Author Retrospective for

Optimizing for Parallelism and Data Locality

Kathryn S. McKinley
Microsoft Research
mckinley@microsoft.com

ABSTRACT

Today there is an urgent need for algorithms, programming language systems and tools, and hardware that deliver on the potential of parallelism due to the end of Dennard scaling. This work (from my PhD dissertation, supervised by Ken Kennedy) was one of the early papers to optimize for and experimentally explore the tension between data locality and parallelism on shared memory machines. A key result was that false sharing of cache lines between processors with local caches on separate chips was disastrous to the performance and scaling of applications. This retrospective includes a short personal tour through the history of parallel computing, a discussion of locality and parallelism modeling versus a polyhedral formulation of optimizing dense matrix codes, and how this problem is still relevant to compilers today. I end with a short memorial to my deceased co-author and advisor Ken Kennedy.

Original paper: http://doi.acm.org/10.1145/143369.143427

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers; Optimization*

Keywords

False Sharing; Loop Transformations; Parallelism; Locality

Parallel Computing: A Personal History

Parallel computing seemed to be entering its heyday in the late 1980s and early 1990s. At Rice in 1989, Ken Kennedy was awarded an NSF Science and Technology Center for the Center for Research on Parallel Computing (CRPC) as the Principal Investigator. The CRPC started with seven sites and eventually included 400 researchers, staff, and graduate students. Their technical expertise spanned parallel algorithms, compilers, runtimes, and hardware. The CRPC vision that Ken, his collaborators, and students shared was to invent parallel algorithms for critical problems in science, coupled with programming language tools, such as compilers, runtime systems, and programming environments, that made them run fast. We were not trying to solve the *dusty-deck* problem of automatically converting sequential algorithms to parallel ones. We understood that parallel and sequential algorithms for the same problem require different solutions. However, tools would do heavy lifting to map application parallelism to hardware parallelism, such that the programmers would not have to reimplement their algorithms for each new parallel architecture. A key aspect of this problem is balancing parallelism, sharing between tasks, and memory usage, which was the topic our paper addressed.

In this same period, a number of established companies and startups, such as Sequent, had introduced parallel machines. The Sequent Symmetry was the machine on which we reported our results. It was not yet clear that the research and development challenges of parallel computing would make it too costly to win in the market place in the short term. By the mid 1990s, this generation of parallel computers together with some of the companies that built them were pulled under by the economic tide and huge success of personal computers with a single processor. Clock scaling was delivering exponential performance improvements. Sequential software was easier to build; the sequential programming interface was stable across generations of hardware and software; and hardware was twice as fast every 18 months. Why invest in parallel computing?

In hindsight, abandoning sustained national funding, much research, and advanced development on parallel computing was a mistake. Today, leakage power and other manufacturing challenges at small feature sizes have ended Dennard scaling and will soon end Moore's Law. Vendors have turned to parallelism as a strategy to continue increasing computer capacity to meet computational demands in science, government, and consumer markets. However, researchers and companies are still struggling to deliver on the potential of parallelism. For example, even though manufacturers can fit 20 to 100 processors on a single chip, many commodity server, client, and mobile chips have 1 to 12 CPUs with 2 to 24 hardware contexts. Building, compiling for, programming, and porting software between parallel systems is still hard.

Parallelism and Data Locality

Part of the reason programming and optimizing parallel machines, even shared memory machines, is hard is due to real sharing and *false sharing* of data between parallel tasks. In a typical shared memory parallel machine, each CPU has some amount of private cache memory and some shared main memory. In the 1990s, the CPUs were on separate chips and today they are on the same chip, but the basic memory hierarchy structure is similar.

Hardware caches exploit the observation that programs often exhibit *temporal locality* – they reuse the same data — and *spatial locality* — they access adjacent data elements — close together in time. For example, a loop that sequentially iterates over a vector of

ICS 25th Anniversary Volume. 2014
ACM 978-1-4503-2840-1/14/06.
http://dx.doi.org/10.1145/2591635.2591646.

integers (allocated as a single vector) exhibits spatial locality with a reuse distance of one iteration. Caches store the most recently used items and cache lines typically contain between 32 B to 256 B of data. The fewer accesses between reuse of data on the same cache line, the more likely the cache will be to still hold the line. Unfortunately, since the architecture only moves and tracks data at a cache line granularity, *false sharing* occurs when two tasks on different CPUs want to write *independent* data that resides on the same cache line at the same time, they cannot. Only one cache can write the line at a time, which requires invalidating and moving the cache line, serializing execution.

Figure 1 in the paper shows the effect of false sharing on 18 processors of a Sequent shared-memory parallel machine executing an addition of three 2D dense matrices and storing the result. For the same amount of work, without false sharing, a speedup of 18 (best) is possible. With false sharing, speedup ranges between 6 and 9 (mem). That figure would have been better labeled "False sharing more than halves speedup."

The remainder of the paper presents an optimization algorithm for dense matrices that seeks (1) to minimize the reuse distance for cache lines for each parallel task, (2) to introduce parallelism at the outermost loop possible, and (3) to eliminate false sharing by strip mining, such that the outer loop is parallel and each parallel iteration never shares a cache line with another parallel iteration.

The algorithm uses data dependence analysis to detect parallelism and temporal reuse [4, 5, 12]. It precisely computes spatial and temporal cache line reuse on inner loops and then generalizes to the entire loop nest. It next determines the loop ordering that maximizes expected cache line reuse, and directly permutes the loops to this order or the closest legal loop order. The next phase introduces parallelism at the coarsest possible granularity (outermost loop) without causing false sharing. It uses a combination of loop permutation and strip mining. A machine specific cost model determines if the parallel loop is of sufficient granularity to deliver speedups. If the parallel loop also provides cache line reuse, the algorithm strip mines the loop based on the line size and number of processors, permuting the iterator loop to the outermost legal position and leaving the *strips* of consecutive iterations with reuse in place to deliver cache locality. This transformation process ensures that none of the outermost parallel loop iterations share cache lines, eliminating false sharing.

Historical Positioning

Our work on optimizing compilers for parallel architectures at Rice was greatly influenced by the research at UIUC, IBM, and Stanford. David Kuck and his PhD students at UIUC introduced the first analyses and optimizations in this area. For example, Uptal Banerjee introduced the first practical dependence tests to detect parallelism [4, 5] and Abu-Sufah introduced the first optimizations for improving data locality [1]. Allen and Kennedy showed how to use dependences to correctly permute loops and introduce parallelism [3], and with Callahan, subsequently showed how to maximize the granularity of parallelism [2].

Irigoin and Triolet introduced the use of a polyhedral model of dependences to increase parallelism [14] and the work of Wolf and Lam extended this theory to introduce parallelism at all possible levels [28]. Wolf and Lam also used this theory to produce all the legal loop permutations and loop skews and then selected the one that maximized cache reuse [27], however they did not consider both locality and parallelism at once.

At the time, the polyhedral and unimodular models were restricted to permuting, reversing, and skewing perfectly nested loops operating on dense matrices. Optimizing one loop in isolation was and is not sufficient for the best performance; data reorganization, loop fusion, and loop distribution add substantial benefits [7, 10, 11, 16, 21, 22, 25]. Our model directly computed the best permutations for locality and parallelism and we showed [7, 16, 20] how to directly derive good loop fusion and distribution choices, which the polyhedral model could not yet perform. Our approach also had the advantage that the resulting code was human readable and suitable for use in an interactive parallelization tool [17]. Several comercial tools implemented our approach, including the DEC Alpha compiler.

Today, the polyhedral approach combines fusion and distribution with other loop optimizations [11, 22, 25] and is in wide use in commercial and research compilers. These techniques use heuristic search based on models, some similar to our model, to specialize code for a variety of parallel architectures.

Loop nest optimization remains an active area of research [11, 22, 25], in part, because each generation of parallel hardware offers new challenges and opportunities. Using the compiler to adapt applications to a parallel machine achieves portability and high performance, substantially reducing developer effort. Loop optimizations are critical to optimization in many modern settings, including GPUs [18], high-level hardware synthesis [8, 19, 24], autotuning [9, 23, 25], and library construction [13, 26].

None of the early work on locality or parallelism considered false sharing and locality and parallelism interactions. Our paper highlighted the problems that false sharing caused, which subsequently other researchers also sought to eliminate. For example, Jeremiassen and Eggers transformed data with padding [15] and Ding and Kennedy performed array regrouping [10] to eliminate false sharing and improve locality. Later my PhD student Emery Berger, others, and I showed how the Hoard memory allocator avoided false sharing to produce more scalable performance, compared to prior C/C++ memory allocators [6]. The Hoard algorithm is now widely used, for example, in Apple's iOS.

Professor Ken Kennedy (1945 - 2007)

Ken loved his research and bringing people together on both a small and large scale. He founded the Rice Computer Science Department in 1985, CRPC, the Los Alamos Computer Science Institute (LACSI), and co-chaired with Bill Joy, the President's Information Technology Advisory Committee (PITAC) from 1997 to 1999. This first PITAC committee initiated the NSF ITR program, which funded the research of 1000s of faculty and graduate students over a 10 year period. In his own research life, he could not bear to cut back on people or research; he would instead raise more money. He felt it was a personal failure if any student who worked with him did not graduate and graduated over 40 PhD students. Together with his students and colleagues, he invented programming language technologies still in use today, such as register allocation, strength reduction, prefetching, loop transformations for scalar, vector, and parallel architectures (this paper is an example), interprocedural analyses, interactive parallel programming tools, FortranD, and telescoping languages.

In the last few years of his life as he struggled with pancreatic cancer, he kept working because he loved his life, his students, and his work. Ken laughed a lot and really loudly, especially at his own jokes. I still miss him.

Acknowledgements I would like to thank again my graduate student colleagues Chau-Wen Tseng, Preston Briggs, Cliff Click, Ervan Darnell, and Nathaniel McIntosh for their contributions to this work. I would also like to thank them as well as David Callahan, Keith Cooper, Mark Hall, Mary Hall, Seema Hiranandani, Marina

Kalem, Amy Pullen, Linda Torczon, and Scotty Strahan (my husband) for making graduate school a wonderful personal and intellectual experience. I thank the anonymous reviewer for improving the first version and Steve Blackburn, Mary Hall, and Todd Mytkowicz for discussions and comments on this version.

References

[1] W. A. Abu-Sufah. *Improving the Performance of Virtual Memory Computers, PhD Dissertation*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[2] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *ACM Symposium on the Principles of Programming Languages (POPL)*, pages 63–76, Munich, Germany, Jan. 1987.

[3] J. R. Allen and K. Kennedy. Automatic loop interchange. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 75–90, Montreal, Canada, June 1984.

[4] U. K. Banerjee. Data depedence in ordinary programs, Master's Thesis. Technical Report 76-837, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1976.

[5] U. K. Banerjee. *Depedence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, 1988.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, Cambridge, MA, Nov. 2000.

[7] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 252–262, San Jose, CA, Oct. 1994.

[8] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Annual Design Automation Conference (DAC)*, pages 1233–1238, 2012.

[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *ACM/IEEE Conference on Supercomputing*, pages 4:1–12, 2008.

[10] C. Ding and K. Kennedy. Inter-array data regrouping. In *Languages and Compilers for Parallel Computing*, pages 149–163, San Diego, CA, Aug. 1999.

[11] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 1995.

[12] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 15–29, Toronto, Canada, June 1991.

[13] K. Goto and R. van de Geijn. High-performance implementation of the Level-3 BLAS. *ACM Transactions on Mathmatical Software (TOMS)*, 35(1):4:1–14, July 2008.

[14] F. Irigoin and R. Triolet. Supernode partitioning. In *ACM Symposium on the Principles of Programming Languages (POPL)*, pages 319–329, San Diego, CA, Jan. 1988.

[15] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 179–188, Santa Barbara, CA, July 1995.

[16] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–321, Portland, OR, Aug. 1993.

[17] K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[18] M. Khan, G. R. P. Basu, M. Hall, C. Chen, and J. Chame. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Transactions on Architecture and Code Optimization*, 9(4):31:1–25, 2013.

[19] Q. Liu, G. A. Constantinides, K. Masselos, and P. Y. K. Cheung. Combining data reuse with data-level parallelization for FPGA-targeted hardware compilation: A geometric programming framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(3):305–315, Mar. 2009.

[20] K. S. McKinley. A compiler optimization algorithm for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8):769–787, Aug. 1998.

[21] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 94–104, Cambridge, MA, Oct. 1996.

[22] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *ACM Symposium on the Principles of Programming Languages (POPL)*, pages 549–562, 2011.

[23] M. Puschel, J. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[24] B. So, M. W. Hall, and P. C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 165–176, 2002.

[25] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2009.

[26] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

[27] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Notices*, pages 30–44, Toronto, Canada, June 1991.

[28] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.