Improving the Performance of Parallel Loops in OpenCilk

by

Luka Govedič

S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology(2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

© 2023 Luka Govedič. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Luka Govedič
Department of Electrical Engineering and Computer Science
May 19, 2023

Certified by: Tao B. Schardl
Research Scientist
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Improving the Performance of Parallel Loops in OpenCilk

by

Luka Govedič

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

For good performance, parallel loop scheduling must achieve low scheduling overheads and multidimensional locality in nested loops. This thesis explores both challenges and contributes an extension to randomized work-stealing for first-class loop support that reduces scheduling overheads.

Randomized work-stealing schedulers traditionally execute parallel-for loops using parallel divide-and-conquer recursion, which is theoretically efficient and scalable but can incur substantial overheads in practice. This thesis extends randomized work-stealing with a custom work-stealing protocol called on-the-fly loop splitting. I introduce loop frames to make work stealing on parallel-for loops more efficient and flexible.

Loop frames make two key changes to work stealing for parallel-for loops. First, loop frames extend work stealing by directly encoding information about intervals of loop iterations in the runtime. Loop frames add first-class support to work stealing for parallel-for loops that composes with classical randomized work stealing. Second, loop frames allow intervals of loop iterations to be split on-the-fly, such that worker threads attempt to steal half of the unexecuted loop iterations rather than a deterministically constructed partition of loop iterations. On-the-fly loop splitting allows for more flexible dynamic load balancing of loop iterations while keeping the work overheads low and maintaining the theoretical efficiency of divide-and-conquer.

I evaluate loop frames in practice by implementing loop frames in the OpenCilk runtime system. In particular, loop frames augment the THE protocol from Cilk to coordinate updates to loop frames. I observe that loop frames and on-the-fly loop splitting incur substantially less overhead than the divide-and-conquer algorithm without sacrificing parallel scalability.

Finally, I study the impacts of increased locality in more than one dimension in nested loop applications. Results show that both cache-aware and cache-oblivious reordering of nested loop iterations can result in performance benefits up to a factor of 1.7×.

Thesis Supervisor: Tao B. Schardl

Title: Research Scientist

Acknowledgments

While I received support from many, no one was as instrumental during this research as Dr. Tao B. Schardl. I am extremely grateful for his continued guidance and countless insightful discussions we had over the course of over three years. He helped ignite my passion for performance engineering research and was indispensable in my becoming the researcher I am today.

I would also like to express my gratitude to Prof. Charles Leiserson for supporting me in every aspect of my academic and professional career and being a limitless source of advice. I am thankful to all other Supertech research group members and Prof. Rezaul Chowdhury for their repeated feedback and interest in my project.

I cannot thank Sam D'Alonzo enough for being a constant source of support. I want to thank all of my friends and communities I have been a part of, and my family for helping this dream happen in the first place.

I would also like to acknowledge that I used ChatGPT to proofread parts of my thesis.

This research was sponsored in part by the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Contents

1 Introduction **15**

2 Background **23**

2.1 The dag model of multithreading 23

2.2 Randomized work stealing 24

2.3 Cache-oblivious and cache-aware algorithms 25

3 Loop frames **27**

3.1 Implementation 27

3.2 Synchronization protocol 29

4 Empirical evaluation of loop frames **35**

4.1 Microbenchmark experiments 36

4.2 Application experiments 37

5 Study of improved locality in nested loop applications **41**

5.1 Locality in multidimensional loops 42

5.2 Implementations 44

5.3 Benchmarks and performance results 46

5.3.1 Matrix multiply 47

5.3.2 Blur 50

5.3.3 Transpose 55

5.4 Takeaways 56

6 Related work **59**

7 Conclusion **61**

List of Figures

1-1 A DAG for a simple fork-join program and the corresponding pseudocode. In this example, there are 4 serial tasks, A-D, that need to be executed. A needs to be executed before B and C (which can be executed in parallel). Both B and C need to finish before D. In the pseudocode, A is called first. Then, B is spawned, meaning C can execute before B is done. Sync on line 4 forces both B and C to complete before the execution moves on to D. 16

1-2 Pseudocode for the typical divide-and-conquer implementation of a parallel-for loop. The function BODY encodes the body of the parallel-for loop. The **spawn** and **sync** keywords allow for parallel execution of the operations in the function. The constant G is used to coarsen the recursion to mitigate performance overheads. 19

3-1 Pseudocode for loop frame operations. LFPARFOR is used by the worker that enters the loop frame, while SIMPLESTEAL is used by the thief when stealing a loop frame from a victim. LFPARFORHELPER is used by both to actually execute all iterations and contains a familiar branch, body, and increment that can be found in all for loops. NEXT returns true if there are more iterations available, and false otherwise. 28

3-2 Pseudocode for the optimized synchronization protocol to coordinate operations on loop frames. The argument D denotes a victim's deque. The functions lockOwnDeque and unlockOwnDeque acquire and release the lock on the executing worker's deque. The function isLoopFrame tests if the given frame is a loop frame. 31

4-1 Median speedup with respect to number of processors for the `daxpy`, `nqueens`, and `mandelbrot`, appearing from left to right. Each plot features loop frames in orange squares and the divide-and-conquer algorithm in yellow triangles. The `daxpy` plot also includes the results of the simpler protocol for loop frames (briefly mentioned in Chapter 3) in red diamonds. The `nqueens` plot includes in blue crosses an alternative implementation of the parallel-for loop where each iteration is spawned off in sequence. 36

5-1 Regular (left), tiled (middle), and Morton (right) iteration orders for a two-dimensional iteration space. In this case, $N = 16$ and $G = 4$. . 45

5-2 Simplified nested loop implementation of matrix multiply. The outer two of the three loops in the nest are parallel. 48

5-3 Speedup of `mm` implementations with respect to grainsize, run on a single worker. 48

5-4 Speedup of `mm` implementations with respect to grainsize, run on 24 workers. 50

5-5 Parallel speedup of `mm` implementations for $G = 128$. 51

5-6 Simplified nested loop implementation of `blur`. `blurPixel` performs the "blurring" of a pixel and is factored out for better readability. . . 52

5-7 Speedup of `blur` implementations with respect to grainsize run on a single worker. 53

5-8 Parallel speedup of `blur` implementations for $G = 1$. 54

5-9 Simplified nested loop implementation of `transpose`. 55

5-10 Speedup of `transpose` implementations with respect to grainsize, run on 24 workers. 56

List of Tables

4.1 Comparison of running times of programs from PBBS [53] with grain-sizes 64, 256, and 2048. Each set of rows contains median running times of the benchmarks run on 1 or 20 workers using DACParFor (DAC) and loop frames (LF), as well as the ratio (*Ratio*) of the DACParFor running time divided by the loop-frame running times. A ratio greater than 1 indicates that DAC is slower than LF. The running times in *italic* represent the fastest run for that benchmark and worker count. 39

5.1 Benchmarks used in the study and their properties. Loop nest rank shows the number of nested loops, which corresponds to the number of dimensions in the iteration space, with the number in parenthesis indicating how many of those loops are parallel loops. Total distinct reads and total reads demonstrate whether there is reuse. 42

5.2 Implementations of nested loops used in the study. Base case size reduces both the recursive call overhead and parallelism. While neither `<name>` nor `<name>-tiled` use recursion, `<name>-tiled` still has a base case, which is one tile. `<name>-best` and `<name>-dac-full` only apply to `mm`. 45

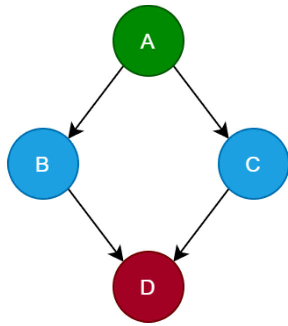
Chapter 1

Introduction

A ***parallel loop*** is a control-flow construct that contains a sequence of independent iterations, which are allowed to be executed concurrently. Parallel loops are common in high-performance parallel programs, and their performance is crucial for the overall performance of parallel code. This thesis tackles two main performance challenges of parallel loops: scheduling overheads and multidimensional locality in nested loops.

Parallel computing is the future of faster code. Multicore computers started appearing around 2005 [39], when clock speeds plateaued, limiting the speedup of serial code and forcing programmers to turn to parallel computing. However, running code in parallel is harder due to race conditions and other issues that occur due to concurrency. Ideally, parallel programs are desired to achieve ***perfect linear speedup***, meaning the program runs P-times faster on P processors (compared to running on a single processor). In reality, however, this isn't possible due to scheduling overheads.

Fork-join parallelism is a flexible task-based approach to parallel computing. Each program is modeled as a ***directed acyclic graph*** (***dag***), where nodes represent fine-grained indivisible tasks and edges represent dependencies between them. The fork-join paradigm supports two main constructs: ***spawn***/***sync*** and ***parallel-for*** loops. ***Spawn*** and ***sync*** are demonstrated in Figure 1-1. Similar to how B and C in Figure 1-1 can execute in parallel, parallel-for loops allow all iterations to execute in parallel. Fork-join parallelism is task-based because the programmer only has to specify tasks and their dependencies, while the scheduling decisions are left to the



FORKJOINDEMO()
1 A();
2 **spawn** B();
3 C();
4 **sync**
5 D();

Figure 1-1: A DAG for a simple fork-join program and the corresponding pseudocode. In this example, there are 4 serial tasks, A-D, that need to be executed. A needs to be executed before B and C (which can be executed in parallel). Both B and C need to finish before D. In the pseudocode, A is called first. Then, B is spawned, meaning C can execute before B is done. Sync on line 4 forces both B and C to complete before the execution moves on to D.

runtime system at the time of execution.

Randomized work stealing [5, 15, 20, 22, 23, 26, 31, 32, 33, 44, 58, 13, 25] is a popular (and asymptotically optimal) scheduling and load-balancing algorithm for ***parallel fork-join*** programs. Typically, a randomized work-stealing scheduler will implement ***parallel-for*** loops using recursive ***divide-and-conquer***, where the iteration space is recursively split into halves. Although efficient in theory [41, 13, 5], this approach can exhibit high overheads in practice. Those overheads are made worse by representing loops with other fundamental fork-join building blocks instead of treating them as first-class parallel constructs.

Multi-dimensional loops present additional challenges to efficient scheduling. Xu [59] argues for a locality-first approach to algorithm design and shows that locality can be more important for good parallel performance than parallelism. In the case of a multidimensional iteration space, good locality across all (or at least multiple) dimensions is often required for fast execution, which can be achieved with a reordering of iterations to achieve better cache efficiency.

Two common approaches to achieving cache efficiency are ***cache-oblivious*** and ***cache-aware*** algorithms. Cache-aware algorithms achieve good cache locality by taking cache size into account, while cache-oblivious algorithms achieve good cache locality without any explicit knowledge of cache parameters. For multidimensional

iteration spaces, ***loop tiling*** and ***recursive divide-and-conquer*** are cache-aware and cache-oblivious approaches to improving cache efficiency by reordering loop iterations.

When programmers implement these cache-efficient approaches manually, they tightly couple the algorithm to the implementation. That coupling essentially eliminates opportunities for other optimizations that operate on loops, e.g. vectorization and Loop-Invariant Code Motion, which avoids recomputation inside the loop by executing loop-invariant code before the loop. Instead, if an automated system, such as a compiler or a runtime system, was capable of transforming nested parallel loops into an implementation with the desired iteration order, programmers could maintain the simplicity of a nested loop implementation while also reaping the benefits of the increased locality. Because the loops are parallel, the compiler is allowed to reorder iterations without expensive aliasing and dependency analysis.

In this thesis, I tackle these two challenges of scheduling parallel loops. To reduce scheduling overheads of parallel loops, I introduce ***loop frames***. Loop frames extend work stealing by adding first-class support for parallel-for loops. To improve the performance of nested parallel loops, I study the impacts of improved locality along multiple dimensions on cache locality and overall performance for a few benchmarks.

Unlike the divide-and-conquer approach, where loops are implemented using existing fork-join constructs, loop frames give loops a special representation in the runtime, allowing them to optimize work stealing of parallel-for-loop iterations. Loop frames support ***on-the-fly loop splitting***, where intervals of loop iterations are dynamically split based on the unexecuted iterations when the split occurs. On-the-fly loop splitting allows unexecuted computation to be dynamically distributed more evenly than the divide-and-conquer algorithm, but it introduces non-determinism that complicates the theoretical analysis of work stealing [13, 5]. This thesis evaluates loop frames in practice and observes that loop frames incur substantially less overhead than the divide-and-conquer algorithm across the board, showing improvements in serial and parallel running times.

I report on a study that examines the impact of improved locality in more than one

dimension of a multidimensional iteration space. I evaluate both ***cache-oblivious*** and ***cache-aware*** implementations. The results show that improved locality along multiple dimensions increases cache efficiency and boosts performance. That motivates a compiler transformation that automatically reorders nested loop iterations to achieve good locality along all dimensions.

Divide-and-conquer parallel-for loops

To motivate loop frames, let us first examine the overheads in the traditional divide-and-conquer implementation of parallel-for loops. Figure 1-2 presents pseudocode for DACParFor, a typical implementation of a parallel-for loop [41, Sec. 8.3]. More specifically, a parallel-for loop over iterations $\{0, 1, \ldots, n-1\}$ is compiled to a call to DACParFor$(0, n, \text{Body})$, where the function Body encodes the loop body. As Figure 1-2 shows, DACParFor uses the parallel keywords **spawn** and **sync** to process the loop iterations using parallel divide-and-conquer recursion. DACParFor starts by splitting the loop into halves, until less than G iterations are left in a single batch. Here, G is a constant used to coarsen the recursion. The **spawn** keyword on line 3 allows the recursive call to execute iterations $\{s, s+1, \ldots, m-1\}$ to run in parallel with the ***continuation*** in the parent caller, which executes iterations $\{m, m+1, \ldots, e-1\}$. The **sync** statement on line 7 acts as a local barrier that joins together the child computations spawned within the function.

A ***work-span analysis*** [18, Ch. 27] of DACParFor shows that this algorithm is theoretically efficient. Consider the execution of DACParFor$(0, n, \text{Body})$, and for didactic simplicity, suppose that the every execution of Body performs $\Theta(1)$ instructions. The ***work*** T_1 of this execution — the total number of instructions executed — is $\Theta(n)$. The ***span*** T_∞ of this execution — the length of a longest path of dependencies — is $\Theta(\lg n + G)$. The ***parallelism*** of the execution, which bounds the maximum possible speedup on parallel processors, is the ratio of the work divided by the span, that is, $\Theta(n/(\lg n+G))$. Randomized work stealing bounds the execution time of a program on P processors by $T_P \leq T_1/P + O(T_\infty)$ [13, 5].

```
DACParFor(s, e, Body)
1  while e − s > G
2      m = (s + e)/2
3      spawn DACParFor(s, m, Body)
4      s = m
5  for i ∈ {s, s + 1, ..., e − 1}
6      Body(i)
7  sync
```

Figure 1-2: Pseudocode for the typical divide-and-conquer implementation of a parallel-for loop. The function Body encodes the body of the parallel-for loop. The **spawn** and **sync** keywords allow for parallel execution of the operations in the function. The constant G is used to coarsen the recursion to mitigate performance overheads.

Figure 1-2 also illustrates how parallel-for loops can exhibit substantial overheads in practice. As Figure 1-2 shows, a call to DACParFor$(0, n,$ Body) spawns $\Theta(n)$ function calls. Each spawned function call incurs overheads due to the function calls themselves and due to the operations each **spawn** performs to enable parallel execution. These overheads are particularly large in comparison to an ordinary serial loop, which typically performs just a few operations on registers per iteration to control the loop's execution. For example, I measured these overheads on a simple `daxpy` microbenchmark, which computes $y[i] \mathrel{+}= a \cdot x[i]$ over all elements of two given arrays x and y and scalar value a. On 1 processor, `daxpy` ran over $25\times$ slower when implemented using DACParFor with $G = 1$ compared to an ordinary serial-loop implementation with identical compiler optimizations.

Previous research has examined ways to improve the performance of parallel-for loops. Typically, the overheads of DACParFor can be mitigated by increasing the constant G for coarsening the recursion. Cilk, for example, typically sets $G = \min\{2048, n/8P\}$, where n is the number of loop iterations and P is the number of worker threads [30]. But increasing G reduces the parallelism of DACParFor, reducing the maximum possible parallel speedup the loop can achieve. As Section 6 discusses, previous work has explored many other approaches to optimizing parallel loops, including static and dynamic schemes to partition loop iterations [48, 28, 57,

40, 27, 7, 8], strategies to exploit common memory access patterns in loops [54], and techniques to reduce the synchronization overheads of work stealing [55, 56, 2].

Loop frames

Loop frames provide an alternative to DACParFor to load balance parallel-for loops dynamically using work stealing. Loop frames aim to address the shortcomings of DACParFor while providing the same theoretical performance guarantees. Loop frames introduce a simple data structure and work-stealing protocol to randomized work-stealing. Rather than build upon **spawn** and **sync** constructs, the loop-frame data structure concisely represents an interval of loop iterations using two integers. In addition, the loop-frame protocol supports on-the-fly loop splitting, which allows a processor to steal half of the frame's unexecuted loop iterations. In contrast, DACParFor splits the n iterations of a parallel-for loop at deterministic points, i.e., at iterations $n/2$, $n/4$, $3n/4$, etc., as Figure 1-2 shows. Loop frames compose with traditional randomized work stealing and serve as a drop-in replacement for DACParFor in a work-stealing scheduler.

On-the-fly loop splitting introduces non-determinism into the scheduling of parallel-for loops, complicating their analysis. Regardless, [47] shows that, despite their non-deterministic behavior, randomized work stealing with loop frames achieves the same theoretical performance guarantees as DACParFor.

Implementation of loop frames

I implemented loop frames in a beta version of the OpenCilk runtime system [51]. As Chapter 3 describes, my implementation extends the THE protocol from Cilk [25] to synchronize operations on loop frames efficiently. I compare the empirical performance of loop frames versus the traditional divide-and-conquer algorithm on 3 microbenchmarks and 14 application benchmarks from the Problem-Based Benchmark Suite [10]. I evaluated loop frames and the divide-and-conquer algorithm with vari-

ous coarsening values G. For the divide-and-conquer algorithm, G is used to coarsen the recursion as shown in Figure 1-2. For loop frames, G is used to ***strip-mine*** the loop, which batches iterations into groups of size G, thereby amortizing overheads associated with loop frames. Empirical results show that loop frames significantly reduce scheduling overheads of DACParFor, reducing the coarsening parameter G required for the best performance.

Study of locality benefits in multidimensional loops

To explore the benefits of increased locality of multidimensional iteration spaces, I performed a study using various implementations of simple benchmarks containing nested loops. The study focused on cache-oblivious and cache-aware approaches, comparing them to a naive nested loop implementation and analyzing their performance. The benchmarks used include `mm`, `blur`, and `transpose`. All benchmarks exhibit better cache locality when locality along multiple dimensions is improved. The implementations used achieve the regular, tiled (cache-aware), and multidimensional divide-and-conquer (cache-oblivious) iteration orders with a mix of recursive and iterative approaches. To eliminate the effects of recursive overheads, the study analyzes the effects of recursive coarsening. Results show that improved locality along more than one dimension improves performance for two of the three benchmarks.

Contributions and outline

This thesis makes the following contributions.

- Loop frames, which extend randomized work stealing with first-class support for parallel-for loops. They support on-the-fly loop splitting, which provides flexible dynamic splitting of loop iterations and allows for scheduling and load balancing with significantly less overhead than the traditional divide-and-conquer algorithm. Finally, loop frames are fully compatible with an existing fork-join scheduler.

- An efficient implementation of loop frames based on the OpenCilk runtime system [51]. This implementation extends the THE protocol from Cilk [25] to coordinate parallel operations on loop frames efficiently.
- An empirical evaluation of loop frames. I observe that on microbenchmarks and application benchmarks, loop frames and on-the-fly loop splitting incur substantially less overhead than the divide-and-conquer algorithm without sacrificing parallel scalability.
- A study on the impacts of reordering iterations of a multidimensional iteration space on cache locality and overall performance. The study measures the performance of multiple benchmarks and various implementations that achieve different iteration orders. It finds that for two out of the three benchmarks, reordering iterations can result in a significant reduction in cache misses and an improvement in bottom-line performance.

The rest of this thesis is organized as follows. Chapter 2 reviews the dag model of multithreading, randomized work stealing, and cache-efficient algorithms. Chapter 3 describes loop frames and their custom stealing and synchronization protocols used in the OpenCilk implementation. Chapter 4 presents the empirical evaluation of loop frames. Chapter 5 presents the study of improved locality in multidimensional iteration spaces. Chapter 6 discusses related work on optimizing work stealing and parallel loops. Finally, Chapter 7 offers concluding remarks.

Chapter 2

Background

This chapter reviews the dag model of multithreading [12, 13], which represents the execution of a recursive fork-join program and classical randomized work stealing. This chapter also reviews cache-oblivious algorithms and recursive divide-and-conquer in multiple dimensions.

2.1 The dag model of multithreading

An ***execution dag*** $G = (V, E)$ represents the execution of a recursive fork-join program, where each vertex $x \in V$ represents a ***strand*** — a sequence of serially executed instructions containing no parallel control — and edges represent control dependencies between strands. The execution of a **spawn** terminates the current strand and produces a vertex with out-degree 2. A **sync** creates a vertex with in-degree greater than 1. The execution dag G is a series-parallel dag [21] with a single source vertex and a single sink vertex. To simplify the analysis, this thesis shall assume that each strand represents a single executed instruction.

The work and span of the execution of a program can be measured in terms of its execution dag G. The work of G is the total number of strands in G. The span of G is the length of the longest path of dependencies in G.

Task-parallel programming platforms that support the dag-parallel model include dialects of Cilk [11, 25, 19, 38, 36, 29], Fortress [4], Habanero [9], Habanero-Java [16],

Hood [14], HotSLAW [42], Java Fork/Join Framework [35], OpenMP [46, 6], Task Parallel Library [37], Threading Building Blocks (TBB) [49] and X10 [17].

2.2 Randomized work stealing

The operation of a randomized work-stealing scheduler can be described in terms of the execution dag $G = (V, E)$. A randomized work-stealing scheduler dynamically load balances a parallel computation across available threads, called ***workers***. At each time step, each worker p maintains an ***assigned strand***, which is the strand that p executes on that time step. A strand s is said to be ***ready*** if all its predecessors in G have been completed. Executing an assigned strand can ***enable*** a strand s' that is a direct successor of s in G by making s' ready. Each worker maintains a ***deque*** – a double-ended queue – of ready strands. Typically, a worker operates on its deque like a stack, pushing and popping strands from the tail of its deque. When a worker runs out of strands on its deque, it becomes a ***thief*** and randomly chooses another worker as its ***victim***. If the selected victim has excess strands on its deque, then the thief can ***steal*** a strand from the head of the victim's deque.

A classical randomized work-stealing scheduler keeps track of a single ready strand using a ***spawn frame***. Hence, the deque of ready strands is stored as a deque of spawn frames. Each spawn frame maintains the necessary information for a thief to begin executing that strand, e. g. the code address and register state to resume execution of the strand. When a thief steals from a victim, it removes the spawn frame at the head of the victim's deque and assigns the stolen frame to itself. The ***extended deque*** is defined to include the assigned strand along with the other spawn frames in the deque.

Worker deques support three methods: PUSHBOTTOM, POPBOTTOM, and POPTOP. A **spawn** statement causes a worker to execute PUSHBOTTOM to push its current frame onto the bottom of the deque. A worker executes POPBOTTOM when it returns from a spawned function. If POPBOTTOM runs on an empty deque, then POPBOTTOM does not return but instead causes the worker to become a thief. A

thief steals a spawn frame by calling POPTOP to remove the frame at the top of the deque.

2.3 Cache-oblivious and cache-aware algorithms

A ***cache-oblivious algorithm*** [24] is an algorithm that achieves good cache locality without explicit knowledge of the cache parameters B and M, cache line length and cache size, respectively. In contrast, ***cache-aware*** algorithms explicitly use those cache parameters to achieve good cache-locality.

Cache-oblivious algorithms have many advantages over cache-aware algorithms. Because they do not require precise tuning to the hardware they execute on, they are very portable. Obliviousness to cache parameters also makes them perform well on more complex memory hierarchies like multi-level caches and heterogeneous hardware. Conversely, cache-aware algorithms must be explicitly tuned for each level of the cache hierarchy.

Optimal cache-oblivious algorithms achieve the lower asymptotic bound of cache misses. Several problems, including matrix multiplication, matrix transposition, and sorting, have known optimal cache-oblivious algorithms. In contrast, others, like the Cooley-Tukey FFT, are optimally cache-oblivious with specific parameter choices. These algorithms may require fine-tuning on particular machines, but the overall goal is to minimize the extent of tuning necessary.

Typically, a cache-oblivious algorithm employs a recursive divide-and-conquer approach, dividing a problem into smaller subproblems until the subproblem fits into the cache, regardless of the cache size. Hybrid algorithms may also combine cache-aware and cache-oblivious algorithms for specific cache sizes.

One cache-aware alternative to nested loops in a multidimensional iteration space where iterations can be reordered is ***loop tiling*** [34]. Instead of iterating from start to finish for each dimension, the iteration space is divided into ***tiles***, which are multidimensional sub-parts of the iteration space. The size of each tile is tuned so that each data accessed in a tile fits in cache. In serial execution, each tile is processed as a

whole before starting the next. Most parallel executions allow tiles to run in parallel while each tile runs serially, which provides coarsening and guarantees the whole tile is executed on a single worker. Multiple levels of tiling are required to exploit all cache levels in a multi-level cache hierarchy.

The cache-oblivious replacement for loop tiling is ***multidimensional recursive divide-and-conquer***. It performs a series of splits, always by the largest dimension, and recursively visits both halves. This approach will traverse the iterations in the Morton order [43], sometimes referred to as the z-order.

Chapter 3

Loop frames

This chapter describes loop frames, which extend randomized work stealing to provide first-class support for parallel loops in the runtime. Section 3.1 describes loop frames and how they support on-the-fly loop splitting in a randomized work-stealing scheduler. Section 3.2 outlines the detailed synchronization protocol between the victim and the thieves that maintains correctness during concurrent execution.

I present loop frames for parallel-for loops without any coarsening (i.e. executing chunks of iterations at a time to reduce the scheduling overhead) for simplicity. In the implementation, coarsening is implemented via loop strip-mining, where the loop is divided into chunks and a remainder. Each chunk is then executed as a whole, and the remainder is executed at the end.

3.1 Implementation

A loop frame is an extension of a spawn frame. All spawn frames contain the execution context, which assists workers in pausing and resuming different sub-parts of the computation. In addition to that, loop frames store a range of loop iterations that remain to be executed and additional flags that assist synchronization. Just like spawn frames, loop frames are pushed onto the deque and popped from it by workers and thieves.

A loop frame extends a spawn frame with *start* and *end* variables, which encode

```
LFParFor(n, Body)
1  F = LoopFrame(1, n, Body)
2  pushBottom(F)
3  LFParForHelper(F, 0)
4  popBottom()
```

```
LFParForHelper(F, i)
1  while Next(F)
2        F.Body(i)
3        i ++
```

```
StealLF_simple(F)
1  F′ = CopyLoopFrame(F)
2  m = (F′.start + F′.end)/2
3  F.end = m
4  F′.start = m + 1
5  pushBottom(F′)
6  LFParForHelper(F, m)
7  popBottom()
```

Figure 3-1: Pseudocode for loop frame operations. LFParFor is used by the worker that enters the loop frame, while SimpleSteal is used by the thief when stealing a loop frame from a victim. LFParForHelper is used by both to actually execute all iterations and contains a familiar branch, body, and increment that can be found in all for loops. Next returns true if there are more iterations available, and false otherwise.

the parallel loop iterations in the half-open interval $[start, end) = \{start, start + 1, \ldots, end - 1\}$. At any time during the execution, *start* and *end* represent the iterations that haven't been executed yet. While the divide-and-conquer implementation will push and pop spawn frames onto the worker's ready deque while executing the iterations, LFParFor only operates on the *start* and *end* variables, reducing the scheduling overhead per iteration.

Figure 3-1 contains pseudocode for the operations on a loop frame. Here, I assume that the Next function and lines 1–3 in StealLF_simple operate atomically. The detailed implementation of the protocol that provides these atomicity guarantees is presented in Section 3.2.

Let us walk through the pseudocode in Figure 3-1. A parallel-for loop over n iterations is implemented using a call to LFParFor(n, Body) followed by a **sync**, where the function Body encodes the body of the parallel-for loop. The **sync** ensures that all workers executing iterations of the loop have completed. When a worker executes LFParFor(n, Body), line 1 first creates a new loop frame F representing iterations $[1, n)$. Line 2 pushes F onto the worker's deque, and then line 3 calls LFParForHelper to direct the worker to start executing the loop from iteration 0.

These lines thus assign iteration 0 to the worker and allow iterations $[1, n)$ to be stolen. Upon returning from LFParForHelper, line 4 pops f off the bottom of the worker's deque or, if the pop fails (because the loop frame has been stolen completely), causes the worker to become a thief.

The function LFParForHelper(F, i) causes the worker to execute loop iterations starting from $i = F.start - 1$ one at a time until reaching iteration $F.end$. In particular, line 1 calls Next(F) to check if there are more loop iterations to execute. Next(F) increments $F.start$ and returns $False$ if $F.start > F.end$, or $True$ otherwise.

To steal from the loop frame F using on-the-fly loop splitting, a thief calls the function StealLF_simple(F), which operates as follows. Line 1 first produces a copy F' of F, and then line 2 computes the midpoint m between $F'.start$ and $F'.end$. Line 3 then performs on-the-fly loop splitting: The thief claims the unexecuted iterations $[m, F'.end)$ by setting $F.end = m$, thereby stealing half of the unexecuted iterations. Finally, like an ordinary worker, the thief pushes F' representing the iterations $[m + 1, F'.end)$ onto its deque (lines 4 and 5) and then calls LFParForHelper on line 6 to start executing its loop iterations from iteration m. Finally, line 7 pops F' off the bottom of the thief's deque and allows the thief to resume work stealing.

3.2 Synchronization protocol

This section describes the synchronization protocol used in the OpenCilk runtime system [51] implementation of loop frames. Section 3.1 described loop frames assuming that two functions, Next and StealLF, execute atomically. This section describes how these functions implement an efficient protocol, based on the THE protocol in Cilk [25], to synchronize updates to loop frames.

Let us first review the THE protocol for coordinating accesses onto worker deques [25]. Each deque maintains a ***head*** pointer H and a ***tail*** pointer T to track the head and tail of the deque, as well as a mutex lock L. Workers normally push and pop frames onto the deque by updating T. A thief can pop a frame from the top of the

deque by incrementing H. To optimize deque operations, the THE protocol allows the worker to speculatively update T, and only requires the worker to acquire L when the deque appears to be empty. After acquiring L, the worker can determine whether the deque was really empty or there was an uncompleted steal attempt in progress that has since failed. A thief, meanwhile, always acquires L before updating H, ensuring a single steal attempt happening at a time and providing a correctness guarantee for the worker's second, protected check.

To ensure the atomicity of NEXT and STEALLF, a simple but inefficient approach is to give a thief exclusive access to a victim's loop frame F during STEALLF. Intuitively, after locking the deque, the thief would increment H, update $F.end$, then decrement H and release the deque lock. The NEXT function, meanwhile, would decrement T before attempting to update $F.start$. If the worker discovered $H > T$ after decrementing T, it would acquire the deque lock before completing its update to $F.start$. Although this approach is simple, it turns out to be inefficient, as the worker will push and pop frames for every iteration.

Figure 3-2 presents pseudocode for the optimized synchronization protocol on loop frames, which I found to be 40% faster than the simple protocol. Conceptually, rather than use H and T to coordinate all operations on loop frames, the optimized protocol operates on the *start* and *end* variables of a loop frame directly, similarly to how the THE protocol operates on H and T. The NEXT function implements the routine used in LFPARFORHELPER for a worker to get the next loop iteration to execute from a loop frame F. A thief steals from a deque D by calling STEAL(D), which calls STEALLF(F, D) on line 8 if line 7 discovers that the frame F at the top of D is a loop frame. The STEAL function thus extends the `steal` function in Cilk-5 [25] to handle loop frames.

Let us walk through the pseudocode in Figure 3-2, starting with NEXT. Line 1 gets the next iteration index i to execute and speculatively increments $F.start$. Line 2 checks if the loop frame is out of iterations by checking whether the new value of $F.start$ exceeds $F.end$. If the test fails, the worker returns $True$ (line 8). If the test succeeds, then the worker locks its own deque (line 3) to gain exclusive access to

```
NEXT(F)
1  i = F.start ++
2  if F.start > F.end
3      LOCKOWNDEQUE()
4      if F.start > F.end
5          UNLOCKOWNDEQUE()
6          return False
7      UNLOCKOWNDEQUE()
8  return True

STEAL(D)
 1  LOCK(D.L)
 2  F = H ++
 3  if H > T
 4      H --
 5      UNLOCK(D.L)
 6      return FAIL
 7  if ISLOOPFRAME(F)
 8      return STEALLF(F)
 9  UNLOCK(D.L)
10  return SUCCESS

STEALLF(F, D)
 1  F' = COPYLOOPFRAME(F)
 2  m = (F.start + F.end)/2
 3  F.end = m
 4  if F.start > F.end
 5      F.end = F'.end
 6      H --
 7      UNLOCK(D.L)
 8      return FAIL
 9  if F.start < F.end
10      H --
11  UNLOCK(D.L)
12  F'.start = m
13  return SUCCESS
```

Figure 3-2: Pseudocode for the optimized synchronization protocol to coordinate operations on loop frames. The argument D denotes a victim's deque. The functions LOCKOWNDEQUE and UNLOCKOWNDEQUE acquire and release the lock on the executing worker's deque. The function ISLOOPFRAME tests if the given frame is a loop frame.

F. The worker rechecks if $F.start > F.end$ on line 4 and then returns either *False* (line 6), if the loop frame is out of iterations, or *True* otherwise (line 8). In either case, the worker unlocks its deque before returning.

Function STEALLF(F, D) in Figure 3-2 is a more detailed version of STEALLF_SIMPLE in Figure 3-1, additionally containing the synchronization protocol on lines 4–11. After setting $F.end = m$ on the victim loop frame F, line 4 checks if F now contains an invalid interval of loop iterations by testing $F.start > F.end$. If so, then lines 5–8 restore $F.end$ and H to their previous values, unlock the victim's deque, and report a failed steal attempt. Lines 5–8 perform the same rollback as lines 4–6. Otherwise, line 9 checks if F still contains any loop iterations after the steal by

testing $F.start < F.end$. If so, then line 10 restores the victim's head pointer to leave F at the top of the victim's deque. If not, the thief leaves H unchanged, effectively removing F from the top of the victim's deque and exposing frames below F on the deque to be stolen. Finally, the thief reports a successful steal (line 13), after which it will begin executing iteration m.

Note that line 12 sets $F'.start = m$ as opposed to Line 4 of the simplified protocol (depicted in Figure 3-1), which sets $F'.start = m + 1$. That's because the former is only stealing the upper half of the iterations while the latter then also immediately takes the first iteration m from the loop frame F' before F' is pushed onto its deque. In other words, it summarizes both operations into one: stealing the upper half of iterations and taking the first one to make it unavailable for stealing. I believe that this distinction makes it clearer which iterations are actually getting stolen from F in the synchronization protocol in Figure 3-2.

The definition of correctness for this protocol follows from preserving the semantics of a for-loop: it executes every iteration in the specified range exactly once, and it doesn't execute any iterations outside this range. The proof of correctness mirrors that of the THE protocol [25], assuming sequential consistency. While the THE protocol guarantees that a frame will only be taken (and executed) by either a worker or a thief, the loop frame protocol provides the same guarantee for iterations.

Because a thief first acquires the victim deque's lock before attempting to steal, only the worker and one thief can operate concurrently on a deque and, therefore, on the same loop frame F. If F contains no iterations, then both the victim and thief will fail to take any iterations from F. Otherwise, suppose a thief attempts to steal iterations $[m, F.end)$ from F. If F contains more than 1 iteration, then $F.start < m$, and the worker can update $F.start$ without impinging on any iterations in $[m, F.end)$, Hence, both updates to F can happen concurrently without issue. If F contains 1 iteration, then the thief and victim must coordinate to determine which worker gets iteration m. If the thief discovers $F.start > F.end$ on line 4 of StealLF, then line 1 in Next happened before this check, and the thief will restore $F.end$ to its original value, allowing the victim to claim the iteration. Otherwise, line 1 in Next happens

after this check, causing the thief to steal iteration m and the victim to discover that F is out of iterations.

Chapter 4

Empirical evaluation of loop frames

This section presents my empirical evaluation of loop frames. I tested my implementation of loop frames in version 1.0 of the OpenCilk runtime [51] and compared its performance to that of the traditional divide-and-conquer algorithm, *DACParFor*. I evaluated both algorithms on both microbenchmarks and application benchmarks.

Results show that loop frames achieve significantly lower scheduling overheads than the reference *DacParFor* implementation. On microbenchmarks, which used no coarsening to directly expose the scheduling overheads, loop frames outperformed the reference implementation by a factor of up to 2.7×. Additionally, application benchmarks from the Problem-Based Benchmark Suite (PBBS) [53] were used to demonstrate that loop frames also perform well in a real-world setting. Those results show loop frames require less coarsening than the reference implementation to achieve good performance, providing more evidence for a reduction in scheduling overheads.

All experiments in this chapter were run on compute nodes on the MIT Supercloud system [50]. Each compute node is a dual-socket Intel Xeon Gold 6248 system with a total of 384 GB of main memory. Each Xeon is a 2.50GHz 20-core CPU. To reduce the effects of noise on the performance measurements, I disabled hyperthreading and pinned all workers to cores on a single socket. All the benchmarks were compiled with OpenCilk [51], based on LLVM 9, using the highest optimization level. All results are median running times, aggregated from 10 trials. Specific compilation strategies are described in each section.

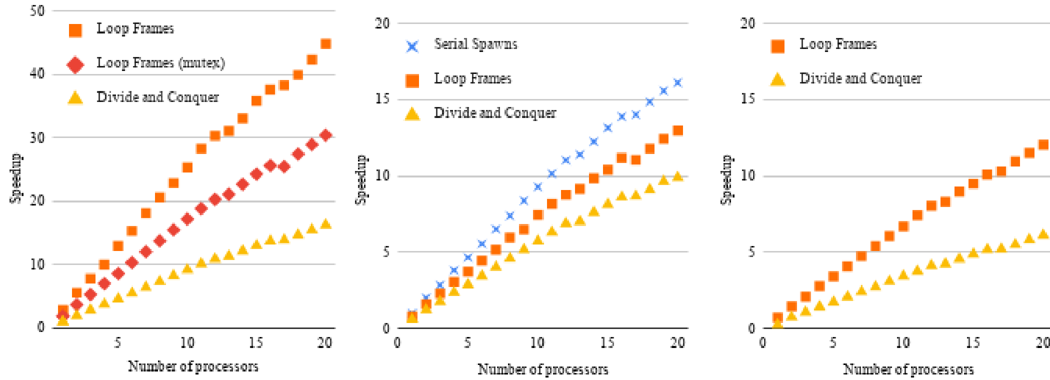


Figure 4-1: Median speedup with respect to number of processors for the `daxpy`, `nqueens`, and `mandelbrot`, appearing from left to right. Each plot features loop frames in orange squares and the divide-and-conquer algorithm in yellow triangles. The `daxpy` plot also includes the results of the simpler protocol for loop frames (briefly mentioned in Chapter 3) in red diamonds. The `nqueens` plot includes in blue crosses an alternative implementation of the parallel-for loop where each iteration is spawned off in sequence.

4.1 Microbenchmark experiments

Figure 4-1 presents the performance results for the three microbenchmarks: `daxpy`, `nqueens`, and `mandelbrot`. I compare the overheads of parallel-for loops implemented using loop frames with the reference *DacParFor* implementation on these programs. Because each program makes heavy use of parallel-for loops, the performance of each program depends substantially on the overhead of the parallel-for-loop implementation. The parallel-for loops in each program were implemented by directly inserting calls to runtime ABI functions into the code to avoid modifications to the OpenCilk compiler. The three programs vary in their computational intensity, that is, the ratio of arithmetic operations to memory operations. To evaluate the overheads of the parallel-for-loop implementations, I study these programs with no coarsening, i.e., with $G = 1$ in DACParFor. I examine the results for each of these microbenchmarks results in turn.

The results for the `daxpy` microbenchmark, which was briefly described in Chapter 1, are shown in the leftmost plot in Figure 4-1. All running times in this plot are normalized to the 1-processor running time of the DACParFor implementation. `daxpy`

runs a geometric mean 2.7× faster when using loop frames instead of DACParFor on all processor counts.

The middle plot in Figure 4-1 presents the results for `nqueens`. This program performs a parallel recursive backtracking search to count the solutions to the n-queens problem. At each level of recursion, the program performs a parallel-for over the n positions in a given row. The figure presents results for $n = 13$, meaning that each parallel-for loop contains just 13 compute-intensive iterations. As the figure shows, when using loop frames, `nqueens` runs a geometric mean 1.3× faster than the DACParFor version on all processor counts. Furthermore, the figure shows that loop frames cut in half the performance gap between the DACParFor implementation and an optimized version of `nqueens` in which the parallel-for loops spawn off each iteration in sequence after performing an earlier viability check. This pruning strategy reduces the parallel overhead of spawning short tasks that don't contribute to the parallelism and is not possible for parallel loops, which is why it outperforms both loop-based implementations.

The rightmost plot in Figure 4-1 presents the performance results for `mandelbrot`, which plots a picture of the Mandelbrot fractal with a specified size. The results in the plot are normalized to the serial projection of the program [25, 52], in which all parallel constructs are replaced with their serial counterparts. `mandelbrot` using loop frames runs a geometric mean 2× faster compared to using DACParFor on all processor counts.

4.2 Application experiments

I evaluated the performance of loop frames on 14 applications from the Problem-Based Benchmark Suite (PBBS) [53]. In particular, I examined the performance of loop frames and DACParFor on these applications with different coarsening values G. I selected 14 applications from PBBS for which changing G for the default DACParFor implementation of parallel-for loops from $G = 2048$ to $G = 64$ produced at least a 2% change in running time. This selection criterion aims to identify

programs for which I anticipate the performance differences between DACParFor and loop frames to have a measurable impact on program running time.

To evaluate loop frames on these benchmarks, I modified the OpenCilk runtime system to implement a runtime-ABI function for `cilk_for` loops [30] using loop frames. I developed a similar runtime-ABI function using DACParFor to provide a fair comparison between the algorithms. All PBBS benchmarks were compiled to use this runtime-ABI function for all `cilk_for` loops. To test a coarsening value G, I directed the OpenCilk compiler to use G when coarsening loop statically, via loop strip-mining, and when using a runtime-computed coarsening for the `cilk_for` loop [30]. In addition, I compiled all programs with exceptions disabled, due to lack of exception support in the modified runtime.

Table 4.1 presents the performance results for the PBBS benchmarks. Many of the PBBS benchmarks have parallel-for loops with large loop bodies, which minimizes the total contribution of parallel-loop overhead to the total running time of the program. Nevertheless, I found that, with smaller values of G, the benchmarks perform better on both 1 and 20 workers when using loop frames than when using DACParFor. With $G = 64$, for example, the benchmarks ran $2-6\%$ faster when using loop frames versus DACParFor. The performance difference between loop frames and DACParFor decreases as G is increased because a larger value of G decreases the total contribution of parallel-loop overhead to the program's running time. Nevertheless, the results indicate that loop frames allow applications to achieve efficiency and scalability with smaller coarsening values. Good performance at lower coarsening could speed up other applications that don't contain as much parallelism as PBBS. However, such applications would need to be benchmarked directly to confirm this hypothesis.

I note that Table 4.1 shows loop frames performing worse than DACParFor on several benchmarks with $G = 2048$. For $G = 2048$, I believe that loop frames are performing worse than intended due in part to a performance bug in the implementation. For a parallel-for loop over n iterations, after strip-mining the loop by G, the loop-frame implementation executes $n \bmod G$ iterations serially after the rest of the loop-frame performs a **sync**. In contrast, the DACParFor implementation allows

those same iterations to execute in parallel with the other iterations of the loop. Any future implementation using loop frames should correct this bug.

	detbfs	ndbfs	chull	irefine	dict	imis	imatch	ndmatch	pkruskal	remdup	ist	ndst	pks	prange
DAC, T_1, 64	15.333	12.266	9.440	48.000	2.045	2.890	10.380	6.343	42.666	3.131	16.666	16.200	26.183	22.018
LF, T_1, 64	14.433	12.033	9.212	47.250	2.002	2.800	9.623	6.120	41.266	2.902	17.200	17.466	25.933	19.081
Ratio	1.062	1.019	1.025	1.016	1.021	1.032	1.079	1.036	1.034	1.079	0.969	0.928	1.010	1.154
DAC, T_{20}, 64	0.976	1.038	0.743	3.150	0.120	0.186	0.611	0.366	3.146	0.199	1.194	9.473	1.892	1.309
LF, T_{20}, 64	0.963	0.809	0.742	3.205	0.116	0.184	0.575	0.356	*3.096*	0.191	1.165	9.383	1.891	1.295
Ratio	1.013	1.009	1.001	0.983	1.034	1.022	1.062	1.028	1.016	1.042	1.025	1.010	1.001	1.011
DAC, T_1, 256	14.700	12.033	9.193	47.750	2.004	2.866	9.500	6.050	41.566	2.995	15.966	15.900	25.716	19.533
LF, T_1, 256	14.500	11.900	9.112	47.250	1.983	2.810	9.216	6.056	41.233	2.900	15.566	16.099	25.500	19.111
Ratio	1.014	1.011	1.009	1.011	1.011	1.020	1.031	0.999	1.008	1.033	1.026	0.988	1.008	1.022
DAC, T_{20}, 256	3.100	0.816	0.743	3.165	0.117	0.186	0.572	0.353	3.126	0.194	1.162	8.943	1.895	1.313
LF, T_{20}, 256	0.970	*0.803*	0.747	3.180	0.115	*0.183*	0.558	*0.349*	3.130	0.191	*1.151*	9.083	1.877	1.298
Ratio	1.000	1.004	0.995	0.995	1.017	1.016	1.025	1.011	0.999	1.016	1.010	0.985	1.010	1.012
DAC, T_1, 2048	14.266	*11.799*	*9.081*	*46.750*	*1.958*	2.796	*9.013*	6.013	*40.233*	2.867	*15.400*	*15.500*	*25.133*	*18.970*
LF, T_1, 2048	*14.233*	11.900	9.090	47.450	*1.958*	*2.706*	9.083	*6.010*	40.433	*2.844*	15.500	15.633	25.250	19.073
Ratio	1.002	0.992	0.999	0.985	1.000	1.033	0.992	1.000	0.995	1.008	0.994	0.991	0.995	0.995
DAC, T_{20}, 2048	*0.956*	0.806	*0.739*	*3.130*	*0.114*	0.184	0.554	0.353	3.106	*0.190*	1.159	9.286	*1.857*	*1.280*
LF, T_{20}, 2048	0.989	0.830	0.746	3.190	*0.114*	0.184	*0.552*	0.351	3.126	0.192	1.162	*8.896*	1.870	1.297
Ratio	0.967	0.971	0.991	0.981	1.000	1.000	1.004	1.006	0.994	0.990	0.997	1.044	0.993	0.987

Table 4.1: Comparison of running times of programs from PBBS [53] with grainsizes 64, 256, and 2048. Each set of rows contains median running times of the benchmarks run on 1 or 20 workers using DACParFor (DAC) and loop frames (LF), as well as the ratio (*Ratio*) of the DACParFor running time divided by the loop-frame running times. A ratio greater than 1 indicates that DAC is slower than LF. The running times in *italic* represent the fastest run for that benchmark and worker count.

Chapter 5

Study of improved locality in nested loop applications

In this chapter, I present a study of the impact of increased cache locality on three simple benchmarks that contain nested loops. I implemented each benchmark using cache-oblivious and cache-aware approaches and compared their performance to a naive nested loop implementation. The results show that improved locality in more than one dimension can result in performance gains up to 1.7× when better temporal and spatial locality can be achieved.

Table 5.1 summarizes the benchmarks used in the study. `mm` computes a product of two matrices. `blur` performs a convolution of a trivial 2D filter on a 2D image. `transpose` computes an in-place matrix transposition. `mm` reuses each element N times, `blur` reuses each element K times, and `transpose` contains no reuse.

All benchmarks satisfy several criteria that make them relevant to this study. First, memory operations represent a significant proportion of the running time, so better locality can measurably affect overall performance. Second, locality along more than one dimension improves spatial or temporal locality. Otherwise, loops in the nests can be reordered to guarantee locality in the desired dimension. This criterion also implies the nested loop implementation does not achieve the optimal number of cache misses.

This chapter is organized as follows. Section 5.1 explains the importance and

Property	`mm`	`blur`	`transpose`
Parameters	N	N, K	N
Loop nest rank (parallel)	3(2)	2(2)	2(2)
Total distinct reads	$2N^2$	N^2	$N^2 - N$
Total reads	$2N^3$	N^2K^2	$N^2 - N$

Table 5.1: Benchmarks used in the study and their properties. Loop nest rank shows the number of nested loops, which corresponds to the number of dimensions in the iteration space, with the number in parenthesis indicating how many of those loops are parallel loops. Total distinct reads and total reads demonstrate whether there is reuse.

terminology of locality in multidimensional loops. Section 5.2 lists the various implementations used in the study and describes their properties. Section 5.3 describes each benchmark in more detail and presents the empirical results for each. Finally, Section 5.4 summarizes the takeaways we can draw from the study.

5.1 Locality in multidimensional loops

This section discusses locality in multidimensional loops and when it benefits performance. It explains the relationship between iteration order and locality. The section also explains the distinction between spatial and temporal locality and how they impact cache operation. Finally, an example of iterating a 2D array illustrates the importance of locality in different dimensions.

The traversal order of a multidimensional iteration space is what determines its locality. Recursive and iterative methods traversing a multidimensional iteration space execute in some linear order. Even when run in parallel, each parallel processor will execute iterations one after the other.

We rely on parallel loops because they allow us to reorder iterations, but we only analyze locality from the perspective of the linear order. In the fork-join parallel-programming model, reordering iterations of nested loops is always legal if all loops are parallel. Conversely, prior knowledge of application semantics or expensive dependency and aliasing analysis are required to reorder serial loops. Finally, randomized work-stealing bounds the additional cache misses incurred due to steals [1].

Locality in a particular dimension of an iteration space is beneficial if executing iterations with different values of that coordinate would result in memory accesses to locations far apart. This almost translates to "executing iterations for the same values of the coordinate results in memory accesses close together or at the same location in memory," except for that to be true, good locality in multiple dimensions at a time is often required.

The distinction between spatial and temporal locality is essential. Temporal locality can only be achieved if there is ***reuse***, meaning that a location in memory is accessed multiple times, specifically during multiple iterations in an iteration space. That reuse can result in temporal locality if those points are executed closely in the resulting iteration order. On the other hand, spatial locality means nearby iterations access nearby elements in memory.

This distinction is important because of the way caches operate. Both types of locality increase the likelihood that a memory location is already in the cache. Still, spatial locality does so by another element on the same cache line already residing in the cache. Hence, the benefits of spatial locality generally do not extend past the size of a cache line B.

For example, let us consider a simple task of iterating a 2D array laid out contiguously in memory in ***row-major layout***. Row-major layout means that the x coordinate of the array is the fastest-running dimension. Let us assume we are iterating the array using two nested loops with dimensions (indices) i and j, where i and j correspond to x and y coordinates of the array, respectively. The best locality is achieved if iterating over i is done in the inner loop, as iterations with varying i and the same j access nearby elements, while the converse is not true. Hence, locality along j is more vital because it results in good spatial locality. Of course, locality along i also matters as we want to access consecutive elements in memory in consecutive order. Still, there is no benefit to executing iterations with the same i and different j soon after the other. Hopefully, this example is familiar to the reader and helps clarify the concepts of locality in different dimensions, but it would not serve well as a benchmark in this study.

5.2 Implementations

This section describes the various implementations used in the study that traverse the multidimensional iteration space. The primary purpose was to study the effect of changing the order in which iterations are executed. However, I also wanted to eliminate other effects on the overall performance, namely function call overheads in recursive implementation. Therefore, I added a "grainsize" parameter (G) to coarsen the recursion. Additionally, multiple implementations may visit the iteration space in the same order but use different approaches to separate the effects of increased locality from the impact of recursive call overhead.

For each benchmark, if the benchmark is called `<name>`, the following implementations are available:

- `<name>` is a simple nested loops implementation.
- `<name>-tiled` iterates the space by dividing the space into $G \times G$ tiles.
- `<name>-dac` uses multidimensional divide-and-conquer recursion, always splitting the largest dimension. All dimensions are split down to size G.
- `<name>-dac-loop` uses divide-and-conquer recursion but fully splits the outer dimension first. Only the inner dimension is coarsened (split to size G); other dimensions are split to size 1. It achieves the same iteration order as `<name>`.
- `<name>-dac-loop-tiled` is like `<name>-dac-loop` but both dimensions are coarsened. It achieves the same iteration order as `<name>-tiled`.
- `<name>-dac-full` and `<name>-best` also recursively split the third (serial) dimension and only apply to `mm`. They are described in more detail in Subsection 5.3.1.

Figure 5-1 shows the three distinct iteration orders in the implementations. `<name>` and `<name>-dac-loop` achieve the regular order, `<name>-tiled` and `<name>-dac-loop-tiled` achieve the tiled order, and `<name>-dac` achieves the Morton order (also referred to as z-order). `<name>-dac-full` and `<name>-best` achieve the Morton order in all 3 dimensions.

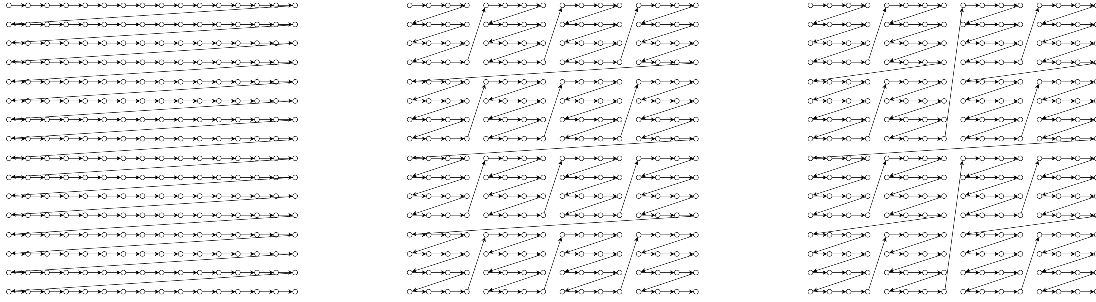


Figure 5-1: Regular (left), tiled (middle), and Morton (right) iteration orders for a two-dimensional iteration space. In this case, $N = 16$ and $G = 4$.

Implementation	Base case size	G affects order
`<name>`	1	N/A
`<name>-tiled`	G^2	Yes
`<name>-dac`	G^2	Yes
`<name>-dac-loop`	G	No
`<name>-dac-loop-tiled`	G^2	Yes
`<name>-dac-full`*	G^3	Yes
`<name>-best`*	G^3	Yes

Table 5.2: Implementations of nested loops used in the study. Base case size reduces both the recursive call overhead and parallelism. While neither `<name>` nor `<name>-tiled` use recursion, `<name>-tiled` still has a base case, which is one tile. `<name>-best` and `<name>-dac-full` only apply to `mm`.

The grainsize parameter G may affect both recursion coarsening and iteration order. G is the size of the tile in both `<name>-tiled` and `<name>-dac-loop-tiled`, but because the latter uses recursion, it also impacts the recursion base-case size, which is equal to one tile. `<name>-dac` also uses a tile as a base case, which means that the resulting order is not exactly the ***z-order***. However, the locality remains similar as long as one tile fits in cache. Finally, `<name>-dac-loop` only allows coarsening in the innermost dimension, and G only affects the coarsening. Table 5.2 summarizes the effects of the grainsize parameter, G, on recursion coarsening and iteration order for all implementations.

Despite having a coarsening parameter, `<name>-dac` is cache-oblivious, as G is only used to reduce the recursion overhead and not to optimize the residual size in cache. However, if G becomes too large, the cache efficiency of `<name>-dac` can be lost.

5.3 Benchmarks and performance results

This section presents more detail on each benchmark and the performance results. All benchmark implementations were compiled with version 2.0 of OpenCilk and `-O3 -DNDEBUG` optimization flags. I ran all experiments on an AWS c5.metal machine, which features an Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz processor with 48 physical cores across 2 NUMA nodes. Each core has its own 32 KiB L1d and L1i caches and a 1 MiB L2 cache. Meanwhile, the 35.75 MiB L3 cache is shared between all cores on one NUMA node. L1 and L2 are 8-way associative, L3 is 11-way associative, and all contain 64 B cache lines. Because all benchmarks operate on 32-bit (4 B) values, the effective cache line size B is 16.

For each benchmark, I explored the impact of G on the overall performance in both serial and parallel settings. I measured the median running time (minimum when running on one worker) from 10 trials for all power-of-two values of G from 1 to 128. I repeated those measurements for all worker counts between 1 and 24, always utilizing distinct physical cores on a single NUMA node to reduce performance variability. All plots in this chapter plot the speedup (larger is better), which is calculated as $S(t) = \frac{t_{norm}}{t}$, where t_{norm} is the running time used for normalization. Generally, t_{norm} is the minimum running time of the serial projection of the nested loop implementation `<name>`. One exception is when plotting the speedup with respect to grainsize on a certain number of workers, where t_{norm} is the median running time of `<name>` for that worker count.

Parallel scalability was not affected by G for most implementations. Because the problem sizes had to be large for the data to not fit in cache, ample parallelism was always available. Therefore, I only evaluate some of the results to decrease the amount of redundant information. Specifically, I only look at speedup with respect to G for 1 and 24 workers and parallel scalability for $G \in \{1, 128\}$.

I chose inputs with sizes that are not precisely powers of two for all benchmarks. It is easiest to implement recursive divide-and-conquer codes if the sizes of inputs are always powers of two, making it easy to divide the iteration space in half recursively.

However, such sizes can also result in conflict misses because real-world caches are not fully associative, and their sizes are often a power of two. Therefore, two elements in the same column of a 2D input might have the same low-order bits and map to the same cache set. One potential solution is to pad the 2D memory to offset the conflicts, but I opted for running the benchmarks on inputs with sizes that are not powers of 2.

5.3.1 Matrix multiply

The matrix multiply benchmark `mm` computes the product of two row-major square matrices A and B of size $N \times N$ and stores the result in matrix C of the same size. Because it performs $O(N^3)$ arithmetic and $O(N^3)$ memory operations on data of size $O(N^2)$, it is an excellent benchmark for this study. It exhibits a factor of N reuse on both reads and writes and enables excellent spatial and temporal locality benefits. Empirical results show that it can significantly benefit from improved locality in multiple dimensions.

Figure 5-2 shows simplified code that implements `mm` using nested loops. It is a three-dimensional loop nest with dimensions i, j, and k. Each loop only iterates over two matrices and always refers to the same element in the last one. For example, the i loop strides through A and C but always refers to the same element in B. It follows that locality in all three dimensions is essential. On the other hand, the j loop is the only one that does not stride through either matrix, meaning that i and k locality are more critical for spatial locality.

The order of loops in the nest may not achieve the best locality but is used for consistent comparisons with other benchmarks. The code would be correct under any permutation of the loop order in the nest. However, because the k loop must be serial, different loop orders might decrease the parallelism or increase the scheduling overheads.

In addition to the common implementations, I implemented `mm-dac-full` and `mm-best`. Those implementations also recursively split the k dimension, achieving good locality along all three dimensions instead of just i and j. The difference between

```
1. void mm(const int *A, const int *B, int *C, int size) {
2.   cilk_for (int i = 0; i < size; ++i) {
3.     cilk_for (int j = 0; j < size; ++j) {
4.       for (int k = 0; k < size; ++k) {
5.         C[i * size + j] += A[i * size + k] * B[k * size + j];
6.       }
7.     }
8.   }
9. }
```

Figure 5-2: Simplified nested loop implementation of matrix multiply. The outer two of the three loops in the nest are parallel.

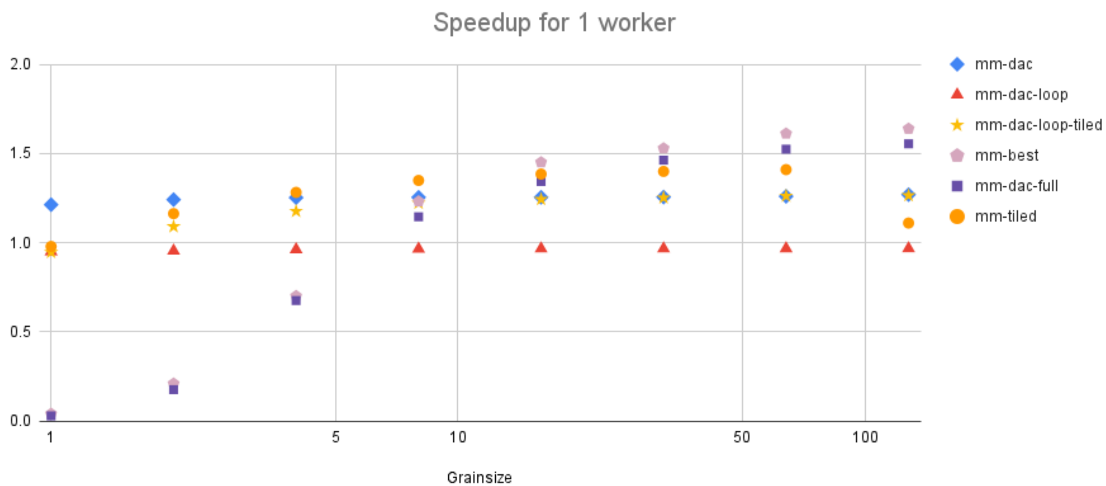


Figure 5-3: Speedup of `mm` implementations with respect to grainsize, run on a single worker.

them is that `mm-dac-full` splits along one dimension (the largest one in the current subproblem) for every level of recursion, while `mm-best` splits all three at the same time. For square matrices, both implementations perform splits in the same order but differ in the number of levels of recursion and spawned tasks by a constant factor.

For the evaluation, I chose a size of $N = 2071$ to avoid conflict misses and make sure that a column of a matrix does not fit in the L1 cache. A whole row does fit, but when iterating over a column, each element is on its own cache line. To cache the entire column, $B \cdot N = 64$ bytes $* 2071$ memory needs to fit in cache, which is larger than L1.

Figure 5-3 shows the impact of the grainsize parameter on the performance of all implementations of `mm` on a single worker. The performance of `mm-dac` and `mm-dac-loop` is independent of G. They run 1.21× and 0.95× faster than the serial elision of `mm`. Because G does not noticeably affect the locality of those implementations, this lack of performance dependence on G means that recursive overhead does not measurably contribute to the overall running time. The large size of the base case can explain that: it performs $O(G^2N)$ work, which is $O(N)$ even with $G = 1$. On the other hand, larger G improves the performance of `mm-dac-loop-tiled` and `mm-tiled` because it improves locality along the j dimension. With large enough G, `mm-dac-loop-tiled` achieves enough locality to catch up to `mm-dac`, which is expected. Finally, `mm-best` and `mm-dac-full` perform much better with larger G because their base case does $O(G^3)$ work, which is not enough to amortize the cost of recursive calls when G is low. They both achieve the best performance with $G = 128$: `mm-best` outperforms the baseline by 1.64× while `mm-dac-full` outperforms it by 1.55×.

The performance difference between `mm-dac-full` and `mm-best` can likely be attributed to a combination of performance variability and different overheads of recursion and spawning parallel tasks. The serial projection of `mm-best` is sometimes slower than the serial projection of `mm-dac-full`.

It is less clear why `mm-tiled` has similar performance to `mm-dac-loop-tiled` for $G = 1$ but outperforms it at larger G. It also measurably dips in performance at very large grainsizes. Unfortunately, I ran out of time to thoroughly investigate these deviations. Still, they could be explained by divergent compiler transformations arising from the differences in control flow between `mm-tiled` and `mm-dac-loop-tiled`.

Figure 5-4 shows the relationship between performance and G, this time for execution on 24 workers. The plot looks almost identical except for `mm-tiled` performing worse for $G \in \{64, 128\}$.

The performance difference is even more apparent in Figure 5-5, which shows the parallel scalability of all implementations for $G = 128$. Except `mm-tiled`, all implementations achieve almost perfect linear speedup, with their self-relative speedups on

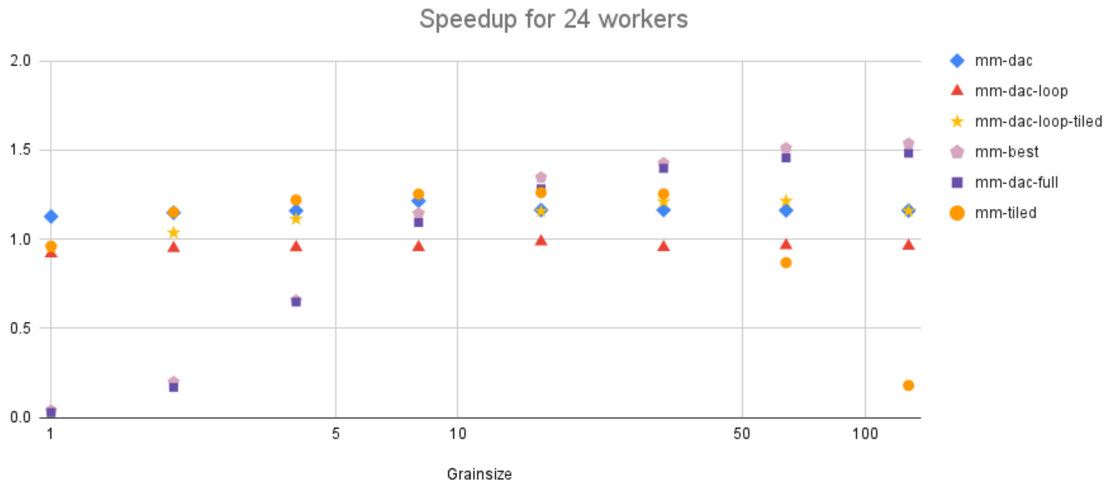


Figure 5-4: Speedup of `mm` implementations with respect to grainsize, run on 24 workers.

24 workers ranging from 22.7× to 24×.

5.3.2 Blur

`blur` is a common operation in computational photography and image processing. The benchmark takes a two-dimensional image of size $N \times N$ and performs a simple 2D convolution that averages the pixels in the 2D neighborhood of size K around each pixel. Despite a high factor of reuse, empirical results show that it does not benefit from improved locality in multiple dimensions. Figure 5-6 shows the simplified nested-loop implementation.

Because each iteration of `blur` accesses nearby cells in both directions, improved locality in both x and y dimensions reduces the number of cache misses. The regular iteration order with x (fastest-running dimension of the image) in the inner loop reuses each element along the x dimension K times, and it only incurs one cache miss for every B elements due to excellent spatial locality. Assuming a row of the image does not fully fit in cache, the number of misses is $O(\frac{N^2K}{B})$, and the fraction of misses to reads is $O(\frac{1}{KB})$. Suppose the tiled order with a sufficient tile size or Morton order is used. Then, elements can also be reused along the y axis due to better locality in the x dimension, and the number of misses is $O(\frac{N^2}{B})$. The fraction becomes $O(\frac{1}{K^2B})$,

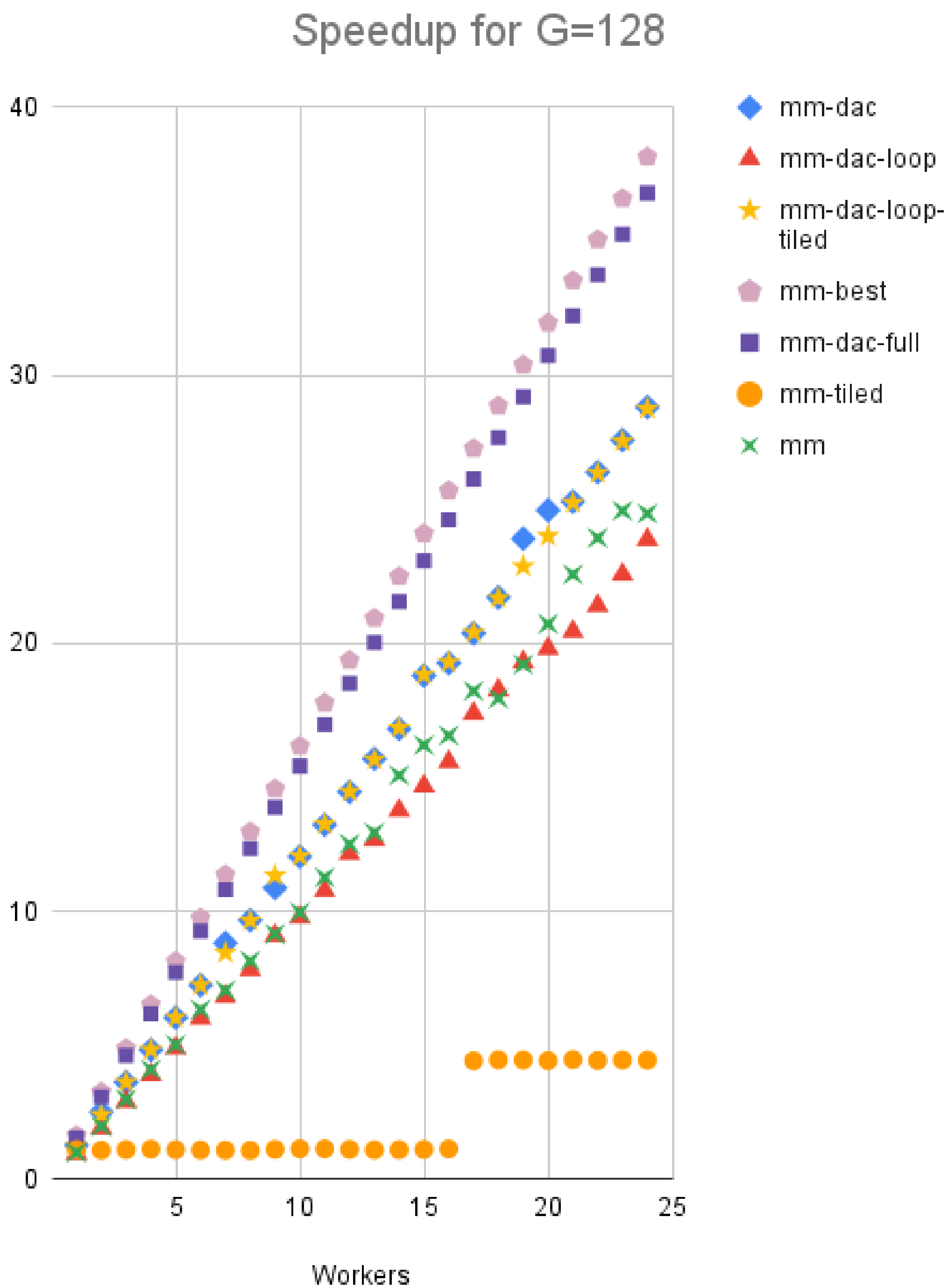


Figure 5-5: Parallel speedup of `mm` implementations for $G = 128$.

```
1. void blur(const float *in, float *out, int k, int size) {
2.   cilk_for (int y = 0; y < size; y++) {
3.     cilk_for (int x = 0; x < size; x++) {
4.       out[x + y*size] = blurPixel(in, x, y, k, size);
5.     }
6.   }
7. }
8.
9. float blurPixel(const float *in, int x0, int y0, int k, int size) {
10.   float sum = 0;
11.   for (int y_diff = -(k - 1) / 2; y_diff <= (k - 1) / 2; y_diff++) {
12.     for (int x_diff = -(k - 1) / 2; x_diff <= (k - 1) / 2; x_diff++) {
13.       int x = std::min(size - 1, std::max(0, x0 + x_diff));
14.       int y = std::min(size - 1, std::max(0, y0 + y_diff));
15.       sum += in[x + y*size];
16.     }
17.   }
18.   return sum / (k * k);
19. }
```

Figure 5-6: Simplified nested loop implementation of `blur`. `blurPixel` performs the "blurring" of a pixel and is factored out for better readability.

an additional factor of K reduction.

Even though `blur` could be written as a 4D loop-nest, the inner two loops are smaller and likely wouldn't benefit as much from improved locality. Their indices represent a relative offset in the calculation of the memory location, meaning that two different values of `x` and `y` with the same `x_diff` or `y_diff` don't have anything in common, unless `x` and `y` are close, in which case locality in those two dimensions is already present.

I chose a size of $N = 16401$ for this benchmark to make sure that a whole row of the image would not fit in the L1 cache. In addition, I chose $K = 11$ to get a significant factor of reuse. However, in hindsight, a high K also resulted in a low number of cache misses, so reducing the cache misses had a minor impact on the overall performance.

Figure 5-7 shows the performance of `blur` with respect to G on a single worker. All implementations achieve very similar performance. These results show that the improved locality does not affect the running time. For $G = 1$, `blur-dac` achieves

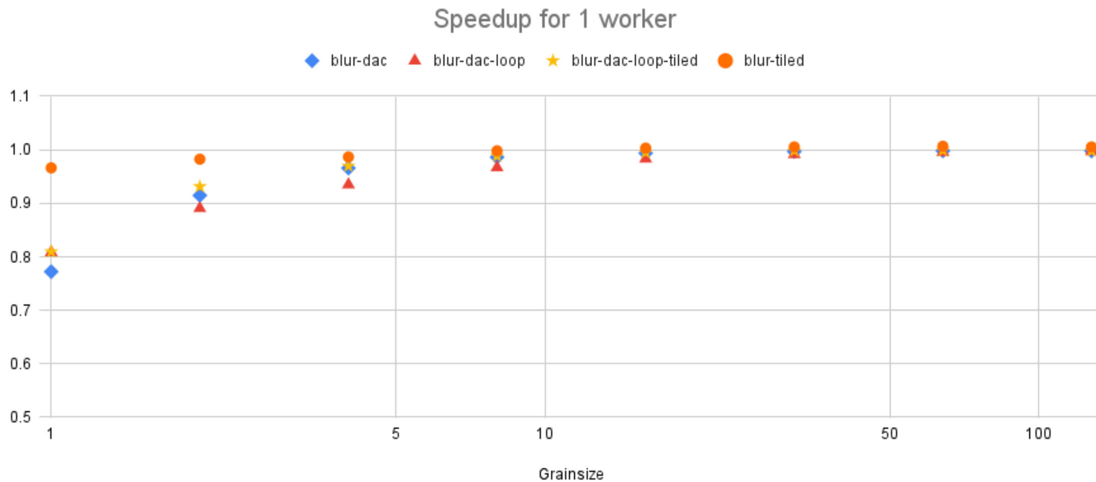


Figure 5-7: Speedup of `blur` implementations with respect to grainsize run on a single worker.

a $0.77\times$ speedup compared to the serial elision of `blur`, whike `blur-dac-loop` and `blur-dac-loop-tiled` achieve a $0.80\times$ speedup. The reason they all perform worse than `blur` at low G can be directly attributed to recursive overhead. As G increases, the difference shrinks, and the performance of those benchmarks approaches that of `blur`. This approach happens more slowly for `blur-dac-loop`, as its coarsening only occurs with a factor of G instead of G^2, and the amortized cost of recursion approaches 0 more slowly.

Meanwhile, `blur-tiled` achieves $0.97\times$ of the performance of the baseline even for $G = 1$, and quickly converges towards the baseline for larger G. It performs best with $G = 64$, where it outperforms the baseline by 0.67%. That improvement is not significant enough to conclude that the improved locality was the reason for improved performance.

Figure 5-8 shows the parallel scalability of all `blur` implementations. They all scale exceptionally well, achieving a speedup of over 23.9 on 24 cores. This happens due to ample parallelism and sufficient arithmetic intensity that does not overwhelm the memory bandwidth of the CPU, which notoriously does not scale with the number of processors.

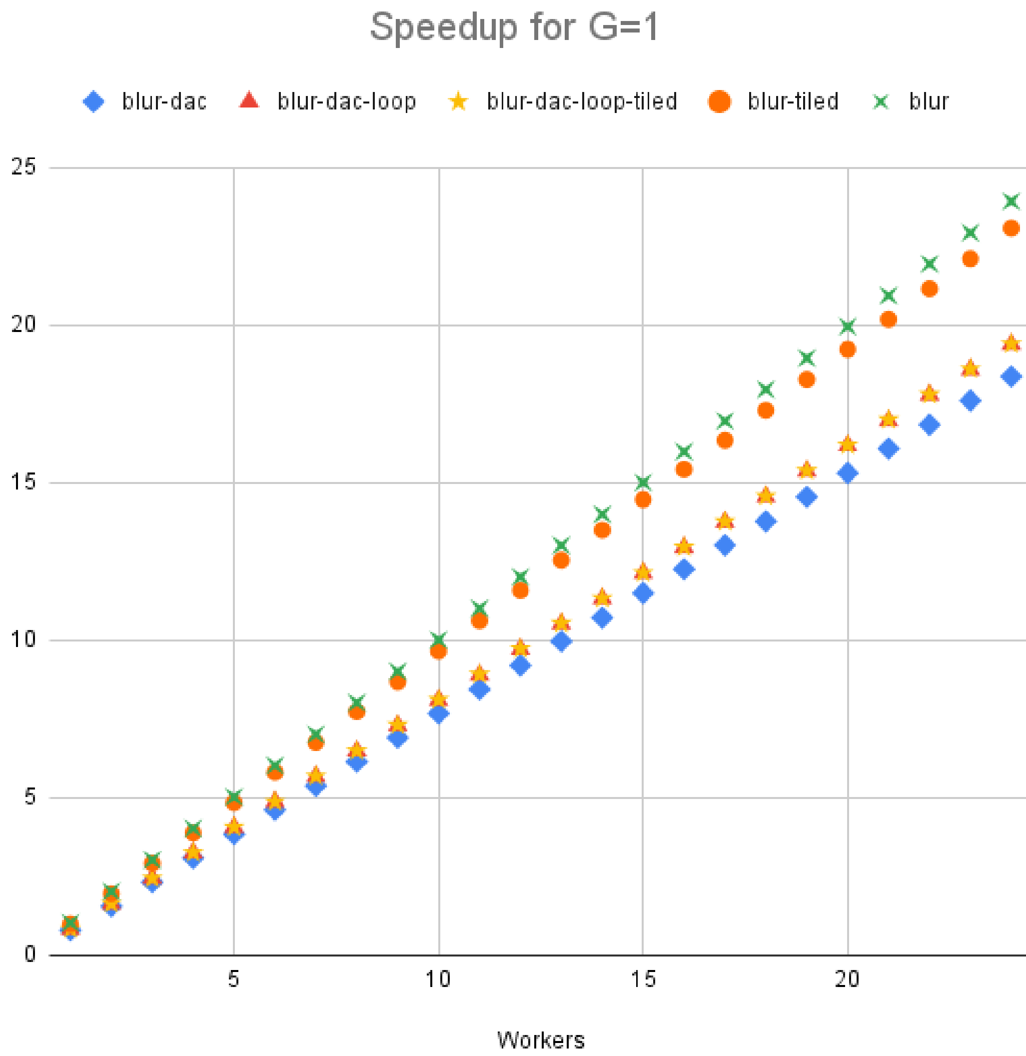


Figure 5-8: Parallel speedup of `blur` implementations for $G = 1$.

```
void transpose(uint32_t *matrix, int size) {
  cilk_for (size_t i = 0; i < size; ++i) {
    cilk_for (size_t j = i + 1; j < size; ++j) {
      std::swap(matrix[i*size+j], matrix[i+j*size]);
    }
  }
}
```

Figure 5-9: Simplified nested loop implementation of `transpose`.

5.3.3 Transpose

`transpose` calculates an in-place transpose of a square matrix. It differs from the other two because it features a ***triangular iteration space***, meaning the iteration range of the inner loop depends on the index in the outer loop. That gives the iteration space a triangular shape. Empirical results show that it can greatly benefit from improved locality in both dimensions. Figure 5-9 shows the simplified nested-loop implementation.

In `transpose`, locality along both dimensions results in good spatial locality because the code writes to the element with the switched coordinates of the element it reads from. On the other hand, there is no reuse; every element is only read and written once, and that read and write happen one after the other in all implementations.

I chose a size of $N = 16401$ for this benchmark to ensure that a whole row of the matrix would not fit in the L1 cache.

Figure 5-10 shows the performance of `transpose` with respect to G on 24 workers. Because the work per iteration is low, recursive overheads are pretty high. With larger G, improved locality of `transpose-dac` outperforms other implementations, achieving up to a 1.7× speedup compared to `transpose`. The second best is `transpose-dac-loop-tiled` with $G = 128$, achieving a 1.62× speedup. From cache-misses alone, `transpose-dac-loop-tiled` is expected to achieve similar per-

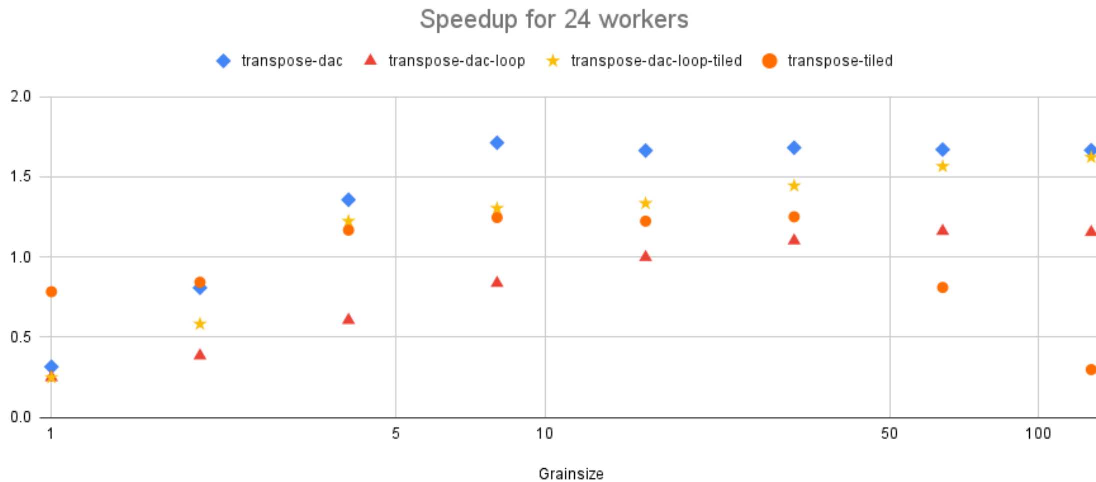


Figure 5-10: Speedup of `transpose` implementations with respect to grainsize, run on 24 workers.

formance after $G > B$, but it keeps noticeably improving in performance. Meanwhile, `transpose-dac-loop` plateaus entirely below them, which is expected due to its worse locality. But it is not likely for it to outperform `transpose`. Finally, `transpose-tiled` performs poorly with large G like other `<name>-tiled` implementations.

Because each iteration of `transpose` performs so little work, other differences between the implementations and overheads may affect these results. Further investigation would be required to uncover the cause of the unexpected side of these results.

5.4 Takeaways

Three main takeaways can be drawn from this study. First, convolutions (like `blur`) don't benefit as much from reordering despite a high factor of reuse. Second, applications where locality along more than one dimension results in either temporal or spatial cache-locality can benefit tremendously from reordering the iterations. Benchmarks in this study achieved up to a factor of 1.7×. Third, manually optimizing implementations and making them cache-efficient can require much work.

Careful tuning is required to achieve the best cache performance, which can also be hard to measure. Cache-efficient implementations, whether cache-aware or cache-oblivious, can get complex and must be tightly coupled to the application logic, making the code prone to off-by-one errors and other bugs. This coupling further justifies compiler transformations that can reorder iterations automatically without programmer input.

I could not thoroughly investigate the poor scalability of `<name>-tiled` for all three benchmarks. I believe that the OpenCilk compiler performed its own coarsening of the parallel loops. This coarsening could result in puny parallelism when combined with the manual coarsening done by the implementation.

Future studies should focus on experimenting with different input sizes for `blur`, other base case approaches for `mm`, and further investigation of `transpose` performance results. A lower K would mean lower reuse and a smaller relative reduction in cache misses for `blur-dac`, but it might increase the absolute decrease in cache misses and significantly impact performance. When experimenting with different orders of loops, I found that changing the order of nested loops inside `mm` yielded an implementation that was even faster than `mm-best`, which was partially due to better vectorization, but it certainly warrants a closer look.

Chapter 6

Related work

This section discusses related work on parallel loops and optimizations to randomized work stealing. It also presents related work on improving locality of nested loops.

A substantial body of previous work has studied a variety of schemes for partitioning parallel loops to improve performance [48, 28, 57, 40, 27, 7, 8, 54]. One common scheme involves statically partitioning the n iterations of a parallel loop into P chunks of n/P iterations each, where P is the number of processing threads, and then assigning each chunk to a processor [45, 3]. Although static partitioning incurs low overheads, its performance suffers when the work of the loop is not balanced across its iterations or the execution of the parallel loop does not have dedicated access to the processors. In contrast, dynamic partitioning schemes, like DACParFor, partition parallel loop iterations into small chunks whose size is independent of the number of processing threads and then allows a scheduler to dynamically schedule chunks onto processors, e.g., via work stealing or work sharing. Dynamic partitioning can overcome the shortcomings of static partitioning by providing automatic load balancing but at the cost of increased overhead. Other schemes, such as guided self-scheduling [48] or factoring [28], partition the loop iterations into variable-sized chunks, whose size is based on P and then the number of unexecuted loop iterations remaining. Previous work has also explored schemes to schedule and load balance parallel loops based on statistics collected at runtime [7, 8] and based on memory-access patterns, such as spatial locality across parallel loop iterations [54].

Loop frames implement dynamic partitioning of parallel loops that load balances work automatically via work stealing. Loop frames and on-the-fly loop splitting also differ from other schemes that dynamically split unexecuted loop iterations, e.g., [48, 28, 54], which typically use a central queue to manage the loop iterations. In contrast, on-the-fly loop splitting distributes unexecuted loop iterations in a decentralized fashion using worker-local deques. Furthermore, in contrast to previous work, on-the-fly loop splitting presented in this thesis has a parallel running time bound with work stealing, derived in [47].

Tzannes *et al.* introduce lazy binary splitting [55], and a generalization called lazy scheduling [56], to reduce the overheads of work stealing. These schemes split parallel loops in a dynamic, lazy fashion by pushing work onto a worker's deque only when the deque is empty. Lazy scheduling differs from loop frames in two ways. First, although lazy scheduling pushes partitions of parallel-loop iterations lazily onto the deque, these partitions are still created in a manner consistent with DACParFor. In contrast, on-the-fly loop splitting partitions the unexecuted iterations in the loop frame at the time of a steal. These partitions do not necessarily correspond with partitions generated using DACParFor, which necessitates a separate analysis of work stealing with loop frames. In addition, loop frames and lazy scheduling reduce work-stealing overheads in different ways. Lazy scheduling reduces overheads by avoiding synchronization operations on a nonempty deque. In contrast, loop frames reduce overheads by using a concise representation of unexecuted parallel-loop iterations in a single deque frame. It remains a topic of future work to see if these techniques can be combined.

Acar *et al.* explore another approach to reducing work-stealing overheads by studying work-stealing with private deques [2]. Private deques allow a work-stealing scheduler to avoid unnecessary memory fences during execution, thereby reducing synchronization overheads. It remains an open research problem to apply similar techniques to loop frames. In particular, future work might investigate designating a subinterval of a loop frame's iterations to be private in order to reduce synchronization costs involved with updating the *start* variable in a loop frame.

Chapter 7

Conclusion

This thesis introduces loop frames and on-the-fly loop splitting, which extend randomized work stealing with first-class support for parallel-for loops. Loop frames and on-the-fly loop splitting maintain the same theoretical guarantees as the traditional divide-and-conquer algorithm for parallel-for loops while providing lower overheads in practice and flexibility in dynamically splitting loop iterations. This particularly impacts applications with lower parallelism where extensive coarsening cannot be used. Although this work introduces on-the-fly loop splitting, it remains an interesting research question to fully leverage the flexibility it provides.

This thesis also investigates the impacts of improved locality of multidimensional iteration spaces on performance. While convolutions like `blur` don't benefit as much from reordering, applications with temporal or spatial cache-locality can benefit tremendously. Cache-efficient implementations can be complex and error-prone, justifying the use of compiler transformations that automatically reorder iterations. Future studies should continue to explore different input sizes and approaches for various benchmarks.

Bibliography

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proc. of the 12th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA 2000)*, pages 1–12, 2000.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP*, page 219–228, 2013.

[3] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. *FastFlow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. John Wiley & Sons, Ltd, 2017.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification Version 1.0.* Sun Microsystems, Inc., March 2008.

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[6] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[7] I. Banicescu and Z. Liu. A dynamic scheduling method tuned to rate of weight changes. In *HPC*, pages 122–129, 2000.

[8] Ioana Banicescu and Vijay Velusamy. Performance of scheduling scientific applications with adaptive weighted factoring. In *IPDPS*, page 84, USA, 2001. IEEE Computer Society.

[9] Rajkishore Barik, Zoran Budimlić, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 735–736, Orlando, Florida, USA, 2009. ACM.

[10] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, pages 181–192, 2012.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[12] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, February 1998.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[14] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[15] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, October 1981.

[16] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The new adventures of old X10. In *PPPJ*, pages 51–61, 2011.

[17] 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 *OOPSLA*, 2005.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[19] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, December 2008.

[20] Rainer Feldmann, Peter Mysliwietz, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation. In *SPAA*, 1994.

[21] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[22] Raphael Finkel and Udi Manber. DIB — A distributed implementation of backtracking. *ACM TOPLAS*, 9(2):235–256, April 1987.

[23] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[24] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

[25] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[26] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[27] Susan Flynn Hummel, Jeanette Schmidt, R. N. Uma, and Joel Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, page 318–328, Padua, Italy, 1996. ACM.

[28] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: A method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, August 1992.

[29] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[30] Intel Corporation. *Intel Cilk Plus Language Extension Specification, Version 1.2*, 2013. Document 324396-003US.

[31] Richard M. Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[32] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, June 1989.

[33] Bradley C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645.

[34] Monica S. Lam, E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algortihms. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, Santa Clara, CA, April 1991. ACM SIGPLAN Notices 26:4.

[35] Doug Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

[36] I-Ting Angelina Lee. *Memory Abstractions for Parallel Programming*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, 2012.

[37] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[38] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.

[39] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? volume 368. American Association for the Advancement of Science, 2020.

[40] Jie Liu, Vikram A. Saletore, and Ted G. Lewis. Safe self-scheduling: A parallel loop scheduling scheme for shared-memory multiprocessors. *International Journal of Parallel Programming*, 22(6):589–616, 1994.

[41] Michael McCool, Arch D. Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[42] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS '11)*, October 2011.

[43] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario, March 1966.

[44] Rishiyur S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[45] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.5*, July 2015. Available from `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.

[46] *OpenMP Application Program Interface, Version 3.0*, May 2008.

[47] Nipun Pitimanaaree. Provably efficient randomized work stealing with first-class parallel loops. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2019.

[48] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, 1987.

[49] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[50] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, et al. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[51] Tao B. Schardl, I-Ting Angelina Lee, and Charles E. Leiserson. Brief announcement: Open cilk. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 351–353, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *PPoPP*, pages 249–265, 2017.

[53] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The Problem Based Benchmark Suite. In *SPAA*, pages 68–70, 2012.

[54] Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec, and Bruno Raffin. A work stealing scheduler for parallel loops on shared cache multicores. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, pages 99–107, 2011.

[55] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. In *PPoPP*, page 179–190, 2010.

[56] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *TOPLAS*, 36(3), September 2014.

[57] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Trans. Parallel Distributed Syst.*, 4:87–98, 1993.

[58] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[59] Helen Jiang Xu. *The Locality-First Strategy for Developing Efficient Multicore Algorithm.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2022.