Radish: Compiling Efficient Query Plans for Distributed Shared Memory

Brandon Myers
University of Washington
bdmyers@cs.washington.edu

Daniel Halperin
University of Washington
dhalperi@cs

Jacob Nelson
University of Washington
nelsonje@cs

Mark Oskin
University of Washington
oskin@cs

Luis Ceze
University of Washington
luisceze@cs

Bill Howe
University of Washington
billhowe@cs

ABSTRACT

We present RADISH, a query compiler that generates distributed programs. Recent efforts have shown that compiling queries to machine code for a single-core can remove iterator and control overhead for significant performance gains. So far, systems that generate *distributed* programs only compile plans for single processors and stitch them together with messaging.

In this paper, we describe an approach for translating query plans into distributed programs by targeting the partitioned global address space (PGAS) parallel programming model as an intermediate representation. This approach affords a natural adaptation of pipelining techniques used in single-core query compilers and an overall simpler design. We adapt pipelined algorithms to PGAS languages, describe efficient data structures for PGAS query execution, and implement techniques for mitigating the overhead resulting from handling a multitude of fine-grained tasks.

We evaluate RADISH on graph benchmark and application workloads and find that it is 4× to 100× faster than Shark, a recent distributed query engine optimized for in-memory execution. Our work makes important first steps towards ensuring that query processing systems can benefit from future advances in parallel programming and co-mingle with state-of-the-art parallel programs.

1 Introduction

The state of the art for query execution on distributed clusters involves in-memory processing exemplified by systems such as Spark [38], which have demonstrated orders of magnitude performance improvement over earlier disk-oriented systems such as Hadoop [2] and Dryad [21]. These systems still incur significant overhead in serialization, iterators, and inter-process communication, suggesting an opportunity for improvement.

Prior systems have demonstrated orders of magnitude performance improvements over iterator based query processing by compiling plans for single processors [10, 23, 27, 34]. Frameworks that generate distributed programs only compile plans for individual processors and stitch them together with communication calls, retaining the iterator model [15, 33]. These systems have a common shortcoming: they depend directly on a single-node compiler (e.g. LLVM, JVM) to perform machine-level optimizations, but these compilers cannot reason about a distributed program.

The alternative, which we explore in this paper, is to generate programs for a partitioned global address space (PGAS) language (e.g., Chapel [11] or X10 [13]), then compile and execute these distributed programs to evaluate the query. A key feature of these languages is a *partition*, which is a region of memory local to a particular processor and far from other processors.

Consider this query with a join and multiplication:

```
SELECT R.a*R.b, R.b, S.b FROM R,S WHERE R.b=S.a;
```

One side of the join corresponds to the following PGAS program:

```
for r in R:
   on partition [ hash(r.b) ]
         for s in lookup(r.b)
               emit r.a*r.b, r.b, s.b
```

This program is a representation of the physical plan: the join is a hash join with the relation `r` as the build relation. From this code, the PGAS compiler is now free to explore an additional class of decisions related to distributed execution on the target machine. The explicit `on partition` construct instructs the compiler to send the iteration to the worker corresponding to the hash of the value `r.b`. The multiplication `r.a*r.b` could be computed either before or after transmission of tuple `r` over the network to worker `hash(r.b)`. In terms of communication, there is no obvious difference between these two choices; in either case, two numbers will be sent over the network: `(r.a*r.b, r.b)` in one case, and `(r.a, r.b)` in the other. However, a compiler that understands this parallel code and the underlying architecture will consider the likelihood that the multiply functional unit is available. This kind of optimization is inaccessible to both a database-style algebraic optimizer that cannot reason about the instruction level and an ordinary shared memory compiler (e.g. LLVM, GCC) that cannot

reason about partitions. While a query compiler can directly generate machine or byte code [27]; this is not the case in any compiler for distributed memory systems.

The approach of targeting PGAS allows us to extend existing query pipeline compilation techniques, reuse existing language compilers and runtimes, and simplify the design of the system. We posit that it may also allow us to integrate and co-compile handwritten parallel algorithms for difficult analytics.

Our goal is to automatically generate code that is on par with good handwritten distributed programs that answer the query. A naïve translation of a query to PGAS code results in unnecessary network messages from shared memory access and CPU overhead from running a task per tuple. We designed distributed data structures for executing queries efficiently on the PGAS model. To reduce CPU overhead, we use lightweight tasks and loop unrolling.

In this paper, we describe RADISH, a compiler system that implements this approach. To evaluate our results, we evaluate the performance of code generated by RADISH against in-memory distributed query processing systems. On the linked data benchmark SP^{2}Bench, our system is 12.5×faster than Shark, a query engine built upon the state-of-the-art in-memory data analysis platform Spark. On relational microbenchmarks, RADISH is between 4× and 100× faster than Shark.

In summary, this paper makes the following contributions,

1. We design and present a new system for compiling queries into fast code for distributed memory systems, by turning pipelines into specific data-centric (as opposed to control-centric) tasks, then expressing these tasks as programs in PGAS languages. By targeting PGAS code, our approach allows distribution-aware compilers to apply additional data layout and distribution-aware optimizations, such as task migration and parallel loop-invariant code motion.

2. We mitigate the CPU and network layer overheads of the fine-grained tasks produced by this code generation technique, primarily through messaging buffering, inlining tasks, and light-weight task scheduling.

3. We implement our technique as RADISH, a query compiler that translates relational queries into PGAS code that runs on distributed memory systems. To the best of our knowledge, RADISH is also the first query processing system that integrates with PGAS languages.

4. We use RADISH to generate code for GRAPPA, a PGAS language and runtime with high-throughput random access. The system executes queries 12.5× faster than Shark, which has been shown to have performance comparable to MPP analytical databases. Although we only evaluate code generated for GRAPPA, RADISH is extensible to target other PGAS languages.

The rest of this paper is organized as follows. Section 2 discusses background on PGAS languages and related work on query compilation. Section 3 explains the PGAS code generation technique and considerations. Section 4 evaluates the performance of RADISHX. Section 6 discusses how to overcome challenges of a fine-grained (tuple-per-task) execution model and the applicability of RADISH.

2 Background and Related work

2.1 PGAS languages

PGAS languages are the dominant shared memory languages for programming distributed memory clusters. Their critical attribute is that the partitioning of the shared address space across nodes of the cluster is explicit to allow for the programmer and language compiler to reason about locality. Creators of PGAS languages have given evidence for better productivity than message passing libraries [12, 11, 13, 17, 16] and in some cases better performance due to a distributed-aware compiler and runtime optimizations enabled by the abstraction of shared memory [37]. For the techniques in this paper, we assume that a target PGAS language provides the following constructs: concurrent tasks, parallel for loops, and control over data partitioning.

2.2 Code generation for queries

Push-based code generation Recent work has explored compiling entire queries into source code or bytecode to take advantage of system-specific compiler optimizations. HIQUE [23] compiled operators into nested loops and preprocessed inputs, which the low-level compiler could optimize better than iterator code. However, the approach materialized results after each operator. Neumann [27] improves data locality by exploiting pipelines in the query plan to generically inline operators. LegoBase [22] improves upon the software design and capabilities of template-based query compilers by using staged compilation. One of its staged optimizations is transforming between a push and pull-based engine. RADISH extends the technique of Neumann to generate PGAS code to run queries in parallel on distributed memory clusters. Since LegoBase emits C code, we could extend LegoBase for distributed systems using the techniques we present.

Targeting code to distributed systems Socialite [32] outputs Java code for the purpose of integrating Java functions with datalog, and it has been extended to run on distributed platforms [33]. TupleWare [15] uses LLVM to analyze UDFs for use by the query planner. Its planner can also choose between pipeline and vector compilation; however, it is limited to block-based fetching of data over the network. 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 provides at least two benefits over others. First, high level parallel code allows the low-level compiler to optimize pipeline code across communication points. Second, parallel code written by the programmer can integrate with generated parallel code.

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 in modern CPUs. MonetDB/X100 [10] produced unaliased, vectorized code that is unrolled and software pipelined by the C compiler. Sompolski et al [34] 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 backend, GRAPPA, uses lightweight threads and batching of network messages to achieve high bandwidth. Techniques that take advantage of CPU vector

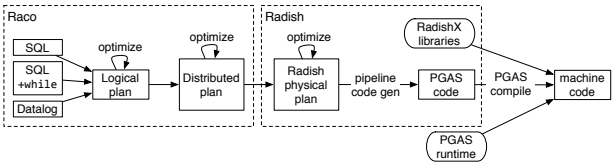


Figure 1: Compilation of queries. Boxes are representations of the query and arrows are transformations. The high-level difference from a conventional query compiler is generation of PGAS code followed by generation of machine code with a distributed-aware compiler.

units are complimentary: since RADISH generates PGAS parallel loops it could use the techniques to make the loops vector-friendly.

3 Code generation

We extend techniques for pipeline-based code generation for query plans [27] to produce distributed parallel programs. RADISH parallelizes query execution using tasks and shared memory in three ways. First, RADISH reduces communication and supports fine-grained updates via careful layout of global data structures: memory locations that are accessed together are placed in the same partition. Second, RADISH evaluates a diverse space of plans by considering both fine-grained (tuple granularity) and coarse-grained (relation granularity) synchronization between pipelines. Third, RADISH produces efficient parallel code for each pipeline: the code is data-centric and exercises the primitives available in distributed-aware languages. Within a processor, data-centric code makes efficient use of memory bandwidth by sending one tuple through a pipeline of operators at a time. Since the PGAS compiler understands communication (i.e., it exposes a shared memory abstraction), its optimization window extends across a whole pipeline regardless of communication boundaries. The query compilation flow is shown in Figure 1. RADISH extends RACO [29], a relational algebra compiler, to generate its plans. RADISH creates a distribution-aware physical plan consisting of distinct pipelines (Section 3.1), generates code for each pipeline (Section 3.2), generates synchronization between pipelines (Section 3.3), and feeds the output to the PGAS compiler (Section 3.4).

Example query We illustrate RADISH code generation with the query in Figure 2a(top). Figure 2b shows a physical query plan consisting of hash aggregate and hash join, and Figure 2a(bottom) shows a candidate program emitted by RADISH. The basic structure of the code is similar to that of the sequential code generated by the technique in [27]: a nested loop for each pipeline in the physical plan materializes an intermediate or a final result. We highlight three aspects of the parallel PGAS code. First, each loop is made parallel, as all iterations are independent. Second, there is concurrency between pipeline tasks: in particular, task 0 (line 1) and task 1 (line 7) can execute concurrently. Pipeline task 2 iterates over the results of the aggregation and probes the join hash table. Pipeline task 2 depends on fully materialized results of the other tasks; this is denoted with `sync` statements. Third, data structures for the physical representation, like the hash table, are implemented similarly to those in shared memory but have are partitioned. The code represents one execution strategy for the join and aggregate where the task migrates (along with required data) to the relevant partition of the structure (using `on partition(...)`). The resulting benefit over naïve shared memory code is that when two partitioned data are required for an operation such as a hash table lookup, the operation can occur in a single message rather than 1-2 for every shared memory read and write.

RADISH sends the emitted PGAS code through the PGAS compiler, which links with the PGAS runtime to generate machine code. Low level optimizations (e.g. machine-dependent) performed by the PGAS compiler are complementary to those of RADISH, and depending on the quality of the code from RADISH, some higher level optimizations (e.g. communication avoidance) are also complementary.

3.1 Physical plan

The RADISH physical algebra includes operators based on global data structures, such as global array-based hash tables (HT). These operators include those with tuple-grain and relation-grain synchronization, such as `HTAggregateUpdate` / `HTAggregateScan` and `HTBuild` / `HTProbe`, and those with only tuple-grain synchronization, such as `SYMLeft` / `SYMRight` (for symmetric hash join). The hash table data structure backing these operators may be any PGAS implementation that implements insert, lookup, and iteration. In our evaluation RADISH's hash table is a global array of cells, each having a list of tuples in the same partition. When the code from RADISH is compiled by the PGAS compiler, the data structure implementation code is optimized along with it.

Figure 2b shows the plan for the publication 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 (`HTBuild`), are called *pipeline breakers*.

RADISH extends the technique of [27] 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. Calls to `produce` at source operators (e.g., `HTAggregateScan`) generate code to iterate over full relations. Calls to `consume` for pipelined operators (e.g., `Select`) generate code to process and send the input tuple to the next operator. Calls to `consume` for pipeline breakers (e.g., `HTAggregateUpdate`) generate code to materialize the input tuple.

3.2 Parallel code for one pipeline

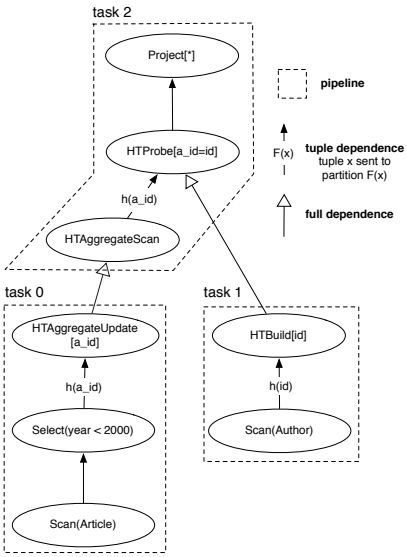
RADISH indicates that each tuple is allowed to proceed through the pipeline independently. This property is denoted with a parallel loop, or `forall` (as in Chapel's [11]). The `forall` loop demands that any schedule of iterations must be serializable to guarantee execution is correct and deadlock-free. Tasks within a pipeline share the materializing data structure, so to ensure correctness, updates to this data structure must be atomic. Updates are discussed further in Section 3.3.

RADISH by default begins an iteration on the partition where the input tuple resides. `Foralls` may be nested, which is required when a pipelined operator, such as the probe side of many-to-many join, produces multiple outputs.

```
1  select * from Author,
2    (select a_id, count(*) from Article
3      where Article.year < 2000
4      group by a_id) Pubcounts
5    where Author.id==Pubcounts.a_id
```

(a)

```
1  spawn task0
2    parallel for each t0 in Article
3      if t0.year < 2000
4        on partition(cnt_table[t0.a_id])
5          aggregate t0 in cnt_table[t0.a_id]
6
7  spawn task1
8    parallel for each t0 in Author
9      on partition(join_table[t0.id])
10       materialize t0 in join_table[t0.id]
11
12 spawn task2
13   task0.sync
14   task1.sync
15   parallel for each t0 in cnt_table
16     on partition(join_table[t0.a_id])
17       parallel for each t1 in
             join_table[t0.a_id]
18         output Tuple(t1, t0)
19
20 task0.sync; task1.sync; task2.sync
```



(b)

Figure 2: Query representations from SQL, plan, and PGAS code. The example is *authors' count of articles published before 2000*. (a) Query for authors' count of articles published before 2000. The lower listing shows pseudocode representing what RADISH would emit to implement the query in PGAS. (b) A physical plan for RADISH. A pipeline is comprised of tuple dependences and broken by a full dependence. The path of a pipeline can span multiple partitions.

Concurrency When executing a `forall`, a RADISH backend needs to expose sufficient concurrency to keep the machine busy, while bounding it to a practical amount (usually linear in the number of processors). To support this bounding for nested parallelism, our evaluated backend uses *recursive decomposition*. In this technique, 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. This technique is similar to *spawn* in the Cilk workstealing algorithm [9] and scheduling policies in X10 [18].

RADISH's data-centric code generation ensures better data locality than pull-based execution while the task stays on a single partition. We extend this concept to the distributed system is 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 our evaluated RADISH backend uses the following heuristic priority: started tasks with local origin, started tasks with remote origin, unstarted tasks.

Pipeline termination `Forall` loops execute iterations across all partitions, so a RADISH backend must implement distributed termination detection. If all iterations are guaranteed to execute on one partition, then coarse-grained completion detection is sufficient, since no partition can receive tasks from other partitions. A common implementation is to use a local counter on each partition to track the number of spawned tasks that are incomplete. Global completion detection is detected with a single parent task for the pipeline on every partition entering a barrier upon the local count reaching zero.

If any given iteration may execute on multiple partitions (e.g., pipeline 2 in Figure 2b probes a global hash table), then distributed termination detection is required. A common technique is a credit scheme [25] where credit is transfered when a task is spawned remotely. For each pipeline, there is a global credit tracker. Any construct that spawns or transfers tasks (e.g., `forall` on line 15 and 16 and `on partition` on line 16), has a reference to the credit tracker for its pipeline so that it may register new tasks. X10's `async-finish` regions and Chapel's `sync` regions rely on lightweight termination detection of global tasks.

Data movement RADISH is row-oriented, so a task needs data movement when it touches two rows in different partitions, as might happen in a join or aggregate. Naive shared memory code will incur a network message for every memory load and store involved in accessing a global data structure. RADISH data structures are laid out to reflect access patterns and operations move computation to the data (`on partition`) to reduce the number of round trip messages between partitions. This transformation has been shown to increase performance by an order of magnitude in a variety of communication-intensive applications [8, 20, 39].

PGAS offers the choice between data movement through blocking tasks or continuation passing [35]. The optimal choice depends upon the data. Consider the hash join in the example query: once the `HTBuild` pipeline is finished, a task in the `HTProbe` pipeline matches its tuple with the assigned bucket of the hash table. If the probe is 1-to-many and the probe tuple is large, then bytes sent may be minimized by sending the build tuple key and reading back the matches, as with a semi-join. If the probe is many-to-1 and the probe tuple is large, then the same scheme enhanced with caching will store retreived matches for other tuples. Finally, if the

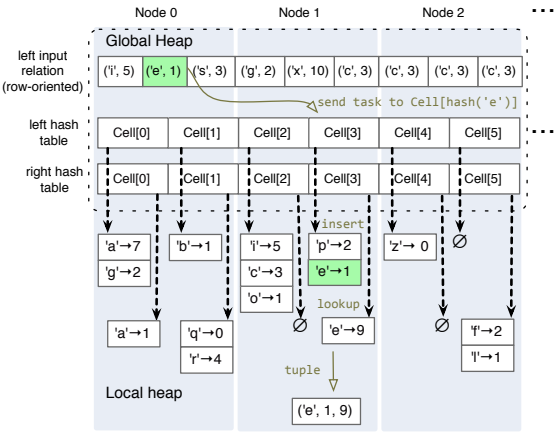


Figure 3: Storage layout for two hash tables in a fully-pipelined symmetric hash join. The relations (only one shown) and cells of the hash tables are partitioned across the global heap. Shown atop the layout is a `left_lookup_insert` of tuple $('e', 1)$. The lookup and insertion together must be atomic 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.

probe is many-to-many or the build tuple is large, then the optimal choice is to migrate the task to the bucket partition and process it there.

3.3 Inter-pipeline coordination

To generate code with inter-pipeline parallelism, RADISH wraps each pipeline inside a task spawn. Physical operators at a pipeline boundary can require either 1) relation-grain synchronization, when one pipeline depends on the full materialized result of another, or 2) tuple-grain synchronization, when two pipelines access data structures concurrently. The choice of what type of operator to use depends upon weighing the cost and benefit of concurrency.

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. The mechanism for the relation-grain synchronization can be one of 3 categories: serialization using static program order, producer-consumer synchronization between the two pipeline tasks, or dependence-aware scheduling of tasks. RADISH uses only the second because it provides full information to the parallel language and does not depend on a special runtime scheduler. Runtime scheduling of pipelines can take into account memory constraints and processor utilization. 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 more fine-grained synchronization, like `SymmetricHashTableJoin`, provide more tasks to the runtime.

Tuple-grain synchronization Tuple-grain synchronization is implemented as atomic operations on mutable data structures (e.g., hash tables). Mutable data accessed in an atomic operation must reside in a single partition whenever possible to avoid distributed transactions. 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.

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 3 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]` rather than multiple remote reads and writes.

3.4 PGAS compiler optimizations

RADISH uses two optimizations to reduce the communication involved with global shared memory and fine-grained parallel loops. First, RADISH puts explicit task migration in the generated code with `on partition(...)`. Tools exist that inspect data layout to automatically infer good migrations in PGAS code [20]. Second, we use `forall` loop-invariant code motion: if multiple iterations share a region of code, each partition can execute the region once for all tasks in that partition. In addition to saving computation, this transformation is also used to reduce memory footprint and reduce communication of broadcasted read-only data used within the body of the loop [7].

4 Evaluation methodology

We have built RADISH as an open-source extension to RACO, a relational algebra$^+$ compiler and optimization framework [29]. We built a backend to RADISH that emits GRAPPA [4] 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.

The primary goal of our evaluation is to measure the performance of RADISHX for executing a variety of data retrieval and analytical queries. We compared the performance of query execution between RADISHX and the data analytics system Shark, which is competitive with MPPs [36] and is optimized for in-memory query execution. A secondary reason to choose Shark is that it is built upon Spark [38], so like RADISH it has the ability to integrate queries with a wider class of applications.

In all performance comparison plots, the error bars represent 95% confidence intervals.

4.1 Setup

Query systems GRAPPA code emitted by RADISH was linked against the MPI implementation MVAPICH2 v1.9b [5] (for network communication). In terms of network stack, Shark uses IP-over-Infiniband, while RADISHX uses Infiniband natively with MPI, allowing kernel-bypass. To run

experiments on Shark, we adopted the methodology from [3]. In particular, Shark was configured to read memory-cached input tables and to write the result of a query to a memory-cached output table. With these settings, the fault tolerance system overhead is limited to tracking lineage of RDD partitions in the control plane. Each worker JVM was assigned 52GB of memory. The number of reducers for shuffles was set to the suggested 3 per worker. For all queries, we verified that the expected number of Spark workers were assigned tasks.

Hardware We ran all experiments on a cluster of AMD Interlagos processors. Each node has 32 2.1-GHz cores in two sockets, 64GB of memory, and a 40Gb Mellanox ConnectX-2 InfiniBand network card. Nodes are connected via a QLogic InfiniBand switch.

4.2 Benchmark queries

Simple queries We used a collection of single-operator queries meant for understanding the basic performance differences of the two systems. The queries are:

- *SEL1* and *SEL99*: selection with 1% and 99% selectivity to vary materialization costs.
- *AGG50M* and *AGG100*: aggregation with 50 million and 100 unique keys to test out-of-cache and in-cache accumulation.
- *COUNT*: aggregation with no grouping.

We generated a relation of 1B rows, 3 integer columns, at a total of 20GB. The selection and aggregation columns are distributed uniformly randomly.

Linked data graph queries We used the SPARQL benchmark SP^{2}Bench [31] for RDF data. The irregular structure of the citation graph makes the benchmark challengingi, and has motivated a numberof specialized RDF systems. For these experiments, we used the RDF format of the generated data, with one relation of three columns. We did not perform the heavy indexing and co-partitioning that is common in the loading step in RDF databases or triple stores built upon RDBMSs. With this restriction, the benchmark serves as a proxy for challenging queries on irregular, network-like data. To limit the number of operators we had to implement to evaluate RadishX, we modified the queries to exclude `DISTINCT` (affects *Q4*, *Q5a*, *Q5b*, *Q9*), `ORDER BY` (affects *Q2*), and range joins (affects *Q4*). We used SP^{2}Bench to generate a dataset of 100 million triples (10 GB).

4.3 Application queries

Naive Bayes To build a naive Bayes classifier in Radish, we wrote a training query and a classifier query in MyriaL, the primary language supported by the Raco. Both queries pivot the input into a sparse format first (`input-id, feature-index, feature-value`) so that the remainder of the query is independent of the number of features. The training query is comprised of three SQL queries to compute the conditional probabilities. The classifier joins the conditional probabilities with the input, and then does 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 [6]. 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.

Iterative query: PageRank We implemented PageRank as an iterative SQL query to compare RadishX to a hand-coded iterative application. Raco supports an input language that resembles SQL with sequences and while loops [19]. Our PageRank implementation uses `while` for iterations, but the Raco compiler is not yet able to do loop-invariant motion of physical operators to avoid unnecessary re-partitioning. To extrapolate performance of PageRank with these optimizations, we isolated the execution time for the main iterative work: performing the aggregate sum over each vertex's neighbors' ranks. We compare this time to the runtime of a hand-coded Grappa version of PageRank that that uses an adjacency list graph representation explicitly. We ran experiments using the 1.4-B edge Twitter follower graph [24].

5 Results

5.1 Performance on simple queries

RadishX performs non-selective scans up to 34× faster than Shark (see Table 1). The systems are impacted similarly when materializing many tuples in memory for *SEL99*. The relative speedup is less in *AGG50M*, given that the query requires all-to-all communication, which is the bottleneck for both systems. Since the Radish relational optimizer does not yet use data statistics to inform the plan, it gives the same plan for both aggregation queries: namely aggregation with a global hash table. This accounts for the near identical performance. Shark, on the other hand, takes advantage of the small number of keys in *AGG100* with local combining to reduce communication. Finally, *COUNT* runs just as fast as *SEL1* on RadishX because it is doing a similar amount of work on every tuple. The all-to-one communication of the single tuple at the end has negligible cost (Radish generates a collective reduction when there is no grouping). Shark, on the other hand, experiences overhead even for a small amount of communication.

Case study: *SEL1* We computed a detailed breakdown of the execution time on the 20GB dataset to understand the differences between these two diverse systems. We took sample-based profiles of both systems and categorized CPU time into five components: *network/messaging* (low-level networking overheads, such as MPI and TCP/IP messaging), *serialization* (message de-aggregation in RadishX, Java object serialization in Shark), *iteration* (loop decomposition and iterator overheads), *application* (actual user-level query directives), and *other* (remaining runtime overheads for each system). Figure 4 shows the results of this analysis. In Shark, a core spends two-thirds of the time in network code: given that this query should do little communication, this component likely includes idle time (appears in profile as polling the network) that is due to poor load balance and task spawning. Even so, relative to RadishX, significant slowdown in Shark is due to overheads that exist in the inner loop of computation: deserialization and iteration. The serialization overheads are not fundamental to Java vs. C++ but rather come from the way Shark stores tuples in memory. RadishX had negligble impact from application code, while Shark spent 5% of its time there (functions directly related

	AGG100	AGG50M	COUNT	SEL1	SEL99
RadishX (20GB)	6.34	6.27	0.10	0.15	1.34
shark (20GB)	22.69	64.68	14.87	5.15	119.78

Table 1: Running time in seconds for microbenchmark queries. Grappa and Shark ran on 16 nodes.

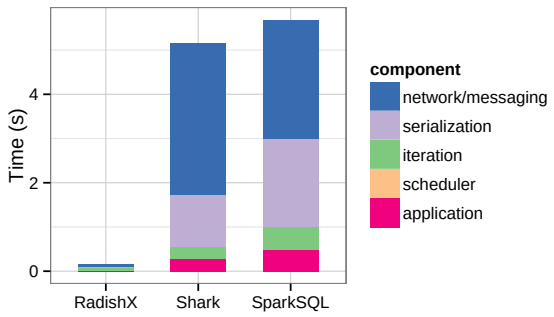


Figure 4: Time breakdown for microbenchmark query *SEL1*, 20GB. Ignoring network time, both Shark and SparkSQL have significant serialization overhead. Compared to RadishX, Shark's application code (the filtering operation) is non-negligible and iteration overhead is 4.3× greater. The percentages for RadishX are 40.2% network, 47.0% iteration, 12.8% scheduler, and negligble for others.

to the filter operator). Ignoring the network component, speedups would be 3.1× for only serialization, 1.1× for only iteration, 1.2× for only application, 0.99× for only scheduling overhead; the total speedup would be 19×.

To see how much Shark's performance was impacted by its storage system design, we ran the same analysis for *SEL1* implemented in *SparkSQL*[1]. SparkSQL allows tuples in an RDD to be stored in native memory format (the `MEMORY` storage level), potentially allowing faster access than Shark's serialized format. SparkSQL still executed *SEL1* with a similar profile to that of Shark, although it has entirely different serialization procedures. While Shark spent a lot of time in object serializations and transforming between column format, nearly all of the serialization time for SparkSQL consists of `productToRowRDD` that transforms the format of tuples.

5.2 Performance on linked data queries

Figure 5 compares the performance of RadishX and Shark on SP^{2}Bench. For this comparison, RadishX was instructed to only generate left-deep join plans. The results are grouped roughly by category:

- *Q3b*,*Q3c*, *Q1*, *Q3a* are select-join and are selective so join input is small
- *Q9*, *Q5a*, *Q5b*, and *Q2* are select-join and join input is large; *Q9* includes a union
- *Q4* allows for indexes to be used more than once and has quadratic output size

[1]At the time of writing, SparkSQL was in alpha and did not yet support enough of SQL to implement all of the experimental queries

RadishX has a geometric mean speedup of 12.5× over Shark. On the highly selective queries *Q3b* and *Q3c*, the geometic mean speedup is 41.4×, which follows the observation for *SEL1*. As the number of inputs to the join increase, the geometric mean speedup is smaller: 18.6× for *Q1* and *Q3a* and 5.3× for the four join queries with large input.

We performed a breakdown of CPU time in Figure 6. We highlight *Q2* because it involves enough communication to be interesting. The systems spend nearly the same amount of CPU time in application computation. About half of RadishX's performance advantage comes from efficient message aggregation and a more efficient network stack. RadishX's message aggregation moves as few cache lines as possible through the memory hierarchy for each small message.

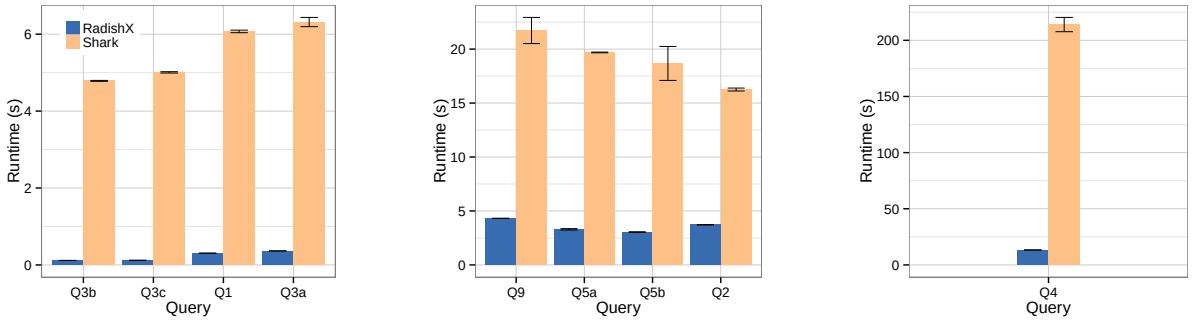
Additional benefit comes from iterating via RadishX's compiled parallel `forall` loops compared to Shark's dynamic iterators. RadishX's scheduling time is higher than Shark due to frequent context switches in the application and messaging code, but other components make up for this overhead. We assign Shark's in-memory column compression in the serialization category: *SEL1* spent negligible time in compression, while *Q2* spent 9% of its time there. This is because the shuffle hash joins of *Q2* require memory to store intermediate results. RadishX spends 22× less time in serialization and 10× less time in iteration; if only these two components could be accelerated in Shark, the total speedup would be 1.5×.

The Shark query plans used shuffle hash join as all tables are estimated to be large. Shark's execution of these queries appears to place bursty demands on the network, and is sensitive to network bandwidth, while we find RadishX to be more consistent (Section 5.3). On query *Q2*, Shark achieves the same peak bandwidth as it can sustain in a random access benchmark (200MB/s/node) [26], but its sustained bandwidth is just over half this amount (116 MB/s/node).

The large gap in the performance of *Q4* suggests that RadishX benefits greatly from pipelining relative to the sequences of shuffle joins performed by Shark. In the next section we find that RadishX achieves even higher performance on this query when we switch to a bushy join plan, which is one that Shark did not try.

The Shark query planner chooses to do a sequence of joins, which are performed using shuffle. Although the map outputs are materialized in memory, these steps are very expensive.

Weak scaling We evaluated weak scaling of the large join queries on RadishX. Using SP^{2}Bench, we generated datasets with size 1.4M rows per node. Results are shown in Figure 7. We expect running time to be flat if RadishX achieves linear scaling with input size. *Q2* and *Q5a* both scale well, 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 8 nodes took on average 21.38s (versus 4.25s on 64 nodes), meaning RadishX achieves scaling with more nodes. The quadratic *Q4* scales well; although intermediate



(a) small join input **(b)** large join input **(c)** quadratic output; shared indexes

Figure 5: Runtime on SP^{2}Bench queries for 16 nodes (a,b) or 64 nodes (c). Error bars mark the 95% confidence interval.

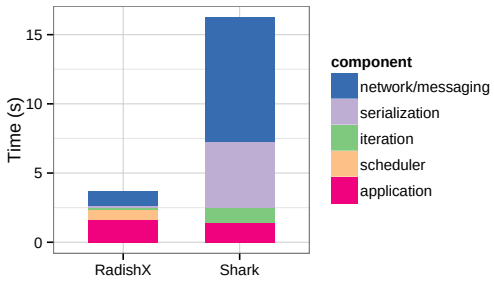


Figure 6: Performance breakdown of speedup of RadishX over Shark on *Q2*.

result size to input size ration increases with small input size, it plateaus at 33 at around 50M rows (SP^{2}Bench generates data chronologically and the overlap window of two authors' active years is limited). For *Q4* we tried other plans: symmetric hash join (SYM) and bushy join shape (bushy). Hash join (HJ) consistently equals or beats SYM. Bushy is better for this query because it reduces the intermediate result size by 3×. SYM scales better in the bushy join shape.

RadishX's throughput for small network messages depends heavily on buffering. In the average case, the system must increase the delay for the buffers as the number of nodes increases to achieve maximum throughput. RadishX continues to achieve higher total-system throughput by waiting longer to send messages. We find that at 64 16-core nodes, optimal throughput is achieved for a buffering delay of around 300 μs.

Compilation time For the SP^{2}Bench queries, Radish takes on average 0.16 seconds to generate the RadishX program. Compiling the RadishX program with the Grappa compiler takes on average 19.2 seconds.

5.3 Join plans

Pipelining joins We explored the performance of two hash join implementations in RadishX. Hash table join (`HJ`) has a full dependence from build to probe pipelines. Symmetric hash table join (`SYM`) has has two independent pipelines synchronizing at a fine grain, as discussed in Figure 3. We expected `SYM` to provide more concurrency by serializing fewer tasks and thus achieve higher throughput of tuples. We configured Radish to use the same join order for both plans and use a left-deep join shape.

Execution traces for SP^{2}Bench *Q2* are shown in Figure 8. The y-axis is the number of outstanding tasks. The black line is the total tasks for all pipelines. The colored lines are number of tasks for individual pipelines. The `HJ` plan is faster. Qualitatively, the pipelines in the `HJ` plan peak at different places and the probe pipeline (in red) is active alone at the end. The `SYM` plan looks different: the pipelines are active together and taper together. The `SYM` plan reached lower peak and sustained number of outstanding tasks compared to the `HJ` plan. The *bytes/sec* at the RDMA interface tracks the outstanding tasks closely and both plans do the same amount of communication. Separation in time of pipelines sometimes improves locality and overall processing rate.

5.4 Analytical queries

We also evaluated the performance of RadishX on analytical applications involving aggregation.

Naive Bayes Figure 9 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).

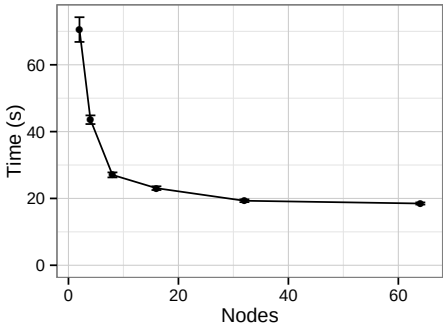
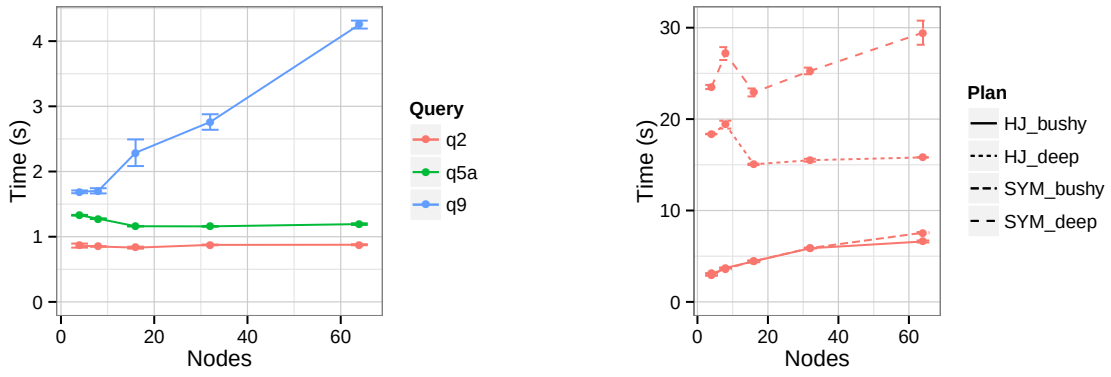


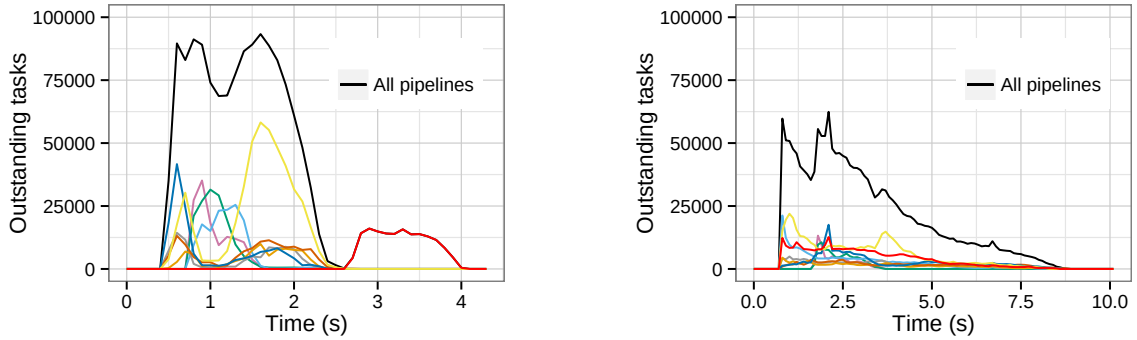
Figure 9: Strong scaling of RadishX query-generated naive Bayes classification

Iterative query: PageRank RadishX gets within 4× the performance of the hand-coded version of PageRank. We find the performance difference is due to two factors. 1) The



(a) Mid queries **(b)** q4

Figure 7: Weak scaling of RadishX on the large join queries of SP^{2}Bench



(a) Hash table join **(b)** Fully-pipelined symmetric hash table join

Figure 8: Execution trace of two left-deep plans for SP^{2}Bench *Q2*. Each colored line is a pipeline of the query.

aggregate sum of ranks of neighbors uses indirect lookups into a hash table (averaging two collisions per access), while the hand-coded does direct array access. 2) The RadishX version uses more memory bandwidth as a by-product of isolating the timed aggregate sum: we forced RadishX to materialize the input, which is vertex ranks and adjacencies in an unnested format, while the hand-coded's vertices have pointers to edge lists. For reference, the hand-coded PageRank performs about 2× faster than PageRank on GraphLab [26], even without using the edge partitioning techniques GraphLab uses to reduce skew.

6 Discussion

We discuss the overheads induced by the task-per-tuple compilation approach, as well as how Radish should be used.

6.1 Fine grained tasks

Translating a query to parallel data-centric code produces tasks and remote calls (`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 to mitigate 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 [8]. In modern parallel databases operators often pull data in chunks over the network [15, 38]. In Radish programs, the messages pushed over the network by a task for single tuple are often 3 orders of magnitude smaller. The Grappa runtime, on which we evaluated RadishX, performs buffering of messages to utilize network bandwidth. The communication layer of Grappa buffers small messages into large network packets with the same destination. If injection rates of commodity network interface cards improve, then this software buffering will become unnecessary.

CPU overhead Assigning a task for every tuple in a pipeline incurs overhead in the CPU for spawning, scheduling, and context switching. RadishX addresses these overheads in two ways. First, chunks of iterations of a `forall` may be optionally fused at compile time into longer sequential tasks, similary to [1]. Besides removing the overhead of tasks, it also enables loop unrolling optimizations for reducing control overhead and software pipelining to use out-of-order processors' reordering window, as in MonetDB/X100 [10]. In the SP^{2}Bench queries, fusing iterations in RadishX improved performance by up to 23%. Second, Grappa provides user-level threading that can switch between contexts in 50 ns [26]. For a hash table join with blocking probes, lightweight

blocking tasks automatically execute similarly to prefetch algorithms [14].

6.2 Using RADISH

Applicability RADISH does not support recovery of failed tasks. Therefore, it is not a replacement for systems like Spark when running long data-parallel jobs on large clusters. 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 [15, 30]. As every pipeline in RADISH begins by pulling from a distributed materialized result, a recovery mechanism could be applied at this level without reasoning about shared memory.

Compilation time for RADISHX is currently quite high (about 19 seconds). Until this improved with a more efficient backend compiler (as in [27]), RADISHX is applicable for repetitive queries (like classifiers) or those where the compilation time is amortized by the performance improvement.

Parallel database This paper has focused on pipelined operators implemented with global data structures. Since RADISH targets a class of general parallel languages, it is capable of generating code for plans with conventional parallel database operators, like broadcast and hash partition. The algorithms used in RADISHX are not fundamentally different from common parallel database operations like hash partition followed by local join. What differs is its push-based execution strategy and compiled pipelines.

Integration with PGAS languages Choosing a PGAS language as an intermediate representation for queries has three unique advantages beyond the scope of this paper. First, by making query optimization modular in this way, queries compiled with RADISH can benefit from advances in the high-performance computing and parallel programming community (like autotuning of collective communication [28]). Second, it provides a method of integration and optimization of queries along with not just conventional UDFs but also irregular parallel programs. Third, RADISH targets a collection of existing constructs in parallel languages (e.g. `forall`, `spawn`, `on partition`) that is sufficiently abstract to have a variety of realizations.

7 Conclusion

We improved the performance of query processing on distributed memory systems by generating partitioned global address space (PGAS) programs. Since the PGAS compiler is distributed-aware, its optimization window extends through communication boundaries in pipelines of query operators. To produce efficient PGAS programs, RADISH partitions data structures according to how they are accessed and applies communication and loop optimizations.

Our evaluation of RADISH on benchmarks and applications showed that this compilation technique offers significant performance improvement over a distributed in-memory query engine based on iterators. We find that RADISH spends less time in iteration and serialization code. This result suggests that targeting parallel langauges as an intermediate representation is a valuable approach for performance, parallel language integration, and simpler design of query engines for distributed systems.

8 Additional Authors

9 References

[1] Intel cilk plus lanugage specification v0.9. Technical report, 2010.

[2] Apache hadoop, jun 2014.

[3] Big data benchmark. https://amplab.cs.berkeley.edu/benchmark, Feb 2014.

[4] Grappa: Scaling data-intensive applications on commodity clusters. https://grappa.io, Jun 2014.

[5] Mvapich: Mpi over infiniband, 10gige/iwarp and roce, sep 2014.

[6] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[7] 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*, pages 1101–1113, May 2011.

[8] 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*, pages 10 pp.–, April 2006.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[10] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[11] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21(3):291–312, Aug. 2007.

[12] 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.

[13] 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*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[14] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17, 2007.

[15] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. Tupleware: Redefining modern analytics. *CoRR*, abs/1406.6667, 2014.

[16] H. P. N. (editor), D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium Language Reference Manual, Version 1.16.8. Technical Report UCB//CSD-04-1163x, Computer Science, UC Berkeley, 2004.

[17] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[18] 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*, pages 1–12. IEEE, 2009.

[19] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the myria big data management service. *SIGMOD*, 2014.

[20] 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*, OOPSLA '14. ACM, 2014.

[21] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[22] I. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. In *Proceedings of the VLDB Endowment*, volume 7, 2014.

[23] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[24] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[25] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *J. Syst. Softw.*, 43(3):207–221, Nov. 1998.

[26] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Latency-tolerant software distributed shared memory. Technical Report UW-CSE-14-05-03, Univeristy of Washington, Apr 2014.

[27] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[28] R. Nishtala, Y. Zheng, P. H. Hargrove, and K. A. Yelick. Tuning collective communication for partitioned global address space programming models. *Parallel Computing*, 37(9):576–591, 2011.

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

[30] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10):853–864, Aug. 2013.

[31] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp2bench: A sparql performance benchmark. *CoRR*, abs/0806.4627, 2008.

[32] 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.

[33] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, Sept. 2013.

[34] 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*, DaMoN '11, pages 33–40, New York, NY, USA, 2011. ACM.

[35] G. J. Sussman and G. L. S. Jr. Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*, 1975.

[36] 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*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[37] K. Yelick. Performance advantages of partitioned global address space languages. In B. Mohr, J. Trff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 6–6. Springer Berlin Heidelberg, 2006.

[38] 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[39] 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*, page 1, 2011.