Jim Fix Thesis Topics: Parallel Algorithms and Concurrent Languages

I've a wide range of interests within computer science. I usually prefer projects informed by mathematics and enjoy taking a mathematical approach. This has often been work in algorithms or in theory, though I've also done work in computer languages and graphics. I've found myself drawn, too, to writing code and making things work more than just on paper. The projects below are ideas from recent work. I am open to overseeing other topics, and you can check http://jimfix.github.io/research for what I've overseen.

Parallel Algorithms for Multicore and GPUs

Hardware designers have faced limits speeding up a processor's execution of a single-threaded program, so they've provided performance by placing several processors on a chip that can run several threads simultaneously. You see this, in stock computers, with multicore CPUs (usually about 8 processors, and perhaps 10s in the near future) and, in a different way, with graphics processors (GPUs; usually 100s of processors). To take advantage of this hardware you write code in a way that uses several cores at once, operating on different parts of the data simultaneously. This requires the theory and practice of parallel algorithms. Even standard problems (sorting or data processing; graph algorithms) are a challenge here because parallelization requires rethinking them considerably and because writing parallel code is tricky.

• **parallelizing Delaunay triangulations:** Isabella Jorissen [‘16] and Joewie Koh[‘17] have each investigated approaches to computing a special triangulation (with some great properties) of data points in the plane. We've written a single-processor version of Guibas & Stolfi's [1] algorithm. It has a "divide and conquer" structure similar to `mergesort` that lends itself to parallelization. I'd like to adapt that code so that runs on multicore (using POSIX threads) or on the GPU (in NVidia's `CUDA`). The key innovation to work out is the merging of two (or several) sub-triangulations into a full triangulation. Alex Pan[‘17] investigated a bi- or multi-merge for parallelizing `mergesort`, and I'd like to adapt it to triangulations.

• **parallelizing graph algorithms:** network data abounds that is ripe for analysis (e.g. social networks, interactomes, roads) and there are very large data sets that essentially require a parallel approach (see http://devblogs.nvidia.com/parallelforall/gpus-graph-predictive-analytics/). Alex Pan[‘17] investigated schemes on the GPU for computing connectivity[2] and shortest paths[3] on sparse graphs, and I'd like to further explore the tradeoff we saw between a Bellman-Ford-like versus a Dijkstra-like approach.

• **parallelizing search structures for long strings:** suffix trees and suffix arrays are the basis for a variety of fast schemes for processing large sequences (think: genetic data), for finding similarities between sequences, or for assembling sequences from fragments of data. Kärkkäinen & Sanders[4] have a general scheme for parallel construction of a suffix array, and I'd like to implememt it, or a variant, on GPUs.

Verifiable Languages for Concurrent and Distributed System

This past summer I worked with several students (Meaza Abate, Laura Israel, Jalan Ziyad) to develop a language (called "Start Now" or `SNo`) that is meant for writing code for small devices, or for concurrent programs, or for simple network services. It has concurrency built in (similar to Ericsson's `Erlang` and Google's `Golang`), allowing you to devise processes that communicate over "channels." These are inspired by Robin Milner's π-calculus[5]. We have written two versions of the language: an interpreted version that runs **`.sno`** files and a compiled version that produces `C` code from **`.sno`** files. I'd like to explore the addition of a type system to check `SNo` programs. I would also like to have the compiler target small, cheap devices (Arduino/AVR, see AdaFruit's site; raspberry Pi) and build fun or useful libraries for them.

• **a tiny operating system for small devices:** SNo needs a threads system with channel synchronization primitives, a network stack (for device communication over serial, bluetooth, wifi), and a concurrent garbage collector. We've these components in Linux, but not for a system like the Arduino. Imagine making it easy to build a Roomba in SNo! We'd want sensor, motor, controller, etc. libraries for it.

• **session typing:** When two processes communicate, they follow a protocol. If one is offering a service to the other, a client, then there is a contract they fulfill during that session. SNo asks programmers to code this way, and we use session types[6,7] to describe the activity on each end of a channel. We check them with the channel analog of the Hindley-Milner system used by Haskell for checking functions. We verify that the client's use of a channel is *dual* to the server's. This, along with internal/external choice provided by "send constructors" and "receipt patterns" relates sessions to linear logic[8]. I've a good sense of how this checking should work in SNo and would like to spend a year with a student implementing it. I have larger goals here, hoping to develop schemes for proving properties of programs beyond session typing.

References

[1] L.Guibas, J.Stolfi. "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams." *ACM Transactions on Graphics 4(2) 1985*.

[2] D. Merrill et al. "Scalable GPU graph traversal." In *Proceedings of PPoPP '12*.

[3] A.Davidson et al. "Work-efficient parallel GPU methods for SSSP." In *Proceedings of IPDPS '14.*

[4] J.Kärkkäinen, P.Sanders. "Simple linear work suffix array construction." In *Proceedings of ICALP '03.*

[5] R.Milner et al. "A calculus of mobile processes.". *Information and Computation 100 (1) 1982*.

[6] P.Wadler. "Propositions as Sessions." In *Proceedings of ICFP '12*.

[7] L.Caires, F.Pfenning. "Session types as intuitionistic linear propositions." In *Proceedings of CONCUR '10.*

[8] J.-Y.Girard. "Linear logic: its syntax and semantics." *Theoretical Computer Science 50(1) 1987.*