In recent years, hardware advances have brought shared-memory (a.k.a., multicore) parallelism to the mainstream. Now, nearly every computing device is a parallel computer, ranging from phones with a few cores to high-end machines with thousands. To benefit from continuing improvements in multicore processors, programmers have to expose parallelism in their applications. Fortunately, there are plenty of applications that stand to benefit from multicore parallelism, such as multimedia processing, computer games, small-scale simulations, graph analytics, build systems and compilers, etc. Yet, in a great deal of modern software development, parallelism remains a specialist activity, and not the default, as is desirable.

Part of the reason may be that the there is a premium to minimize the time to develop an application, as well as other factors, such as to write in a declarative style, to structure code for reuse and readability, etc. As such, programmers rely on high-level programming languages and abstractions in their programs. Programming languages can offer a great deal by supporting natural programming idioms and ease of programming. However, both supporting programs in high-level languages and achieving efficient parallel programs remains challenging: modern software often entails workloads that are irregular and hard to predict, e.g., as is typical in data analytics workloads or in any computation involving sparse matrics and real-world graphs. Moreover, such applications expose parallelism at different levels, leading naturally to nested parallelism, e.g., by nesting of parallel loops or recursive calls. Research on parallel programming languages has addressed these concerns by integrating parallelism deep into either the compiler, the runtime system, or both, as represented by the Cilk extensions to C++ [1] and languages, such as X10 [2], Haskell [3], Data-Parallel Haskell [4], etc.

My research focuses on both enhancing programmer productivity and improving program performance, especially concerning task parallelism. I focus on issues relating to scheduling parallel tasks efficiently on cores, but also on language design for task parallelism, and, more recently, on language support for accessing serialized data formats. In my work, I use techniques from both the programming languages and parallel algorithms communities. Of particular importance for my work is the use of language-based cost models, as exemplified by the work of Guy Blelloch [5, 6]. This approach has enabled scheduling techniques I have introduced, such as Heartbeat Scheduling, to be backed by end-to-end performance bounds, which crucially apply to all programs and all inputs, a property I believe to be crucial to achieve a future where parallel programming is ubiquitous.

1 Granularity control

Programmers using task-parallel languages to target multicore systems could increase their productivity if it were possible to simply express all opportunities for parallelism in their applications, no matter how fine or coarse grained. In the current state of the art, however, such fearless parallelization comes with significant performance risks. The problem is that it is not always profitable to exploit every opportunity for parallelism. Parallelize too much, and the program may be much slower than a serial program. But parallelize too little, and cores may be underutilized. Finding a balance between these two extremes, and doing so consistently, is one of the keys to expanding the scope of parallel programming in the multicore era.

Granularity control has received much attention in the research literature, from the 80's to the present [7, 8, 9, 10, 11, 12, 13]. Basic approaches include "outer parallelism", whereby only the

outermost loops or calls are parallelized, and "manual pruning". Outer parallelism suffices when there is guaranteed to be large amounts of parallelism in outer loops, but falls short when used for modern multicore software, which often features highly irregular parallelism. Use of manual pruning fails for similar reasons, as well as reasons of performance portability: threshold settings that perform well on one platform may not on another [10]. There are more sophisticated heuristics that involve varying levels of compiler and runtime techniques. But such heuristics suffer from worst-case inputs, which can be difficult to diagnose. This situation complicates the job of performance engineering: any given performance deficiency could be a consequence of one of many potential problems, including granularity control. Is there an approach that is backed by suitable formal guarantees of efficiency and that supports a practical implementation for multicore?

I answered this question in the affirmative by devising Heartbeat Scheduling [14]. The basic idea of Heartbeat Scheduling is to spawn new tasks only at a beat (i.e., the "heartbeat"), and to make sure to do useful work in between beats. In an implementation of Heartbeat Scheduling, the program always starts executing in the conventional, serial fashion, but remembers latent opportunities for parallelism, e.g., instances of parallel calls and loops, by leaving behind a little bookkeeping. Every time a task finishes executing a small chunk of its local computation, the language runtime makes an attempt to liberate some of the latent parallelism by interrupting the task, at which point the handler inspects the call stack of the task. If it finds some latent parallelism, the handler promotes the parallelism into a proper task, e.g., by placing a record of the task in a load balancing pool. From the pool, the task can be executed by a remote core, if migrated by the load balancing algorithm.

In an experimental study, I demonstrated that a prototype implementation performs as well or better than the state-of-the-art Cilk scheduler across a range of modern benchmarks, and could do so entirely automatically, requiring no manual grain settings and no special-purpose heuristics. Crucially, Heartbeat Scheduling is the first solution for granularity control that is backed by end-to-end upper time bounds, which apply to all task-parallel programs and all inputs. The bounds guarantee the two key desired properties. First, all task-related overheads are well amortized. Second, only a marginal amount of parallelism is pruned away, amounting in particular to a small constant factor decrease in the parallelism of an application. I believe that such performance bounds are crucial to the success of the approach, because they provide the rigor needed to deliver predictable performance, and thereby provide the assurance needed to simplify performance engineering for multicore.

Heartbeat Scheduling is not the only approach backed by formal guarantees. I also helped devise and implement Oracle Guided Scheduling, a different approach with some complementary tradeoffs [15, 16, 17]. A recent publication of this work was awarded the SIGPLAN Research Highlight [17]. In Oracle Guided Scheduling, the idea is to amortize task overheads by predicting the cost of executing tasks. Our prediction algorithm requires the programmer to annotate their functions with asymptotic-cost functions, and to use a small runtime system that collects data from both the cost functions and the cycle counter. The algorithm uses this data to incrementally improve granularity, before converging on provably good settings. Although both have the same goal, Oracle Guided Scheduling and Heartbeat Scheduling are largely complementary: the former requires programmer annotations but the latter does not, whereas the latter can be implemented as a standalone library, for any language but the former requires compiler and runtime support.

2 Task scheduling

In task-parallel programming, the order in which tasks are executed and on which processor they run is determined by the scheduler. The scheduling policy has a major impact on performance and is therefore the subject of a large body of research. Of the alternatives, perhaps the most popular and well-studied scheduler is work stealing. Many state-of-the-art implementations of work stealing, such as that of Chase and Lev [18], use sophisticated concurrent deque data structures, both to buffer pending tasks and to allow tasks to migrate across processors. Although work stealing with concurrent deques performs well for many applications, there are some important applications, such as parallel graph traversal, and there are some languages, such as Parallel ML with split-heap garbage collection [19], for which concurrent deques do not perform well. Part of the reason is that even a small modification to the deque data structure is likely to be challenging: it is a delicate concurrent algorithm whose verification has spanned multiple research works. Useful extensions, such as steal-half work stealing [20], have been proposed, but such proposals require significant technical work to design a new concurrent data structure. Furthermore, concurrent data structures impose a costly fence operation on local accesses, which can harm performance for fine grain parallel applications.

In collaboration with Acar and Charguéraud, I helped with the design and implementation of work stealing with private deques [21]. In this algorithm, the deque is represented by a conventional, serial data structure, and tasks are migrated between cores by message-passing communication. We proved that the algorithm is backed by theoretical guarantees similar to those of state-of-the-art work stealing with concurrent deques. In experimental work, we demonstrated that our algorithm performs as well as the state-of-the-art implementation used by Cilk.

These theoretical and practical advances opened a new space for customizable work stealing variants. In subsequent work, we demonstrated an application of our private-deques algorithm to in-memory graph traversal [22]. I believe our algorithm remains the fastest algorithm for reachability analysis of in-memory graphs on multicore systems. A key feature of our algorithm is its robustness in the face of arbitrary graph structures: for graphs, such as a long chain, for which there is no available parallelism, our algorithm performs as well as the best serial algorithm, for graphs with scarce parallelism, our algorithm achieves best possible speedup, and for graphs with abundant parallelism, our algorithm performs as well or better than the best parallel algorithm. This ability to scale both up and down is, I believe, a crucial and underrepresented feature of multicore algorithms.

As a byproduct of our graph-reachability work, we also introduced a novel data structure, the Chunked Sequence [23, 24], which is useful in its own right. A Chunked Sequence provides a representation of a sequence, with the operations of a double-ended queue, logarithmic-time split, and logarithmic-time concatenation. Given its collection of operations, the structure is useful for implementing the frontier of a parallel graph traversal, and is flexible enough to support arbitrary sequences, interval trees, and dynamic dictionaries. Our Chunked Sequence structure has also been used in scientific computing, in a plasma simulation application [25].

3 Language design and implementation

As programming adapts to multicore parallelism, so must programming languages. A major challenge facing such adaptation relates to the design of language constructs. Languages need constructs that can harness the various forms of hardware parallelism, yet provide semantics that support high-level reasoning and modular programming. I began my research career working toward such goals

in the context of the Manticore project [26, 27, 28, 29, 30, 31, 32]. Manticore is a parallel variant of the functional programming language, Standard ML. In helping with the development of Manticore, I designed and implemented a new method for handling exceptions in the parallel setting, where a raised exception can lead to early termination of some tasks. During my PhD, I implemented large parts of the backend of the Manticore compiler, and its runtime system. The system continues to serve as a research vehicle for students at the University of Chicago.

In recent work, I helped design the Dag Calculus [33], a new task-based language that features flexible, high-level control over complex synchronization patterns, such as parallel pipelines and loosely structured graph traversals. The Dag Calculus was motivated by our realization that some popular linguistic mechanisms, namely async-finish and parallel futures, are in fact closely related, and complementary, even though previously they had been designed independently for different settings (imperative and functional languages, respectively). We demonstrated that Dag Calculus can efficiently encode synchronization patterns of both async-finish and parallel futures. In follow-up work, I helped design a lightweight synchronization barrier for async-finish (and by extension, our Dag Calculus) [34]. Our barrier data structure is based on the Scalable Non-Zero Indicator (SNZI) concurrent data structure. Our adaptation of SNZI implements a special kind of barrier synchronization, which allows for existing tasks to add new tasks to a barrier on the fly. Crucially, we proved that our barrier has amortized constant-time complexity, including contention, which is a rarity for concurrent data structures, and guarantees efficient synchronization for multicore applications that use the async-finish pattern.

In a recent, and ongoing collaboration, I helped with a novel typed functional language for programming directly on serialized data. This language has a compiler, named Gibbon [35], and it supports high-level programming with algebraic data types, a classic feature from typed functional programming, but in a memory-efficient fashion. Conventional algebraic data types store objects using a pointer-based representation. Such pointer-based representation is unfriendly to modern chips, which favor inline, serial access patterns. In our language, tree data structures arising from algebraic data types are laid out serially, in a format that is much friendlier to modern chips, and can easily move from in-memory to on-disk storage. The language also supports a mix of pointers and serialized layout, with a type-safety result that rules out the common pitfalls of programming with complex serialized data structures. I proved the first type-safety result for the Gibbon type system. A key feature of the type system is its end-witness logic, which specifies the well-formedness criteria for heap objects.

4 Future work

In the past year, I initiated a collaboration with researchers specializing in algorithms, systems and compilers at Carnegie Mellon University, Northwestern University, and IIT Chicago. The initial goal of our collaboration is to bring Heartbeat Scheduling from prototype to production grade. To this end, we have focused on two key challenges: the compiler support and the alarm mechanism needed to provide a regular heart beat. This project has been partly funded by a planning grant from the National Science Foundation, which we were awarded based on our proposal reporting initial results. We plan to submit our results for publication soon. Ultimately, I want to see Heartbeat Scheduling become widely used, for example, as an extension of the OpenCilk code base. Completion of this work will require further development of compiler support.

High-level languages, and functional ones in particular, are my passion. It has long been my

goal to see functional programming enter into new niche areas of programming. I believe that the functional language Gibbon and the like can bring functional programming to a new niche area, such as high-performance and distributed functional programming. The growing interest in this niche area is exemplified by emerging functional languages, such as Gibbon and Unison [36], which rethink the relation between language and program data. A key idea is to retain high-level, type-safe constructs, such as algebraic data types, while enabling low-level control (or tight coupling) with data layout. A first step toward realizing the promise of such languages is to bring support for parallelism. My goal is to bring parallelism as the default to Gibbon. I believe that the combination of automatic granularity control and finer control over serialized data layout can bring functional languages, such as Gibbon, to a much higher level of performance and productivity, thereby opening new, attractive use cases, such as parallel compilers and lightweight data analytics tools.

Over the past decade, there have been exciting proposals for parallel languages that support mutable data, but guarantee deterministic-by-construction semantics, which has great potential for improving safety and enhancing programmer productivity. Such language proposals include LVars [37] and Concurrent Revisions [38]. Although both have mature theoretical foundations, neither has gained widespread support in general-purpose programming. One major hurdle is performance. Both approaches require sophisticated support from the compiler and runtime, and currently state-of-the-art techniques deliver performance that is poorer than lower-level, unsafe methods. I want to explore compilation and runtime approaches, such as fine-grain blocking, and extensions of Heartbeat Scheduling that can significantly raise the efficiency of LVars and Concurrent Revisions.

To fully harness multicore systems, it is essential to utilize SIMD instructions. Recent work by Ren et al. proposes a new method for extracting SIMD parallelism from recursive task-parallel programs [39]. This work is exciting for me because it frames the problem of SIMD parallelization as a task-scheduling problem. As such, I believe there is a path to integrate it with my current approach to Heartbeat Scheduling. The goal is to have compiler and runtime support that can both utilize Heartbeat Scheduling and SIMD parallelism from high-level task-parallel codes, such as those of Cilk.

References

[1] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 207–216, 1995.

[2] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek 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. ACM, 2005.

[3] Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 339–401, 2011.

[4] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M T Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In Ramesh Hariharan, Mad-

havan Mukund, and V Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 383–414, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Guy E. Blelloch and John Greiner. Parallelism in sequential functional languages. In *Functional Programming Languages and Computer Architecture*, pages 226–237, 1995.

[6] John Greiner and Guy E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, March 1999.

[7] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[8] Marc Feeley. A message passing implementation of lazy task creation. In *Parallel Symbolic Computing*, pages 94–107, 1992.

[9] Vladimir Janjic and Kevin Hammond. Granularity-aware work-stealing for computationally-uniform grids. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 123–134, 2010.

[10] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Conference record of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, New York, New York, USA, June 1990. ACM Press.

[11] P. Lopez, M. Hermenegildo, and S. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 21:715–734, June 1996.

[12] José E. Moreira, Dale Schouten, and Constantine D. Polychronopoulos. The performance impact of granularity control and functional parallelism. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '95, pages 581–597, 1996.

[13] A. Duran, J. Corbalan, and E. Ayguade. An adaptive cut-off for task parallelism. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2008.

[14] Umut A Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. Heartbeat scheduling: Provable efficiency for nested parallelism. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '18. ACM, 2018.

[15] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, volume 46 of *OOPSLA '11*, pages 499–518. ACM, 2011.

[16] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. Cambridge University Press, 2016.

[17] Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. Provably and practically efficient granularity control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 214–228, New York, NY, USA, 2019. ACM.

[18] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *SPAA '05*, pages 21–28, 2005.

[19] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language, 2007.

[20] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289, 2002.

[21] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 48 of *PPoPP '13*, pages 219–228. ACM, 2013.

[22] Umut A Acar, Arthur Charguéraud, and Mike Rainey. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 67:1–67:12. ACM, 2015.

[23] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Theory and practice of chunked sequences. In *The 22nd Annual European Symposium on Algorithms*, ESA '14, pages 25–36. Springer, 2014.

[24] Arthur Charguéraud and Mike Rainey. Efficient Representations for Large Dynamic Sequences in ML. ML Family Workshop, September 2017.

[25] Yann Barsamian, Arthur Charguéraud, Sever A. Hirstoaga, and Michel Mehrenberger. Efficient strict-binning particle-in-cell algorithm for multi-core simd processors. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 749–763, Cham, 2018. Springer International Publishing.

[26] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in manticore. In *The 13th ACM SIGPLAN International Conference on Functional Programming*, volume 43 of *ICFP '08*, pages 119–130. ACM, 2008.

[27] Matthew Fluet, Mike Rainey, and John Reppy. A scheduling framework for general-purpose parallel languages. In *The 13th ACM SIGPLAN International Conference on Functional Programming*, volume 43 of *ICFP '08*, pages 241–252. ACM, 2008.

[28] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. volume 20, pages 537–576. Cambridge University Press, 2010.

[29] Mike Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, 8 2010.

[30] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[31] Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy tree splitting. In *The 20th ACM SIGPLAN International Conference on Functional Programming*, volume 45 of *ICFP '10*, pages 93–104. ACM, 2010.

[32] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-only flattening for nested data parallelism. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 48 of *PPoPP '13*, pages 81–92. ACM, 2013.

[33] Umut A Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. Dag-calculus: A calculus for parallel computation. In *The 26th ACM SIGPLAN International Conference on Functional Programming*, ICFP '16. ACM, 2016.

[34] Umut A Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic nonzero indicators for nested parallel computation. In *22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17. ACM, 2017.

[35] Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: A language for programs operating on serialized data. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19. ACM, 2019.

[36] Unison Programming Language. https://www.unisonweb.org/.

[37] Lindsey Kuper and Ryan R Newton. Lvars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*, pages 71–84. ACM, 2013.

[38] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *ESOP*, pages 116–135, 2011.

[39] Bin Ren, Gagan Agrawal, James R Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. Simd parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE, 2013.