Automatic Parallelization to Asynchronous Task-Based Runtimes Through a Generic Runtime Layer

Charles Jin
Reservoir Labs
New York, NY
jin@reservoir.com

Muthu Baskaran
Reservoir Labs
New York, NY
baskaran@reservoir.com

Benoit Meister
Reservoir Labs
New York, NY
meister@reservoir.com

Jonathan Springer
Reservoir Labs
New York, NY
springer@reservoir.com

***Abstract*—With the end of Moore's law, asynchronous task-based parallelism has seen growing support as a parallel programming paradigm, with the runtime system offering such advantages as dynamic load balancing, locality, and scalability. However, there has been a proliferation of such programming systems in recent years, each of which presents different performance tradeoffs and runtime semantics. Developing applications on top of these systems thus requires not only application expertise but also deep familiarity with the runtime, exacerbating the perennial problems of programmability and portability.**

This work makes three main contributions to this growing landscape. First, we extend a polyhedral optimizing compiler with techniques to extract task-based parallelism and data management for a broad class of asynchronous task-based runtimes. Second, we introduce a generic runtime layer for asynchronous task-based systems with representations of data and tasks that are *sparse* and *tiled by default*, which serves as an abstract target for the compiler backend. Finally, we implement this generic layer using OpenMP and Legion, demonstrating the flexibility and viability of the generic layer and delivering an end-to-end path for automatic parallelization to asynchronous task-based runtimes. Using a wide range of applications from deep learning to scientific kernels, we obtain geometric mean speedups of 23.0× (OpenMP) and 9.5× (Legion) using 64 threads.

***Index Terms*—compiler, mapping, asynchronous task-based runtimes.**

I. Introduction

The quest for exascale computing has ignited a parallel effort to develop programming models that can support the increasingly complex architectures. Such a system would need to contend not only with the challenge of programming applications that can make efficient use of massive amounts of parallelism, heterogeneous nodes with specialized accelerators, and deep memory hierarchies, but also emerging runtime concerns such as global power constraints, diverging clock speeds, and resiliency against node failures.

Asynchronous task-based execution has gained support in recent years as a paradigm for addressing such concerns. The approach consists of decomposing a computation into smaller pieces, called *tasks*, which are submitted for execution asynchronously to a separate software layer, the *runtime system*. The runtime system is then responsible for dynamically mapping tasks to the available resources, which allows it to adjust to variations in the runtime environment.

The development of asynchronous task-based runtimes is an active subject of research in the exascale community and beyond, with efforts including: OpenMP [1], Legion [2], Open Community Runtime (OCR) [3], Concurrent Collections (CnC) [4], Charm++ [5], OmpSs [6], Habanero [7], PaRSEC [8], X10 [9], DARMA [10], among others. This proliferation of runtimes has posed a challenge for application developers, as writing efficient codes requires not only domain expertise but also deep familiarity with the runtime, exacerbating issues of maintainability and portability.

This work seeks to address this challenge by presenting an end-to-end framework for the automatic parallelization of sequential codes to asynchronous task-based runtimes. Our contributions are threefold:

- We augment R-Stream [11], an auto-parallelizing polyhedral compiler, to express task-based parallelism and data management for a generic runtime;
- We design a new generic task-based runtime layer corresponding to the polyhedral output, with higher-level abstractions for representing both tasks and data as tiles;
- We provide two implementations of this generic runtime layer: a mature implementation for shared memory systems in OpenMP, and a proof-of-concept for distributed memory systems in Legion, demonstrating the flexibility and expressiveness of the API.

Using a variety of benchmarks, we demonstrate an average (geometric mean) speedup of 23.0× (64 threads) for the automatically generated OpenMP task code over sequential. As a baseline, we implement versions of the benchmarks that are parallelized using OpenMP's do-all parallel construct (21.0× average improvement). The task-based versions perform especially well on benchmarks with irregular computational footprints or insufficient do-all parallelism, while achieving comparable performance on more regular benchmarks. We also provide a prototype implementation for distributed memory system using Legion. While the average speedup of the generated Legion code over the sequential benchmarks is less dramatic at 9.5×, this gap is mainly attributable to the additional runtime overhead in the generic layer for supporting distributed memory; the main contribution is a demonstration

This material is based in part upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Awards Number DE-SC0018480 and DE-SC0019522.

of automatically generating several thousands of lines for distributed memory from tens of lines of sequential source.

II. Generic Tile-based Runtime Layer

We first describe a new runtime layer which introduces higher-level abstractions for tasks and data that are *tiled by default*. By this, we mean that tasks and data are identified both by a type identifier as well as a point in some coordinate space. This representation is a compact way to express task-based parallelism for a large class of loosely "regular" applications (e.g. most scientific kernels and stencil computations), and serves as the abstract target for the polyhedral analysis.

Because the generic runtime layer will be implemented separately for each underlying runtime system, the design of the abstractions should not leak the implementation details of any particular lower-level interface, e.g., the overhead of managing distributed memory should not be incurred when the generic runtime is used for shared memory.

The runtime layer components can be separated into three main categories, namely, tasks, data, and dependences. Appendix A describes the full API, while Appendix B provides implementation details for Legion and OpenMP.

A. Tasks

Tasks represent tiled units of computation, and are associated with a `taskTypeId` and a `taskId`:

- `taskTypeId` identifies a general type of task. At a high level, this is a lightweight handle to the function run by each task of the given type.
- `taskId` identifies a specific instance of a task, abstractly represented as a point within the (potentially sparse) space shared by other tasks of the same type.

This is a natural way to express parallelism in nested loop structures, i.e. decomposing the computation into tasks by tiling the iteration space. For instance, a 16×16 iteration space may be decomposed into 4 tiles of size 8×8. Each tile would share the same `taskTypeId`, while the `taskId` is in the range $[0, 3]$. Tasks have access to their `taskId` at runtime to determine their shares of the original computation.

B. Data

Datablocks are tiles of data. Similar to task tiles, a datablock is identified by the datablock type (`dbTypeId`) and a list of `coords`, which permits an abstraction of a datablock as a point within the (potentially sparse) space identified by the `dbTypeId`. This logical representation very naturally captures the common case of tiling dense arrays when decomposing a larger computation into smaller tasks. For instance, a 32×32 array may be decomposed into 8 tiles of size 16×8. The size of the coordinate space would be 2×4 while the dimensions of each tile would be 16×8. Each tile shares the same `dbTypeId`, while the `coords` to identify a specific tile would run from $(0, 0)$ to $(1, 3)$, e.g. tile $(1, 2)$ refers to $[16 : 32, 16 : 24]$ in the original array.

Before any tasks can be spawned, all datablock types must be declared to the runtime by providing the `dbTypeId` as well as some bounds on the size of the coordinate space. After the runtime initialization, a task is then able to fetch a specific datablock by providing a `dbTypeId`, a set of `coords`, and the size of the datablock in bytes. The decision to specify sizes at runtime rather than initialization is motivated by the desire to enforce the abstraction of datablocks as points within any arbitrary coordinate space, which is useful for tiling irregular spaces. For instance, a sparse space may have many points which are never created, and tiles within the space may even be of different sizes and shapes (indeed, tiles need not even be quadrangular, as in the case of triangular matrices). It is thus also important to defer the creation of datablocks until they are explicitly needed, as an eager approach could allocate datablocks corresponding to unused points.

Identifying data with a `dbTypeId` and a list of `coords` also allows lightweight handles (i.e., references) to be passed without moving the underlying data, which can be very expensive on distributed memory systems. For runtimes that support such functionality, the generic layer can defer data movements until the data is actually needed, reducing unnecessary intermediate movements.

Support for distributed memory systems is implemented using the underlying runtime's primitives. Though distributed memory systems generally require additional overhead for data management, the generic runtime layer is flexible enough to avoid any overhead when targeting runtimes for shared memory.

C. Dependences

We consider two classes of dependences:

- **Data Dependence.** When a new instance of a task is created, the runtime is passed handles to all its necessary datablocks by the spawning task. The runtime will not schedule the task for execution until the underlying data is coherent and available for the task to use.
- **Control Dependence.** Every task is associated with a count for the number of predecessors. Upon completion, tasks auto-decrement the count of its successor tasks (we refer to this as `autodec`). A task is only scheduled for execution when its predecessor count is zero.

Concretely, the data dependences are specified by passing a datablock *enumeration* function, parameterized by the child task's `taskTypeId` and coordinates (via the `taskId`), to the runtime. The runtime is then able to use this function to convert the `taskTypeId` and `taskId` into the necessary set of `dbTypeId` and `coords`.

Similarly, the control dependences are specified by providing the runtime with a predecessor *count* function, again parameterized by the child task's `taskTypeId` and `taskId`. Every task is then required to `autodec` all its successors; the runtime tracks the number of tasks that have called `autodec` for the child task and compares this to the total number of expected predecessors to determine when a child task can be submitted for execution to the underlying runtime.

The runtime satisfaction of dependences allows for the creation of a *self-unfolding task DAG*, where the frontier

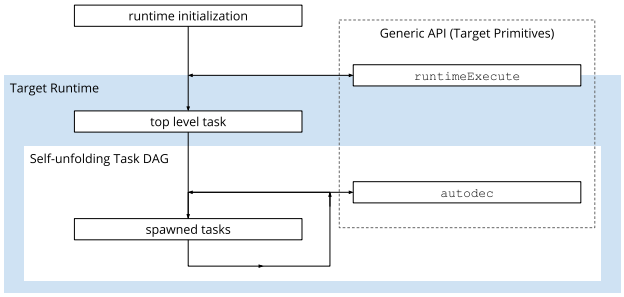


Fig. 1. Flow diagram for the self-unfolding task DAG.

nodes (tasks) and edges (dependences) are created dynamically during execution by the encountering tasks, rather than all at once by the master thread during start up. Specifically, the *enumeration* mechanism for data dependences allows for datablocks to be created on-the-fly, while the *count* mechanism for control dependences removes the need to statically specify any single parent responsible for spawning a given task.

Note that a sequential interleaving of tasks and their data dependences induces a set of control dependences between the tasks (i.e., by analyzing the sequence of reads and writes). Rather than use these induced dependences, the runtime layer still requires the control dependences be explicitly provided via the system of *count* functions and `autodec` calls. The purpose is two-fold: first, this design affords the developer the option to manage synchronization of data accesses between tasks by hand, when performance is critical; second, this avoids the need for any dynamic dependence analysis as tasks are created, making the runtime layer extremely lightweight.

III. Polyhedral Flow

In this section, we describe the techniques implemented in R-Stream, an auto-parallelizing polyhedral optimizing compiler, that target a generic task-based runtime. The compiler accepts as input sequential C and performs polyhedral loop analysis to extract parallelism. This thus completes an end-to-end pathway for automatic parallelization from sequential source code to asynchronous task-based runtimes.

A. Background

The polyhedral model is an optimization framework that relies on a mathematical representation of a computation as a polyhedron. For instance, each iteration of a nested loop body can be represented as lattice points in space, the collection of which is a polyhedron. Though representations of other programs are possible, the model is most precise on affine regions and access patterns, which encompasses a wide range of scientific kernels and other common computationally intensive workloads.

This compact abstract representation allows optimizations to be applied as transformations to the polyhedron rather than as a pass on some program IR, as is common in traditional compilers, permitting some global transformations that are too difficult or costly to perform on standard IRs. Optimizations expressible include fusion and array contraction, as well as any combination of common loop transformations such as fusion, fission, interchange, and skewing. Furthermore, data dependences have an exact representation in the polyhedral model, enabling the automatic extraction of parallelism as well as data-oriented optimizations such as loop tiling and data layout transformations.

B. Polyhedral Analysis

We build upon a prior work by Baskaran et al. [12] and modify an existing mapping path that produces task-based parallelism and data management for the OCR runtime. This delivers a successful proof-of-concept for using a polyhedral compiler flow to target a wide range of task-based runtimes.

First, R-Stream performs the standard polyhedral **raising** and **dependence analysis** to generate a polyhedral representation for the input program. **Scheduling** performs transformations on the polyhedral representation to extract parallelism and improve locality. Next, **tiling** heuristically aggregates the resulting loop nests into larger chunks to increase the granularity of parallelism. Tiling also picks sizes for the datablocks while taking into consideration caching behavior, the resulting data dependences, and data management overhead. Certain auxiliary information extracted from the dependence polyhedra at this step (e.g. liveness, data access requirements) is also passed through to the target runtime as hints. Finally, **dependence generation** recomputes the dependence relations between the resulting task tiles.

C. Code Generation

The polyhedral representation is translated back into source code in **code generation**, which is a modified application of the standard polyhedral scanning. The final step is specialized by target to account for runtime differences (see Appendix C). The following subsections describe these techniques for the runtime components in greater detail.

1) Tasks: The polyhedral compiler tiles computation loops to improve both spatial and temporal locality. This decomposition is suggestive of our coordinate representation of tasks, except that the size of the resulting tiles is generally too fine for task-based execution (e.g., inner loops are tiled to fit within cache lines). In order to increase the granularity of parallelism, we first perform an optional **grouping** pass, which aggregates smaller tiles into larger groups that then become tasks. This pass may be parameterized by, among other things, the characteristics of the underlying runtime, as different runtimes will have different trade-offs between overhead and increased granularity. Next, the coordinates of the tile in the aggregated loops are translated into coordinates for tasks, and the loop bounds and computational body for each task type is generated via standard polyhedral scanning techniques.

It is important to note that the ability of the generic runtime to represent a sparse set of task tiles is particularly important when the resulting tiles are irregularly shaped. For instance, one artifact of the grouping pass is that tasks at the boundaries of the original loop may be irregular, as the resulting task

sizes often do not evenly divide the original loop dimensions. Additionally, for more complex dependence structures, the transformed loop bounds may not even be quandrangular, with the most common example being the triangular loop bounds of many matrix methods. Designing the runtime for sparse coordinate spaces for tasks is thus important for maximum expressibility while avoiding overhead for points that are never created.

2) Data: The polyhedral compiler, in addition to tiling computation loops, also produces tiles for arrays. Hence the process to convert the data tiles into datablocks is very similar to the one used to convert computation loop tiles into tasks. Auxiliary information gathered during this phase also includes the corresponding modes of access for each data tile (e.g., read-only, read-write), and the liveness characteristics of each data block. In particular, for runtimes which manage data entirely internally, this liveness information is used to determine whether any data needs to be copied in from the outer context before computation begins, and similarly, copied out once the computation has concluded. This is useful when only a portion of the application is mapped, though it introduces overhead.

3) Dependences: Dependence information is extracted from the source program during polyhedral analysis and converted back into code via the usual polyhedral scanning. Data dependences are inherent in the determination of data tiles and have a parametric form in the polyhedral model for each task type, so the *enumeration* functions can be generated via standard polyhedral scanning.

Control dependences need to be inferred in both directions. First, a task needs to know all its successors in order to `autodec` them to create the self-unrolling task DAG. This is accomplished by projecting the polyhedron for a given task type *along* the directions of dependence, and scanning the resulting reduced polyhedron to create a loop that invokes `autodec` for each successor. This loop is parameterized by the task coordinates and can be inserted as an epilogue in the function for a task type. Similarly, whenever a specific task is spawned, the runtime needs to be passed a function that provides the number of predecessors. This can be computed by projecting the polyhedron *against* the directions of dependence; the reduced polyhedron is then lowered into a *count* function parameterized by the task type and coordinates. Tasks for which the number of predecessors is zero, i.e. boundaries of the polyhedra which are oriented along the directions of dependence, are spawned in a function which serves as the entry point into the generic runtime.

This process results in each task type being associated with a single piece of code for each of datablock enumeration, predecessor counting, and successor spawning, allowing the analysis to be run only once per task type. Thus, the complexity of code generation scales with the number of distinct task types and not the size of the computation (up to a certain point). Because dependence information is inferred statically at compile time (and indeed, comes "for free" as a byproduct of the polyhedral analysis), we strive, wherever possible, to handle these dependences in the generic runtime layer, rather than deferring to underlying runtimes, which must build and manage the control and data dependences dynamically.

TABLE I
BENCHMARK RESULTS[†§]

	sequential	OpenMP do-all	OpenMP task	Legion
SpMV	0.0608	0.0243	0.0277	0.7285
GoogLeNet	6.8911	0.5023	0.5914	2.3966
ResNet-50 v2	137.1970	4.0172	2.5630	8.3584
SW4	17.8179	0.1109	0.1027	2.9793
Kripke	1.9067	0.1708	0.1565	0.8043
HOMME	5.0319	0.3045	0.2427	0.3805

[†]Results for OpenMP, Legion implementations are with 64 threads.
[§]Execution time, in seconds.

IV. RESULTS

We present performance results for a variety of common workloads, loosely falling into the categories: embarrassingly parallel, deep learning, and scientific kernels (Tables I and II). As a baseline for comparison, we provide timings for the unmapped sequential codes, as well as a hand-tuned version of each benchmark using do-all parallelism in OpenMP (`#pragma omp parallel for`). Unless otherwise noted: parallelizing adjacent loops is done using the `collapse` directive to avoid nested parallel regions; loop scheduling is set to `static` to improve locality; and threads are bound to sockets using `OMP_PLACES` to reduce NUMA overhead.

All experiments were run on an 8-core quad socket (16 hyperthreads) Intel Xeon (Ivy Bridge) server. Code was compiled with GCC 7.3 (OpenMP 4.5) with the `-O6` flag.

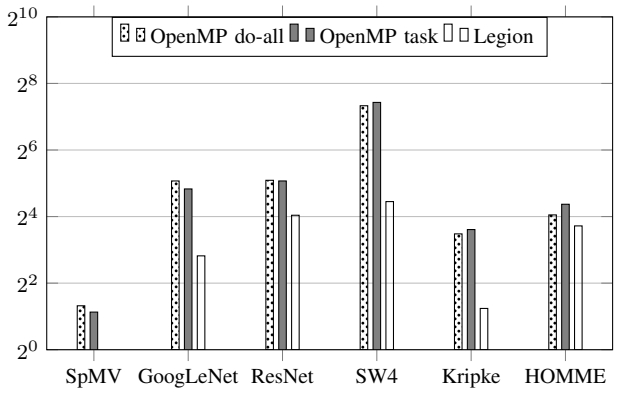
A. SpMV

We implement a sparse matrix-vector multiply taken from the HPCG benchmark [13], a tool intended to complement the High Performance LINPACK (HPL) benchmark used for TOP500. The computation uses a standard compressed sparse row representation and is embarrassingly parallel over the rows, and so is well-suited for do-all parallelism. The kernel also has array indirections that make the computations non-affine, an additional challenge for the polyhedral model.

For the hand-mapped OpenMP implementation, we find that it is sufficient to parallelize this outer loop. The results of the experiment show that the automatically-generated OpenMP task version performs comparably with the hand-mapped version ($2.2\times$ vs. $2.5\times$ speedup, respectively), despite the array indirection. Because there is sufficient parallelism in the outer loop for all variants to exploit (1024×1024 parallel loop iterations), the gap in performance is attributable mainly to runtime overhead. The small gap between OpenMP variants indicates that the overhead of the generic layer is comparable to the standard OpenMP `#pragma omp for` construct.

The experimental Legion target shows a significant slow down. Empirically, we find that the cost of simply entering the Legion runtime nearly exceeds the total execution time of the sequential variant (0.0536 vs. 0.0608 seconds, respectively).

TABLE II
LOG SPEEDUP OVER SEQUENTIAL†§



†Results are with 64 threads.
§Negative speedups not displayed.

TABLE III
LEGION COPY OVERHEAD†

benchmark	copy only	full kernel	% copy
SpMV	0.1779	0.7285	24.4%
GoogLeNet	0.9622	2.3966	40.2%
ResNet	2.4494	8.3584	29.3%
SW4	0.8146	2.9793	27.3%
Kripke	0.1247	0.8043	15.5%
HOMME	0.2633	0.3805	69.2%

†First two columns give execution time, in seconds.

B. Convolution

We implement two convolution kernels: a smaller convolution kernel from GoogLeNet [14], a deep convolution neural network; and a larger dual-layer convolution / bias-add kernel from ResNet-50 v2 [15], a deep residual neural network. These kernels are characterized by the highest data-to-compute ratio of all benchmarks used, and operate on the largest regions of memory of all benchmarks implemented (>1TB for ResNet).

Both kernels have four independent outer loops. Experimentation yields that the optimal number of loop dimensions to parallelize in the hand-mapped implementations is four in the GoogLeNet kernel and three in the ResNet kernel.

On the smaller GoogLeNet convolution kernel, the results for the two OpenMP variants are similar; however, with the larger ResNet kernel, the OpenMP task variant outperforms the do-all variant with a $53.5\times$ vs. a $34.2\times$ speedup. This is in line with the expectation that, as the problem size increases, the overhead of the asynchronous task-based runtime is quickly subsumed by improved load balancing.

The convolution kernels are large enough that the prototype Legion implementation is now able to deliver improvements of $7.0\times$ and $16.4\times$ for the GoogLeNet and ResNet kernels, respectively. However, we find that performance quickly becomes dominated by the cost of moving data into and out of datablocks at runtime initialization and shutdown, respectively. One lower bound for this overhead is to perform only copy operations without the computation (which also removes some dependences). We find that, under this measure, the copy operations contribute approximately 30% and 40% to the execution times of the GoogLeNet and ResNet convolution kernels, respectively (Table III).

C. SW4

SW4 is a scientific application for modelling 3D seismic wave equations on a Cartesian mesh. We evaluate the performance of our techniques on a 4th order stencil kernel. The kernel is embarrassingly parallel—every iteration of the loop can be performed independently. In contrast to the convolution kernels, however, each iteration consists of 666 floating point operations. We find the best performance for the hand-mapped OpenMP version comes from parallelizing three outer loops.

Comparing the hand-mapped OpenMP against the tasking version, we see that the automatically generated version performs about 8.0% better ($160.6\times$ and $173.5\times$ speedup, respectively). It is interesting to note that the improvements exceed what would be expected given 64 threads; this can be explained due to the improved locality of data access from (1) the `schedule` directive in the case of the hand-mapped version, and (2) the tiling generated by the polyhedral model in the OpenMP task version. The prototype Legion version is significantly slower, though still able to deliver a $21.9\times$ speedup over the sequential code.

D. Kripke

Kripke [16] is a mini-application developed by LLNL as a proxy for 3D Sn deterministic particle transport codes. The key sweep kernel has an outer loop of do-all parallelism and inner loops that follow a wavefront dependence pattern. The advantage of applying task-based parallelism comes from the overdecomposition of the inner loops to improve load balancing; a more detailed exploration of tasking in Kripke is discussed in previous work by Jin et al. [17].

The problem size is selected such that the outer loop is of sufficient size to saturate all 64 threads; thus, it suffices to parallelize over the outer do-all loops in the hand-mapped version. However, because the inner loops contain computationally-intensive operations, the finer granularity of a task-based approach should still improve performance due to superior load balancing. Indeed, the best performance comes from the OpenMP task-based runtime with a $12.2\times$ speedup, versus $11.2\times$ for the reference OpenMP implementation and $2.4\times$ for Legion. The generated task-based code shows that the polyhedral model successfully tiles the loops to extract parallelism from the wavefront dependence pattern.

E. HOMME

The High-Order Method Modeling Environment (HOMME) is a benchmark for climate modeling codes. We implement a kernel which calculates a spherical Laplacian then performs a biharmonic viscosity update. The do-all version would require nested parallel regions to fully exploit the available parallelism. We find that this increases the overhead significantly;

TABLE IV
LINES OF CODE†

	sequential	OpenMP task	Legion
SpMV	20	280	1100
GoogLeNet	25	200	1120
ResNet-50 v2	50	200	1320
SW4	300	800	7100
Kripke	80	300	1600
HOMME	100	500	1650

†LOC for hand-mapped OpenMP is within a few lines of sequential.

given there is sufficient parallelism in the outer loops to saturate 64 threads, we only map the outer parallel regions.

Conversely, the task-based implementations are able to exploit this additional parallelism at no additional cost. The automatically generated OpenMP task version performs the best, at 0.24 seconds, with the reference OpenMP (0.30 seconds) and Legion (0.38 seconds) versions following behind. We also find that the experimental Legion version comes closest to the reference OpenMP implementation on this benchmark, with a $13.2\times$ vs. $16.5\times$ speedup, respectively. Looking at the overhead numbers for Legion, we find that the copy code consists of almost 70% of the total runtime (Table III). Inspecting the generated code reveals that this performance is due to a significant overlap in computation and data movements.

F. Discussion and Future Work

The automatically generated code for the OpenMP task variant was either close to or exceeded the performance of the hand-mapped OpenMP codes, even though the hand-mapped versions benefited from improved affinity and locality hints at the OpenMP runtime level, which confirms that the runtime remains lightweight even after layering a generic layer on top of the OpenMP task runtime.

The experimental Legion target, though able to deliver speedups in all but the smallest kernel, still lags far behind the OpenMP task target, albeit on a shared memory architecture. However, it is important to mention the Legion target already delivers significant gains in terms of maintainability and portability. The key kernels of many scientific applications are simple loops that can be expressed in tens of lines of code, but production codebases are often several degrees of magnitude larger, due to the substantial effort needed to manage data and synchronizations in a distributed environment.

Table IV illustrates these maintainability and portability benefits for our end-to-end approach by starting from a bare-bones sequential implementation. The sequential source can also be recompiled through the polyhedral model to target different architectures, automatically adjusting the granularity of parallelism without the need to perform brittle parametric decompositions, as is standard practice today.

V. RELATED WORK

Improving support for task-based runtimes can be split into two general approaches. The first is the strategy used in this paper, which involves generating code directly in the language of the underlying runtime system. Work by Kong et al. [18] uses a polyhedral compiler to generate code for distributed runtimes through CnC, another asynchronous task-based runtime, however their approach is less general, involving new lower-level language extensions for distributed memory clusters. Baskaran et al. [12] also use a polyhedral compiler and build higher-level abstractions for task-based parallelism, but their focus is limited to OCR.

The second approach is to explicitly annotate an existing language with parallel constructs, which requires porting existing codes or is otherwise less transparent to the developer. One common approach is to use pragmas to express parallelism, as in the case of Cilk [19], OpenMP [1], and OpenACC [20], and others. We find this approach complementary to ours in that these primitives can be used to implement our generic layer.

There have also been attempts to unite task-based runtimes from the opposite direction by providing a common underlying layer, most recently with OCR [21] and DARMA [10]. We believe this is a harder problem, since such a common runtime system would need to provide one set of primitives to efficiently implement all existing runtime systems. In contrast, our runtime layer has multiple implementations, each of which only needs to be as efficient as the underlying runtime system.

VI. CONCLUSIONS

Asynchronous task-based parallelism is a promising paradigm for upcoming heterogeneous exascale systems; however, properly decomposing computations and expressing parallelism for asynchronous task-based execution remains a challenge for application developers. We present an end-to-end solution by building a compiler tool that automatically generates code for asynchronous task-based runtimes starting from plain sequential source written in C. Our approach leverages R-Stream, an auto-parallelizing polyhedral compiler, to extract task-based parallelism and data management; we also design a new generic runtime for asynchronous task-based runtimes that serves as an abstract target for the polyhedral output. The generic runtime layer provides higher-level abstractions for sparse, tile-based representations of tasks and datablocks, which enables (1) a concise logical representation of computation and data for task-based programming, as well as (2) an efficient execution policy for on-the-fly creation of tasks and data. In order to demonstrate the flexibility and performance of this generic layer, we implement the generic runtime layer using two runtime systems: OpenMP tasking and Legion. Compared to hand-written implementations of benchmarks using OpenMP do-all parallelism, the automatically generated OpenMP task implementations perform 12.0% faster on average over a wide variety of workloads, particularly larger scientific kernels, with a geometric mean improvement of $23.0\times$ over the sequential code. The experimental Legion target shows more measured performance gains ($9.5\times$), but demonstrates the ability to automatically generate thousands of lines of scaffolding for distributed memory systems from only tens of lines of sequential source.

REFERENCES

[1] OpenMP Architecture Review Board, "The OpenMP specification for parallel programming," 2015, http://www.openmp.org/.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, November 2012.

[3] T. Mattson, R. Cledat, V. Cave, V. Sarkar, Z. Budimlic, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tarislar, J. Teller, and N. Vrvilo, "The Open Community Runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, September 2016.

[4] Intel, "Concurrent Collections," http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[5] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in Charm++," in *Proceedings of the Second International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2. Piscataway, NJ, USA: IEEE Press, 2016, pp. 40–43. [Online]. Available: https://doi.org/10.1109/ESPM2.2016.7

[6] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with OmpSs," in *Euro-Par 2011 Parallel Processing*, ser. Lecture Notes in Computer Science, E. Jeannot, R. Namyst, and J. Roman, Eds. Springer Berlin Heidelberg, 2011, vol. 6852, pp. 555–566. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23400-2_52

[7] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The new adventures of old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '11. New York, NY, USA: ACM, 2011, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/2093157.2093165

[8] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in PaRSEC: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17. New York, NY, USA: ACM, 2017, pp. 6:1–6:8. [Online]. Available: http://doi.acm.org/10.1145/3148226.3148233

[9] IBM, "X10," http://x10-lang.org/.

[10] Sandia National Laboratories, "DARMA," https://share-ng.sandia.gov/darma/.

[11] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, "R-Stream compiler," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1756–1765.

[12] M. Baskaran, B. Pradelle, B. Meister, A. Konstantinidis, and R. Lethin, "Automatic code generation and data management for an asynchronous task-based runtime," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, Nov 2016, pp. 34–41.

[13] M. A. Heroux and J. Dongarra, "Toward a new metric for ranking high performance computing systems," 2013.

[14] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[16] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke - a massively parallel transport mini-app," in *Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method (ANS MC '15)*, Nashville, Tennessee, Apr. 2015.

[17] C. Jin and M. Baskaran, "Analysis of explicit vs. implicit tasking in openmp using kripke," in *4th Workshop on Extreme Scale Programming Models and Middlewear (ESPM2)*, 11 2018, pp. 62–70.

[18] M. Kong, L. Pouchet, P. Sadayappan, and V. Sarkar, "PIPES: A language and compiler for task-based programming on distributed-memory clusters," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 456–467.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/277650.277725

[20] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_85

[21] N. Vrvilo, "A hands-on introduction to the CnC programming model," in *9th Annual Concurrent Collections Workshop*, 2017, https://cnc-workshop.github.io/cnc2017/media/cnc-2017-tutorial-slides.pdf.

[22] B. Meister, A. Konstantinidis, M. Baskaran, T. Henretty, B. Pradelle, T. Ramananandro, S. Tavargeri, A. Johnson, and R. Lethin, "A gpu runtime for event-driven task parallelism," 02 2015.

APPENDIX A
API DESIGN

We provide a high level overview of the design of our generic runtime layer API.

A. Dependences

Dependence information is passed to the runtime whenever a new task is spawned via two special functions of the following types:

- **`predFn`**: accepts as input the `taskTypeId` and coordinates, and returns the number of predecessors for the task instance being spawned.
- **`enumFn`**: accepts as input the `taskTypeId` and coordinates, and fills a data structure with handles to all the data blocks needed by the task instance being spawned.

B. Tasks

Tasks are spawned via the `autodec` system, and the API requires the insertion of a prologue and epilogue for proper runtime management of resources. We use the `taskFn` type to refer to the signature of task functions in the underlying target runtime (e.g., OpenMP tasks can be arbitrary C functions, but all tasks in Legion have the same signature: a Legion runtime variable, context, etc.).

- **`declareTaskType`**: registers a new task type with the generic runtime layer. Parameters are a `taskTypeId` and a pointer to the corresponding `taskFn`.
- **`autodec`**: submits a task for spawning to the generic runtime. More precisely, this call notifies the runtime of the satisfaction of an in-dependence for the specified task instance. The function accepts the `taskTypeId` and coordinates of the task to be spawned, pointers to the `predFn` and `enumFn`, and a pointer to an args struct created by `allocArgs` for passing scalar arguments.
- **`allocArgs`**: takes a pointer and a size parameter and returns a pointer for passing scalar arguments to `autodec`.
- **`enterTask`**: a prologue function that has the same signature as a `taskFn`. Returns a pointer to the private generic runtime context.
- **`unpackArgs`**: takes the private runtime context and unpacks the args struct passed to the task at creation.
- **`exitTask`**: an epilogue function that cleans up the private runtime context.

The behavior of `autodec` is non-blocking: the runtime first checks if this satisfies the predecessor count: if not, the call returns immediately; otherwise, the runtime layer invokes the underlying runtime API to spawn a task, which is asynchronous in the underlying runtime.

C. Data

The main abstraction for data is the *datablock*, which represents a tile of data in a sparse coordinate space:

- **`declareDBType`**: registers a new datablock type with the generic runtime layer. Parameters are a `dbTypeId` and a bounding box for the coordinate space.
- **`fetchDB`**: fetches a specific datablock, as specified by a `dbTypeId` and a set of `coords`.
- **`freeDB`**: destroys a datablock, as specified by a `dbTypeId` and a set of `coords`.

The generic runtime guarantees that all enumerated datablocks are valid for fetching by the time a task begins execution. In particular, the call to `fetchDB` does not block on the creation of a datablock. Attempts to either (1) fetch a datablock that was not enumerated, or (2) enumerate a datablock that was previously freed give undefined runtime behavior.

D. Runtime

The generic runtime has its own lightweight context for managing tasks and data:

- **`runtimeInit`**: initializes the runtime and returns the shared runtime context. Must be the first API call.
- **`runtimeExecute`**: begins execution with a specific task. The entry point is specified by a `taskTypeId` and coordinates. This task implicitly has no predecessors.
- **`runtimeExit`**: cleans up the runtime context. Must be the last API call.

Finally, in order to introduce flexibility when targeting specific implementations of the generic API, many of the generic APIs also accept an additional parameter for unstructured flags to enable further target-specific optimization.

Appendix B
Generic Runtime Implementations

We describe implementations of the generic tile-based runtime introduced above using primitives from two different underlying runtimes: OpenMP's tasking runtime and the Legion runtime.

A. OpenMP

OpenMP follows the fork-join model of parallelism, whereby a master thread creates a team of slaves to workshare a region. One common idiom is to create a team of threads to workshare the iterations of a loop. This work focuses on the `#pragma omp task` construct introduced in OpenMP 3.0. Roughly speaking, a thread which encounters this construct packages the marked block into a deferrable task. The OpenMP runtime then assigns deferred tasks to any available threads within the encountering thread's team. For instance, the following code creates a parallel region with 4 threads, then selects a single "master" thread to execute the loop and create the tasks. The runtime then assigns the 10 tasks between the team of 4 threads.

```
#pragma omp parallel num_threads(4)
#pragma omp single
for (int i = 0; i < 10; i++) {
#pragma omp task
  A[i] *= 2;
}
```

Since the preceding example had no dependences, any schedule is a valid schedule, including an overlapping schedule with concurrent execution. For tasks which are not trivially parallel, dependences for OpenMP tasks can be expressed using the `depend` clause. However, for reasons explained below, we choose to manage the dependences in the generic runtime layer, rather than passing this responsibility to the OpenMP runtime system.

The generic runtime context is separated into a shared and a (task-)private context.

- The **shared context** is populated with metadata about the tiled representation of tasks and datablocks during runtime initialization, as well as runtime information about specific instantiations of the tasks and datablocks. Because the shared context can be accessed by multiple tasks concurrently, methods that alter the shared context are implemented using thread-safe OpenMP primitives.
- The **private context** is constructed and passed to a child task by the parent, and includes the child's `taskId` and a pointer to the args struct.

As OpenMP operates under the shared memory model, the implementation of the other components for the generic runtime layer using OpenMP primitives is relatively straightforward.

1) Tasks: Tasks are registered at runtime initialization with a pointer to the task's function. Spawning tasks is done by invoking the `autodec` operation on the generic runtime, which tracks dependences to determine when the task is ready for scheduling. Once the conditions have been satisfied, the generic runtime blocks the parent task's execution to set up the child's context, including the creation of any new datablocks. The final invocation of the task's function is wrapped inside a `#pragma omp task` block, which passes a deferrable task to the OpenMP runtime, freeing the parent task to continue execution.

2) Data: Datablocks are registered at runtime initialization with a bounding box for the coordinate space. In order for a child to use a datablock, the task's parent enumerates all the child's datablocks when the child is being spawned, which allows for just-in-time creation of those datablocks that have yet to be instantiated. This guarantees that the child task's datablocks are valid when the child task runs. The child task is then able to fetch a datablock by ID and coordinates, returning a C-style pointer for reading and writing. Fetching a datablock that was not specifically enumerated results in undefined behavior.

Because the OpenMP backend targets shared memory systems, datablocks can be instantiated using the standard C library malloc.

3) Dependences: While OpenMP has the built-in `depend` clause for expressing control dependences between tasks and provides its own runtime scheduling analysis, we elected to handle these dependences in the generic runtime layer, for two reasons:

First, it is not necessarily the case that the "master" thread will have the opportunity to spawn all the child tasks. This is because if the queue of deferred tasks is high, the master thread may switch from spawning new tasks to execution. Performance is then dictated by how much parallelism is exposed in the already-explored portion of the task DAG, and one can construct pathological schedules where no parallelism is extracted, despite sufficient inherent parallelism in the complete task DAG. Indeed, this is more than just a theoretical concern (a detailed exploration of this phenomenon is discussed in a work by Jin et al. [17]). The solution is to spawn the tasks in an order which exposes maximal parallelism at any given step. However, this is just a static solution to the scheduling problem, which renders performing dynamic scheduling analysis in the OpenMP runtime redundant.

Second, the key to the self-unrolling task DAG is the ability for any single predecessor to create a common successor. However, this necessarily requires the generic runtime layer to perform some coordination between the predecessors to ensure only one attempt to spawn the task succeeds, which is outside the scope of the OpenMP runtime; proper dependence management is then a "free" byproduct.

Due to the shared memory semantics of OpenMP, data dependences are trivial to resolve (given that the control dependences are respected), as references to data are valid from anywhere in the program.

B. Legion

We provide a prototype implementation of the generic runtime layer that targets distributed memory systems through Legion, a parallel programming framework for writing portable applications on distributed heterogeneous systems. Legion applications are separated into two components: standard program specification using a rich set of APIs, most notably for describing and manipulating program data, and a separate mapper interface that determines runtime decisions for assigning tasks to processing elements and placing data within the memory hierarchy. In this work, we focus on program specification and code generation to the C++ Legion API; the mapper interface is left to future work.

Legion is designed around its representation for program data. The primary data type in Legion is a *region*. Similar to the representation of datablocks in the generic runtime layer, regions in Legion have a dual representation: a `LogicalRegion` is a lightweight handle for argument passing; this must be "mapped" to a `PhysicalRegion` for access to a physical instance. Which tasks can have concurrent access to a `LogicalRegion` (and the corresponding `PhysicalRegion`) is determined by the specific permissions requested when the task was spawned. Because the `PhysicalRegion` refers to an actual memory location, for instance, two tasks on different nodes may not have simultaneous write access, though they may have simultaneous read access, if a copy is made first. A more detailed exploration of Legion's data model is given in the Data section.

Tasks in Legion follow a hierarchical structure, where each task has a single parent, forcing the task DAG into a tree structure. This hierarchy is necessary due to the scoping rules for task data permissions: a child can only be given a subset of the parent's data privileges. This requirement allows Legion to analyze dependences efficiently, enabling a distributed scheduling algorithm for scale. However, this restriction also significantly affects the design and implementation of the generic layer as detailed in the Dependence section.

The generic runtime context is separated into shared, (task-)spawn, and (task-)private contexts:

- The **shared context** is populated with metadata about the tiled representation of datablocks during runtime initialization. We avoid modifying the shared context after runtime initialization so that the Legion runtime can pass read-only copies to tasks without creating artificial dependences.
- The **spawn context** is created by the parent task before spawning, and contains metadata about the datablocks for which the task will have permissions upon spawning.
- The **private context** is created by the generic runtime during the initialization of the child task, and contains data about the specific datablocks which the task can access, e.g., Legion-specific region access and coherence properties.

Separating the spawn and private contexts allows the spawn context to be as lightweight as possible, which is important as it may be migrated across nodes in a distributed context.

1) Tasks: Implementing tasks in the generic runtime using Legion primitives is straightforward. First, task registration in the generic runtime is transparently passed through to the Legion runtime. Similarly, spawning tasks is performed via a Legion `task_launcher` object; this object accepts the task ID and information about the `LogicalRegions` needed by the task, which is provided by the *enumeration* function. The placement of tasks in a distributed context is handled in the Legion mapper interface, which is left to future work.

2) Data: Implementing the generic runtime's data API using Legion primitives is more involved due to the support for distributed memory. Recall that regions in Legion have a dual representation: (1) a `LogicalRegion`, which is a lightweight handle that can be used anywhere in the program, and (2) a `PhysicalRegion`, which references an actual physical instance of a region that is only valid within the enclosing task context. Management of this distinction between a `LogicalRegion` and `PhysicalRegion` is transparent in the generic runtime layer—from the perspective of interfacing with the generic runtime, datablocks are simply enumerated

and fetched, and then guaranteed to be valid when reading or writing within tasks.

More concretely, in our prototype implementation, each datablock is simply a `LogicalRegion` of type `char` with size equal to the number of bytes requested. To read or write the datablock, a raw pointer to the underlying data of the corresponding mapped `PhysicalRegion` is returned by invoking the underlying Realm API upon which Legion is built. This allows the generic runtime to defer the management of distributed data movements and coherence to the Legion runtime, while still presenting a consistent interface for data access.

In order to reduce unnecessary overhead, the generic runtime does not perform speculative mapping of a `LogicalRegion` to a `PhysicalRegion`, instead using the lightweight `LogicalRegion` for as long as possible. In particular, creation of a `LogicalRegion` is delayed until the first time the corresponding datablock is enumerated. Mapping to the `PhysicalRegion` is further delayed, until either automatically mapped by the Legion runtime upon entering a task that has permissions on the region; or, for datablocks that are used immediately within the same context in which the `LogicalRegion` is created, delayed until the datablock is actually fetched. This design allows for region mapping to be interleaved with spawning tasks, in order to start parallel execution as soon as possible.

The tiled representation of a datablock is saved in the global shared context at runtime initialization, specifically a bounding box for the coordinate system of the `dbTypeId`. Note that supporting irregular tile sizes makes it difficult to use Legion's built-in region partitioning mechanism for handling datablocks, as this requires specifying sizes for each individual tile at initialization. It is the programmer's responsibility to ensure that the same coordinate for a `dbTypeId` is always enumerated with the same size; for performance reasons (i.e., avoiding global synchronization), the behavior is undefined otherwise.

3) Dependences: We choose to defer the management of both data and control dependences to the Legion runtime.

First, in order to access a region, a task must be spawned with a corresponding `RegionRequirement`, which is an encapsulation of a data region along with the `coherence` and `privilege` properties. Loosely speaking, `coherence` determines how strong of a guarantee the task requires for concurrent data access (e.g. `ATOMIC`, `EXCLUSIVE`), while the `privilege` says what kind of access a task needs on the region (e.g. `WRITE_ONLY`, `REDUCE`). Once a task has been created, the Legion runtime guarantees that all the requested regions will be mapped and ready for access, obviating the need to handle any data dependences in the generic layer.

Second, control dependences in Legion are automatically inferred from the `RegionRequirements` of a task. For example, with `EXCLUSIVE` coherence, two adjacent tasks with `READ_ONLY` privileges on a region may be reordered or even scheduled simultaneously (so long as the tasks are either on the same node, or a copy of the region is made on separate nodes); however, a `READ_WRITE` task cannot be reordered around adjacent tasks which use the same region. The only way to avoid the runtime overhead of dependence analysis in Legion is to specify tasks with `RELAXED` coherence, which says that the programmer will be responsible for all synchronization and coherence; unfortunately, this coherence mode is not yet available in Legion.

Thus, we elect to spawn all the tasks in the top level task in a correct sequential order. This allows the generic runtime to give each task its minimal set of requirements, exposing maximal parallelism and deferring the job of determining an efficient, correct execution schedule to the Legion runtime; any additional dependence management performed by the generic runtime layer would necessarily be redundant.

C. Other Runtimes: Prior Work

We also adapt two existing runtimes to the generic runtime layer: OCR [12] and a custom GPU runtime [22]. Though the details of these implementations fall outside the main contributions of this work, that the generic runtime layer can be built on top of such a wide range of runtimes further demonstrates the viability of its design.

APPENDIX C
TARGET SPECIALIZATION OF CODE GENERATION

Though the polyhedral analysis applies for a generic runtime, the code generation phase must be specialized for each target runtime to account for differing runtime semantics and to improved performance.

A. OpenMP

When generating code for the OpenMP implementation of the generic runtime, we introduce several adjustments in the code generation to take advantage of the shared memory semantics. The main optimization is to add a separate path that converts all the datablocks back into regular array accesses. Because the OpenMP runtime works only on shared memory systems, tasks are allowed concurrent access to the same memory, and the control dependences suffice to guarantee a valid sequence of reads and writes. Additionally, while datablocks may be a useful logical abstraction for application developers, improving programmability and readability, this is unnecessary when the code is generated automatically, and removing this redirection reduces the runtime overhead. Removing a datablock also has the effect of removing its corresponding copy in and copy out operations (though these can be made very efficient in the case of shared memory). Finally, exposing the exact data access patterns enables more downstream optimizations in later compilation.

Another OpenMP-specific optimization is providing additional hints to the OpenMP runtime, specifically the data affinity hint introduced in OpenMP 5.0. This hint suggests to the runtime that a task should be colocated with the particular location in memory referenced by the hint. This information is readily extractable from the polyhedral model and a simple heuristic can be used to colocate tasks close to

the largest datablock needed by the task. However, as of the writing of this paper, support for OpenMP 5.0, particularly data affinity, is incomplete and experimental, so we leave the actual implementation of this feature to future work.

B. Legion

Code generation for the experimental Legion target is specialized for the particulars of the runtime in a few key ways. First, because the Legion runtime infers dependences entirely from the set of data accesses (i.e., `RegionRequirements`), it is important to provide as much information as possible to the runtime to avoid introducing false dependences between tasks. We thus extract additional information about the particularly data access type when generating the *enumeration* functions, specifically, whether a task needs more than `READ_ONLY` access for each data dependence. This allows the Legion runtime to schedule tasks that share only `READ_ONLY` dependences concurrently. We also make use of the `WRITE_DISCARD` access type, which allows Legion to provide a `PhysicalRegion` filled with uninitialized values.

Second, due to Legion's scoping rules, the Legion target does not support the self-unrolling task DAG. Instead, the entire task DAG is created at once, from a "top-level" task. This can be accomplished by projecting the entire polyhedron along the dependences, rather than in portions for each task type. The resulting loop is then inserted as an epilogue into the top-level task; in some sense, this can be viewed as introducing a new task upon which every other task depends, and then doing the regular code generation for this single task. Finally, because the Legion runtime handles the rest of the control dependence analysis, we do not generate any of the predecessor count functions.