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

Charles Jin, Muthu Baskaran, Benoit Meister, Jonathan Springer

Twenty-third Annual IEEE High Performance Extreme Computing Conference (HPEC '19)

Waltham, MA

25 September 2019

Acknowledgement of Support: 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.

Motivation

- Post-Moore exascale challenges: load balancing, locality, scalability...
- Active research to address these via asynchronous task-based runtimes
- Auto-parallelization tools to reduce reliance on "hero" developer(s)

Motivation

- Post-Moore exascale challenges: load balancing, locality, scalability...
- Active research to address these via asynchronous task-based runtimes
- Auto-parallelization tools to reduce reliance on “hero” developer(s)
- This work seeks to unify the two efforts via an **end-to-end framework** for **automatic parallelization** to **asynchronous task-based runtimes**

Contributions

- Augment R-Stream, an auto-parallelizing polyhedral compiler, to express task-based parallelism and data management for a generic runtime
- Design a generic task-based runtime layer corresponding to the polyhedral output
- Provide two implementations of generic runtime using OpenMP tasks (shared memory) and Legion (distributed memory)

Contributions

- Augment R-Stream, an auto-parallelizing polyhedral compiler, to express task-based parallelism and data management for a generic runtime
- Design a generic task-based runtime layer corresponding to the polyhedral output
- Provide two implementations of generic runtime using OpenMP tasks (shared memory) and Legion (distributed memory)

End-to-end auto-parallelization of sequential source to **asynchronous task-based parallelism**

Generic Runtime Layer

Background

- Generic runtime layer aims to capture common paradigms of task-based parallelism and data management
  - Launch tasks, allocate data, and express dependences
  - Shared and distributed memory
- Also designed to serve as an abstract code generation target for a polyhedral compiler workflow

Generic Runtime Layer

Tasks

- Tiled units of computation
  - `taskTypeId`: lightweight handle to function
  - `taskId`: passed to task at runtime, allowing it to determine its share of computation

Datablocks

- Tiles of data
  - `dbTypeId`: datablock identifier
  - `coords`: a list of tile coordinates
- Passed by reference
- Size is specified when fetched

Generic Runtime Layer

Tasks

- Tiled units of computation
  - `taskTypeId`: lightweight handle to function
  - `taskId`: passed to task at runtime, allowing it to determine its share of computation
- Ex: a loop from [0, 63] is decomposed into 4 tasks covering [0, 15], [16, 31], [32, 47], [48, 63]

Datablocks

- Tiles of data
  - `dbTypeId`: datablock identifier
  - `coords`: a list of tile coordinates
- Passed by reference
- Size is specified when fetched
- Ex: a 64-by-64 array is decomposed into 4 datablocks of size 32-by-32

Generic Runtime Layer

Dependences

When a new instance of a task is created:

- **Data Dependence.** Runtime is passed references to all necessary datablocks by parent.
- **Control Dependence.** Runtime is passed number of predecessors of child task by parent.

Generic Runtime Layer

Dependences

When a new instance of a task is created:

- **Data Dependence.** Runtime is passed references to all necessary datablocks by parent.
- **Control Dependence.** Runtime is passed number of predecessors of child task by parent.

When a task completes:

- It "auto-decrements" (`autodec`) the count of **all** successor tasks.

Generic Runtime Layer

Dependences

When a new instance of a task is created:

- **Data Dependence.** Runtime is passed references to all necessary datablocks by parent.
- **Control Dependence.** Runtime is passed number of predecessors of child task by parent.

When a task completes:

- It "auto-decrements" (`autodec`) the count of **all** successor tasks.

The runtime does not schedule a task for execution until **predecessor count equals zero** and requested **datablocks are coherent and available**.

Generic Runtime Layer

Self-unfolding Task DAG

The `autodec` operation permits a self-unfolding task DAG

- Static task DAG = overhead at start up
- Dynamic task DAG = overhead for runtime dependence analysis

Generic Runtime Layer

Self-unfolding Task DAG

The `autodec` operation permits a self-unfolding task DAG

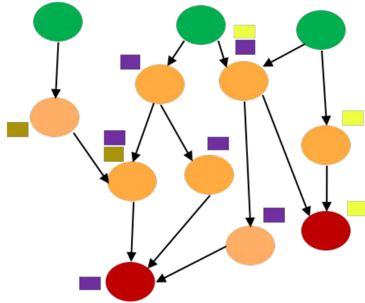
- Static task DAG = overhead at start up
- Dynamic task DAG = overhead for runtime dependence analysis
- Self-unfolding task DAG = frontier nodes (tasks) and edges (dependences) are dynamically created by encountering task

Generic Runtime Layer

Self-unfolding Task DAG

The `autodec` operation permits a self-unfolding task DAG

- Static task DAG = overhead at start up
- Dynamic task DAG = overhead for runtime dependence analysis
- Self-unfolding task DAG = frontier nodes (tasks) and edges (dependences) are dynamically created by encountering task



For frameworks which provide the functionality, R-Stream does not create the entire task DAG and datablocks at runtime initialization

Generic Runtime Layer

Self-unfolding Task DAG

The `autodec` operation permits a self-unfolding task DAG

- Static task DAG = overhead at start up
- Dynamic task DAG = overhead for runtime dependence analysis
- Self-unfolding task DAG = frontier nodes (tasks) and edges (dependences) are dynamically created by encountering task

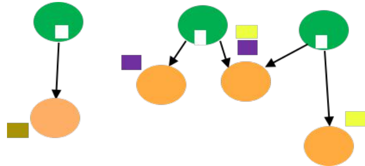
Initially, only root tasks (and their required datablocks) are created

Generic Runtime Layer

Self-unfolding Task DAG

The `autodec` operation permits a self-unfolding task DAG

- Static task DAG = overhead at start up
- Dynamic task DAG = overhead for runtime dependence analysis
- Self-unfolding task DAG = frontier nodes (tasks) and edges (dependences) are dynamically created by encountering task



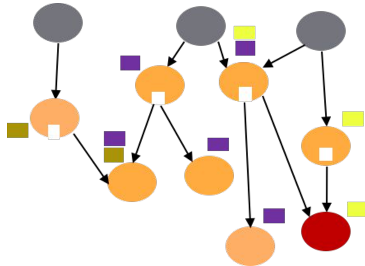
As tasks complete, the task graph "self-unfolds" to generate a frontier of uncompleted tasks, adjusting the predecessor counts

Generic Runtime Layer

Self-unfolding Task DAG

The `autodec` operation permits a self-unfolding task DAG

- Static task DAG = overhead at start up
- Dynamic task DAG = overhead for runtime dependence analysis
- Self-unfolding task DAG = frontier nodes (tasks) and edges (dependences) are dynamically created by encountering task



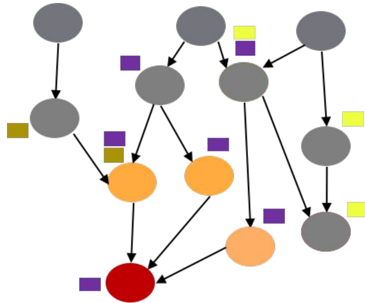
Completed tasks are freed by the runtime to keep the active space compact

Generic Runtime Layer

Self-unfolding Task DAG

The `autodec` operation permits a self-unfolding task DAG

- Static task DAG = overhead at start up
- Dynamic task DAG = overhead for runtime dependence analysis
- Self-unfolding task DAG = frontier nodes (tasks) and edges (dependences) are dynamically created by encountering task



Completed tasks are freed by the runtime to keep the active space compact

Datablocks are also freed once they are no longer needed

Polyhedral Flow

Background

- **Polyhedral model** is an optimization framework that uses a compact mathematical representation of a computation as a polyhedron
    - e.g., each iteration of a nested loop body is a point in space
    - Data and dependences represented as well
- Optimizations are applied as transformations to the polyhedron
    - e.g., loop fusion, array contraction, loop fusion, interchange, fission…
- Most precise on affine regions and access patterns

Polyhedral Flow

Polyhedral Analysis

- Starting from sequential source...
- **Raising** and **dependence analysis** generate polyhedral representation
- **Scheduling** exposes parallelism and improves locality
- Heuristic **tiling** increases granularity of parallelism while balancing data reuse, cache sizes, and runtime overhead
- **Dependence generation** identifies successors for task tiles
- Finally, **code generation** is performed via standard polyhedral scanning

Based on prior work by 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.

Results

Benchmark Performance

Geomean speedups of

- 23.0x (OpenMP task)
- 9.5x (Legion)
- 21.0x (hand-tuned OpenMP, do-all)

	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.

8-core (16 threads) quad socket Intel Xeon (Ivy Bridge) server. Compiled with GCC 7.3 (OpenMP 4.5). Threads were bound to sockets to reduce NUMA overhead.

Results

Qualitative Discussion

OpenMP task

- On-par with hand-tuned OpenMP do-all, which benefits from richer APIs for affinity and locality hints at runtime level
- Especially good on kernels with irregular dependences

Legion

- Mapper interface to better tune to architecture and applications
- Distributed overhead in a shared-memory environment

Portability

- 96 lines of sequential source
- 380 lines of shared memory parallelism
- 2315 lines of distributed memory parallelism

Concluding Remarks

End-to-end auto-parallelization to asynchronous task-based runtimes

- Proof of concept using polyhedral flow to target a generic task-based parallelism
- Both shared and distributed memory

Concluding Remarks

End-to-end auto-parallelization to asynchronous task-based runtimes

- Proof of concept using polyhedral flow to target a generic task-based parallelism
- Both shared and distributed memory

Future Work

- Refinement of polyhedral optimizations
  - Particularly for distributed memory
- Other runtimes: Kokkos, CudaGraphs, etc.

Questions?

Generic Runtime Layer

High level API

Predecessor count function: parameterized by `taskTypeId` and `taskId`, returns number of predecessors.

Datablock enumeration function: parameterized by `taskTypeId` and `taskId`, fills child context with requested `dbTypeId` and `coords`.

Autodec: accepts as input the (1) child `taskTypeId` and `taskId`, (2) the predecessor count function, and (3) the datablock enumeration function.

Datablock fetch: takes `dbTypeId`, `coords`, and size; returns a region of memory for read / write.

Generic Runtime Layer

Datablock API

- Registered with the runtime
- Represented as a tiled array
- Covers both shared and distributed memory with no extra overhead

```
// 4x8 array of 5x5 tiles
// total dimensions: 20x40
declareDBType(0,        /*dbId*/
              5, 5, /*tileDims*/
              4, 8, /*numTiles*/);
```

- `fetchDB` returns a C-style array pointer for read / write
- Compiler will never generate two fetches which lead to a data race
- Implemented using target framework's primitives

```
// returns tile (1, 3)
// which is [5:10, 15:20] in
// original array
fetchDB(0,      /*dbId*/
        1, 3      /*tileId*/);
```

Generic Runtime Layer

Task API

- Registered with the runtime
- Tasks represent automatically tiled units of work from original program
- `autodec` is implemented using target framework's primitives

```
// original code
for (i = 0; i < 100; i++)
  A[i] *= 2;

// tiled tasks using generic API
declareTaskType(0,      /*taskTypeId*/
                task0   /*fn*/);

for (i = 0; i < 5; i++)
  autodec(0,   /*taskTypeId*/
          i    /*taskId*/,
          ...);

task0 (taskId, ...):
  for (i = 0; i < 20; i++)
     A[taskId * i] *= 2;
```

Generic Runtime Layer

Dependence API

- Dynamic creation of task DAG
- All predecessors try to spawn the task, but only one succeeds
- Dynamic enumeration of the required datablocks for the spawned task
- `wait`, `spawn`, (+ other context set up) implemented using target runtime primitives

```
autodec(..., predCntFn, dbEnumFn):
  taskCtx.count++;
  if (taskCtx.count = predCntFn(...)) {
     dbEnumFn(taskCtx, ...);
     wait(taskCtx.dbs);
     spawn(taskCtx);
  }

predCntFn(taskTypeId, taskId, ...):
  // returns number of predecessors
  // for the given task

dbEnumFn(childCtx, taskTypeId, taskId):
  childCtx.addDB(...);
  childCtx.addDB(...);
```

Polyhedral Flow

Code Generation

Tiling

- Tile boundaries for iteration spaces = tasks
- Tile boundaries for data arrays = datablocks

Dependence generation

- Control dependence polyhedra
  - Projecting along direction of dependence = autodec
  - Projecting against direction of dependence = predecessor count function
- Data dependence polyhedra = datablock enumeration function

Final code generation of generic APIs is specialized by target runtime