Automatic Code Generation to Dynamic Task-Based Runtimes: Recent Results

Charles Jin*, Muthu Baskaran

Tenth Annual Concurrent Collections Workshop
Stony Brook University
Nov 30, 2018

Background

- Dynamic task-based parallelism is a leading contender as a paradigm for exascale computing
  - Runtime variations (clock speeds, node failures)
  - CnC, OpenMP, Legion, Charm++, PaRSEC, Kokkos, OCR, etc.
- However, extracting and expressing parallelism is difficult
  - Optimal decomposition depends on problem size and target system
  - Which framework is best for the characteristics of the given problem?

How can we deliver the benefits of task-based parallelism without burdening the programmer?

Contributions

1 / 2

- Key observation: many task-based runtimes use similar APIs and primitives, though the underlying implementations, optimizations, and philosophies may differ
    - Tasks
    - Data
    - Dependences / synchronization

Contributions

1 / 2

- Key observation: many task-based runtimes use similar APIs and primitives, though the underlying implementations, optimizations, and philosophies may differ
  - Tasks
  - Data
  - Dependences / synchronization

Contribution 1: a generic, lightweight task-based runtime API layer that provides a common programming abstraction for “arbitrary” frameworks

Contributions

2 / 2

- R-Stream: sequential C => parallel program
  - Automatic source-to-source parallelization and optimization
  - Productivity, performance portability, extensibility
- Techniques to extend R-Stream to exascale
  - Targets deep memory and processor hierarchies
  - Power-aware scheduling
  - Heterogeneous computing

Contributions

2 / 2

- R-Stream: sequential C => parallel program
  - Automatic source-to-source parallelization and optimization
  - Productivity, performance portability, extensibility
- Techniques to extend R-Stream to exascale
  - Targets deep memory and processor hierarchies
  - Power-aware scheduling
  - Heterogeneous computing

Contribution 2: new backends that target this generic runtime layer for automatic code generation to dynamic task-based parallelism

Talk Outline

Part 1

- Brief introduction to different task-based runtimes
- Design of generic runtime layer
- An overview of R-Stream
- Code generation techniques

Part 2

- Case study: OpenMP tasking in Kripke
- Comparison of implicit and explicit tasking in OpenMP
- Performance results

Conclusion

- Future work and directions

Task-based runtimes

OpenMP

Tasks

- Blocks marked by `#pragma omp task`
- Encountering thread gives a deferrable task to the runtime

Data

- Shared memory
- No data movement primitives

Dependences

- Specified via data read/writes
- Follow sequential semantics

```
for (i = 0; i < iMax; i++) {
#pragma omp task depend(in:  a)
                 depend(out: b)
  // task 1
}

for (j = 0; i < jMax; j++) {
#pragma omp task depend(in:  b)
                 depend(out: a)
  // task 2
}
```

Task-based runtimes

Legion

Tasks

- Multiple variants can be registered with the runtime
- Request explicit permissions and coherence on data when spawned

Data

- Distributed memory model
- LogicalRegion (handle to memory)
- PhysicalRegion (LR mapped to a physical memory location)
- Futures (return and control)

Dependences

- Inferred from data permissions
- Follow sequential semantics

```
register_task(0 /*taskId*/,
              task0);
lr = create_logical_region(...);

task_launcher(0 /*taskId*/);
task_launcher.add_requirement(
  lr,        /*data*/
  READ_ONLY, /*permission*/
  EXCLUSIVE  /*coherence*/);
future = task_launcher.launch();
future.get(); // blocking call

// deferred until lr is ready
// follows data dependence model
task0(...) {
  lr = regions.get(0);
  pr = lr.map();
  // do work on pr
  return 0;
}
```

Task-based runtimes

OCR

Tasks

- Registered with the runtime
- GUID = common handles for objects
- Data and control dependences expressed via “slots”

```
// templates are declared with
// number of in-dependences
templateCreate(
  &t0,  /*template guid*/
  task0,/*fn ptr*/
  1     /*depc*/);
templateCreate(&t1, task1, 1);

// tasks can have an out-event
taskCreate(
  &e0,  /*task guid*/
  t0,   /*template guid*/
  NULL, /*in-event guid*/
  &o0   /*out-event guid*/);

// task1 depends on the out-event
// of task0, and will receive
// whatever GUID task0 returns
taskCreate(&e1, t1, o0, NULL);
```

Task-based runtimes

OCR

Tasks
- Registered with the runtime
- GUID = common handles for objects
- Data and control dependences expressed via “slots”

Data
- Distributed memory model
- GUID (handle to memory)
- Datablock (physical instance)

Dependences
- Directed edges between tasks
- Can be satisfied by a datablock (data dependence) or another task (control dependence)

```
templateCreate(&t0, task0, 1);
templateCreate(&t1, task1, 1);
taskCreate(&e0, t0, NULL, &o0);
taskCreate(&e1, t1, o0, NULL);

dbCreate(&d0,  /*data guid*/
         data_ptr, data_sz);

// this satisfies the single
// in-dependence of task0
// with the created DB
addDependence(d0, /*in guid*/
              e0  /*out guid*/);

// task0 receives data_ptr
// returning the GUID will satisfy
// task1's dependence with a
// reference to data_ptr
task0(...){
  // do work on data_ptr
  return d0;
}
```

Generic API design

Data

- Registered with the runtime
- Represented as a tiled array (individual tiles = "datablock")
- Covers both shared and distributed memory with no extra overhead

```
// 4x8 array of 5x5 tiles
// total dimensions: 20x40
registerDB(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 API design

Tasks

- Registered with the runtime
- Tasks represent automatically tiled units of work from original program
- Explicit dependences determine scheduling contraints
- `spawn` is implemented using target framework's primitives

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

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

for (i = 0; i < 5; i++) {
  spawn(0,   /*taskTypeId*/
        i    /*taskId*/);
}

task0 (...) {
  for (i = 0; i < 20; i++) {
    A[this.taskId * i] *= 2;
  }
}
```

Generic API design

Dependences

- Dynamic creation of task DAG using “autodecs”

```
autodec(taskTypeId, taskId) {
  preds = getPred(taskTypeId,
                  taskId);
  preds--;
  if (preds == 0) {
    spawn(taskTypeId, taskId);
  }
}

task0(...) {
  for (i = 0; i < 4; i++)
    autodec(1, i);
}

// task 1 is spawned 4 times
task1(...) {
  // do work
}
```

Generic API design

Dependences

- Dynamic creation of task DAG using "autodecs"
- All predecessors try to spawn the task, but only one will succeed

```
autodec(taskTypeId, taskId) {
  // ...
  if (isSpawningPred) {
    while (preds > 0) spin();
    spawn(taskTypeId, taskId);
  }
}

task0(...) {
  for (i = 0; i < 4; i++)
    autodec(2, i);
}

task1(...) {
  for (i = 0; i < 4; i++)
    autodec(2, i);
}

// task 2 is spawned 4 times
task2(...) {
  // do work
}
```

Generic API design

Dependences

- Dynamic creation of task DAG using "autodecs"
- All predecessors try to spawn the task, but only one will succeed
- The compiler also automatically inserts a dynamic enumeration of the required datablocks for the spawned task
- `wait_for` (and other context set up) is implemented using target framework's primitives

```
autodec(..., dbEnumFn) {
  // ...
  if (isSpawningPred) {
    dbs = dbEnumFn();
    wait_for(dbs);
    spawn(task, dbs);
  }
}

task1(...) {
  for (i = 0; i < 4; i++)
    autodec(2, i, dbEnumFn);
}

// the runtime will ensure that
// the requested dbs are
// available
task2(...) {
  db0 = fetchDB(0, this.taskId);
  // do work with db0
}
```

R-Stream capabilities

- Sequential C => parallel program
  - Performs polyhedral analysis to automatically **extract dependence information** from source
  - Uses loop transformations to **expose and express parallelism**
  - Forms **tiles of the data and iteration space** to maximize locality
- Targeted compilation
  - Platforms: x86 (single node and cluster), GPU, Intel Traleika Glacier, ...
  - Generates: OpenMP, CUDA, Global Arrays, Pthreads, ...

Code generation

1 / 3

Polyhedral analysis generates "tiles" of both computations and data

- Automatic computation partitioning to generate tasks
- Automatic data partitioning to generate datablocks
- Transformations take into account target architecture to ensure parallelism and good data locality
  - Granularity of tasks
  - Size of datablocks

Automatic expression of control and data dependences

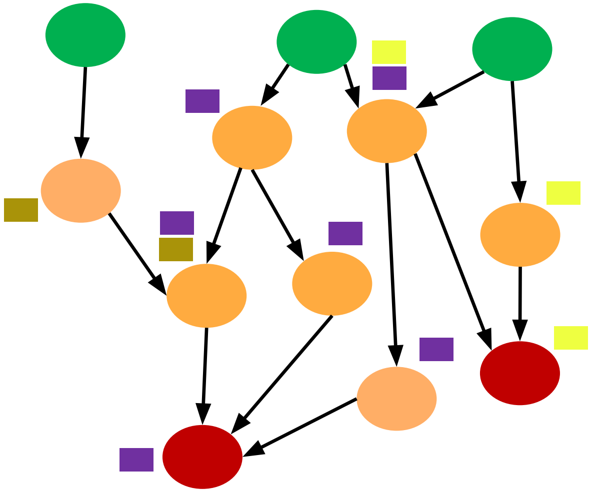
- Graph of tasks and their dependences
- Enumerating datablocks: `Task x` depends on `DB y`
- Fetching datablocks: tile guaranteed to be valid by runtime

Code generation

2 / 3

Self-unfolding tasks and data

- All information necessary for correct execution is packaged in autodec
- Dynamic creation of task DAG and data blocks
- Reduces runtime overhead on critical path



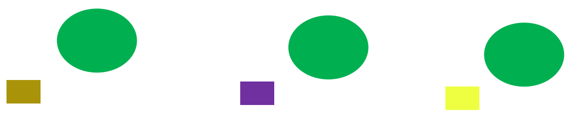
For frameworks which provide the functionality, R-Stream does not create the entire task DAG and datablocks at runtime initialization

Code generation

2 / 3

Self-unfolding tasks and data

- All information necessary for correct execution is packaged in autodec
- Dynamic creation of task DAG and data blocks
- Reduces runtime overhead on critical path



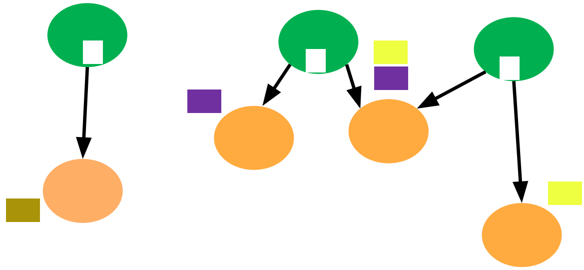
Initially, only root tasks (and their required datablocks) are created

Code generation

2 / 3

Self-unfolding tasks and data

- All information necessary for correct execution is packaged in autodec
- Dynamic creation of task DAG and data blocks
- Reduces runtime overhead on critical path



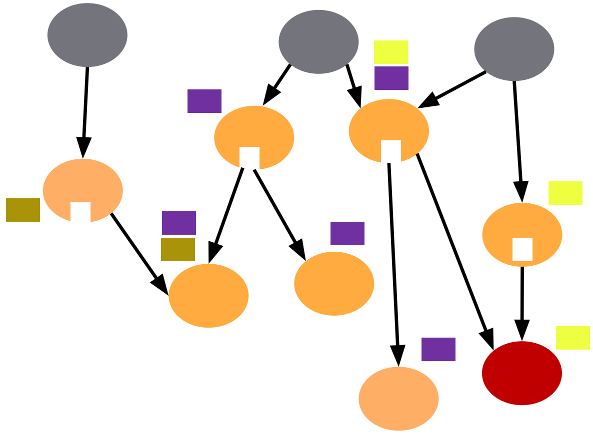
As tasks complete, the task graph "self-unfolds" to generate a frontier of uncompleted tasks, adjusting the predecessor counts

Code generation

2 / 3

Self-unfolding tasks and data

- All information necessary for correct execution is packaged in autodec
- Dynamic creation of task DAG and data blocks
- Reduces runtime overhead on critical path



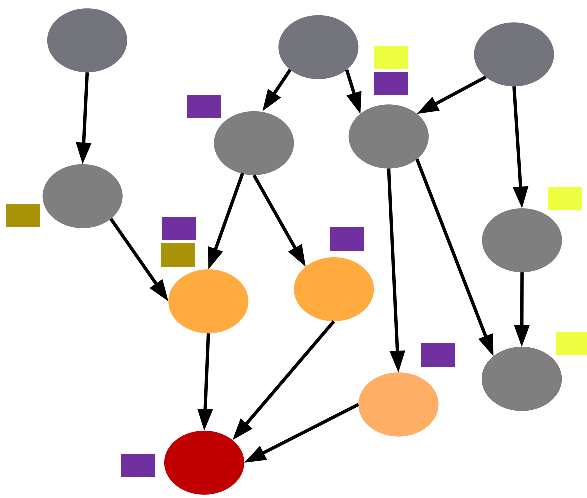
Completed tasks are freed by the runtime to keep the active space compact

Code generation

2 / 3

Self-unfolding tasks and data

- All information necessary for correct execution is packaged in autodec
- Dynamic creation of task DAG and data blocks
- Reduces runtime overhead on critical path



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

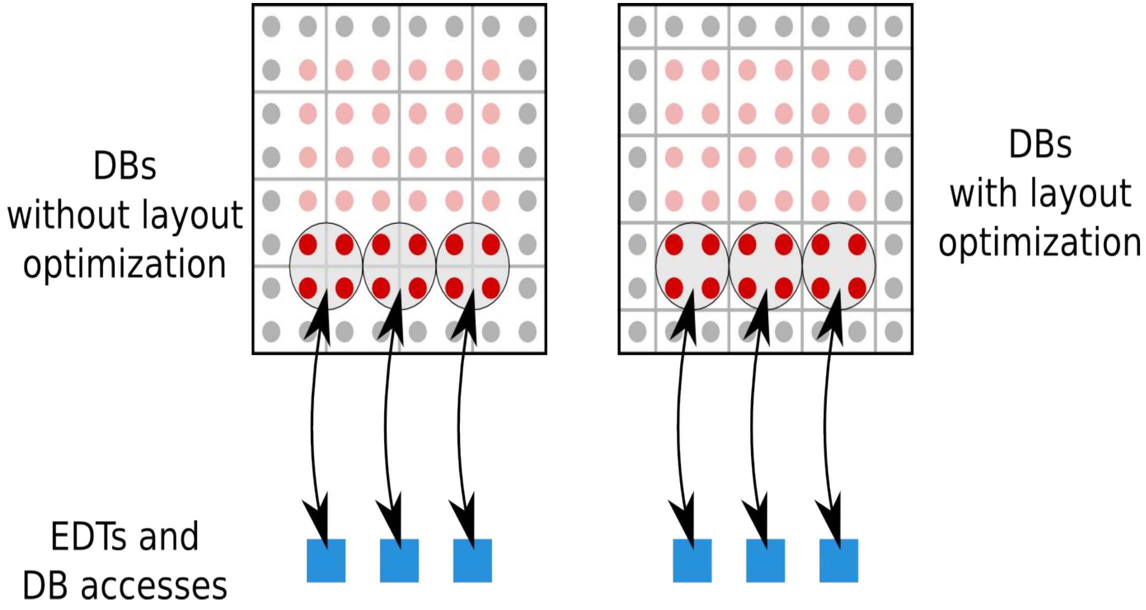
Datablocks are also freed once they are no longer needed

Code generation

3 / 3

Further optimizations

- Aligning task and data boundaries to reduce the number of overlapping data dependences between tasks



Code generation

3 / 3

Further optimizations

- Aligning task and data boundaries to reduce the number of overlapping data dependences between tasks
- Runtime hints (where applicable)
    - Task affinity (OCR, OpenMP soon-to-come)
    - Data affinity (OCR)
    - Data requirement relaxation (Legion READ_ONLY / WRITE_DISCARD)

Summary

- Architecture-specific decomposition of computation and data
- Automatic extraction and expression of task-based parallelism
- Scalable, asynchronous runtime layer
  - Tasks, data tiles, and dependences are created on-the-fly
- Runtime hints (affinity, locality)

Code generation to “arbitrary” dynamic task-based runtimes through a generic runtime API layer

Talk Outline

Part 1

- Brief introduction to different task-based runtimes
- Design of generic layer runtime layer
- An overview of R-Stream
- Code generation techniques

Part 2

- Case study: OpenMP tasking in Kripke
- Comparison of implicit and explicit tasking in OpenMP
- Performance results

Conclusion

- Future work and directions

Background

- Case study: Kripke benchmark (LLNL)
- Base implementation in MPI + OpenMP
- Scope of study is the OpenMP component (i.e. performance for shared memory parallelism)

Central question: compare the flexibility of dynamic task-based parallelism against the runtime overhead savings of do-all parallelism

Tasking in OpenMP

Implicit Tasking

- Available since initial release
- Tells compiler that the loop body can be safely workshared, i.e. no restrictions when scheduling iterations

```
#pragma omp for
for (i = 0; i < iMax; i++) {
  // loop body
}
```

Explicit Tasking

- Available in OpenMP >= 3.0
- Whenever the construct is encountered, the runtime creates a deferrable task (overhead is incurred per task)
- Runtime free to schedule subject to dependences specified via data read/writes

```
for (i = 0; i < iMax; i++) {
#pragma omp task depend(in:  a)
                 depend(out: b)
  // loop body
}
```

Kripke

1 / 2

- Mini-app developed by LLNL as a proxy for particle transport codes
- Designed to explore how data layout affects performance
- 3D position, 2D direction, 1D group => 6D unknown space for phase, so standard approach is to discretize and solve via iterative methods

$$[\Omega \cdot \nabla + \sigma(r, E)]\, \psi(r, \Omega, E) = \int dE' \int d\Omega' \sigma_s(r, \Omega' \cdot \Omega, E' \rightarrow E) \psi(r, \Omega', E') + q(r, \Omega, E)$$

Steady state form of the Boltzmann transport equation in three dimensional geometry.
See https://computation.llnl.gov/projects/co-design/kripke for additional references and details.

Kripke

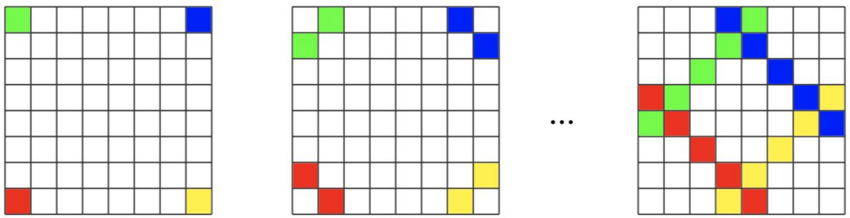
2 / 2

- Mini-app developed by LLNL as a proxy for particle transport codes
- Designed to explore how data layout affects performance
- 3D position, 2D direction, 1D group => 6D unknown space for phase, so standard approach is to discretize and solve via iterative methods
- Scalability is determined by the matrix inversion of H on the LHS, implemented using diamond differences method in the sweep kernel

$$\boxed{H}\Psi = L^{+}\Sigma_s L\Psi + Q$$

Sweep kernel

- Sweep in 2D: 4 independent wavefronts starting from each corner
- e.g. for the sweep starting from `(0,0)`, `(i,j)` depends on
  - `(i-1,j)` and `(j-1,i)` from current iteration
  - `(i+1,j)` and `(j+1,i)` from previous iteration
- Kripke sweeps over 3D position, with one independent wavefront per (group, direction) pair (termed "subdomain")



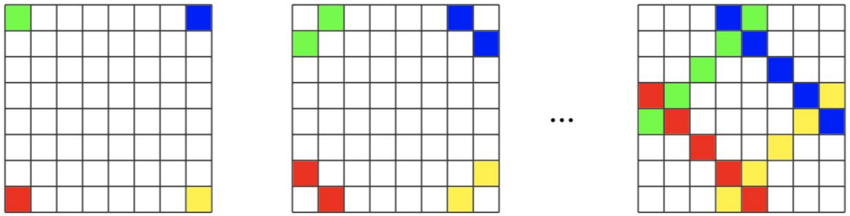
Implicit implementation

1 / 3

```
// nIter   number of iterations
// nSdom   number of subdomains
// iMax    length in i dimension
// jMax    length in j dimension
// kMax    length in k dimension
diags = iMax + jMax + kMax - 2;

for (n = 0; n < nIter; n++)
#pragma omp for
  for (s = 0; s < nSdom; s++)
    // sweep kernel
    for (d = 0; d < diags; d++)
#pragma omp for
      for (i + j + k = d)
        sweepInner(s, i, j, k);
```

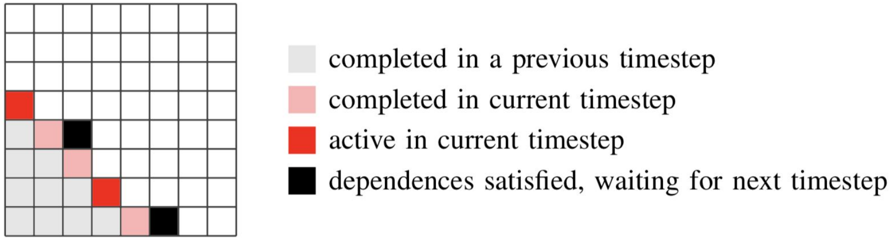
- The implicit implementation uses the `omp for` construct
- Independent sweep per subdomain
- Within each sweep diagonal, the the blocks are independent
- Overhead for a given subdomain is incurred once per diagonal



Implicit implementation

2 / 3

- The `omp for` construct contains an implicit barrier at the end of the block
- => artificial dependence between the sweep diagonals
- => limits the amount of parallelism the runtime can exploit



Implicit implementation

3 / 3

- Thread affinity is an important consideration on NUMA machines
- Pin threads for a given subdomain (= an independent sweep) to the same socket to reduce overhead
- Share work between threads on the same socket to increase parallelism

```
// export OMP_PLACES=sockets
// export OMP_PROC_BIND=spread,master

for (n = 0; n < nIter; n++)
#pragma omp parallel for num_threads(8)  // spread over sockets
  for (s = 0; s < nSdom; s++)
    for (d = 0; d < diags; d++)
#pragma omp parallel for num_threads(8)  // tied to master socket
      for (i + j + k = d)
        sweepInner(s, i, j, k);
```

Explicit implementation

1 / 3

```
// there are 8 sweep directions
// sweep from (0,    0,    0)
//       to   (iMax, jMax, kMax)

for (i = 0; i < iMax; i++)
  for (j = 0; j < jMax; j++)
    for (k = 0; k < kMax; k++)
#pragma omp task
  depend(in: A[i-1][j][k],
             A[i][j-1][k],
             A[i][j][k-1])
  depend(out:A[i][j][k])
      sweepInner(sdom, i,j,k);
```

- The explicit implementation uses the `omp task` construct
- Each subdomain has one sweep direction, and each subdomain can be done in parallel (not shown)
- The runtime is free to schedule the tasks subject to the data dependencies
- Overhead incurred once per task

Explicit implementation

2 / 3

```
// first jMax tasks are not
// parallel
for (i = 0; i < iMax; i++)
  for (j = 0; j < jMax; j++)
#pragma omp task
  depend(in: A[i-1][j],
             A[i][j-1])
  depend(out:A[i][j])
    sweepInner(i,j);
```

- Problem: overhead to create the entire dependence graph at startup can be very high
- OpenMP's solution: if the queue of deferred tasks is large, the master thread may stop spawning tasks and switch to execution
- But this means that if the parallelism isn't exposed until later tasks, you may not get any parallelism at all!

Explicit implementation

3 / 3

```
// 2nd and 3rd tasks are already
// parallel
for (d = 0; d < diags; d++)
  for (i + j = d)
#pragma omp task
  depend(in: A[i-1][j],
             A[i][j-1])
  depend(out:A[i][j])
      sweepInner(i,j);
```

- Better solution: apply loop transformations to help the runtime extract parallelism
- Conclusion: though OpenMP pragmas make it easy to ensure correctness, getting optimal performance still requires non-trivial tuning of source

Results

1 / 2

- Hybrid variant uses both implicit and explicit tasking to reduce overhead to one parallel region per iteration, but still synchronizes after diagonals

	Small Problem	Large Problem
Serial	17.036	485.348
Explicit tasking*	**2.760***	**70.841***
Explicit tasking (code gen)		
Implicit tasking (pure)	4.044	77.073
Implicit tasking (hybrid)	3.178	75.204
Implicit tasking (pure, code gen)		

* best performance over all explicit tasking variants (see paper for further details).

Results are average execution time (in seconds) over 10 runs on an 8-core (16 threads) quad socket Intel Xeon (Ivy Bridge) server using 64 threads; compiled with GCC 7.3 (OpenMP 4.5). Large problem is ~32x size of small problem.

Results

2 / 3

- Codegen for implicit version is pretty good!
- Codegen for explicit version is a WIP: task affinity; conservative copy ops

	Small Problem	Large Problem
Serial	17.036	485.348
Explicit tasking*	**2.760***	**70.841***
Explicit tasking (code gen)	**3.294**	
Implicit tasking (pure)	4.044	77.073
Implicit tasking (hybrid)	3.178	75.204
Implicit tasking (pure, code gen)	**3.284**	

* best performance over all explicit tasking variants (see paper for further details).

Results are average execution time (in seconds) over 10 runs on an 8-core (16 threads) quad socket Intel Xeon (Ivy Bridge) server using 64 threads; compiled with GCC 7.3 (OpenMP 4.5). Large problem is ~32x size of small problem.

Results

3 / 3

Overhead

- Hybrid: once per subdomain, per iteration (**GOOD**)
- Implicit: once per diagonal, per subdomain, per iteration (**OKAY**)
- Explicit: once per task (**NOT GOOD**)

Results

3 / 3

Overhead

- Hybrid: once per subdomain, per iteration (**GOOD**)
- Implicit: once per diagonal, per subdomain, per iteration (**OKAY**)
- Explicit: once per task (**NOT GOOD**)

Parallelism

- Explicit: complete dependence graph available to runtime (**GREAT**)
- Implicit, Hybrid: synchronization barrier after each diagonal limits parallelism, load balancing (**NOT GOOD**)

Results

3 / 3

Overhead

- Hybrid: once per subdomain, per iteration (**GOOD**)
- Implicit: once per diagonal, per subdomain, per iteration (**OKAY**)
- Explicit: once per task (**NOT GOOD**)

Parallelism

- Explicit: complete dependence graph available to runtime (**GREAT**)
- Implicit, Hybrid: synchronization barrier after each diagonal limits parallelism, load balancing (**NOT GOOD**)

Code generation

- No manual decomposition of tasks (**GREAT**)
- No manual targeting of architecture (**GREAT**)
- Room still to improve on performance (**OKAY**)

Talk Outline

Part 1

- Brief introduction to different task-based runtimes
- Design of generic layer runtime layer
- An overview of R-Stream
- Code generation techniques

Part 2

- Case study: OpenMP tasking in Kripke
- Comparison of implicit and explicit tasking in OpenMP
- Performance results

Conclusion

- Future work and directions

Future work and directions

Current work

- Extend support to distributed memory systems via cross-runtime interactions and hierarchical parallelism
- Extend support to heterogeneous architectures (e.g. GPUs)

Future directions

- OpenMP 5.0: task `affinity`, more expressive `depend` clause
- Support for additional runtimes (CnC, Kokkos)
- Support for “structured” tasks (Charm++, Legion)

Questions?