Research Statement

I am interested in the architecture of massively parallel computer systems. Moore’s Law is waning and individual cores will not get faster; parallelism is the only way forward to improve performance. However, despite decades of research into parallel and distributed systems, most programs remain sequential. Modern multicores and GPUs exploit only a small fraction of the parallelism available in applications and are hard to program. This impedes talented domain experts, for whom it is often arduous, and sometimes impossible, to write efficient and scalable parallel programs. Industry has reacted with massive engineering teams that build specialized accelerators for the few common applications with easy-to-exploit parallelism, like deep learning. Although these efforts boost performance by orders of magnitude, they are costly and only apply to a tiny minority of problem domains. The majority of applications are left behind.

My research seeks to make parallelism pervasive, enabling all programmers to write scalable parallel programs by default. My approach is to consider the full software-hardware stack, redesigning the programming model, system software, and architecture, to unlock types of parallelism that are extremely difficult to harness in current systems. For my dissertation I have built Swarm [1, 4, 5, 6, 7, 10], a new architecture that (i) exploits far more parallelism than conventional multicores, and (ii) is almost as easy to program as a sequential machine. Swarm achieves this by exploiting fine-grain ordered parallelism. It outperforms conventional multicores running state-of-the-art parallel algorithms by one to two orders of magnitude and sequential algorithms by up to $600 \times$ at 256 cores. Swarm accelerates applications that are conventionally deemed sequential, and it makes parallel applications scale even further. This architecture lays out fundamental building blocks for next-generation parallel systems. My future work will democratize the efficiency of accelerators and the massive scale of data centers, bringing their benefits to a broad set of problem domains.

Dissertation Research: A Hardware and Software Architecture for Pervasive Parallelism

Fundamentally, parallelizing a program consists of two steps: dividing work into tasks that may run concurrently, and specifying synchronization among tasks with potential data dependences to preserve correctness. Current multicores provide limited support in both dimensions, and thus limit performance for applications that are challenging in either. First, current multicores are amenable only to programs that consist of coarse-grain tasks. However, many programs are more naturally expressed using fine-grain tasks of a few tens to hundreds of instructions, and exploiting fine-grain parallelism is often the only way to make them scale. Unfortunately, fine-grain tasks cause large overheads in current systems, because software task management overheads (e.g., scheduling and load balancing) overwhelm the benefits of parallelism. Second, current systems lack efficient support for synchronization, and only work well for applications that synchronize infrequently. Unfortunately, many applications require frequent synchronization that is specified conservatively, and thus scale poorly beyond a few cores. Amdahl’s Law makes it clear that targeting only the easy parallelism yields diminishing returns; current multicores and GPUs choke performance with sequential bottlenecks.

In my dissertation I designed a new multicore, programming model, and cross-layer techniques that combat Amdahl bottlenecks. These techniques tackle problems spanning microarchitecture and ISAs up to operating systems, compilers, and programming languages. The resulting system, called Swarm, supports fine-grain tasks with minimal overheads and provides efficient, easy-to-use synchronization. Swarm achieves this by targeting a type of parallelism that is extremely taxing in both its task decomposition and synchronization: ordered irregular parallelism. Ordered parallelism is general and thus abundant, but hard to exploit. It allows expressing programs whose tasks (i) have data dependences that are unknown a priori, (ii) may create new tasks at runtime, and (iii) must execute in a programmer-defined partial order.

Swarm programs consist of tiny tasks that are ordered through timestamps. Swarm hardware uncovers parallelism by speculatively running tasks out of order, even thousands of tasks ahead of the earliest speculative task. It scales to large core counts due to its distributed structures and speculation mechanisms [5, 6]. Building on Swarm’s implicit synchronization through task order, I solved other fundamental problems in parallel computing. First, order enables breaking tasks more finely than ever before and sending compute close to the data that it accesses, which improves locality and is crucial to scale [4]. Second, I enhanced the system to combine speculative and non-speculative parallelism, executing speculatively only when required to scale, and otherwise improving efficiency with non-speculative parallelism [7]. Finally, order enables seamless composition of parallel algorithms, opening the door to parallelizing large complex programs [10]. Combining these techniques, my work achieves near-linear scaling to 256 cores on a diverse set of over 20 applications that span domains such as machine learning, graph analytics, databases, simulation, genomics, and more. What started as a small project has grown to include six graduate and two undergraduate students. This work has received both an IEEE Micro Top Picks award [6] and honorable mention, and is featured reading in courses at Carnegie Mellon University, the University of Wisconsin-Madison, MIT, and others.

Mark C. Jeffrey https://people.csail.mit.edu/mcj Massachusetts Institute of Technology
Mining ordered parallelism: I began my dissertation research by designing techniques to make ordered parallelism practical. The combination of order constraints and unknown data dependences among tasks makes ordered parallelism extremely hard to exploit in current multicores; runtime overheads like task scheduling negate the benefits of parallelism. To this end, we designed Swarm [5,6], a new programming model and microarchitecture to mine ordered parallelism. A Swarm program is expressed as a set of short, timestamped tasks. Each task may create new children tasks for later execution. Tasks appear to execute in timestamp order; those with equal timestamp appear atomic.

Swarm hardware is a tiled multicore. It is optimized to support tasks as small as tens of instructions. It extracts parallelism by speculatively executing available tasks out of order, while committing them in order. Doing this efficiently requires overcoming several hurdles. How can we perform scalable order-aware conflict detection, manage tiny tasks efficiently, keep cores busy with their work, and commit many tasks in order in a scalable fashion? Prior work solved some of these problems at small scale with centralized tracking structures. Swarm is the first system to solve these problems using distributed techniques that are amenable to multicores with hundreds of cores. For example, whereas prior work would abort all future tasks upon a mispeculation, Swarm selectively aborts descendant and data-dependent tasks with constant-overhead tracking structures. Swarm exploits the symmetry between regular conflict-checked writes, and the corrective writes of an undo log walk. As a result of our novel contributions, Swarm efficiently speculates thousands of tasks ahead of the earliest speculative task.

To evaluate Swarm, we built a cycle-level simulator based on Pin, with detailed models for cores, caches, the on-chip network, memory, and task and speculation overheads. A 64-core Swarm system is 3-18× faster than a conventional multicore running state-of-the-art parallel implementations of six challenging workloads. Moreover, because hardware accelerates task management and scheduling, Swarm is up to 117× faster than tuned sequential implementations of the same algorithms. Scalable Swarm programs are easy to port from their sequential counterparts. Swarm demonstrates that fine-grain ordered tasks provide a simple programming model with a highly scalable implementation.

Exploiting locality: Parallelism must not come at the expense of locality. Systems that support speculative parallelization—like Swarm, transactional memory, or thread-level speculation—can uncover abundant parallelism in challenging applications. However, these systems scale poorly beyond tens of cores because, unlike their non-speculative counterparts, they do not exploit the locality available in speculative programs. At first glance, locality and speculation seem to be at odds: exploiting locality requires knowing the data accessed by each task, but a key advantage of speculation is precisely that one need not know the data accessed by each task. However, with Swarm’s support for tiny ordered tasks, we find that in fact most of the data accessed is known at runtime when a task is created.

Building on this insight, we extended Swarm with spatial hints [4], a technique to enable the programmer to convey locality to hardware, to run tasks likely to access the same data in the same place. This exploits cache locality and avoids global communication. When a new task is created, software provides an integer hint that abstractly denotes the data that the task is likely to access. Hardware maps tasks with the same hint to the same tile to localize data accesses, and balances load by moving hints, and their tasks, around the system. For example, by leveraging task order, many graph algorithms can be restructured so that each task operates on a single vertex; using the vertex ID as a hint sends tasks for the same vertex to the same place, reducing data movement. Swarm with spatial hints empowers programmers with a semantically rich programming model to scale programs to hundreds of cores by exposing even more opportunities to exploit locality than in non-speculative architectures. At 256 cores, hints achieve near-linear scalability and outperform Swarm’s baseline random task mapping by up to 16×. Hints also make speculation far more efficient, reducing wasted work by 6.4× and network traffic by 3.5× on average.

Mixing speculative and non-speculative parallelism: Parallel systems should support both speculative and non-speculative parallelism. Even applications that need speculation to scale have some work that is best executed non-speculatively. Consider three examples. First, some tasks are well synchronized and running them speculatively adds overhead and needless aborts. Second, other tasks need speculation to scale but perform short actions, such as memory allocation, whose dependences are best hidden from hardware because they commute. Finally, non-speculative parallelism is required to perform irrevocable actions, such as file or network I/O, in parallel. Work on hardware transactional memory has considered these issues, but is limited to the domain of unordered parallelism. All systems for ordered parallelism, including Swarm, disallow non-speculative parallelism.

Espresso and Capsules enhance Swarm to bring the benefits of non-speculative execution to systems that support ordered parallelism [7]. Espresso is an expressive execution model that generalizes Swarm timestamps and spatial hints; it efficiently coordinates concurrent speculative and non-speculative ordered tasks. Espresso lets the system decide whether to run certain tasks speculatively or non-speculatively, reaping the efficiency of non-speculative parallelism when it is plentiful, while exploiting speculative parallelism when needed to scale. Capsules bring ideas from OS system
calls and virtual memory to speculative architectures. They enable speculative tasks to safely transition out of hardware-managed speculation and protect certain memory regions from speculative accesses. Espresso outperforms Swarm’s speculative-only execution by up to $2.5\times$. Capsules enable important system services, like a speculation-friendly memory allocator that improves performance by up to $69\times$. Espresso and Capsules provide a unified architecture for speculative and non-speculative, ordered and unordered parallelism.

**Composing parallelism:** Large complex programs can have large blocks that must appear atomic. Left sequential, these become Amdahl bottlenecks. For example, a database transaction could be millions of cycles long. Database system designers target parallelism among sequential transactions, but *there is plentiful parallelism within each transaction* if we break up each query and update into its own ordered task. But how do we ensure that all of the tasks representing one transaction appear atomic with respect to those of other transactions?

We developed Fractal [10], a general execution model that captures complex synchronization and order constraints beyond Swarm. Examples include composition (i.e., calling an ordered algorithm from within an ordered algorithm), or running independent ordered algorithms concurrently. Fractal groups tasks into *domains*, with tasks in each domain executing in the domain’s timestamp order. Tasks can create domains and set optional order constraints among domains. Our careful hardware-software interface virtualizes domains cheaply, allowing an unbounded number of domains in software with simple hardware support. Fractal accelerates challenging workloads by up to $88\times$.

**Improving throughput-oriented speculative parallelism:** How does a multithreaded core thread-issue policy affect global system performance? Although commercial hardware transactional memory systems use multithreaded cores, we find that conventional multithreading policies have damaging effects on speculative parallelism. This is because unlikely-to-commit tasks consume scarce resources and hurt the throughput of likely-to-commit ones. We designed speculation-aware multithreading (SAM) [1], a policy that aligns thread issue priority with commit order priority. SAM avoids pipeline interference from more- to less-speculative tasks, reducing wasted work. SAM makes speculative execution efficient on throughput-oriented systems, achieving GPU-like efficiency with serial-like programmability.

**Additional research:** While a graduate student at the University of Toronto, I developed new theory to aid speculative parallelism. A requirement in any parallelization tool, from Swarm and transactional memory to race detectors, is to efficiently track read and write address sets of parallel tasks or critical sections. Bloom filters are prevalent in this role, as they answer set membership queries with bounded space requirements. However, many prior parallelization systems employ Bloom filter intersection, a deviation from the intended design, that hurts performance due to previously misunderstood high rates of false conflicts. To clarify the performance implications, I developed an analytical probability model for Bloom filter null-intersection tests [3]. To mitigate the performance problems, I recommended and evaluated alternative Bloom filter configurations [2]. Our other work considers how to reduce Bloom filter sizes on soft (FPGA-based) transaction processors by exploiting application-specific memory access streams [8, 9].

**Future Research Directions**

It is an exciting time for computer architecture research. As Moore’s Law flatlines, system designers are increasingly drawn to three grand opportunities to improve performance within a limited transistor budget: *(i)* extracting parallelism, *(ii)* exploiting specialization, and *(iii)* distributing computation to hundreds or thousands of nodes. Interestingly, all three opportunities suffer from the same problems that I addressed in the context of parallelism. Hardware lacks general architectural support for specialization or distributed computing. As a result, programming models are restricted and these opportunities are extremely hard to pursue: they are only accessible to a tiny minority of expert programmers and hardware designers, and they only work well on a small set of applications.

My goal is to *bridge the widening gap between programmers and the scalable compute platforms of the future*. I will apply the same cross-layer approach that enabled me to broaden the scope of parallelism. This will make specialization and distributed computing accessible to all programmers, and useful for many applications that are off limits today. What is the right hardware and software architecture that exposes plentiful specialization, while maintaining the programmability of general-purpose sequential processors? What architectural mechanisms are necessary to unlock unseen forms of parallelism in both emerging and classic domains? Finally, can we scale fine-grain parallelism to entire clusters or datacenters, leveraging data-centric execution to forgo coherence across distributed memories? My work will look across layers of the stack, spanning applications, compilers, and language design, all the way down to architecture and accelerator design. I am excited to collaborate with faculty in other areas to bring these ideas to fruition.

**Making specialization pervasive:** I want to design the compute platform of the future, which will leverage on-the-fly specialization to accelerate challenging applications. *On the hardware side, an enticing prospect is a reconfigurable...*
FPGAs have shown the potential of reconfigurable computing. They accelerate many applications that are inefficient on multicore or GPUs. However, these fabrics are extremely hard to program: peak performance is only accessible to hardware designers. **On the software side, we need new techniques to make specialization accessible to everyone.** Current high-level synthesis (HLS) tools can compile high-level code into circuits. However, they can only extract limited forms of parallelism, like regular data and pipeline parallelism. Most applications are left behind.

This goal is ambitious, but we are making progress toward it. Swarm provides crucial building blocks for ongoing work on FPGA accelerators and a parallelizing compiler. On the hardware side, one of my collaborators is designing a Swarm FPGA accelerator that provides order-of-magnitude speedups on hard-to-parallelize applications like simulation and path finding. It exploits reconfigurability by replacing general-purpose cores with processing elements that are specialized to particular Swarm tasks. On the software side, I am collaborating on compiler-aided parallelization of sequential programs. Our LLVM-based compiler decomposes sequential code into highly parallel Swarm tasks, while respecting sequential semantics via task timestamp order. This work leans on our Fractal execution model to compose speculative parallelism across the whole program.

As a faculty member, I will first integrate these hardware and software thrusts by designing an FPGA-based heterogeneous platform that automatically compiles C++-based Swarm tasks into specialized processing engines, and uses few general-purpose cores to run less frequent tasks. This opens the door to further collaborative research in both hardware and software. First, what is the right fabric of future reconfigurable hardware? FPGAs target flexible gate-level customization, but this low level of abstraction is often unnecessary. Can task-level customization provide the right abstraction between high-level languages and reconfigurable hardware to make the fabric more efficient? Second, what is the right system architecture? Can the general-purpose cores and reconfigurable fabric tolerate latency to enable cheaper distributed designs, or should they be tightly integrated? Third, how should languages change to uncover task-based parallelism and specialization, while providing easy semantics for programmers? Some automated parallelization decisions benefit from profile-guided information. Can we use machine learning, trained on a pre-profiled corpus of code, to predict the tasks that are profitable to parallelize or specialize?

**Exploiting challenging types of parallelism:** Across domains in computer science and computational sciences, many important algorithms are rarely used today because they appear to be inherently sequential. In my dissertation, I have shown that ordered parallelism can make many of these algorithms scale. But task order is not sufficient to parallelize all applications.

I will explore new techniques to extract parallelism from challenging applications that goes beyond task order. I will continue to **take a holistic approach that starts at applications and inspires new hardware-software interfaces and microarchitectures.** For the seemingly sequential applications that are not amenable to Swarm’s methods of synchronization, what architectural support or new synchronization can relieve some burden from the programmer? Is there a more general framework or theory of synchronization that is easier to apply more widely? Conversely, do certain applications have simpler needs that enable reconfigurable synchronization mechanisms, consuming resources only when needed? I see opportunities in emerging and classic domains: machine learning (e.g., Gibbs sampling in sparse graphical models), numerical optimization (e.g., linear and integer programming approximations), aerial drones (e.g., 3D exploration and navigation), computational geometry (e.g., Voronoi diagrams), electronic CAD (e.g., place and route), bioinformatics, and more. An applications-to-microarchitectures strategy is critical to support a wide range of programmers and domain experts in a post-Moore era.

**Scaling fine-grain parallelism to distributed systems:** With Moore’s Law coming to an end, an enticing approach to improve performance is to scale applications across the thousands to millions of cores available in clusters and datacenters. However, these systems are much harder to program than multicore or GPUs: they have distributed memories and high communication latencies. Furthermore, fault tolerance becomes a first-class concern when we scale to this size. Swarm’s techniques are attractive to introduce fine-grain parallelism to cluster-scale computing for improved performance and ease of programming. However, Swarm leverages the cache coherence protocol for its speculation, which is scalable on chip, but multi-board coherence has high complexity and overheads. Fortunately, spatial hints and Espresso mechanisms enable sending tasks to the data that they access, minimizing data movement, and make it feasible to partition program data across distributed memories. Can we build on these ideas to render global coherence unnecessary and bring accessible fine-grain parallelism to cluster-scale systems? What execution model can facilitate fault tolerance for fine-grain parallel tasks?
References


