize runtime.

E. Mechanisms for Specifying Loop Chains

Any mechanism that implements loop chaining will need some way to specify the sequence of loops that are part of the chain. For each loop, four elements need to be specified: the loop body, the set of iterations in the loop, whether the loop is a reduction, and most importantly the data access functions that specify how each iteration accesses data.

First, the code within the loop body must be specified in some way. This can be through a function name or function pointer. The loop body may also be delimited directly in source code through the use of a keyword or pragma surrounding the code.

Second, the iteration space of the loop needs to be declared. For simple loops, lower and upper bounds on the loop iteration variables suffice. Commonly, a loop will iterate over every element in a set or container. For example, a loop may traverse every cell in a mesh or every particle in a molecular dynamics simulation. Threading Building Blocks [21], the OP2 unstructured mesh library discussed in Section IV, and the Chombo library discussed in Section V, all support this type of iteration. For these cases, specification of the set or container may adequately define the iteration space.

Third, the loop must be identified as either fully parallel or a reduction. This ensures that each iteration of the loop can be executed independently, or, in the case of a reduction, in any order.

The final pieces of information needed are the data access functions. These expressions provide a mapping from a loop iteration to each data item that iteration reads or writes. For regular access patterns, this expression can be a simple affine expression. In the more general case or for irregular accesses, an indirection array containing an explicit mapping from iteration to array element is required. It should be noted that these mappings are already needed by the code itself and do not represent additional work or storage.

There are a variety of ways that loop chain information can

be expressed. Pragmas or language extensions are possibilities and can be implemented using a source to source compiler. Figure 3 shows an example of once such hypothetical `chain` pragma, similar to OpenMP's pragmas, used to declare a chainable loop. The iteration space is inferred from the loop bounds and the loop body is identified by the placement of the pragma. The `access` argument to the `chain` pragma defines an access function and requires arguments specifying the read/write direction, the array name, and a specification of whether a single or multiple elements are accessed per iteration of the loop. If multiple elements are accessed per iteration, the range is specified. The access function itself is given next, `JA[p]`. Note on line 3 that the array U is accessed indirectly via the JA indirection array. The second loop in the loop chain, though not shown, is specified with a similar pragma.

There are several other approaches for specifying loop chains. In Section IV, we present a domain-specific library approach, OP2. In Section V, we discuss detecting loop chains using a domain specific compiler for Chombo.

III. Case Study: Jacobi Benchmark

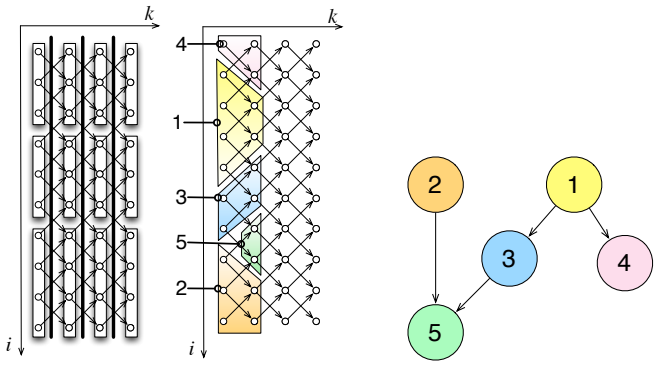
One optimization enabled by the loop chaining concept is full sparse tiling. In this section we briefly describe the full sparse tiling optimization and then show how the scheduling across loops that loop chaining enables can result in significantly better parallel scaling for the Jacobi benchmark.

A. Full Sparse Tiling Overview

One optimization technique for irregular codes is the *inspector/executor* (I/E) [22] approach. Under an I/E scheme, code is added to an application to *inspect* the data set at runtime. This is needed to determine data dependencies in the case of sparse or irregular access patterns. Based on these patterns, a new schedule for execution is created and passed to an *executor* that runs the code according to the schedule generated by the inspector. *Full sparse tiling* (FST) [23], [24], [25] is an I/E optimization that improves temporal and spatial locality by placing the execution of loop iterations that access the same data, even across different original loops, together into a scheduling entity called a *sparse tile*. The tiles together perform the same computation iterations as the original code, just in an optimized order. The new tile-based execution schedule must still satisfy all of the original data dependences.

The full sparse tiling approach can be illustrated on a sparse Jacobi solver algorithm. One implementation of a Jacobi solver was previously discussed in Section II and shown in Figure 1. There, the main iteration loop of the code implementing the solver is unrolled by a factor of two. The two unrolled loops, beginning at lines 2 and 13, can be chained together.

The left side of Figure 4a shows a typical blocked traversal through the iteration space of the two loops. The data dependency arrows in the figure result from a tridiagonal sparse matrix, a band matrix with a bandwidth of three. All elements of u^k are computed in parallel, followed by a barrier. Next, all elements of u^{k+1} are computed. This pattern is repeated for



(a) At left, the Jacobi iteration space for a tri-diagonal sparse matrix with a simple blocked schedule. On the right, a full sparse tiling of the computation.

(b) Task graph for the full sparse tiled Jacobi kernel. Each node represents a tile of computations while edges represent the partial ordering due to data dependencies.

Fig. 4: Full sparse tiling of the Jacobi solver.

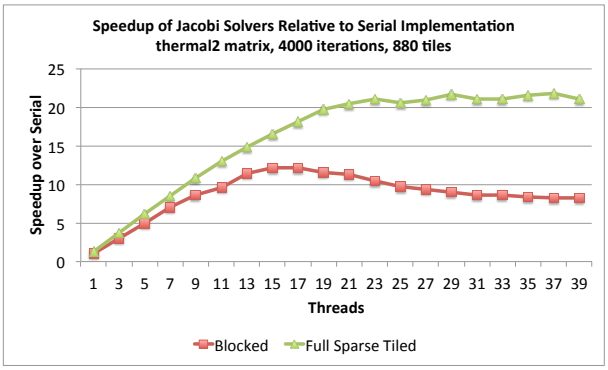


Fig. 5: Speedup of a row-blocked Jacobi solver implementation and a full sparse tiled implementation compared with a serial version of the same algorithm. The FST version is consistently faster than the blocked version and scales better on a 40 core system.

every two convergence iterations. The right side of Figure 4a shows a full sparse tiling of the same iteration space. Rather than computing all elements of u^k, then all elements of u^{k+1}, elements of the computation of both passes are combined into tiles. The data flow arrows seen in Figure 4a indicate that some tiles rely on data from other tiles. Since not all data dependencies can be satisfied within a tile, the inter-tile dependencies create a partial ordering on tile execution. This ordering is captured in a task graph. The task graph for the Jacobi example is shown in Figure 4b.

B. Methodology

In order to evaluate the performance benefits of full sparse tiling, we wrote two parallel versions of the Jacobi solver. One used a straightforward row-blocking technique and OpenMP `parallel for` pragmas for parallelization. The other used a full sparse tiled schedule and a task graph execution engine built on the OpenMP 3.0 task model. These implementations correspond to the two schedules shown in Figure 4a. A set of five sparse matrices, drawn from the University of

Florida Sparse Matrix Collection [26], was used to evaluate the solver. Our experiments were run on a 40 core machine with four 10 core Intel Xeon E7-4860 processors and 256 GB of system memory made available for our research by Intel's Manycore Testing Lab. The Intel icpc compiler, version 12.1.0 (20110811), was used to compile the code.

C. Experimental Results

The relative performance of the solvers varied slightly across the different matrices. We present the results for only one representative sparse matrix, *thermal2*. This matrix is used in finite element method thermodynamics problems. It has 1228045 rows and columns and 8580313 nonzero entries, with a memory footprint of 130 MB.

Figure 5 shows the speedup of both implementations relative to a serial Jacobi solver. Even at one thread, the FST version is 1.27x faster than the serial version. It shows excellent scalability out to 20 cores. The blocked solver also scales well to 15 cores, but then begins to lose performance as more cores are added. The largest difference in performance is 2.68x.

IV. Case Study: OP2 Unstructured Mesh Library

OP2 [27], [28] is a library for C/C++, FORTRAN, and Python for the development of unstructured mesh applications. It includes functions to *declare* an unstructured mesh and a parallel loop call to *compute* over it. The declaration of a mesh is done by specifying its sets (e.g. vertices, cells, edges), their mappings, for instance expressing for each edge its incident vertices, and the data arrays associated with each of the sets.

Figure 6 shows an example of mesh declaration and a parallel loop. In the main program, lines 15-17, we first declare the mesh by creating five variables whose type is one of the three OP2 structured data types: op_set, op_map, and op_dat. These are then declared in the following way:

- The edge and vertex sets are declared through the op_decl_set call (see lines 21 and 22).
- The mapping between edges and vertices is declared through the op_decl_map call (line 23). The first two parameters of the `op_decl_map` call are the *from* and *to* sets of the mapping. The third parameter indicates that there are two vertices for each edge.
- The data array structures associate with vertices and edges, respectively. This is done through the op_decl_dat call that uses the input information to fill a structured data variable of type op_dat (lines 24 and 25). For instance, when declaring the *edgeData* op_dat, we are declaring 6 elements (second parameter) for each edge (first parameter).

Using this data structures, the user can implement the parallel loop call shown in Figure 6 (lines 28-31). In general terms, a parallel loop iterates over one of the mesh sets (also called iteration sets) and can access data arrays associated with the iteration set itself. It can also access data arrays associated with mesh sets different from the iteration set by using mapping information. In this case the parallel loop

```
1  subroutine incrementKernel (a1, a2, c)
2    real(8), dimension(6) :: a1, a2
3    real(8), dimension(6) :: c
4
5    integer(4) :: i
6
7    do i = 1, 6
8      a1(i) = a1(i) + c(i)
9      a2(i) = a2(i) + c(i)
10   end do
11
12 end subroutine
13
14 program meshProgram
15   type(op_set) :: vertices, edges
16   type(op_map) :: edges2Vertices
17   type(op_dat) :: vertexData, edgeData
18
19   ! read in the mesh information and arrays
20   ! and declare the OP2 data structures
21   call op_decl_set (vertices, meshFile, "vertices")
22   call op_decl_set (edges, meshFile, "edges")
23   call op_decl_map (edges, vertices, 2, edges2Vertices,
       meshFile, "edges2Vertices")
24   call op_decl_dat (vertices, 6, vertexData, meshFile,
     "vertexData")
25   call op_decl_dat (edges, 6, edgeData, meshFile, "
     edgeData")
26
27   ! loop over edges
28   call op_par_loop (edges, incrementKernel, &
29    op_arg_dat (vertexData, 1, edges2vertices, OP_INC),
       &
30    op_arg_dat (vertexData, 2, edges2vertices, OP_INC),
       &
31    op_arg_dat (edgeData, -1, OP_ID, OP_READ))
```

Fig. 6: Example of OP2 loops and user kernel. The user kernel is applied to each edge and increments its two vertices. The parallel loop call specifies how the mesh is accessed through mapping (i.e. edges2vertices) providing OP2 mapping arrays to be used to bind appropriate actual parameters to the formal parameters of each kernel invocation.

makes an *indirect access* and a mapping is effectively used as an array of pointers into the target data array.

In the example, the parallel loop iterates over edges (first parameter), i.e. edges are the iteration set for this loop. The kernel function (second parameter, *incrementKernel*) is applied once for each element in the edges set. The kernel increments the data associated with the two incident vertices of each edge, thus indicating a reduction. Notice that the kernel implementation does not use any indirection *explicitly*, and it is programmed with a single edge in mind. It is the job of the OP2 implementation to pass the correct arguments to each kernel invocation.

The actual data accesses are *declared* through the op_arg_dat calls passed as arguments to the op_par_loop (see lines 29-31 in Figure 6). For each op_arg_dat the user specifies the op_dat to be accessed (vertexData or edgeData) as first parameter. If the op_dat was declared as associated to the parallel loop iteration set, then we do not need an indirect access (e.g. line 31). In this case each edge will access its own data variable. This is expressed by the OP_ID (or identity mapping) parameter, which requires the second parameter of the op_arg_dat to be -1. The remaining two arguments of the parallel loop (lines 29 and 30) access vertex data (vertexData,

first parameter). The user declares this access using a mapping between edges (the iteration set) and vertices. This is expressed as the third parameter in the two op_arg_dat calls. The second parameters select one of the two vertices of each edge. For each op_arg_dat invocation the user has also to express the modality with which the kernel will access the op_dats. In the example, we read the edgeData array (OP_READ) and we increment the vertexData array (OP_INC). The kernel implementation must reflect this information.

Notice that in OP2 the information on how to access vertex data through an edge identifier is *declared* in the parallel loop call by the user's specifying the mapping to be used. We call an op_arg_dat, specifying this information, an *access descriptor*. Thanks to the use of access descriptors, the OP2 compiler does not need to analyze the user kernel functions to synthesize implementation code for parallel loops. The access descriptors contain all the necessary information for code generation.

By analyzing access descriptors of parallel loops, the compiler and/or run-time support are able to combine sequences (or *chains*) of parallel loops and deduce dependency information. Figure 7 shows an example of an OP2 loop chain. In this example, the compiler can easily infer that the *temp* data array is read by the second parallel loop after being written by the first one, i.e. it can easily infer a read-after-write dependency looking at the op_arg_dat declarations. This dependency information can be used in many ways, including compile-time loop fusion (see [29]), or other run-time techniques such as communication-avoiding algorithms [30], or full sparse tiling. All these techniques improve data locality across successive loops.

As OP2 is a library provided for different *host* languages (e.g. Fortran), its calls are interspersed with host statements. We can thus expect significant complexity in user code analysis, for instance due to the presence of conditional blocks, procedure calls, etc. A first attempt at reducing the OP2 compiler complexity is to introduce a *super-loop* construct that the user can insert in the code to signal the OP2 compiler that optimizations across multiple loops can be safely applied, i.e. without checking the host code. For instance, the sequence of loops in Figure 7 includes parallel loops accessing the *temp* dataset associated with vertices, the *x* dataset associated with edges, and the *res* dataset associated with cells. Here, we omit the declaration of mappings, sets and datasets for brevity - the access descriptors expressed by the op_arg_dat calls contain the needed information for dependency analysis.

However, to use super-loop calls in legacy or industrial applications might require, in more complex cases, an additional effort from the user in terms of procedure inlining, conditional block movement, and code replication. For instance, in a typical case, a loop is executed only if a conditional value is set to true. To use the super-loop construct, the user has to transform the code, possibly resulting in duplications that affect the general readability of the application structure. Our future plans are to relieve the user from explicitly calling the super-loop statement by appropriately coordinating compiler and run-time mechanisms to automatically infer the start and end of a super-loop.

```
begin_superloop ()
  ! loop over edges
  call op_par_loop (edges, kernel1, &
   op_arg_dat (temp, 1, edges2vertice, OP_INC), &
   op_arg_dat (temp, 2, edges2vertice, OP_INC), &
   op_arg_dat (x, -1, OP_ID, OP_READ))

  !loop over cells
  call op_par_loop (cells, kernel2, &
   op_arg_dat (temp, 1, cells2vertices, OP_READ), &
   op_arg_dat (temp, 2, cells2vertices, OP_READ), &
   op_arg_dat (temp, 3, cells2vertices, OP_READ), &
   op_arg_dat (res, -1, OP_ID, OP_WRITE))


  !loop over vertices
  call op_par_loop (vertices, kernel3, &
   op_arg_dat (temp, -1, OP_ID, OP_READ))
   op_arg_dat (q, -1, OP_ID, OP_WRITE))
end_superloop ()
```

Fig. 7: Example of loops that can be encapsulated in a super-loop call to enable the compiler to apply optimizations.

V. Case Study: Chombo AMR Library

Another application framework to which loop chaining will be applied is the Chombo parallel adaptive mesh refinement library, which is designed for parallel solutions of partial differential equations on structured grids. This type of scientific software consists of hundreds of thousands to millions of lines of software and was developed over several decades by numerous developers. Similar to the motivation in Kelly et al. [17], the Chombo library has a modular design to enable the sustainability and reusability of existing software [31], [32], [33], [34], [35].

There are two types of common usage patterns for Chombo: (1) Standard applications that arise from compressible and incompressible fluid dynamics are developed and maintained as part of the top layer of the library. These applications see incremental and ongoing modifications to improve the precision of solutions and cope with modern computer architectures. (2) New applications are implemented from scratch to enable specific scientific discoveries and/or engineering practices.

In Chombo-based code, small mathematical functions are written as modular pieces of code and combined to produce larger algorithms. Figure 8 illustrates the dominant style of programming in Chombo, which is looping over a level of boxes[1] and applying a single kernel/function, with the kernel often being inside a `SpaceDim` loop that repeats for each physical dimension. This strategy both accelerates development by enabling rapid prototyping of algorithms and also eases maintenance since components can often be separately modified and unit tested in isolation.

With a large user community already actively developing with Chombo, incremental changes are preferred in both the library and applications. Yet at times, performance issues arise that cannot be addressed with the languages and

[1]Each box is a two or three-dimensional, rectangular domain. The levels are due to adaptive mesh refinement.

```
1  for (int idir = 0; idir < SpaceDim; idir++) {
2      FORT_CELLDIVINCRVTOP(CHF_FRA1(a_div, 0),
3                           CHF_CONST_FRA(phi),
4                           CHF_CONST_REAL(m_dx),
5                           CHF_CONST_INT(idir),
6                           CHF_CONST_BOX(grid));
7  }
```

Fig. 8: Example loop chain in Chombo Fortran. In this example, `a_div` and `phi` are structured arrays and `grid` gives the sizes of the arrays. Hidden in the kernels are nested loops over the indices specified by `grid`. Each kernel call computes a difference in a specific spatial direction, and accumulates the result in `a_div`. A problem is that `phi` is loaded a number of times equal to the number of spatial dimensions, but may not be in cache upon execution of subsequent kernel calls.

programming models in current use. Fortunately, Chombo already uses a version of source-to-source translation to write dimension-independent Fortran code, called *ChomboFortran*. ChomboFortran is a domain-specific language for Chombo developers. This provides a means of identifying high-level data abstractions in the subroutine calling interfaces.

For the example in Figure 8, a domain-specific source-to-source translator based on the ROSE compiler infrastructure [36] [37] can detect the loop chain structure in common loops such as this, unroll the loop since the spatial dimension is a runtime constant, and then schedule across instances of the loop. In Figure 8, `a_div` is an accumulation variable (the cell-centered divergence) and `SpaceDim` is known at compile/transform time. The rest of the arguments are read-only, as indicated by the user with the `CONST` part of the ChomboFortran macro. The stencil width and therefore the access functions can be determined by parsing the Chombo macros `CHF_AUTOMULTIDO` and `CHF_IX` in the kernel code (not shown in Figure 8).

Having the domain-specific compiler recognize the loop chain instances in existing Chombo programs will enable the compile time creation of tasks that span loops. This is important to improve the *arithmetic intensity* of the program. Arithmetic intensity is calculated as the ratio of floating-point operations to total data movement nominally to/from DRAM (see the Roofline model [38], [39] for more details). Often, scientific applications may loop over N-dimensional data structures k times (multiple operators or iterative methods).

Figure 8 shows one such example where the `a_div` array is unnecessarily touched `SpaceDim` times. Nominally, one can calculate an $O(N)$ lower bound to data movement and thus an $O(k)$ upper bound on arithmetic intensity. However, finite cache capacities can result in $O(kN)$ data movement. This has the negative effect of reducing the actual arithmetic intensity to $O(1)$.

Unfortunately, the realities of existing compilers often make manual optimization (e.g. inter-procedural, loop fusion, blocking, tiling, unroll-and-jam) a prerequisite for improving arithmetic intensity. Moreover, synchronization points (reductions or MPI communication) endemic to high-performance distributed numerical methods demand one restructure algorithms in order to reduce data movement and improve arithmetic intensity. These include communication-avoiding algorithms that perform redundant work in order to minimize data movement from DRAM. The recognition of the loop chaining abstraction will enable us to avoid manual optimization, while still not requiring complex program analysis.

VI. Related Work

The loop chaining abstraction complements the application of previously developed automated task transformation strategies. The abstraction promises to enable the same level of performance as manual approaches, while remaining portable to new architectures and not requiring users to define tasks. This section describes work previously done to expose asynchronous parallelism by scheduling manually defined tasks, scheduling automatically discovered tasks, and by rewriting existing algorithms to avoid communication.

A. Programming Models to Expose Asynchronous Parallelism

The following describes several projects that aim to expose the available parallelism in code through the application of new programming models that involve the programmer defining tasks.

The Tarragon system [8] provides an API for programmers to create a task graph with edges labeled with what data needs to be communicated from one task to the next. Distributed memory parallelism is handled by having any data on an edge be sent as a message. The run-time system schedules the task graph so that there is computation and communication overlap. This approach requires the programmer to decide what computations will be aggregated into tasks.

In the SuperMatrix work [4], the programming model is to specify matrix computations as submatrix computations. The run-time system can determine how many times subdivision occurs and then can create a task graph at run-time to expose asynchronous parallelism. This solution removes the burden of selecting task sizes from the user, but is more restricted than the loop chaining programming abstraction we are proposing that will work more generally for sequences of data parallel loops.

Other programming models that enable the user to specify asynchronous parallelism through either explicit or implicit task graphs are the Concurrent Collections (CnC) programming model [7], the StarPU [9] run-time system, and the StarSs [3] framework. Each of these systems use different mechanisms for letting the user specify tasks, but in the end the user needs to specify the tasks in a way that improves data locality. The loop chaining programming construct can complement any and all of these approaches by providing the needed information for creating tasks to the compiler and/or runtime system and then using the appropriate task graph based system as a backend.

B. Automatic Approaches for Task Detection

An alternative to providing a new programming model is to automatically examine application code and create task graphs.

Iteration space slicing [40], [41] is an optimization that would be applicable in loop chains where the access functions are affine. Overlapping iteration space slices would have duplicate computation, but plenty of parallelism. The non-overlapping iteration space slices would exhibit good temporal data locality but no parallelism in cases where the data accesses are not pointwise.

A method of partitioning loops with irregular accesses for parallel execution on distributed systems was presented in [42]. This method takes advantage of OpenMP directives and combines both compile-time and runtime techniques to translate OpenMP code to MPI. Data accesses and computations are reordered in order to increase the overlap of communication and computation. This work could leverage the loop chaining abstraction to reduce the program analysis necessary.

The code generation technique presented in [43] examines the data affinity among loops and partitions the execution with the goal of minimizing communication between processes, while maintaining load balancing. This technique is also aimed at irregular applications executing in a distributed memory environment, and it could leverage the loop chaining abstraction.

C. Communication Avoidance

A number of techniques for scheduling across loops to avoid communication have been prototyped by hand or with specialized code generators.

In structured codes, using multiple layers of halo, or ghost, cells is a common optimization [44], but when done by hand breaks the typical software modularity in large applications. Structured codes can also use overlapped tiling techniques that reduce communication at the expense of performing redundant computation [45]. These techniques work by dividing a loop nest's iteration space into a number of overlapping regions called tiles. These tiles are shaped such that each of them can execute in parallel without requiring communication. Several researchers have examined using overlapped communication techniques within single loop nest (often stencil) computations [46], [47], [48]. We believe that an overlapped tiling optimizer could use information from the loop chaining abstraction to do overlapped tiling across loops.

For unstructured codes, there has been various inspector/executor strategies [49] that reschedule across loops to improve data locality while still providing parallelism [50], [24], [51], [25]. The term communication avoidance was coined by Demmel et al. [51] to refer to such schedules, some of which have processors perform some amount of overlapped computation to avoid communication. All of these techniques that schedule across loops could be more easily automated given the loop programming abstraction where the loops to schedule across are identified and the memory access functions are provided.

VII. Conclusion

Loop chains enable cross-loop performance optimizations that balance parallelism and data locality. The loop chaining programming abstraction can be thought of as a dependency analysis pattern as discussed in the Parallel Patterns book [52]. The loop chain construct has the programmer provide enough ordering information and data sharing information to the compiler and the run-time system that the compiler/run-time system can then automate the process of applying various task grouping patterns.

We provide case studies where existing, production codes exhibit loop chains and those loop chains can be exposed through the use of pragmas, library functions, or domain-specific compilation. Moving forward, additional work is needed to develop algorithms for selecting, parameterizing, and applying appropriate transformations to loop chains. Additionally, we have work underway to apply loop chaining enabled optimizations to significant, full sized scientific simulations to better understand loop chain performance tradeoffs and opportunities.

Acknowledgements

The authors gratefully acknowledge the use of Intel's Manycore Testing Lab. 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.

References

[1] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[2] M. Forum, "MPI : A message - passing interface standard," *The International Journal of Supercomputing and High Performance Computing*, vol. 8, no. 3-4, pp. 159–416, 1994.

[3] 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. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 111–122.

[4] 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.

[5] D. Andrade, B. B. Fraguela, J. Brodman, and D. Padua, "Task-parallel versus data-parallel library-based programming in multicore systems," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 101–110.

[6] M. Huang, V. K. Narayana, H. Simmler, O. Serres, and T. El-Ghazawi, "Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 3, no. 4, pp. 20:1–20:25, Nov. 2010.

[7] 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.

[8] 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.

[9] 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. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631

[10] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*. New York: John Wiley & Sons Inc., 2005.

[11] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[12] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 519–538.

[14] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[15] P. Feautrier, "Automatic parallelization in the polytope model," in *The Data Parallel Programming Model*, 1996, pp. 79–103.

[16] G. Belter, E. Jessup, I. Karlin, and J. G. Siek, "Automating the generation of composed linear algebra kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. New York, NY, USA: ACM, 2009, pp. 1–12.

[17] P. H. J. Kelly, O. Beckmann, T. Field, and S. B. Baden, "Themis: Component dependence metadata in adaptive parallel applications," *Parallel Processing Letters*, vol. 11, no. 4, pp. 455–470, 2001.

[18] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[19] H. Zima and B. Chapman, *Supercompilers for parallel and vector computers*. New York, NY, USA: ACM, 1991.

[20] B. Pottenger. Parallelism in loops containing recurrences.

[21] *Intel Threading Building Blocks Reference Manual*, Intel Corporation, 2009.

[22] 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.

[23] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck, "Sparse tiling for stationary iterative methods," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 95–113, 2004.

[24] 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)*. Berlin / Heidelberg: Springer, July 2002.

[25] 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.

[26] 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.

[27] C. Bertolli, A. Betts, G. Mudalige, M. Giles, and P. Kelly, "Design and performance of the op2 library for unstructured mesh applications," in *Euro-Par 2011: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7155, pp. 191–200.

[28] C. Bertolli, A. Betts, N. Loriant, G. Mudalige, D. Radford, D. Ham, M. B. Giles, and P. Kelly, "Compiler optimizations for industrial unstructured mesh cfd applications on gpus," in *Accepted for publication at Languages and Compilers for Parallel Computing Workshop*, 2012.

[29] C. Bertolli, A. Betts, P. H. Kelly, G. R. Mudalige, and M. B. Giles, "Mesh independent loop fusion for unstructured mesh applications," in *Proceedings of the 9th conference on Computing Frontiers*, ser. CF '12. New York, NY, USA: ACM, 2012, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/2212908.2212917

[30] M. B. Giles, G. R. Mudalige, C. Bertolli, P. H. J. Kelly, E. Laszlo, and I. Reguly, "An analytical study of loop tiling for a large-scale unstructured mesh application," in *PMBS'12 - Proceedings of the 3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, Co-located with SC'12*, (to appear) 2012.

[31] P. Colella, D. Graves, T. Ligocki, D. Martin, D. Modiano, D. Serafini, and B. V. Straalen, "Chombo software package for AMR applications: design document," http://davis.lbl.gov/apdec/designdocuments/chombodesign.pdf.

[32] P. Colella, D. Graves, T. Ligocki, and B. V. Straalen, "AMR Godunov unsplit algorithm design and implementation," http://davis.lbl.gov/apdec/designdocuments/amrgodunov.pdf.

[33] P. Colella, D. Graves, T. Ligocki, D. Modiano, and B. V. Straalen, "EBChombo software package for Cartesian grid embedded boundary applications," http://davis.lbl.gov/apdec/designdocuments/ebchombo.pdf.

[34] ——, "EBAMRTools:EBChombo's adaptive refinement library," http://davis.lbl.gov/apdec/designdocuments/ebamrtools.pdf.

[35] ——, "EBAMRGodunov design," http://davis.lbl.gov/apdec/designdocuments/ebamrgodunov.pdf.

[36] D. J. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 2/3, pp. 215–226, 2000.

[37] G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, Eds., *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, DE, USA, October 8-10, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 5898. Springer, 2010.

[38] S. Williams, A. Watterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the ACM*, April 2009.

[39] S. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, University of California, Berkeley, December 2008.

[40] 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.

[41] ——, "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.

[42] A. Basumallik and R. Eigenmann, "Optimizing irregular shared-memory applications for distributed-memory systems," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM Press, 2006, pp. 119–128.

[43] 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 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 72:1–72:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389094

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

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

[46] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 256–265. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542313

[47] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250761

[48] L. Chen, Z.-Q. Zhang, and X.-B. Feng, "Redundant computation partition on distributed-memory systems," in *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, oct. 2002, pp. 252 –260.

[49] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley, "Principles of runtime support for parallel processors," in *Proceedings of the 2nd International Conference on Supercomputing*, 1988, pp. 140–152.

[50] 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.

[51] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, 2008.

[52] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*, 1st ed. Addison-Wesley Professional, 2004.