Compiling queries for high-performance computing

Brandon Myers, Mark Oskin, Bill Howe
University of Washington
{bdmyers, oskin, billhowe}@cs.washington.edu

Abstract

Data-intensive applications motivate the integration of high-productivity query languages with high-performance computing runtimes. We present a technique *Compiled parallel pipelines* (CPP) for compiling relational query plans to programs suitable for high-performance computing platforms. Rather than compose a sequential query compiler with a high-performance communication library like MPI, we take a holistic approach that leverages the capabilities of parallel languages. For each pipeline in the query plan, CPP generates a parallel *partitioned global address space* (PGAS) program. This approach affords modular design, and it allows the compiler to reason about whole pipelines that include parallelism and communication. Using PGAS to efficiently execute queries requires designing efficient shared data structures, generating code that avoids extra messages, and mitigating the overhead of an execution model based on fine-grained tasks.

We implement our technique as a system called RADISH. Our evaluation shows that CPP is $5.5\times$ faster than compiled iterators on TPC-H queries. To show that RADISH is a practical system for in-memory analytics, we also compare the performance of RADISH on TPC-H with the MPP system DBX and find it to be competitive. Our work takes important first steps integrating query processing and distributed HPC.

1. Introduction

Data-intensive applications are motivating new interactions between the models of databases and the algorithms and platforms of high-performance computing. One such case is early, in-situ analysis of output from a large scientific simulation. Considering the cost of development, ad hoc analytics tasks are a severe mismatch for the distributed programming models of HPC. When analytics tasks can be expressed as collection-oriented dataflow, there is a cost to using them: (1) existing dataflow systems are not equipped to make efficient use of HPC environments characterized by fast interconnects, fast messaging libraries, and high CPU to IO capacity [1]; and (2) crossing system boundaries incurs the cost of impedance mismatch in data representation. Hand-tuned algorithms continue to have a role to play in production applications; intermingling high-level dataflow programming with point-to-point message-passing algorithms is an emerging requirement [2].

To address these costs, we present new techniques for compiling queries for distributed HPC environments, aiming to exploit both database-style algebraic optimization and the performance benefits of parallel compilers. We target partitioned global address space (PGAS) languages (e.g., Chapel [3], X10 [4], and [5]), which provide shared-memory programming as well as partitioned memory to enable the programmer (and compiler) to perform locality-aware communication optimizations. By targeting PGAS we exploit (1) *language constructs* for flexible parallel execution, data layout, and task migration, (2) *runtimes* built upon the high-performance messaging standard MPI, and (3) *compilers* that are parallel aware and offer optimizations that complement those of databases.

A distributed query compiler for PGAS environments requires the design of a new architecture for distributed query evaluation, raising a number of challenges. First, expressing query execution in distributed memory using the shared memory model results in expensive fine-grained global memory operations. Second, although use of shared data structures enables flexible execution and optimization, fine-grained sharing has a cost, exacerbated greatly by distributed memory. Third, up until now a distributed query execution strategy that capitalizes on the capabilities of the PGAS languages and compilers does not exist. Previous distributed query processing systems that incorporate query compilation do so by generating sequential program fragments and stitching them into a distributed program using calls to a networking library [6, 7, 8]. Applying this existing approach to generation of PGAS code would obfuscate optimization opportunities around parallelism and communication the PGAS compiler was designed to exploit.

In this paper we introduce *Compiled parallel pipelines* (CPP), a technique that generates efficient parallel programs. We implement CPP in a new parallel query processor called RADISH. To evaluate our technique and implementation, we compare the performance of code generated by CPP to that of *Compiled Volcano iterators* (CVI), a query processor built in PGAS using the iterator model. CVI uses the same query plan and same data structures. It also has compiled expressions and tuple materialization so that we eliminate interpretation overheads and focus our evaluation on the novel aspects of CPP relative to other query compilers.

To evaluate whether our implementation of RADISH achieves acceptable performance, we also compare it to DBX, a commercial parallel database (or, *MPP*) that uses code generation, using the popular benchmark TPC-H.

We make the following contributions:

1. CPP, an approach for translating queries into efficient PGAS code leveraging existing parallel compilers
2. A library of distributed data structures for PGAS query processing supporting efficient fine-grained operations
3. An end-to-end, open source implementation integrating a distributed query optimizer (RACO) and a HPC runtime system (Grappa) built upon MPI
4. An experimental evaluation of CPP compared to a conventional iterator strategy. On TPC-H queries, executing queries by CPP is on average $5.5\times$ faster than CVI. We also compare RADISH with a state-of-the-art database platform on TPC-H queries and find that its performance is competitive.

2. Background: PGAS languages

PGAS languages are a class of shared memory languages for programming distributed memory clusters. Their defining feature is that the partitioning of the shared address space across nodes of the cluster is exposed. Partitions allow the programmer and language compiler to reason about locality while still realizing the productivity benefits of a shared address space model. The high level of PGAS languages—relative to message passing libraries—enables programmer productivity and the opportunity for distributed-aware compiler and runtime optimizations [3, 4, 9, 10, 11].

We target PGAS[1] languages as our runtime model for four reasons: First, the model provides a means for data layout, a flexible execution model based on task-level parallelism, and support for migrating tasks, which we use for communication. Second, PGAS runtimes are built on MPI [12] or other HPC messaging APIs, for which most high-performance networks have an optimized implementation. Third, compilers for these languages are distribution and parallelism aware and may increasingly offer optimizations that complement the algebraic optimization techniques associated with databases, especially amidst UDFs. Fourth, targeting shared memory code and relying on a compiler for certain low-level optimizations simplifies the design of the system, making a comprehensive distributed query compiler feasible.

The techniques in this paper will refer to our abstract model of a PGAS language in Table 1. In addition to these language constructs, our model includes the features of a typical shared memory language: local heaps, stacks, and control structures.

3. Query evaluation in PGAS

Consider the following SQL query:

[1]more precisely, we refer to *global-view PGAS* [3] as PGAS in this paper

```
select cnt, c_name
from Customer,
 (select count(*) as cnt, o_custkey
   from Order
   where o_totalprice > 20
   group by o_custkey) CountByCust
where c_custkey = o_custkey;
```

We illustrate the steps of RADISH's code generation for this query in Figure 1. In (i), a logical plan is formed and optimized by pushing a projection operator. In (ii), the PGAS physical plan is chosen: in this case, hash join and hash aggregate are converted to a combination of `Shuffle` to partition tuples by key and hash table algorithms to perform the matching within each partition. The query optimizer also applies distribution-aware optimizations common to MPP databases, such as removing a redundant `Shuffle` on `o_custkey`. (iii) shows a candidate program emitted by RADISH. Within each pipeline, the operator graph is traversed to emit PGAS code that evaluates the operator and materializes an intermediate or a final result. The high-level algorithm this program implements is essentially a parallel hash join followed by a parallel hash aggregate.

Pipelines run concurrently via calls to `spawn` (lines 5, 11, 18) and their interdependencies are encoded by calls to `sync` (lines 19 and 20). All inputs and intermediate results are stored in global arrays distributed across partitions using the data structure in Figure 2. These data structures are updated using `atomic` operations and iterated over in parallel using `forall`. Third, both `Shuffles` are translated to `on partition` calls; the execution of the iteration "moves" (as a continuation) to the partition where the accessed element is stored. Accessing data inside the continuation implies communication. For example, `t0` is read on line 9 and because `t0` originally resides on a different partition determined by the distribution of `Order`, `t0` must be transferred between partitions.

Figure 3 shows the end-to-end system. RADISH sends the emitted PGAS code and RADISH data structures through the PGAS compiler, which links with the PGAS runtime to generate machine code. Low level optimizations performed by the PGAS compiler are complementary to those of RADISH. Depending on the quality of code emitted by RADISH, a number of higher level optimizations (e.g. communication avoidance) can also be applied, although we do not explore them in depth in this paper.

RADISH receives an optimized query plan from the RACO cost-based optimizer. Its plan rewriting library includes logical heuristics (moving filters, applies, and projects, join ordering, symmetric or build/probe hash join), as well as distributed heuristics (avoiding redundant `Shuffle` and using broadcast for cross product with a small relation). We extended RACO to make choices specific to RADISH, such as hash table sizing using rough unique value statistics on single columns.

`spawn t body`	Asynchronously run `body` as task `t` on the local partition
`t.sync`	Block until task `t` is finished
`forall i in I body`	Run an iteration of `body` for each element in `I`. Iterations *may* run in parallel, so they are allowed to synchronize with each other but may not have inter-dependences, such as a barrier. The iteration for element I_j will start running on the partition where element I_j resides.
`on partition(L) body`	Move to the partition `L` before executing `body`. `L` may be a constant expression or dynamically evaluated expression of address type. If there are further statements after an `on partition`, then the task moves back to the original partition.
`global_new [T](N)`	Allocate global array where elements have type `T` and where the `N` elements are distributed evenly between partitions in a block pattern
`global_delete globalptr`	Free the global data pointed to by `globalptr`
`atomic { body }`	Execute `body` atomically

Table 1: PGAS model to target with code generation

3.1. Physical operators

The RADISH physical algebra includes operators based on two interfaces for global data structures and two granularities of synchronization. Global data structures may provide implementations of one or both of the following interfaces: *global-view* and *partition-view*. Global-view updates encapsulate communication and partition-view updates touch only the local partition. In Figure 1(ii), we have shown only partition-view for clarity: by using operators that do partition-view updates, the `Shuffle` operators are explicit in the physical query plan, allowing for communication optimizations during query optimization. Since `Shuffle` is pipelined, data structures with an partition-view update method do not typically require a global-view update method: a partition-view update composed with a `Shuffle` generates the code for a global-view update. Thus, a global-view update method is used when the data structure requires other kinds of communication or the target language happens to provide an implementation (e.g., Chapel has distributed associative arrays).

These operators include those with *tuple-grain synchronization* and *relation-grain synchronization*. Tuple-grain synchronization is where individual tuples are added to a shared data structure. Relation-grain synchronization is the situation where a full intermediate result can be written before it is read. `HashTableAggregateUpdate` / `HashTableAggregateScan` (Figure 1(ii)) involves both types of synchronization and fully pipelined operators like symmetric hash join (see subsection 3.4) use only tuple-grain.

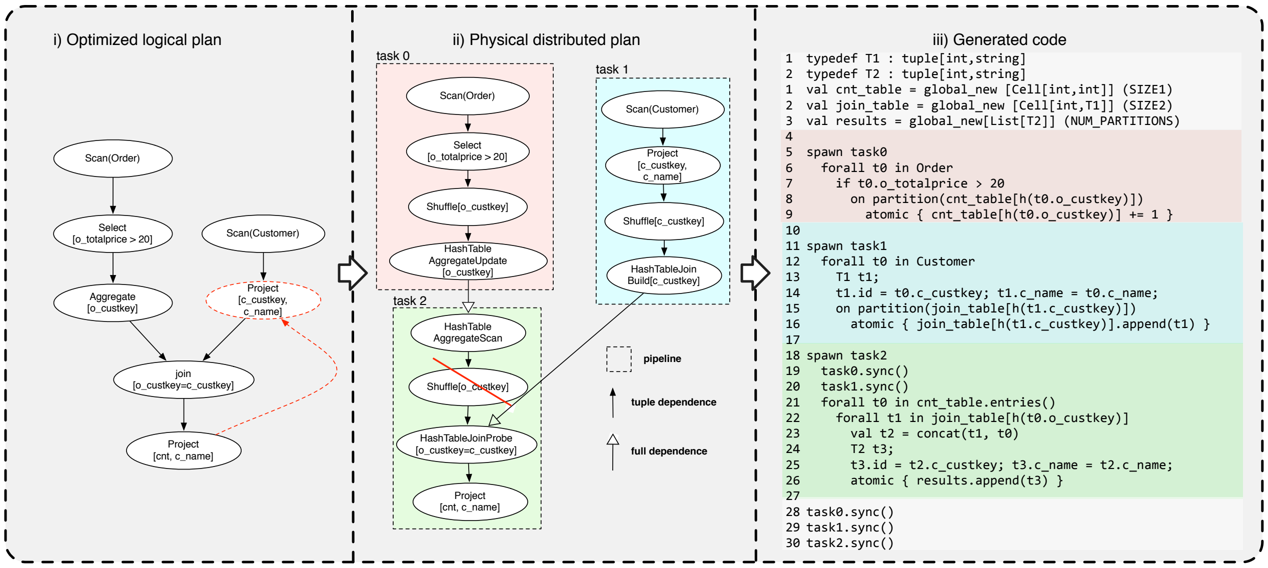
Relation-grain synchronization breaks the physical plan into pipelines. Figure 1(ii) shows the plan for the customer order count query. Black arrows indicate tuple dependences, where pipelining may occur, and white arrows indicate full materialization dependences. Full materialization dependences are created when a logical operator is broken into two physical operators, one of which depends on the full results of the other. These operators, such as the build materialization for hash table join (`HashTableBuild`), are called *pipeline breakers*. Pipeline-breaking is performed during code generation.

3.2. Pipeline code generation algorithm

RADISH extends the algorithm of [13] for translating query plans into data-centric, pipelined code. In the compiler design, every physical operator has a two-sided iterator interface (`produce` and `consume`), and the operators generate *push-based* code where consuming operators are inlined into producing operators. Code generation begins at the single sink operator of the plan. Top-down traversal proceeds by calling `produce` inputs, and at sources bottom-up traversal proceeds by calling `consume` on successors. The signatures are:

```
produce(state) : unit
consume(input_tuple, state) : string
```

Compiler `state` includes pipeline-global properties and unresolved symbols, and `input_tuple` is an abstract tuple provided by the upstream operator. Pipeline-breaking is performed by the behavior of the `produce` and `consume` operators. For source operators (e.g., `HashTableAggregateScan`), `produce` generates code to iterate over full relations. For pipelined operators (e.g., `Select` and `Project`) `consume` generates code to process and send the input tuple to the next operator. For pipeline breakers (e.g., `HashTableAggregateUpdate` and `HashTableJoinBuild`), `consume` generates code to materialize the input tuple. Critically, this code generator design is modular with respect to the operators: each operator implementation solely generates code to produce and/or consume tuples; it is not necessary to implement fused operators (e.g., projecting join) unless the desired algorithm for a fused operator significantly differs from that of the composition of the operators.


i) Optimized logical plan
Scan(Order)
Select [o_totalprice > 20]
Aggregate [o_custkey]
Scan(Customer)
Project [c_custkey, c_name]
join [o_custkey=c_custkey]
Project [cnt, c_name]
ii) Physical distributed plan
task 0
Scan(Order)
Select [o_totalprice > 20]
Shuffle[o_custkey]
HashTable AggregateUpdate [o_custkey]
task 1
Scan(Customer)
Project [c_custkey, c_name]
Shuffle[c_custkey]
HashTableJoin Build[c_custkey]
task 2
HashTable AggregateScan
Shuffle[o_custkey]
HashTableJoinProbe [o_custkey=c_custkey]
Project [cnt, c_name]
pipeline
tuple dependence
full dependence
iii) Generated code

```
typedef T1 : tuple[int,string]
typedef T2 : tuple[int,string]
val cnt_table = global_new [Cell[int,int]] (SIZE1)
val join_table = global_new [Cell[int,T1]] (SIZE2)
val results = global_new[List[T2]] (NUM_PARTITIONS)

spawn task0
  forall t0 in Order
    if t0.o_totalprice > 20
      on partition(cnt_table[h(t0.o_custkey)])
        atomic { cnt_table[h(t0.o_custkey)] += 1 }

spawn task1
  forall t0 in Customer
    T1 t1;
    t1.id = t0.c_custkey; t1.c_name = t0.c_name;
    on partition(join_table[h(t1.c_custkey)])
      atomic { join_table[h(t1.c_custkey)].append(t1) }

spawn task2
  task0.sync()
  task1.sync()
  forall t0 in cnt_table.entries()
    forall t1 in join_table[h(t0.o_custkey)]
      val t2 = concat(t1, t0)
      T2 t3;
      t3.id = t2.c_custkey; t3.c_name = t2.c_name;
      atomic { results.append(t3) }

task0.sync()
task1.sync()
task2.sync()
```

Figure 1: System overview: translation from query plan to PGAS code. (i) query plan is optimized to reduce communication and processing. (ii) logical operators are translated to physical operators; operators that depend on all inputs before executing break the plan into pipelines. (iii) for each pipeline do a bottom-up traversal to generate push-based code.

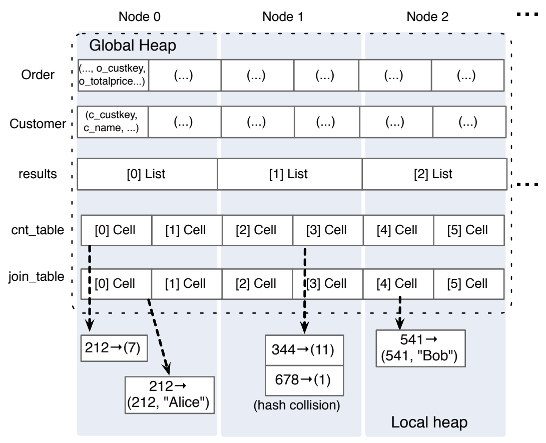


Figure 2: Physical layout of data across compute nodes in a distributed memory cluster for the example query. Each node is a PGAS *partition*, although other mappings are possible. The diagram represents a snapshot in time while `task0` and `task1` are still executing.

Translating tuples. During code generation, abstract tuples are passed between operators through `consume`. Abstract tuples appearing in the expressions of operators are translated to code using the interface:

```
getattr(position) : string
setattr(position, compiled_val): string
```

This encapsulates how tuples are accessed, making operator implementations decoupled from how tuples are stored (e.g., in array of structs or struct of arrays). This genericity exists only in the query compiler, so it does not create runtime overhead.

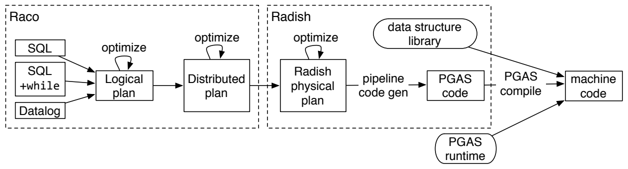


Figure 3: Query evaluation in RADISH. Boxes are representations of the query and arrows are transformations. The distinguishing feature is the generation of PGAS code that is compiled by a distribution-aware compiler.

3.3. Parallel code for one pipeline with CPP

For each pipeline, the operator implementations in RADISH generate parallel code. This code is push-based and data-centric, like previous approaches for sequential query compilation [14]. RADISH has a novel approach to parallel, distributed execution: the code is data-parallel and communication is performed inline with the processing code. The result is holistic code that is compiled and optimized as a whole by the PGAS compiler. RADISH operators use async-finish parallelism, which makes fine-grained parallelism more efficient.

Push-based, data-centric. RADISH generates push-based code, which is known to be more efficient than pull-based iterator execution [15]. The strategy is also data-centric in that it is tuple-at-a-time, keeping data close to the processor, as opposed to operator-at-a-time. RADISH uses `forall` to iterate over inputs and intermediate results. At some level the PGAS runtimes execute `forall` using task spawning and synchronization (section 4), but `forall` is semantically more meaningful and allows scheduling to be decided by the compiler and runtime system. `Forall`s may be nested, which is required when a pipelined operator, such as the probe side of a join, produces multiple outputs.

Each tuple is allowed to proceed through the pipeline independently. The `forall` loop only demands that any schedule of iterations must be serializable to guarantee execution is correct and deadlock-free. Physical operator implementations emit calls to the global data structure interface.

Data-centric code generation ensures better data locality than pull-based (i.e., iterator) execution as long as a task remains on a single partition. RADISH extends this concept to the distributed system by keeping the working set as small as possible (section 4).

Async-finish parallelism. All the operators we have implemented in RADISH have a property known as *async-finish* parallelism [16]. This means that nested `forall`s spawn tasks (`async`) and all tasks only need to be synchronized (`finish`) at the outermost `forall`.

The analogy to basic tuple processing is intuitive: within one pipeline, tuples are processed independently and the only data dependence on a tuple is the data dependence from the whole pipeline to dependent pipelines. The async-finish property allows for very efficient concurrent execution. In particular, although the runtime may spawn many tasks, they can be synchronized in bulk.

It is of course possible to write PGAS algorithms in RADISH that are not async-finish. Non async-finish algorithms can be more memory-efficient if finishing a task frees memory. For the query

```
select x,y,sum(z) from R group by x,y;
```

a memory-efficient algorithm that is not async-finish is

```
1 cluster R by x
2 forall xc in clusters:
3   allocate small hash table ht
4   forall y, z in xc: ht[y] += z
5   forall y, sz in ht: emit (x,y,sz)
```

This is not async-finish because line 4 spawns new tasks that must finish before line 5 can execute.

3.4. Shared data structures

RADISH physical operators are implemented with shared data structures, designed for efficient synchronization and data movement. Tasks within a pipeline and between two pipelines sharing a physical operator coordinate through shared data structures. All shared data structures depend upon tuple-grain synchronization, where producers and consumers coordinate at a fine grain. Some shared data structures also support cheaper bulk producer-to-consumer synchronization, called relation-grain synchronization. When producers synchronize with consumers only by relation-grain synchronization, there is an opportunity for the data structure to apply writes more efficiently; this is directly tied to async-finish parallelism. Management of data structure memory is tied to the pipelines that share it.

Data movement. Naïve shared memory code will incur a network message for every memory load and store involved in accessing a global data structure. Consider the `cnt_table` in Figure 2 used for aggregation: adding a new tuple involves a read of the cell, a read for each element in the collision list, a read of the current value, and a write of the new value. To reduce the number of round trip messages between partitions, RADISH uses distributed continuation passing [17]. RADISH data structures are laid out to exploit spatial locality, and the entire data structure operation is moved to the data using `on partition`. This transformation increases performance by up to an order of magnitude in communication-intensive applications [18, 19, 20]. `on partition` simplifies data movement in operator implementations because the PGAS compiler takes responsibility for detecting the data accessed in the continuation. One example is passing shared references to dynamically allocated data structures (e.g. `join_table`) through `forall` and `on partition`: the PGAS compiler detects that the data structure reference never changes. It broadcasts the global reference to all partitions before any loops so that they reference the local pointer. In a case study on the join hash table of *Q17*, we found this optimization gave a 13% reduction in average message size.

Relation-grain synchronization. If the physical operator calls for relation-grain synchronization, RADISH generates a scheduling dependence from the producer to the consumer. An example of this is `HashTableJoin`, where the build must complete fully before the probe starts. RADISH uses generates producer-consumer synchronization between the two pipeline tasks to indicate such a dependence (as in line 13 and 14 of Figure 1(iii)). For deep plans, `HashTableJoin` with relation-grain synchronization will miss opportunities for concurrency by delaying the probe pipeline: only the lookup itself is actually dependent on the corresponding build. Other algorithms that do only tuple-grain synchronization, like `SymmetricHashTableJoin`, provide more tasks to the runtime.

Tuple-grain synchronization. Data structure operations with tuple-grain synchronization have an interface like conventional concurrent data structures. These are indicated by atomic mutations (lines 9, 16, 26 in Figure 1(iii)). Mutable data accessed in an atomic operation must reside in a single partition whenever possible to avoid distributed transactions. This

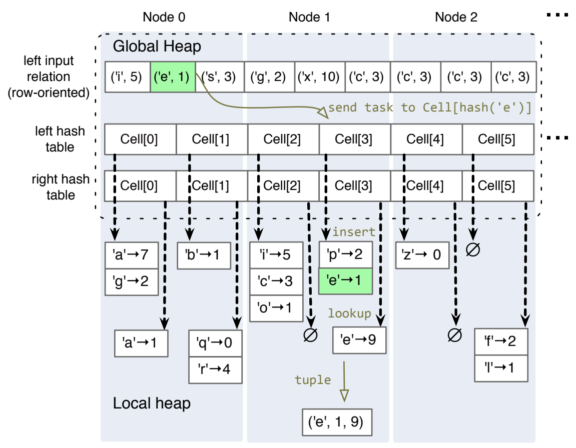


Figure 4: Storage layout for two hash tables in a fully-pipelined symmetric hash join. The relations (only one shown) and cells of the hash tables are partitioned across the global heap. Shown atop the layout is a `left_lookup_insert` of tuple $('e', 1)$. The lookup and insertion together must be atomic to avoid missing or duplicating matches. The two tables are partitioned the same way so that the atomic `left_lookup_insert` can run in a single partition.

locality is exploited by RADISH's use of `on partition` for `Shuffle`.

We illustrate tuple-grain synchronization using the example of `SymmetricHashTableJoin`, which uses one hash table per input. The input pipelines are concurrent, so to avoid duplicating and missing matches, the insert and lookup in the two hash tables must be atomic. Figure 4 shows the layout of two hash tables used in symmetric hash join. By identically distributing the arrays `left hash table` and `right hash table` and their adjacency lists, the atomic region can be implemented as a single migration of the probing task to the owner of `left[3]` and a local lock.

Memory management. RADISH uses coarse-grained reference counting to delete intermediate data structures that are no longer needed. When a pipeline is scheduled, for each sink data structure that does not yet exist, it allocates it with a reference count equal to the number of pipelines that will share it. When a pipeline finishes, it decrements the count on all data structures it touches.

3.5. Other implementation strategies

We implemented a baseline execution strategy, CVI, in RADISH using conventional techniques for query processing. We also discuss how to implement column-oriented processing in RADISH with few changes.

CVI. We implemented a query engine using a conventional technique based on iterators [21]. In CVI, each operator is implemented as a sequential iterator with a `nextTuple` method. Intra-pipeline parallelism is achieved by having multiple instances of each iterator. Communication is achieved by global operators, which push tuples to workers and materialize them in the appropriate stucture. Pipelines are scheduled to run when all full dependences are finished. One task on each instance polls the `nextTuple` method of the sink operator until it returns `false`.

To isolate the benefits of the CPP technique from the known benefits of compilation over interpretation, CVI shares much of the same library code and is compiled. This system uses the same messaging and data structure code as RADISH but operators are written generically using the iterator interface. Some fragments of individual operators are still generated: specifically, apply expressions, predicates, tuple concatenation, and attribute access.

Column-oriented execution. The RADISH technique is compatible with column-oriented execution (roughly, that a relation is a struct-of-arrays). For analytics workloads, column-oriented execution has been shown to be faster for a variety of reasons: no wasted IO bandwidth for projected columns, compression, late-materialization, and block iteration [22]. Regarding projected columns, row-oriented RADISH only wastes memory bandwidth for a fragmented cacheline. Late-materialization (lazy copying of attributes) must be weighed against the possibility of additional network messages; this decision is made during optimization through our extensions to RACO. Benefits from block iteration (avoiding a function call per tuple to extract an attribute) are not relevant because tuple extraction overheads are compiled away.

Given that RADISH is solely in-memory, implementing column-oriented execution requires relatively few changes. The RADISH compiler's interface for an abstract tuple includes methods `getattr`, `create`, and `setattr`. For column-oriented execution, these methods are implemented to emit code for array access rather than struct field access. Data structures can also be reused unmodified: instead of storing tuples, the column-oriented hash table stores an index into partition-local vectors for each attribute.

4. Runtime issues

In this section, we discuss issues in concrete PGAS languages and runtimes Grappa, Chapel, and X10. We also describe the PGAS language, Grappa, in more detail because we use it in our experiments.

4.1. Parallel loops and tasks

RADISH processes tuples with tasks, but rather than using an explicit execution strategy (such as worker threads taking chunks of a table at a time [23] for sequential processing), RADISH uses `forall`. The abstract `forall` gets mapped to the data parallel loops `Grappa::forall`, Chapel `forall`, and X10 `ateach` over ranges. These constructs are primarily implemented with tasks consumed by threads. We discuss two issues regarding task-oriented execution: concurrency management and loop termination.

Concurrency management. When executing a `forall`, a RADISH runtime needs to expose sufficient concurrency to keep the machine busy, while bounding it to a practical amount (usually linear in the number of processors). One technique for bounding concurrency for nested parallelism is *recursive decomposition*. A `forall` is executed by splitting the iterations into two disjoint subsets: spawning one subset as a new task and proceeding recursively with the other [24, 25]. Use of `forall`, in addition to the valid-anywhere nature of addresses in PGAS, allows RADISH to—with no changes—take advantage of dynamic load balancing mechanisms provided by the runtime system. Dynamic load balancing in data-intensive programs is a complex problem alone. We do not explore it in this paper and refer the reader to "morsel-driven query execution" [23], in which threads share input tables using a parameterized work queue.

As discussed in subsection 3.3, RADISH generates data-centric code for each pipeline. Maintaining data-centricity for pipelines that include communication also involves task scheduling. To see why, observe that `on partition` sends a running task to a remote partition, but that remote partition will most likely be running other tasks. The best that the task scheduler can do is try to keep the working set as small as possible. Recursive decomposition limits the spawning of *new* tasks, but there is also the question of how to schedule *existing* tasks. The task scheduler for a single partition in our Grappa back end uses the following heuristic priority order for existing tasks: started tasks with local origin, started tasks with remote origin, and lastly local tasks that are not yet started.

Most PGAS runtimes assign address space partitions to nodes (one hardware shared memory domain) or to individual threads within a node. For partition-per-node runtimes, threads in the partition must synchronize among themselves. For partition-per-thread runtimes, synchronization is necessary only at the network interface.

Loop termination. `Forall` loops execute iterations across all partitions, so they rely upon distributed termination detection. If for all iterations i, i is guaranteed to execute on one partition, then coarse-grained completion detection is sufficient. This requires only a coarse-grained global barrier.

If any given iteration i may execute on multiple partitions (e.g., pipeline 1 in Figure 1(ii)), then distributed termination detection is required to prevent races causing early termination. A common technique is a credit scheme [26] where credit is transferred when a task is spawned remotely. In async-finish parallel loops (subsection 3.3), a single credit tracker is necessary.

4.2. Grappa and fine-grained tasks

Translating a query to parallel tuple-at-a-time code produces tasks and remote messages (`on partition`) at the granularity of a tuple. This approach avoids ingraining low level details, such as block-based communication, into the intermediate parallel program, allowing the PGAS compiler and runtime flexibility in how it implements loops and communication. A challenge of fine granularity tasks and remote calls is the additional overheads they induce on the network and CPU. We explain how our evaluated PGAS backend, GRAPPA, mitigates these overheads.

Network overhead. Commodity network interfaces critically rely on sufficiently large packets (on the order of 10^5 bytes) to achieve most of the available bandwidth of the network [18]. In modern parallel databases operators often pull data in chunks over the network [6, 27]. In RADISH programs, the messages pushed over the network by a task for single tuple are often 3 orders of magnitude smaller. GRAPPA performs buffering of messages to utilize network bandwidth. Small messages with the same destination are bundled into large network packets. RADISHX's average message size on TPC-H queries was 40-200 bytes, so GRAPPA buffering has an enormous benefit. We find that at 64 16-core nodes, optimal throughput is achieved for a buffering delay of around 300 μs.

CPU overhead. Assigning a task for every tuple in a pipeline incurs overhead in the CPU for spawning, scheduling, and context switching. GRAPPA helps in two ways. First, chunks of iterations of a `forall` may be optionally fused at compile time into longer sequential tasks. In the SP^{2}Bench queries, fusing iterations in RADISHX improved performance by up to 23%. Second, GRAPPA reduces the overhead of scheduling many fine-grained tasks with user-level threading that can switch between contexts in 50 ns [5].

5. Evaluation methodology

We have built RADISH as an open-source extension to RACO, a relational algebra$^+$ compiler and optimization framework [28]. We built a back end to RADISH that emits GRAPPA [5] code. In the evaluation we refer to GRAPPA programs generated by RADISH as RADISHX. RADISHX includes a variety of hash-based algorithms for joins and aggregations.

Our primary goal is to implement both CPP and CVI in RADISHX and compare performance. Our secondary goal is determine that RADISHX is a practical system for analytics by comparing its performance with other relevant systems: the parallel database DBX. DBX is a state-of-the-art commercial database system that features parallel execution of individual queries over a cluster and code generation. This makes it the closest system to compare to RADISHX. In all performance comparison plots, the error bars represent 95% confidence intervals.

5.1. Setup

Query systems. GRAPPA code emitted by both RADISH and RADISH-ITER was linked against the MPI implementation MVAPICH2 v1.9b [29] (for network communication).

For each trial using DBX, we ran the query twice and recorded the second result. Full table and column statistics were collected for every input table. All input tables use random partitioning, as does RADISHX. Our DBX experiment code is available online as a fork of an open source TPC-H implementation.

Hardware. We ran TPC-H experiments on a cluster of 16 AMD Opteron processors connected by QDR Infiniband. Each node has 16 3.1-GHz cores across two sockets and 128GB of memory. We ran SP^{2}Bench experiments on a cluster of AMD Interlagos processors connected by QDR Infiniband. Each node has 32 2.1-GHz cores across two sockets and 64GB of memory.

5.2. Benchmark queries

TPC-H. We assume the conventional ad hoc TPC-H scenario with no tuning of the physical storage; RADISHX uses full scans to implement selections and all joins and aggregates require partitioning. We used scale factor 10 for comparing CPP with CVI and scale factor 100 for comparing to DBX. The TPC-H substitution parameters are the validation parameters [30].

Graph data benchmark. We evaluated scaling of RADISHX using the benchmark SP^{2}Bench [31] for RDF data. We used the RDF format of the generated data, with one relation of three columns. We did not perform the indexing and co-partitioning that is common in the loading step in RDF databases. To limit the number of operators we had to implement to evaluate RADISHX we removed `DISTINCT` from the queries. We used SP^{2}Bench to generate a dataset of 100 million triples (10 GB).

Analytics Application. To evaluate runtime on an analytics task, we implemented a naïve Bayes classifier in RADISH in two steps: a training task and a classification task. Both queries first pivot the input into a sparse format (`input-id, feature-index, feature-value`) so that the remainder of the query does not depend on the number of features. The training task is comprised of three SQL queries to compute the conditional probabilities. The classifier task joins the conditional probabilities with the input, and then uses a user-defined aggregate to compute argmax over outcomes and likelihoods.

We use a subset of the Million Song Dataset prepared by the UCI Machine Learning Repository [32]. The task is to predict song year in a $515,345$-song dataset from eight timbre features. Feature values were discretized into intervals of size 10.

6. Results

6.1. CPP vs. CVI performance

Figure 5 shows the runtime of CPP for each query normalized to the runtime of CVI. We find that CPP is at least $1.4\times$faster on all queries and on average $7.7\times$faster.

There is one difference between the query plans of CPP and CVI: CVI only supports symmetric hash join, while CPP also supports asymmetric hash join (build/probe as in Figure 1 ii). The difference in performance between the two join algorithms is notable when the large `lineitem` table is joined because it must be materialized in the symmetric hash join but can be the probe side in the asymmetric hash join. To support asymmetric hash join in CVI would have required significant changes to the communication layer of RADISHX to support queuing, which would make it less comparable to CPP. Instead, to make a fully apples-to-apples comparison, we also measured the results for CPP using the symmetric hash join. In most cases, this plan is nearly as fast, and when there is a difference (e.g., *Q19*), CPP still has an advantage. This result demonstrates that there is a performance advantage of CPP due to the compilation strategy. On average, CPP with the identical plan is $5.5\times$faster than CVI.

6.2. Contextual performance

We compared running times for TPC-H queries between RADISHX and DBX to validate that our implementation was competitive with a commercial query processor. The running times for RADISHX and DBX on TPC-H are shown in Figure 6a. Running times for sf10 for queries we could not get to run on DBX at sf100 are shown in Figure 6b. In the 15 queries that we could run on both systems, RADISHX is on-par-or-faster on all but 3 queries. Even though RADISHX is improved by RACO's local-global group by decomposition optimization *Q12* and *Q14* and significantly on *Q15* and *Q16*, without it RADISHX is still significantly faster than DBX. In all queries, DBX's plan was at least as efficient as the one used by RADISHX. In some cases RADISHX may be using a less efficient plan, but this is acceptable since the purpose of this experiment is simply to benchmark RADISHX's implementation of query execution. We ran DBX with its LLVM-based *run-*

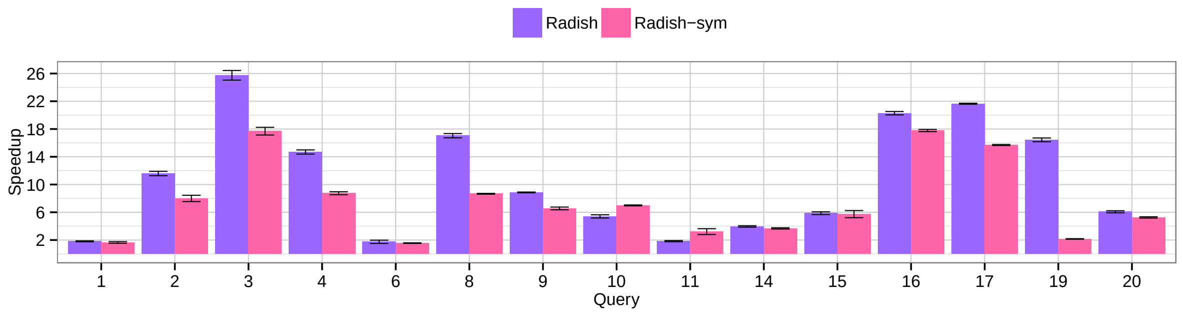


Figure 5: Speedup of RADISHX CPP over CVI on TPC-H queries

time code generation [33] toggled. We found that *Q1* showed significant speedup but there was no discernible benefit for any other queries. This speedup is unsurprising because *Q1* is embarrassingly parallel arithmetic-intensive code.

There are some data points missing from the plots. RADISHX ran indefinitely on *Q7* and crashed on *Q18* due to bugs that we are investigating, and DBX crashed on *Q17*. We have not implemented *Q13* or *Q22* in RADISHX as RACO does not yet support nulls (although there are workarounds), and DBX does not support cross products necessary in *Q11* and *Q22*.

6.3. Weak scaling

We evaluated weak scaling of large queries on RADISHX, using SP^{2}Bench with 1.4M rows per node. Results in Figure 7 show that *Q2* and *Q5a* both scale well (flat), even though for *Q2* the intermediate result size to input size ratio increases with input size. RADISHX does not scale as well on *Q9*, which also has an increasing ratio of result to input size. This is not simply due to small input size: running the full input size on 64 nodes gave a speedup of $4.9\times$ over 8 nodes. The quadratic *Q4* scales well; although intermediate result size to input size ration increases with small input size, its author-author degree is around 33 at around 50M rows. For *Q4* we tried other plans: symmetric hash join (SYM) and bushy join shape (bushy). Bushy is better for this query because it reduces the intermediate result size by $3\times$. SYM scales better than HJ in the bushy join shape. We found that on the other SP^{2}Bench queries, an all-`HJ` plan was consistently faster (up to $2\times$ faster) than the equivalent all-`SYM` plan. Although the fully pipelined `SYM` plan provides more concurrency by using only tuple-grain synchronization, it must materialize both sides of the join into hash tables, as discussed in subsection 6.1.

6.4. Analytic query

Naive Bayes. Figure 7c shows strong scaling for the classification query on the Million Song Dataset. The order of the join between sparse inputs and the conditional probabilities table was critical to performance. By building the hash table from the conditional probabilities rather than from the sparsified inputs, each probe finds a constant number of matches (one per outcome).

6.5. Compilation time

Although we have not spent effort on minimizing compilation time, we include it for completeness (Figure 8). For the TPC-H queries, RACO and RADISH take on average 1.26s to parse the query, optimize the plan, and generate the RADISHX program. Compilation of the RADISHX program with optimization flag `-O3` takes 10s of seconds. This significant time is partly attributable to extensive inlining of C++ template headers for the Grappa runtime and can be reduced with engineering effort.

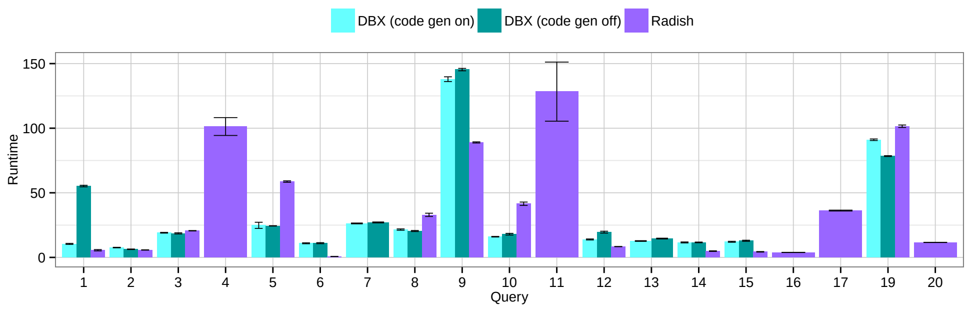
7. Discussion

We discuss how RADISH should be used and other PGAS compiler optimizations.

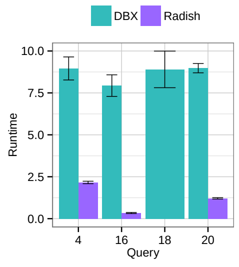
7.1. Using RADISH

Applicability. As with other MPI-based HPC systems and most MPP databases, RADISH does not support recovery of failed tasks. Therefore, it is not a replacement for systems like Spark [34] when running long data-parallel jobs on large clusters ([5] did a performance comparison of GRAPPA and Spark on a query workload). However, recent efforts have argued that most clusters in practice are 20-50 machines, a scale at which fault tolerance is not a dominant concern [6, 35].

Compilation time for RADISHX is currently quite high (subsection 6.5). Sub-second query compilers [13] gain efficiency by generating only "glue code" and pre-compiling most of the operator implementations. RADISHX is applicable for repetitive queries (like classifiers) or those where the compilation time is amortized by the performance improvement.

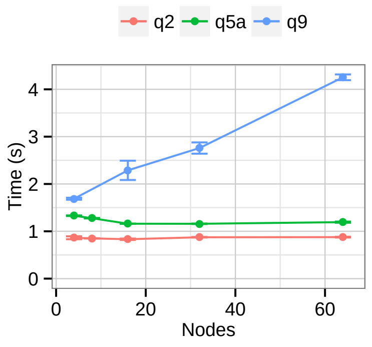


(a)

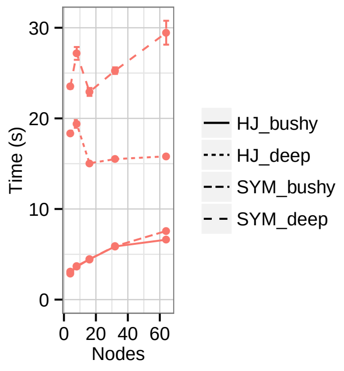


(b)

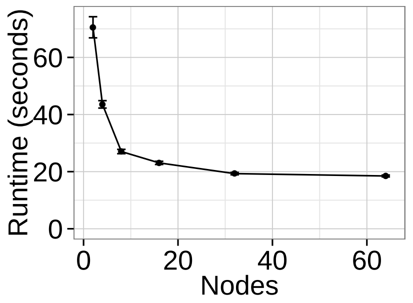
Figure 6: Performance of RADISHX and DBX on TPC-H queries, 16 nodes. (a) sf100, (b) sf10 for queries that would not run on DBX at sf100



(a) Mid queries



(b) q4



(c) Strong scaling of RADISHX query-generated naive Bayes classification

Figure 7: a) Weak scaling on queries with medium join result size, b) weak scaling on the quadratic join query, and c) strong scaling on Naive Bayes

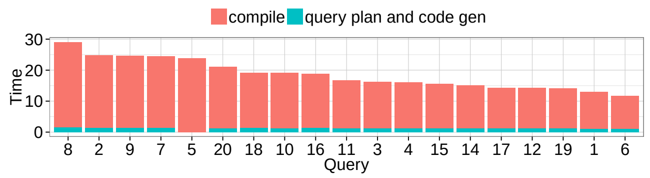


Figure 8: Total compilation time for TPC-H queries

We expect RADISH to be effective as the back end to language-integrated queries in analytics pipelines that include parallel *table-valued functions* (TVFs) or whole stages written in PGAS code that cannot be run efficiently in dataflow systems; for example, array computation, many of the domains in [36], and algorithms with frequently updated shared state like parameter servers [37]. All the user may need is a statistic or summarization over a large output after a simulation. While, a developer could do HPC and dataflow processing in different systems, doing so would incur the costs of non-summarized data crossing system boundaries [38] and prohibit co-optimization of handwritten and generated parallel code.

7.2. PGAS compiler optimizations

When there are UDFs, PGAS optimizations could provide optimizations inaccessible to a query planner. By inlining a UDF into the generated code, the PGAS compiler can send only live fields instead of the whole tuple during `on partition(...)` by using *scalar replacement* [39]. GraphX achieves a similar projection by bytecode introspection [40], rather than inlining.

RADISH always benefits from accessing a global hash table using `on partition`, but it is not obvious where to run the rest of the pipeline after the match. A similar decision must be made for hash joins in columns stores and MPPs. [19] does such an optimization on general PGAS code by inspecting data layout to infer good task migrations and would be complementary because it works at the granularity of memory accesses and works in the presence of UDFs.

8. Related work

Targeting code to single processors. Recent work has explored compiling entire queries into source code or bytecode to take advantage of system-specific compiler optimizations. Most query processing engines are built by implementing an iterator interface for each operator. HIQUE [15] compiles operators into nested loops and preprocessed inputs. A C compiler can optimize this code better than iterator code. However, the approach materializes results after each operator, inefficiently utilizing the memory hierarchy. The code generation in HyPer [13] improves data locality of generated code and also has a modular design for inlining operators. RADISH extends this code generation technique to generate PGAS code to run queries in parallel on distributed memory clusters. HyPer and RADISH compose code templates for program generation from queries. LegoBase [41] improves upon the software design and capabilities of template-based query compilers by using multi-stage generative programming that relies on types instead of code templates. Certain abstractions in RADISH embody concepts in LegoBase; for example, the abstract tuple (subsection 3.2) would be a *future-stage* expression during query plan optimization and a *present-stage* expression during code generation when `getattr` is called. The final step in LegoBase compilation is to take the optimized Scala code that implements the query and lower it to C code. One could use the techniques of RADISH to instead emit distributed shared memory code. Although this requires partition-awareness, this can be hidden inside of data structure implementations (e.g., a Scala `HashMap` can be lowered to the RADISH library's partition-aware hash table).

RADISH's execution model is most closely related to morsel-driven parallelism for NUMA shared memory machines [23]. Both are task-based and exploit shared data structures for flexible parallel execution. RADISH is different in that it expresses the task-parallelism in `forall` so that the PGAS runtimes chooses how to run tasks, and it produces parallel programs for distributed memory clusters.

Targeting code to distributed systems. Socialite [42] outputs Java code for the purpose of integrating Java functions with datalog, and it has been extended to run on distributed platforms using a master-slave model [7]. Tupleware [6] uses LLVM to analyze UDFs for use by the query planner. Its planner can also choose between pipeline and vector compilation; however, for all communication it is limited to library calls that perform block-based fetching of data over the network. By having a monolithic design that utilizes introspection rather than inlining, Tupleware does optimizations that neither query optimizers nor general compilers alone are capable of, such as using statistics to decide whether to vectorize a UDF. Since RADISH generates parallel code for a parallel compiler, it has the unique potential to optimize handwritten parallel code along with generated code.

Steno [8] translates LINQ queries to loops in C# and relies on DryadLINQ for distributed execution. All of these systems take the same approach of compiling fragments with a sequential compiler and stitching the fragments together with a communication library. RADISH takes a holistic approach to code generation for distributed systems where the entire program is compiled by a distributed-aware language compiler. This approach allows pipelines written in low-level *parallel* code to be further optimized even across communication points.

The HyPer transactional/analytic database is similar to RADISH in that it uses a task-based execution model, JIT compiles queries, and is NUMA-aware [23]. However, the JIT generates code for a single sequential "morsel" rather than a holistic parallel program, and HyPer only runs on a single multi-core machine.

Removing overheads of tuple-at-a-time. While the benefits of compilation to push-based machine code are to reduce overheads of reduce iterator overheads and increase data locality, other systems perform block-based or vectorized execution to exploit the efficiency of data parallelism and data caches in modern CPUs. MonetDB/X100 [43] produced unaliased, vectorized code that is unrolled and software pipelined by the C compiler. Sompolski et al [44] explored generation of data-parallel code that makes use of single-processor SIMD units. While RADISH does not generate vectorized code for the CPU, our evaluated back end, GRAPPA, uses lightweight threads and batching of network messages to achieve high bandwidth. Vector-at-a-time techniquesc are complimentary: a PGAS compiler could vectorize `forall` loops emitted by RADISH.

Data analytics on HPC systems. A number of systems have adapted in-memory data analytics platforms to HPC environments. [45] improves the task scheduling of in-memory Spark jobs on an HPC system. [46] extends MPI with features to support dataflow jobs. [47] enhances Spark performance by reimplementing its data shuffle operation directly on RDMA. In this paper, we provided a data analytics programming model on an HPC stack by instead exploiting existing HPC languages and runtimes, but we did not explore scaling up to very large deployments.

9. Conclusion

We implemented a fast query processor atop PGAS languages. Our particular approach is a compiler that generates code that takes advantage of the parallel-awareness of PGAS languages. Our code generation technique executes queries $5.5\times$ faster than conventional compiled execution. Using PGAS to efficiently execute queries required efficient data structures, generating code that avoids extra messages, and mitigating the overhead of an execution model based on fine-grained tasks.

This result is the first step toward parallel language-integrated queries, which will be useful for writing high-performance analytics codes. We also hypothesize that targeting parallel langauges as an intermediate representation for queries is a valuable approach for performance and simpler design of query engines for distributed systems.

References

[1] X. Lu, N. S. Islam, M. Wasi-ur Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of hadoop rpc with rdma over infiniband," in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 641–650.

[2] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox, "A tale of two data-intensive paradigms: Applications, abstractions, and architectures," in *Big Data (BigData Congress), 2014 IEEE International Congress on*. IEEE, 2014, pp. 645–652.

[3] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel Language," *International Journal of High Performance Computing Application*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538. [Online]. Available: http://dx.doi.org/10.1145/1094811.1094852

[5] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *USENIX ATC*, 2015.

[6] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, "An architecture for compiling udf-centric workflows," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1466–1477, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824045

[7] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed SociaLite: A Datalog-based language for large-scale graph analysis," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1906–1917, Sep. 2013. [Online]. Available: http://dx.doi.org/10.14778/2556549.2556572

[8] D. G. Murray, M. Isard, and Y. Yu, "Steno: Automatic optimization of declarative queries," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 121–131. [Online]. Available: http://dx.doi.org/10.1145/1993498.1993513

[9] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*. Hoboken, NJ, USA: John Wiley and Sons, Inc., 2005. [Online]. Available: http://dx.doi.org/10.1002/0471478369

[10] B. L. Chamberlain, S. eun Choi, S. J. Deitz, and L. Snyder, "The high-level parallel language ZPL improves productivity and performance," in *In Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[11] P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Z. Su, and K. A. Yelick, "Titanium language reference manual, version 2.19," UC Berkeley Tech Rep. UCB/EECS-2005-15, Tech. Rep., 2005.

[12] M. P. I. Forum, "MPI: A message-passing interface standard, version 3.0," Tech. Rep., 2012.

[13] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, Jun. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002938.2002940

[14] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, April 2011, pp. 195–206. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2011.5767867

[15] K. Krikellas, S. Viglas, and M. Cintra, "Generating code for holistic query evaluation," in *ICDE*, 2010, pp. 613–624.

[16] J. K. Lee and J. Palsberg, "Featherweight x10: A core calculus for async-finish parallelism," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693459

[17] S. Jagannathan, "Communication-passing style for coordination languages," in *Coordination Languages and Models*. Springer, 1997, pp. 131–149.

[18] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006, pp. 10 pp.–. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2006.1639320

[19] B. Holt, P. Briggs, L. Ceze, and M. Oskin, "Alembic: Automatic locality extraction via migration," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '14. ACM, 2014. [Online]. Available: http://dx.doi.org/10.1145/2660193.2660194

[20] J. Zhang, B. Behzad, and M. Snir, "Optimizing the Barnes-Hut algorithm in UPC," *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '11*, p. 1, 2011. [Online]. Available: http://dx.doi.org/10.1145/2063384.2063485

[21] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '90. New York, NY, USA: ACM, 1990, pp. 102–111. [Online]. Available: http://dx.doi.org/10.1145/93597.98720

[22] D. J. Abadi, S. R. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 967–980.

[23] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 743–754.

[24] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999. [Online]. Available: http://dx.doi.org/10.1145/324133.324234

[25] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161079

[26] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *J. Syst. Softw.*, vol. 43, no. 3, pp. 207–221, Nov. 1998. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(98)10034-1

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[28] Raco, "Raco: The relational algebra compiler," https://github.com/uwescience/raco, June 2014.

[29] MVAPICH. (2014, sep) MVAPICH: MPI over Infini-Band, 10GigE/iWARP and RoCE. [Online]. Available: http://mvapich.cse.ohio-state.edu

[30] "Tpc benchmark h standard specification, revision 2.17.1," Tech. Rep., 2014.

[31] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel, "SP2Bench: A SPARQL performance benchmark," *CoRR*, vol. abs/0806.4627, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04329-1_16

[32] K. Bache and M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[33] S. Wanderman-Milne and N. Li, "Runtime code generation in cloudera impala." 2014.

[34] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: SQL and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1145/2463676.2465288

[35] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 853–864, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536206.2536213

[36] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[37] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola, "Scalable inference in latent variable models," in *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, ser. WSDM '12. New York, NY, USA: ACM, 2012, pp. 123–132. [Online]. Available: http://doi.acm.org.offcampus.lib.washington.edu/10.1145/2124295.2124312

[38] F. Nagel, G. Bierman, and S. D. Viglas, “Code generation for efficient query processing in managed runtimes,” *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1095–1106, 2014.

[39] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar, “Communication optimizations for distributed-memory X10 programs,” in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 1101–1113. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.105

[40] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, “Graphx: Graph processing in a distributed dataflow framework,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685096

[41] I. Klonatos, C. Koch, T. Rompf, and H. Chafi, “Building efficient query engines in a high-level language,” in *Proceedings of the VLDB Endowment*, vol. 7, no. EPFL-CONF-198693, 2014.

[42] J. Seo, S. Guo, and M. S. Lam, “SociaLite: Datalog extensions for efficient social network analysis,” in *29th IEEE International Conference on Data Engineering*. IEEE, 2013. [Online]. Available: http://mobisocial.stanford.edu/papers/icde13.pdf

[43] P. A. Boncz, M. Zukowski, and N. Nes, “MonetDB/X100: Hyper-pipelining query execution,” in *CIDR*, 2005, pp. 225–237.

[44] J. Sompolski, M. Zukowski, and P. Boncz, “Vectorization vs. compilation in query execution,” in *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, ser. DaMoN ’11. New York, NY, USA: ACM, 2011, pp. 33–40. [Online]. Available: http://dx.doi.org/10.1145/1995441.1995446

[45] Y. Wang and R. Goldstone, “Characterization and optimization of memory-resident mapreduce on HPC systems,” *Parallel and Distributed . . .* , 2014. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=6877311

[46] F. Liang, C. Feng, X. Lu, and Z. Xu, “Performance benefits of DataMPI: a case study with BigDataBench,” *Big Data Benchmarks, Performance . . .* , 2014. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-13021-7{_}9

[47] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, “Accelerating spark with RDMA for big data processing: Early experiences,” in *High-Performance Interconnects (HOTI), 2014 IEEE 22nd Annual Symposium on*. IEEE, 2014, pp. 9–16.