The rapid increase of stored data has been met with a bevy of new large-scale data processing systems to manage and analyze it, but these systems have three problems. First, their programming models consist of coarse grained parallelism; performing parallel computations in lock-step does not cover the space of efficient algorithms for data-intensive problems. Second, they do not utilize modern high-performance architectures. Third, the distributed aspect of these systems makes them difficult to build. My research mission is to design data-intensive systems to solve these problems; my particular approach is to apply ideas from high-performance computing and compilers. In this direction, I have developed new compiler and runtime techniques to build high-performance computing systems that are easier to program than the existing solutions.

**Easier parallel programming for data-intensive applications**

Data-intensive applications pose a challenge for programmers trying to achieve high performance. Achieving efficiency on a cluster of multi-core compute nodes requires the programmer to have a complex mental model of the machine. In this model, performance is sensitive to how data and computation are partitioned across a deep memory hierarchy and how parallel processes communicate. New high-productivity, data-parallel processing frameworks (e.g., Spark, Flink, Pregel)—for various data models like relations, collections, and graphs— make programming easier. Yet, some applications have no efficient mapping to these paradigms: they require primitives that allow more control. Thus, there is still a gap of performance and compatibility between high-performance parallel programs and high-productivity parallel programs. My core research has focused on closing this gap, first by building a common platform for data-intensive applications running on small-to-medium-scale clusters and second, by integrating query processing natively in high performance languages.

**Latency-tolerant distributed shared memory.** I co-developed Grappa\(^1\), a new language embedded in C++ and distributed runtime system that makes it easier to write high-performance data-intensive applications, such as graph processing, that compose different kinds of data [1]. Grappa is a *partitioned global address space language* (PGAS) language. Programmers write PGAS programs using parallel loops, global memory reads/writes, and synchronization. In practice, for programmers to achieve good performance on data-intensive PGAS codes, they need to consider the size of network messages, the frequency that concurrent tasks synchronize, and task size for scheduling overhead. Grappa’s runtime features enable the programmer to write efficient programs more easily by ignoring some of these issues. For example, Grappa encourages the use of fine-grained tasks, which are easy to use. But fine-grained tasks are inefficient because commodity processors rely on spatial locality and networks rely on large packets. The key insight in the design of Grappa’s runtime is that in data-intensive applications, individual shared memory accesses do not require low latency when the performance metric is application throughput. Grappa makes fine-grained tasks more efficient in three ways. First, lightweight threads make frequent context switches less costly [2]. Second, bulk synchronization allows many small tasks to communicate with fewer messages. Third, cache-aware batching of small messages forms large packets from small ones, which better matches the design parameters of network interfaces.

We built three in-memory data-intensive computing frameworks on Grappa: relational query processing, scale-free graph processing, and iterative data-parallel, and applications atop each one. We compared performance to popular platforms for each paradigm, like Spark and Graphlab. Grappa was 1.33-10X faster than purpose-built parallel frameworks, due mostly to its network optimizations. Because Grappa is a shared memory model rather than a graph-specific one, we were able to implement algorithmic optimizations not possible in the graph API. Optimizing breadth first search gave a 2X speedup.

The other students and I presented our work in regular grant reviews through Pacific Northwest National Laboratory, funded by the DoD. Grappa has made an impact in the parallel computing community. Throughout the development of the system and applications, we also engaged researchers at Sandia, Lawrence Berkeley, and Cray Supercomputers. Grappa’s implementation strategy as a library-based extension of C++ is attractive to adopters and new PGAS languages like UPC++ [3] and DASH [4] have followed suit.

**Compiling queries for high-performance computing.** Data-intensive applications are motivating new interactions between the models of databases and the algorithms and platforms of high-performance computing

---

\(^1\) [http://grappa.io](http://grappa.io)
(HPC). Considering development cost, exploratory queries needed by data analysts are a severe mismatch for the distributed programming models of HPC. Having HPC applications communicate with databases for analytics is also undesirable due to the high cost of moving data and converting between physical data representations.

I set out to find a way of integrating query processing efficiently into HPC languages. I focused on partitioned global address space languages (PGAS) because (1) they provide control over how data is distributed, (2) are built upon high-performance messaging libraries like MPI, (3) and have parallel-aware compilers, which simplifies the implementation of database operators and may afford additional optimizations not accessible to query optimizers. I implemented query processing for PGAS in a system called Radish [5], work funded by the DoD. Radish initially used demand-driven dataflow, the conventional technique for parallel query processing. However, this system executed SQL queries far slower than if the programmer implemented them in PGAS manually and did not exploit the parallel-awareness of the PGAS compiler.

To make Radish fast, I developed compiled parallel pipelines (CPP), a technique to generate PGAS code from SQL queries. The advantage of code generation over the conventional approach to parallel query processing is that the code is amenable to optimization by the PGAS compiler. Compiling queries has recently become critical to the performance of databases; as more of the working set fits in fast main memory the bottleneck shifts from disk IO to the CPU. Query compilers work by generating code for the query in an imperative language and then compiling it with a general-purpose compiler. Radish-CPP differs from previous query compilers for parallel databases because it generates and compiles a holistic parallel program, rather than sequential programs stitched together by a communication library. My experiments with the standard database benchmark TPC-H show that Radish-CPP answers queries 2-18X faster than Radish with demand-driven dataflow (even when compiled). Compared with Impala, a commercial database that uses column storage, parallelization, and code generation, Radish-CPP is competitive or faster.

To perform experiments on up to 4096 cores and 8TB of DRAM in the Radish and Grappa work, I’ve used PNNL’s supercomputing resources extensively. This expertise enables me to advise new students to leverage large-scale computing resources—including cloud—in research projects and coursework. Students in a parallel computing class can run scaling experiments on multiple machines, and students in a database class can run queries over large data using distributed systems.

Other projects

In addition to my thesis research, I’ve explored other aspects of data management. My work includes developing a software stack for database research, formal verification of query processors, and efficient multi-query processing for streaming data.

**Myria: the UW stack for big data research.** I am an active contributor of UW’s NSF-funded open-source big data management stack, which we use to facilitate diverse research. This stack includes the MyriaX parallel database, deployment in Amazon’s cloud, the Raco relational query optimizer, a number of backend compilers for Raco, including Radish, and the myria-web interface for interactive querying, browsing, and performance visualization. Myria as a whole has enabled research across data management, astronomy, and oceanography, myria-web has enabled research in database visualization, and Raco has enabled research in query compilation, iterative query processing, federated databases, and join optimization. I am the current owner for Raco and have helped new researchers, including undergraduates, contribute to it.

I was the primary supervisor for one such undergraduate student. He worked on an independent project to extend myria-web for federated data management. Motivated by the need to analyze diverse data (the clinical dataset MIMIC II has relations, text, and signals), our lab is exploring techniques for processing queries across multiple databases. With my student, I outlined a project with a well-defined goal but an open design space. The project introduced him to multiple parts of the Myria stack and our lab has since built

---

3 [http://github.com/uwescience/raco](http://github.com/uwescience/raco)
4 [https://physionet.org/mimic2/](https://physionet.org/mimic2/)
upon his work for a frontend to query patient and signal data from MIMIC II, a demo at VLDB 2015.

Our work on federated data management is funded by Intel Science and Technology Center (ISTC), a partnership between Intel, Brown, MIT, Portland State, and UW. This collaboration has given me the opportunities to be in contact with more researchers from other institutions, build a cross-institution research tool (the BigDawg Polystore, or federated database), and to present my thesis research at IEEE High-performance Extreme Computing and the ISTC Big Data annual retreat.

**Verified query compiler.** It is important for compilers to translate their input program correctly. Correctness of an implementation is typically ensured with testing, but bugs sneak through because it is not possible to exhaustively test every query on every input database. Formal verification can prove correctness (according to the spec) of the compiler mathematically, but building verified software is tedious (you need the program, the spec, and the proof). In widely used and security-critical programs (compilers, web browsers, OS kernels, databases), formal verification is arguably worth the effort. There now exists a verified compiler for C [6], but there is not yet end-to-end verification of SQL queries to machine code.

We built Crimp\(^5\), which soundly translates queries to imperative C-like code. Crimp includes a semantics for SQL and a semantics for Imp (a C-like language), as well as a compiler between the two languages. Using the automated proof assistant Coq, we proved that for all queries and all inputs, that the translated query gives the same output as the original query. Crimp currently supports query plans with row selection, column projection, and “nested loops” join. Our implementation in Coq is extractable to give an executable implementation of our verified code generator.

**Efficient execution of multiple streaming queries.** In applications with streaming data, such as real-time traffic monitoring, insights are found using window queries. A window query continuously provides a summarization over a span of time, such as the average speed of cars on a road over the last 30 seconds. Often analysts need information at multiple granularities—multiple windows. Computing two windows over the same data involves repeated work, yet most streaming systems execute multiple window queries separately. Previous work shares partial aggregates between multiple queries by slicing the data stream into sub-windows in a clever way [7]. However, computation is still repeated when computing aggregations over the results from of the sub-windows.

I helped graduate students at Seoul National University to build multi-query execution that shares both sub-windows and their aggregations. Our approach was to build a dependence graph of sharable results. The dependence graph is infinite, so our system had to maintain it dynamically and manage its size. Our system required 2X fewer CPUs but more memory to support a set of window queries in road network applications at a target latency and throughput.

**Future work and involving undergraduates and Master’s students**

I look forward to instilling in undergraduate and Master’s students a basic research mindset where they are problem-focused, looking across the computing stack and between disciplines for solutions. Good researchers are strong technical communicators: I will instill the values of high quality writing and presentation in my students by providing feedback and frequent opportunities to practice these two skills among peers. In addition to applying for NSF and DARPA grants, I will apply to other sources like NSF’s REU and support students in applying for summer research programs locally and externally. I received an honorable mention for the PhD NSF GRFP and read new UW PhD students’ NSF proposals each year, so I have experience to help Master’s students apply for the NSF GRFP. I have presented at DoD grant reviews through PNNL, and I will help my students apply to research internship opportunities like NSIP. Below I give directions for new research and how students will contribute to my research plan.

**Queries in data-intensive parallel programs.** My thesis work introduced fast standalone query processing for shared memory languages, but integration of queries into data-intensive parallel programs raises new questions. General parallel programs contain a variety of data structures, which may be nested, as well as partitioned. How do we integrate queries within general parallel programs that have diverse distributed data

\(^5\) [http://github.com/uwplse/crimp](http://github.com/uwplse/crimp)
structures? Initially, students will implement existing parallel algorithms from scientific simulation, graph processing, and machine learning in a new way: a hybrid of SQL and parallel code. In the next step, we will analyze these implementations to find where the bottlenecks are and motivate specific compiler optimizations. Our approach will draw from work on language-integrated queries and data structure synthesis but will need to incorporate partitioning of data.

New algorithms for distributed query processing. Compared to the conventional shared-nothing approach to implementing distributed query processing, using PGAS has the advantages of programmability, compiler support, and flexible parallelism. However, algorithms for PGAS query processing are not yet efficient. Straightforward hash join and aggregation algorithms suffer data skew and inefficient use of the memory hierarchy. I propose to develop efficient relational query processing algorithms for PGAS. Our approach will be to adapt algorithms that have been recently developed for multicore processors with non-uniform memory access (NUMA) (e.g., [8]). Multicore NUMA is similar to PGAS in its notion of memory regions, but our adaptations will have to consider two differences: PGAS targets clusters with more extreme NUMA than multicores, and PGAS has flexible control of data granularity. There are a number of new multicore NUMA algorithms, so this line of work will produce multiple self-contained projects for students. These projects have a clear path to workshop or conference posters and papers in data management and parallel computing conferences.

Data management in future computer systems. Looking further out, I plan to study data management in computer systems of the future. The “three V’s of Big Data”—velocity, volume, and variety—remain a challenge in part because our computer systems are not getting faster. The end of Dennard scaling (performance per watt improves) and the imminent end of Moore’s law (transistors on a chip increases) will change the shape of computer architectures. Right now architects only know how to increase performance two ways: parallelize or specialize. Over the last decade and a half, data management researchers have re-designed systems according to new architectural features of general-purpose processors—increased DRAM, deep cache hierarchies [9], multiple cores and NUMA, and SIMD instructions. If data management systems can only go faster with architectures specialized for them, then computer architects and data management researchers need to design across existing abstraction boundaries. To form a framework for tractably exploring this area, I will identify 2-3 emerging data analysis scenarios that demand interactive latencies (geographical data is a well-explored example). For each scenario, a number of students will work on 1 or 2-person projects contributing to the end-to-end system. This work allows students to work on multiple facets of computer science and engineering.


