Automatic Optimization and Code Generation for Asynchronous Task-Based Runtimes

Muthu Baskaran
Reservoir Labs

CnC 2021: The Thirteenth Annual Concurrent Collections Workshop
October 27–28, 2021 at SUNY Global Center

CONTEXT

Addressing HPC/Exascale Challenges

Challenges in Petascale, Exascale, and beyond – with increasing complexity

- Performance
  - Parallelism, locality, load balancing, algorithmic scalability
  - Latency of local & remote memory accesses
- Productivity
  - DSLs, with their flexibility vs performance tradeoff
  - Parallel debugging
- Hitting some hardware boundaries
  - Process scaling continues
  - But energy envelope is bounding HW capabilities
  - Node failures

Addressing HPC/Exascale Challenges

Working around power constraints

- Lower voltage as much as possible
  - Near Threshold Voltage
    - Performance variability across PEs increases
    - Heterogeneity, even in a homogeneous array of PEs
- Increase parallelism as much as possible, lower frequency
  - Use of hierarchies to get to scale
    - Affects latencies
  - Fork-Join, Loop parallelism often not enough to produce that much concurrency
- Fine-grained power controls for SW optimization
  - Need to optimize program for use of HW knobs (Power API)
- Explicitly managed memories and communications
  - Need to generate complex data decomposition and movement

Addressing HPC/Exascale Challenges

Direct impact on software requirements

- Parallel programming model must enable
  - Fine-grain load balancing
  - Non-loop (task) parallelism
  - Hiding long memory latencies (even in loop codes)
- Productivity is key (in addition to performance and other factors)
  - Programming models allowing to express separation of concerns
  - Tools enabling a productive ecosystem
- One of the widely adopted concept: Task-based runtimes
  - Declare tasks and their dependences
  - Tasks are scheduled asynchronously
    - Work-stealing variants

Embracing the Trend with HPC Community

Our first target to task-based runtime concepts

- Collaboration with the CnC team to target advanced concepts via R-Stream
  - Regular sessions with Kath Knobe and her team

Continued the collaboration with the community

- DOE X-Stack and FF2 programs
- Brainstorming concepts and developing tools with
  - OCR team (Rice and Intel)
  - ETI SWARM
  - Various teams in the community working on runtimes

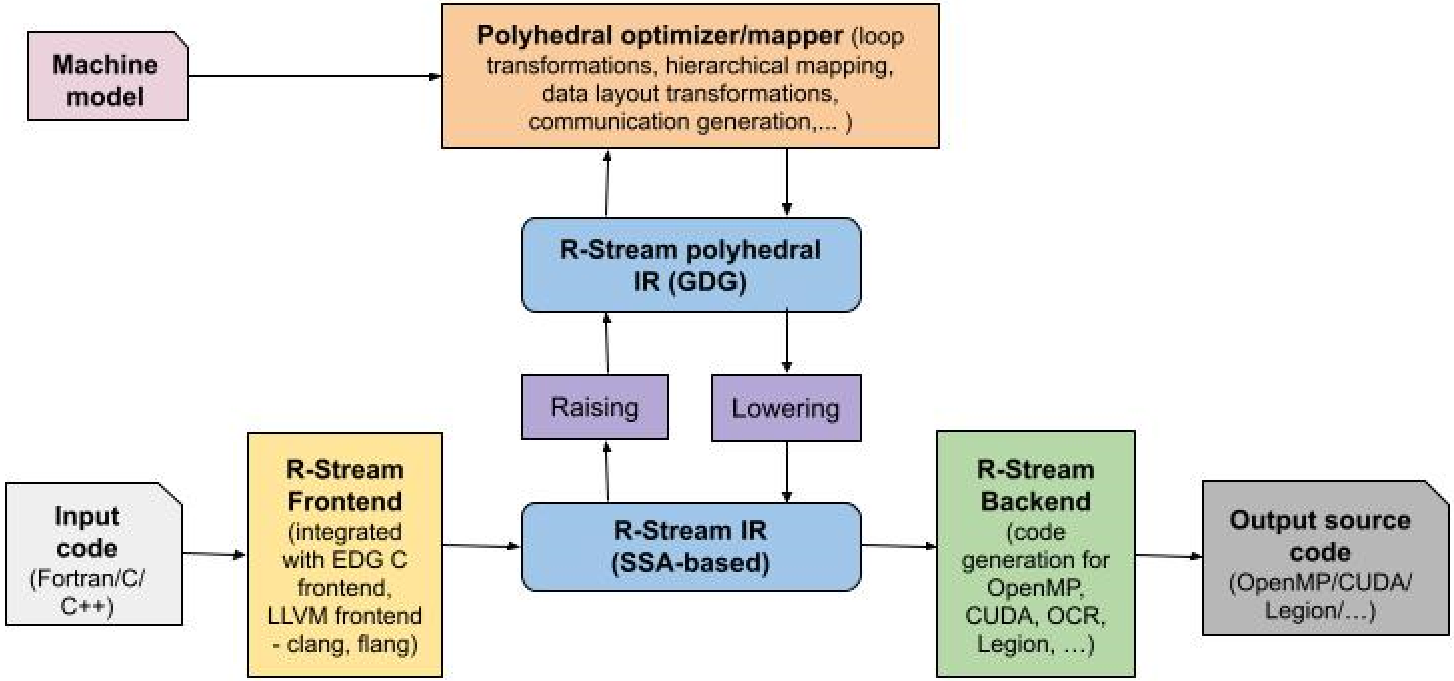
R-Stream Extensions for Addressing Exascale Challenges

Developed techniques to extend R-Stream to map to Exascale

- Developed new techniques to generate code for multiple asynchronous task-based runtime models (CnC, OCR, SWARM, ...)
  - Ability to target deep memory and processor hierarchies
- Techniques generic to be applied broadly and extended across different models and targets
  - Legion/Realm
  - PaRSEC
  - Task-based runtime for GPUs
  - OpenMP tasks
- Developed new optimizations for energy efficiency
  - Power controls (DVFS) through a runtime Power API
  - Hierarchical energy-proportional scheduling and many more ...

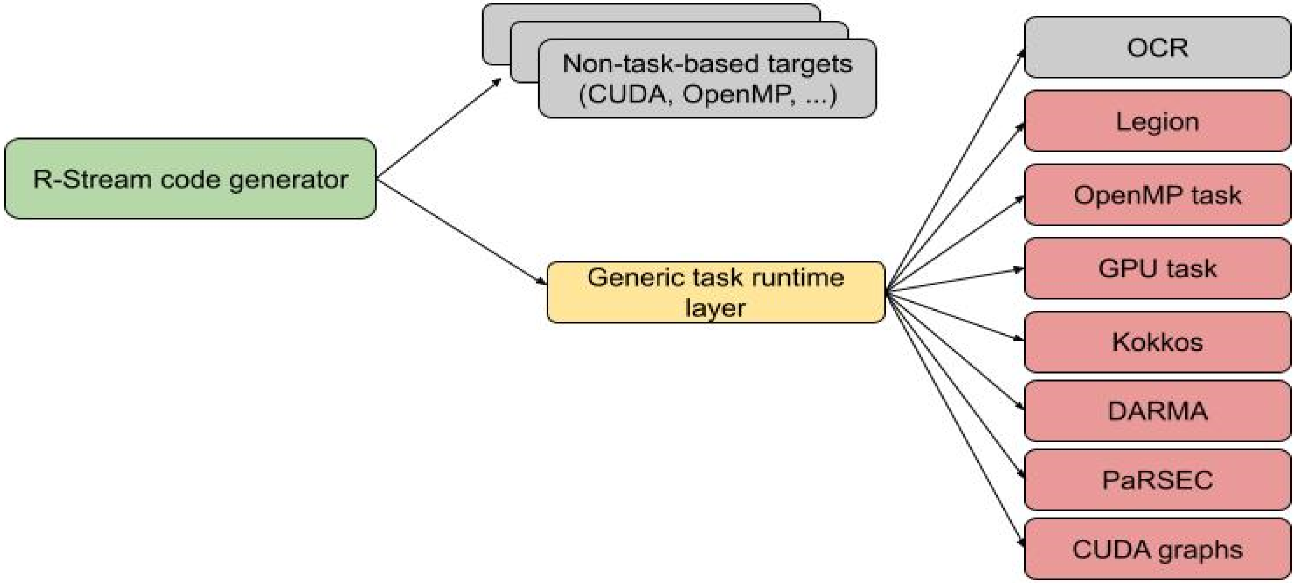
R-Stream Extensions

- Augment R-Stream to express task-based parallelism and data management for a generic runtime



R-Stream Extensions

- Augment R-Stream to express task-based parallelism and data management for a generic runtime
- Design a generic task-based runtime layer corresponding to the polyhedral output



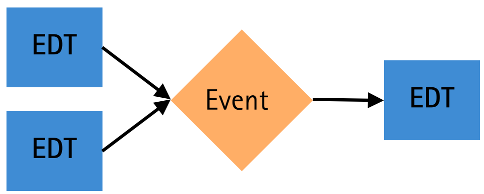
R-STREAM + OCR

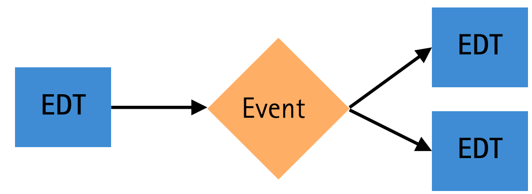
OCR Paradigms

Event-driven task (EDT) runtime

- Tasks (EDTs), Data Blocks (DBs), events, policy domains (nodes)

Control dependences





Data dependences

Runtime support

- EDTs start when their dependences are satisfied
- EDTs acquire all the DBs they need before starting

R-Stream Optimizations for OCR – The Big Picture

R-Stream techniques concretely offer

- Automatic code generation and optimization
  - Parallelism, Data locality
- Explicit management of hierarchical communications
- Scalable asynchronous EDT execution
  - Tasks, dependences, and data blocks are created on-the-fly
- Positive interference with runtime through hints and affinity
- Improved capability for testing and developing runtime features

Demonstrated through generation of optimized code

- HPCG, HPGMG, CoSP2
- SW4
- Stencils (2D, 3D, 4D), linear algebra kernels, ...

Automatic Generation of Runtime Paradigms

Decomposition of computations and data through tiling

- Automatic computation partitioning to form tasks
- Automatic data partitioning to form datablocks
- Transformations including tiling ensure
  - Extraction of enough parallelism and good data locality

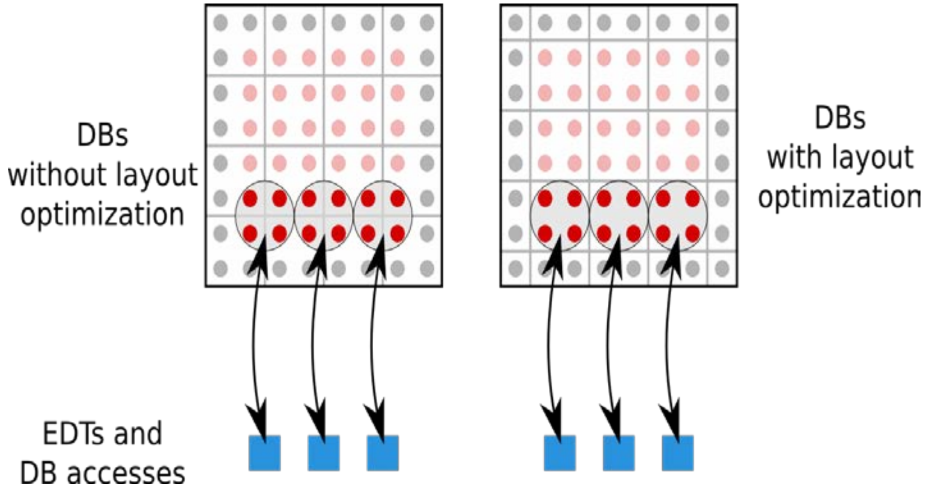
Control and data dependences

- Task graph – tasks and their dependences
- Dependences between DBs and EDTs
  - Enumeration: EDT x depends on DB y
  - Fetching: DB coordinates to data pointer

Optimization on Data Decomposition

Choose a DB layout according to read/write pattern

- greatly reduce the number of DBs an EDT depends on



Characterize DB accesses

- reuse factor of DBs
- communication volume and pattern of DBs
- Dominant DBs

Data Placement

Several data distribution strategies

- Blocked
- Block-cyclic
- Round-robin
- No affinity: OCR automatically handles data placement

Use OCR hint (affinity) to place DBs on nodes

Implicit load balancing

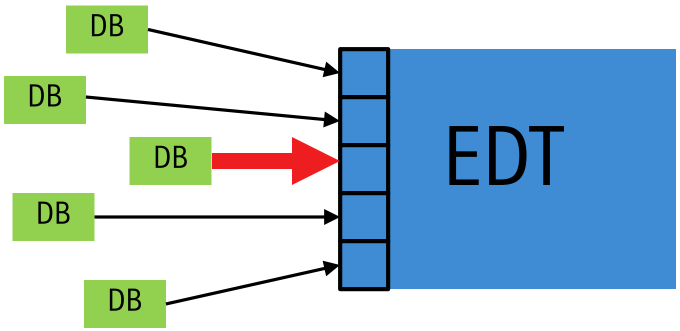
- EDT-DB affinities: EDTs are co-located with one DB

Explicit data placement has a significant impact on performance

Automatic Generation of OCR Hints

Positive interference with runtime

- Compiler-generated “hints” and “affinity” to the runtime



Useful hints

- DB-EDT affinity
- DB-Policy Domain affinity
- EDT priority

GENERIC RUNTIME SUPPORT

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
- 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

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

```
// 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(...);
```

KEY CAPABILITIES

Scaling Task Dependence Computations

Loops have inter-task (outer) and intra-task (inner) dimensions

State of the art

- Produce a dependence polyhedron
  - Tiled iteration spaces
- Project out intra-task dimensions

Computation of task dependence was too slow

- Tiled dependence polyhedron dimensionality can be high
- Projection is relatively expensive

Scaling task dependence computations using pre-tiling iteration spaces

On-the-fly Task Creation

Single node: *first predecessor that is done*

- Decrement successor counter but create it if necessary
  - "Autodec" operation
  - Based on atomics

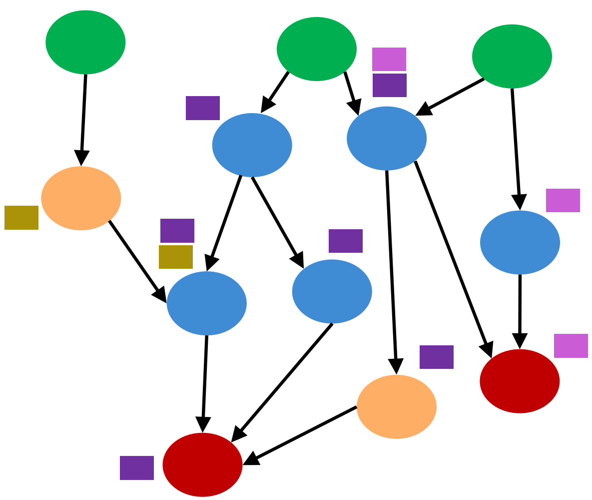
Multi-node: *agreed upon* predecessor

- All predecessors must know it statically **to avoid syncs**
- E.g., lexicographic min of the predecessors
  - But PILP is costly, can produce ugly code
- **Lexico min can be computed at runtime**
  - Early-exited loop
  - Cheap, readable

Self-unfolding Tasks and Data

Dynamic creation of tasks and data blocks

- Scalable & Flexible
  - Tasks, data blocks, and dependences created on-the-fly *as needed*
    - *"autodecs" (counted dependences and decrement with automatic creation)*
  - Minimum runtime overhead (space, in-flight work, garbage collection)

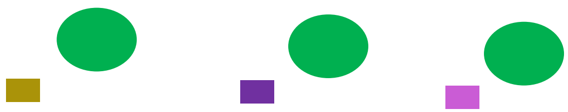


R-Stream does not create the entire task graph and data blocks at the beginning

Self-unfolding Tasks and Data

Dynamic creation of tasks and data blocks

- Scalable & Flexible
  - Tasks, data blocks, and dependences created on-the-fly *as needed*
    - *"autodecs" (counted dependences and decrement with automatic creation)*
  - Minimum runtime overhead (space, in-flight work, garbage collection)

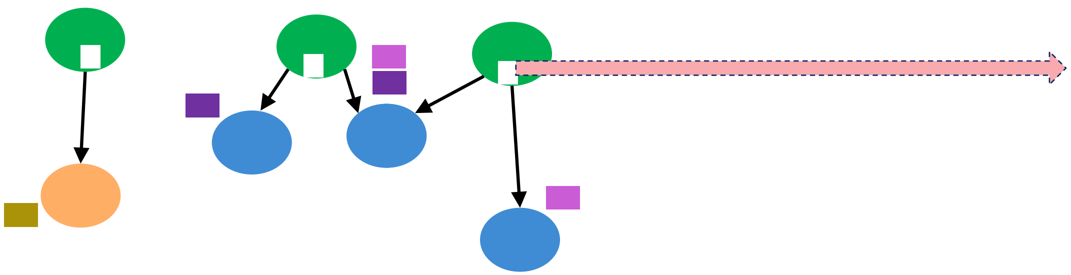


First create only tasks that do not have predecessors and create only necessary data blocks

Self-unfolding Tasks and Data

Dynamic creation of tasks and data blocks

- Scalable & Flexible
  - Tasks, data blocks, and dependences created on-the-fly *as needed*
    - *"autodecs" (counted dependences and decrement with automatic creation)*
  - Minimum runtime overhead (space, in-flight work, garbage collection)

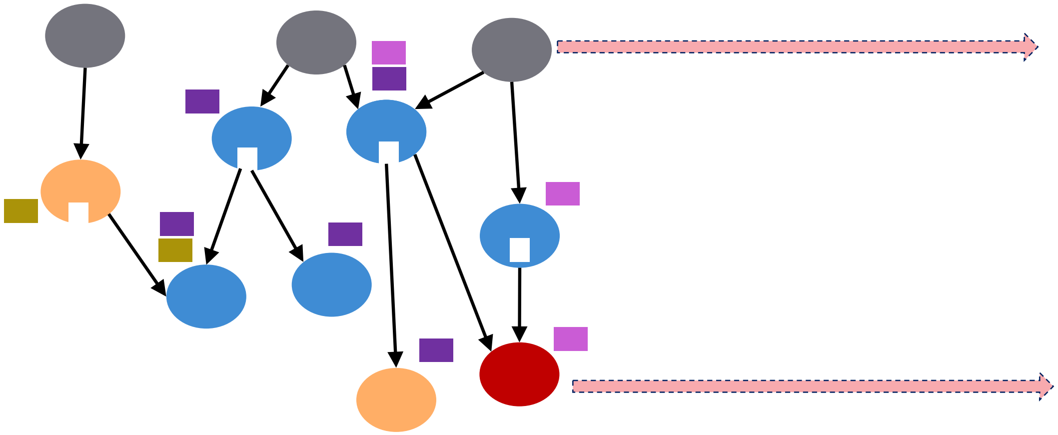


"Autodec" code that dynamically enumerates its successors and adjusts the dependence count

Self-unfolding Tasks and Data

Dynamic creation of tasks and data blocks

- Scalable & Flexible
  - Tasks, data blocks, and dependences created on-the-fly *as needed*
    - *"autodecs" (counted dependences and decrement with automatic creation)*
  - Minimum runtime overhead (space, in-flight work, garbage collection)



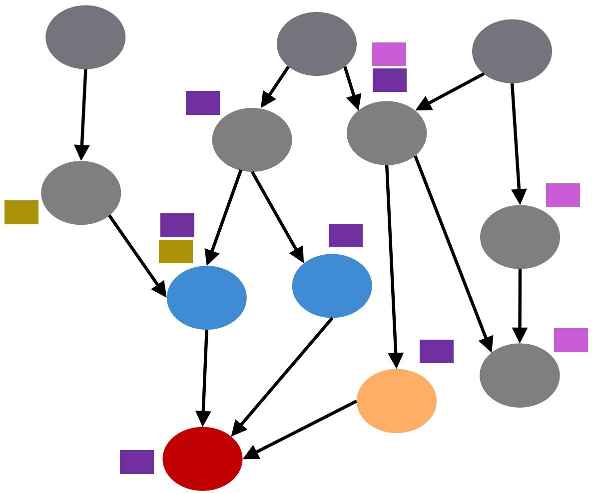
Some tasks could be freed by the runtime keeping the active space compact

Data blocks are explicitly freed as their active use is complete (does not wait until the end of the program)

Self-unfolding Tasks and Data

Dynamic creation of tasks and data blocks

- Scalable & Flexible
  - Tasks, data blocks, and dependences created on-the-fly *as needed*
    - *"autodecs" (counted dependences and decrement with automatic creation)*
  - Minimum runtime overhead (space, in-flight work, garbage collection)



In-flight work is kept to minimum (tasks freed as they are done)

Conclusions

Automatic parallelization and optimization tools

- Great productivity and performance enablers
- Need to evolve with the changing trend of architectures, programming models and runtimes
- Many challenges are addressed and a lot of interesting challenges need to be addressed

Gathering and brainstorming forums with the community is key

Would like to thank the organizers and the steering committee of this workshop!