Some Research Problem Areas for Students

Quentin F. Stout
Computer Science and Engineering
University of Michigan
qstout@umich.edu

My students and I work on parallel algorithms (involving several models of parallelism and distributed computing), parallel computers, serial algorithms, computational science (usually in large multidisciplinary projects, but I'm not involved in any right now though hope to find some new projects soon), and random topics. Parallelism is my main emphasis, but I've had students work on quite different topics that they suggested. Feel free to suggest a topic.

Why parallel? Suppose you went through the usual undergraduate curriculum, studying and inventing standard serial algorithms, and you were a star who invented the fastest algorithm possible for a problem. No matter how large a problem you can solve, very quickly people want larger (I've had several first-hand experiences of this). Serial computers are not getting faster (our computer architects can explain the relevant physics), and have similar memory limits, so if you need more computation in a usable amount of time, you need parallelism.

Most of my students think about parallelism, either abstractly (parallel algorithms), implemented (parallel computing), or both. Parallel is more interesting because there are many models, some not at all what you normally think of. Supercomputers have many boards connected together, each similar to a board in a desktop computer. The Fugatu computer has 7,630,848 cores, mostly GPU cores. Several students have solved large problems on such systems (though not this large), but most of my students work on more theoretical models with a vast number of tiny cores which can't solve anything interesting on their own but the overall system can. Algorithms for such fine-grained parallelism are very different than the ones you've learned, and vary widely across the different models.

First, a short mention of **quantum:** I've never done quantum algorithms. For a vast number of problems in graph theory, geometry, matrix operations, etc., the models below are provably faster (in O-notation). People who work in quantum computing rarely mention this. Quantum computers are primarily being used for approximate solutions (and the ability to crack RSA, if they can ever build one large enough with sufficient error correction), while I'm usually interested in exact solutions.

Some of the more abstract or unusual models we *have* worked on:

2- or 3-dimensional grids of compute elements: with either a fixed number of bits of memory no matter how vast the grid is (cellular automata, like Conway's Game of Life), or a fixed number of words of size logarithmic in the system size (mesh-connected computers). Some cellular automata algorithms first have them self-organize into larger units which function like more powerful processors, even though no automata has enough bits to store the units' size. Once the compute units are formed then they start solving the problem, though there are fewer units than automata, making the efficiency complicated. Units organize by bouncing messages against the edges of the mesh, forming a barrier when they collide, then bouncing messages against barriers, etc. Biology does this: how do the cells of the body form units like bones, kidneys, etc. when no cell has more information than the pressure on it, what its neighbors are, what proteins are floating by, etc.? There isn't any central processor controlling it all (unless you get into philosophical arguments about reality).

Von Neumann first considered this fundamental model of parallelism and space that now shows up in the systolic multipliers in Google's Tensor Processing Unit (TPU) and Amazon's Inferentia. These multipliers use an algorithm which is the fastest possible (in O-notation) using classical physics on a 2-dimensional surface (such as a chip). I've developed the fastest possible in 3-dimensional space. Very little is known about the limits of computing in 3-space. Now I need a student to create a universe with more dimensions

Randomly scattered compute elements: such as the internet of things. Tim Lewis looked at algorithms for these systems, where each processor had a small amount of power to broadcast messages. Broadcasts might interfere with each other, and processors didn't even know how many other processors were working on this task. Other processors may be helping different tasks, failed, or run out of power. This is a form of distributed computing, but not one of the standard models used there so we thought of it as a disorganized parallel system.

Neural nets: The brain has about 10^{11} neurons and 10^{15} synapses which somehow recognize faces, read research descriptions, etc. There are also somewhat fewer axons going through the white matter, forming long connections. Standard computer neural net models have a sequence of stages with a dense matrix of connections from one to the next, but this is impossible for giant nets and not at all how the brain is organized. Its connections are sparser and somewhat structured. Amy Nesky looked at aspects of this to reduce the time for deep learning. There are many other aspects of parallelism in learning that I'm interested in.

Randomly scattered, moving, simple compute elements: this is something I'd like to look at if I find a student interested in it. What is an appropriate model? How do they solve problems? Biology has many successful examples, but we don't.

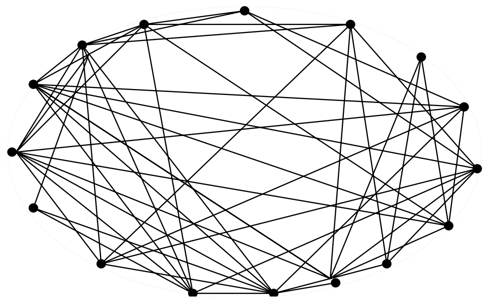
PRAMS: they have a shared memory that all processors can access. It's the standard theory model, but unrealistic. Phil MacKenzie won the university-wide Best Thesis award for his work using this model for graph problems. One of his algorithms took only $\Theta(\log^* n)$ ("log star") time. The model isn't a high priority for me now but this could change if a student wants to look at an interesting problem. Yujie An used a more realistic model which combined shared memory and mesh-connected processors.

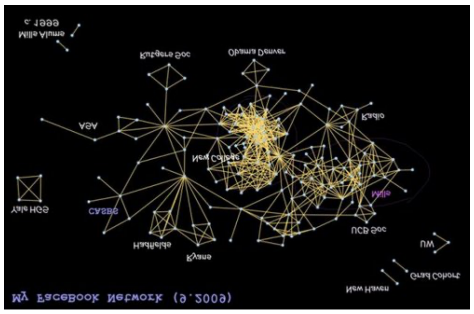
What Sort of Problems Have We Looked At?

Basically, the problems you consider in a good algorithms class, and some that are more complex. For many of the models, even for basic problems no one yet knows good algorithms. For example, for the scattered, but static, small compute elements with broadcasting, suppose each processor can determine the distance to neighbors. How fast can a minimal spanning tree be found? If the coordinates of a set of points is stored on one of the models, how fast can they find an MST (or a more complex graph or geometric construct)? Some implementations utilize space-filling curves (I thought that Hilbert's curves would be more useful than Z-ordering, but an undergrad showed they aren't). For some models, such as moving tiny compute units, the problems of interest might be quite different. Often one of the first problems is to determine what the interesting problems are.

Some algorithms use randomization, such as ones that minimize the energy used to determine the median (energy is an important resource, just like time and space, another thing you probably haven't studied). Randomized parallel algorithms are often more difficult to analyze because you need to determine the ending time of the slowest processor, not the sum of times on a serial processor. I.e., extremal statistics, not averages. Tim Lewis' algorithms, mentioned above, used randomization extensively.

Sometimes algorithms assume the input is random. However, the standard mathematical random models may be unreasonable. For example, the left image is based on the classical mathematical random graph model, but looks nothing like the large social graph on the right. On parallel computers, what Is the expected time to, say, find a minimal spanning tree in random social graphs? I'd like to find a student interested in such problems, and hopefully will find a faculty collaborator who needs large-scale parallelism to solve their problems.





Random models of points are also relevant. For example, for the scattered simple processors model mentioned above, trying to find a minimal spanning tree, what if they were randomly dropped from a helicopter, and their positions were determined by a uniform probability distribution on the unit square? This is a commonly used distribution (or uniform on the unit circle, 2-d normal distributions, etc). What if you use weaker assumptions, such as a uniform distribution on an unknown underlying region? For uniform distribution on the unit square a CRCW PRAM with common or collision detection write can determine the convex hull in $\Theta(1)$ expected time, but I don't know about the general case.

Some Applied Problems Using Parallelism

Below I mention some parallel work on ethical clinical trials. We've also done applications including: parallelizing Ford Motor Company's crash simulation; climate modeling; ground water remediation; creating the Earth Systems Modeling Framework which is used as the glue to combine and coordinate the various physics modules that are used at the National Center for Environmental Prediction (NCEP), part of NOAA, to produce the nation's weather forecasts (all the commercial forecasts base their work on this); and develop the Space Weather Modeling Framework, used to coordinate the modules used at the nation's Space Weather Prediction

Center. Many of the doctoral students in my Parallel Computing class are from other departments and do term projects related to their thesis work, so I've been on thesis committees of students in many other areas.

Some of The Non-Parallel Work We've Done

We've looked at efficient algorithms, and highly optimized implementations, for ethical clinical trials. These trials are a form of learning, trying to put more patients on the better drug (or treatment, or whatever is being tested) while still reaching a statistically valid outcome. Bob Oehmke was finalist for the best paper award at the Supercomputing conference for his very efficient parallel implementation, and very detailed analysis of how it was made so efficient, of one of these algorithms.

I've looked at algorithms for isotonic regression where instead of standard least squares regression to find a best line or plane through points, you try to find the best increasing function. It's more appropriate in many settings since assuming the relationship is linear is a very rigid assumption. For some problems this involves creating Steiner spanning graphs of points in d-dimensional space. Point
$(x_1, \ldots x_d)$ *dominates* point $(y_1, \ldots, y_d)$ iff $x_i \geq y_i$ for all $1 \leq i \leq d$. For a set of n points, representing domination ordering explicitly as a directed graph with edges (y,x) can require $\Theta(n^2)$ edges, but by adding Steiner points this can be reduced to Θ(n polylog n), while only increasing the number of points to Θ(n polylog n). This tradeoff can result in significantly faster algorithms. Adding another log factor gives a 2-spanner, where if x dominates y then there is a path of the form y -> s -> x, where s is a Steiner vertex. This special structure can be exploited when the L_∞ metric is used, and in some problems that are quite different.

If you're interested in some of these research topics, or have your own that you think is interesting, feel free to contact me. I won't consider you as a possible graduate research assistant until you officially apply, or are already a graduate student here (typically in CS, but sometimes in math or elsewhere). For some problems there are possibilities for undergraduates.