POSTER: Automatic Parallelization Targeting Asynchronous Task-Based Runtimes

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

***Abstract*—In a post-Moore world, asynchronous task-based parallelism has become a popular paradigm for parallel programming. Auto-parallelizing compilers are also an active area of research, promising improved developer productivity and application performance. This work seeks to unify these efforts by delivering an end-to-end path for auto-parallelization through a generic runtime layer for asynchronous task-based systems. First, we extend R-Stream, an auto-parallelizing polyhedral compiler, to express task-based parallelism and data management for a broader class of task-based runtimes. We additionally introduce a generic runtime layer for asynchronous task-based parallelism, which provides an abstract target for the compiler backend. We implement this generic runtime layer using OpenMP for shared memory systems and Legion for distributed memory systems. Starting from sequential source, we obtain geometric mean speedups of 23.0× (OpenMP) and 9.5× (Legion) on a wide range of applications, from deep learning to scientific kernels.**

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

I. Introduction

Asynchronous task-based execution has gained support as a solution to the challenges of programming future exascale systems. Computations are decomposed into *tasks*, which are asynchronously submitted for execution to the *runtime system*. The runtime dynamically schedules the tasks subject to dependences, offering load balancing, locality, and scalability.

At the same time, designing and building efficient parallel applications poses a significant challenge to developers, requiring not only application expertise but also familiarity with a specific parallel framework. One area of active research is leveraging the compiler, either through implicit parallelism (e.g., OpenMP pragmas) or full auto-parallelization.

This work addresses this landscape by presenting an end-to-end framework for automatic parallelization to asynchronous task-based runtimes via polyhedral analysis:

- We modify an existing mapping path in R-Stream [1], a polyhedral optimizing compiler, to extract task-based parallelism and data management for a generic runtime;
- We design a new generic task-based runtime layer that corresponds to the polyhedral code generation;
- We provide two implementations of the generic runtime layer using OpenMP tasks [2] and Legion [3].

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.

Using benchmarks including scientific and deep learning kernels, we find the automatically generated OpenMP task code yields a geometric mean speedup of $23.0\times$ over sequential code, versus $21.0\times$ for hand-optimized do-all OpenMP and $9.5\times$ for our prototype distributed memory Legion target.

II. Generic Task-based Runtime Layer

The runtime layer is designed to capture the common paradigms of task-based parallelism and data management and act as an abstract code generation target for a polyhedral compiler workflow (described in the next section). The following is a high-level description of the runtime components.

Tasks are tiled units of computation. Each task is parameterized by a task type identifier (`taskTypeId`), which is a lightweight handle to the task function, and a task identifier (`taskId`), which is passed to the task at runtime, allowing it to determine its share of computation.

Datablocks are tiles of data. A datablock is identified by a datablock identifier (`dbTypeId`) and a list of datablock coordinates (`coords`). The specification of the size of a datablock is delayed until it is fetched during execution, to (1) support tiling of irregular spaces (e.g., triangular matrices) and (2) avoid overallocation of unused coordinates. The `dbTypeId` and `coords` allow datablocks to be passed by reference without moving the underlying data, which can be expensive on distributed memory systems.

We consider two classes of dependences:

- **Data Dependence.** When a new instance of a task is created, the runtime is passed references to all its necessary datablocks by the parent.
- **Control Dependence.** The runtime is also passed the number of predecessors of the child task.

Upon completion, a task "auto-decrements" (`autodec`) the count of its successor tasks; the runtime does not schedule a task for execution until its predecessor count has reached zero and the enumerated datablocks are coherent and available.

The use of `autodec` allows for the creation of a *self-unfolding task DAG*, whereby the frontier nodes (tasks) and edges (dependences) are dynamically created by the encountering tasks. This (1) avoids the overhead of creating and analyzing the entire static task DAG at startup, but also (2) reduces the need for dynamic dependence analysis, making the runtime layer extremely lightweight.

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00047

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

†64 threads. Execution time, in seconds.

III. POLYHEDRAL FLOW

Building upon prior work by Baskaran et al. [4], we modify an existing mapping path in R-Stream for OCR [5] to produce generic task-based parallelism and data management. This work is thus a successful proof-of-concept for targeting a wide range of task-based runtimes using a polyhedral compiler flow.

Starting from sequential source, R-Stream first performs standard **raising** and **dependence analysis** to convert sequential code into a polyhedral representation. **Scheduling** transforms this representation to expose parallelism and improve locality. Next, heuristic **tiling** of the resulting loop nests increases the granularity of parallelism while balancing data reuse, cache sizes, and runtime overhead. The tiling heuristic also selects data blocks sizes to balance control dependences against datablock management overhead. Auxiliary information extracted from the dependence polyhedra during this phase includes liveness and access requirements for each data tile, which can be passed as hints to the target runtime. Finally, **dependence generation** identifies successors for task tiles.

Code generation is performed via standard polyhedral scanning, except that tile boundaries delineate tasks. Scanning the data dependence polyhedra gives the data enumeration functions. Control dependences are inferred in both directions. Projecting the polyhedron for a task type along the directions of dependence yields a loop that invokes `autodec` for each successor, which can be inserted as an epilogue in the task function. Similarly, projecting the polyhedron against the directions of dependence gives the count function.

The final step in code generation is specialized by target to account for runtime differences. For instance, the Legion target requires all tasks to be spawned in the main task to avoid spurious dependences. The count function is also not generated as the Legion runtime directly manages control dependences.

IV. RESULTS

We present performance results in Table I for a variety of common workloads, loosely categorized into embarrassingly parallel (SpMV), deep learning (GoogLeNet, ResNet), and scientific kernels (SW4, Kripke, HOMME). We also provide timings for sequential and hand-optimized OpenMP do-all versions. Experiments were run on an 8-core (16 threads) quad socket Intel Xeon (Ivy Bridge) server with 64 threads. Code was compiled with GCC 7.3 (OpenMP 4.5). Threads were bound to sockets to reduce NUMA overhead.

The automatically generated code for the OpenMP task variant was on par with the performance of the hand-mapped OpenMP codes, even though the do-all versions benefited from richer API support for affinity and locality hints at the OpenMP runtime level. The task variant performed especially well for the scientific kernels with irregular dependences [6]. Results confirm that the generic runtime is lightweight.

The experimental Legion target, though able to deliver absolute speedups, still has much room for improvement. In particularly, targeting the Legion runtime mapper interface, which can be tuned for specific architectures and applications, will enable better coordination with the underlying Legion runtime system and improve locality and affinity. However, it is important to mention the Legion target already delivers significant gains in terms of maintainability and portability: starting from an average 96 lines of sequential source, we are able to generate code in a higher-level API for both shared (380 lines) and distributed memory systems (2315 lines).

V. CONCLUSIONS

We present an end-to-end path for auto-parallelization to asynchronous task-based runtimes. Our approach leverages R-Stream, a polyhedral optimizing compiler, to extract task-based parallelism and data management; a new generic task-based runtime layer serves as an abstract target for the compiler backend. We provide implementations for the generic runtime in OpenMP and Legion, successfully demonstrating a polyhedral compiler approach to targeting task-based runtimes for both shared and distributed memory systems. The generated OpenMP code outperforms hand-optimized OpenMP by 12.0%, particularly on irregular scientific codes. The experimental Legion target is less optimized but demonstrates automatically generating thousands of lines of scaffolding for distributed memory from tens of lines of sequential source. Future work includes refinement of the polyhedral flow for task-based parallelism, improved heuristics for distributed memory management, and additional implementations of the generic layer using task-based programming systems.

REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.