SINGLE-JOB DYNAMIC PARALLELISM SCALING THROUGH LOCK CONTENTION MONITORING

By

Mahesh Khanwalkar

Georgia Institute of Technology

May 2020

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

Multicore systems have become ubiquitous, with most modern computing systems boasting multiple cores. Therefore, the ability to effectively harness the available parallelism resources (cores and/or threads) becomes incredibly important. However, the goal of extracting the most out of the available resources is complicated due to many potential issues that degrade performance and waste resources.

One such issue — lock contention — is a major concern due to the prevalence of locks and locking constructs in multithreaded code. Lock contention is characterised by an increasing demand for a particular shared resource. That is, multiple threads are attempting to access or modify a resource, and therefore have to compete with each other to acquire the lock that protects this resource. In general, increased parallelism can substantially decrease execution times, under the assumption that there is not too much lock contention. However, for many problems dealing with irregular data structures, like tree and graph algorithms, the level of lock contention is not constant. For example, MST algorithms start with low levels of contention but end with very high levels. This creates two problems: performance loss near the end of execution and resource waste. The high level of lock contention implies that threads will be spending most of their time waiting to acquire the locks, thereby hogging resources, while not performing meaningful work: more time is spend waiting than performing useful computations, in regards to whatever the algorithm is doing.

From this issue arises the question of whether dynamically adjusting the number of threads allotted to the algorithm's execution could potentially reduce the impact of lock contention. Specifically, having many threads allocated at the beginning but then slowly decreasing the available threads until the algorithm completes. The principle of dynamically adjusting allotted resources has been explored extensively within work-stealing with

various feedback mechanisms [1, 2, 3] proposed that inform the scheduler how to better allocate resources to the running tasks, maximising resource utilization while minimising waste. While work stealing is an entirely different problem, in that it does not directly deal with the issue of lock contention, the idea of feedback mechanisms could be useful in reducing the impact of lock contention.

The work presented here introduces the idea of dynamically scaling up or down the number of threads using the metric of failed lock acquisition attempts, by borrowing the idea of feedback mechanisms from work stealing. Lock acquisition failure is a count of the number of times acquiring a particular lock failed because it was already held by some other thread. Therefore, the reasoning behind using this metric is that it should act as a reasonable estimate for the level of lock contention. That is, high counts of failure occurs if and only if there is a high level of lock contention. A special thread pool was designed to track this metric, automatically disabling internal worker threads upon encountering a high threshold and enabling internal worker threads upon encountering a low threshold. The performance impact of dynamic scaling was evaluated on Boruvka's MST algorithm [4, 5] as a proof of viability. The algorithm was tested with different input graphs, and the execution time speedup for both normal parallel and dynamic scaling was recorded, relative to the serial execution, to show the performance benefit of using the proposed dynamic scaling approach.

CHAPTER 2
LITERATURE REVIEW

The task of easily creating correct, thread-safe code has been a complicated issue. Using solely low-level parallelism constructs, like locks can potentially be problematic, since they open up a range of possible issues, like deadlock and livelock. Therefore, allowing for program design using higher level constructs becomes an important objective. There are two main approaches to this problem: domain specific languages (DSLs) [6] and existing language extensions, as well as, library support. The proposed dynamic scaling approach in this thesis adopts the library approach, due to its simplicity and broad applicability to existing workflows and ecosystems.

DSLs and existing language extensions allow for these high-level constructs to live as a part of the language itself, often times allowing for shorter, more readable code. OpenMP [7], a library and extension of C/C++, can be employed for task parallelism, data parallelism, and hybrid parallelism use cases. OpenCL [8], an extension of C, facilitates the creation of compute kernels and the use of parallelism with vector types. Parallel programming languages, like Chapel [9], NESL [10], and X10 [11], have support for various parallelism paradigms, like data and task parallelism.

The primary disadvantage of DSLs is that they require a new toolchain to be used, which involves some sort of specialized compiler [12]. In addition, there are no IDEs to increase productivity and there is a relative lack of documentation compared to general purpose languages. In contrast, popular existing language extensions like OpenMP and OpenCL are well supported by different vendors [7, 8], both in terms of compiler extensions and other tooling, allowing for a richer ecosystem than smaller, more specialized DSLs. In order to reduce the amount of work required by the user, the dynamic scaling approach purposefully does not employ DSLs, although it is technically still a possible option to consider in the

future. The main reason for choosing a library is that existing code can be easily modified to work with the proposed library, rather than a full port to a new programming language.

Libraries that implement high level parallelism constructs, like task parallelism, have been introduced to many different general-purpose programming languages. Intel TBB [13], a template library for C++, provides functions to support constructs like parallel loops and provides implementations of thread-safe data structures. Java's concurrency package [14], also provides various thread-safe data structures, as well as synchronisation constructs like semaphores, barriers, and phasers. One primary advantage of libraries is that they do not require specialized toolchains [12]. Therefore, library-based solutions enjoy the full support of existing mature toolchains and IDEs. However, libraries may incur verbosity and work-arounds [12], since the constructs are being retroactively implemented using standard language features, rather than being first-level members in a specialised language. This does not happen to be a problem in the dynamic scaling library, since it does not need to expose complex parallelism constructs. Rather, a simple clean interface is the goal to ensure ease of use and minimal effort on the library user's part.

While there are many different choices in terms of languages and libraries to develop parallel applications in, not all problems can be easily parallelised in a simple manner. For example, the structure of parallelism and locality in algorithms dealing with irregular data structures, like graphs and trees, are not well known [15]. This lack of insight has been a roadblock towards the development of tools and techniques for implementing parallel versions of these algorithms.

M. A. Hassaan et al. [15] explore the parallelism profile of the Boruvka's MST algorithm. The paper finds that, at the beginning of the computation, the algorithm can use a large amount of available parallelism, but this does not stay true for the entirety of the execution. As the computation continues, the amount of available parallelism that can be used decreases rapidly. To optimise the parallel implementation of the algorithm, the algorithm was split into two phases, Match and Merge, which allows for the work chunking

optimisation to be applied. There was a speedup of 2.7 and 11.9 recorded for small graph inputs and large graph inputs, respectively over the serial version of the algorithm.

Since Boruvka's MST algorithm initially can exploit a large degree of available parallelism but then can only use a small degree towards the end of execution, dynamically adjusting the available parallelism resources becomes a potential avenue of performance improvements. The principle of dynamically adjusting resources to match tasks' requirements has been explored by various algorithms that extend basic work-stealing [1, 2, 3].

K. Agarwal et al. [1] present an adaptive work-stealing scheduling strategy (A-STEAL), where there is cooperation between the thread and job scheduler, and where the thread scheduler provides continual feedback (input) to the job scheduler, informing it of situations where it has been overallocated certain resources. This allows the job scheduler to reassign those wasted resources, like idle cores, to other jobs, allowing for fuller resource usage. The main investigation in the paper is how to provide effective parallelism feedback to the job scheduler. Specifically, this paper proposes a classification method for previous quanta, making changes based on this classification to adapt for the next quantum. The paper also introduces a new evaluation technique, trim analysis, to evaluate the algorithm in terms of time and waste, which allows the authors to prove certain bounds on the speedup.

G. Varisteas and M. Brorsson introduce Palirria [3], an adaptive work-stealing implementation that attempts to estimate the needs of a certain task and can dynamically adjust the number of allotted resources accordingly. This allows for better performance for different kinds of parallel tasks, even ones with irregular patterns of parallelism. The proposed algorithm is able to make certain estimations using a Deterministic Victim Selection (DVS) policy and also uses a quantum-based approach to estimate resource requirements. The paper also has an in-depth comparison of A-STEAL and details that it has similar performance (as a baseline) and does draw performance benefits over A-STEAL under certain workloads. Both algorithms are similar in their use of quanta; however, the metrics by which they estimate resource requirements are different, potentially explaining the perfor-

mance benefits that were observed.

A. Tzannes et al. [2] propose the idea of lazy scheduling, which tries to solve the problem of manual pruning to eliminate excessive parallelism, by making the process dynamic, allowing it to adapt to varying load conditions. This lazy scheduling is integrated into the standard paradigm of work-stealing, based on calculations of total system load. The paper presents a few different lazy scheduling models, evaluating where each model is most productive in efficiently extracting the usage of available parallelism in the system. In addition, the paper covers existing industry technologies, like Intel TBB and Cilk Plus [16], evaluating how these technologies handle effectively the same problem of job scheduling and specifically pointing out certain limitations in these existing designs.

While these papers propose different methods of extending work-stealing, the adaptive nature described does not need to be tied to work-stealing: it can potentially be applied, in general, to dynamically scaling up threads (adding) or scaling them down (disabling), using some sort of feedback mechanism and a metric for determining the parallelism needs.

For example, this idea of feedback driven concurrency control has been explored within the context of Software Transactional Memory (STMs). K. Ravichandran and S. Pande introduce F2C2-STM [17] which uses techniques previously employed within TCP congestion control to dynamically monitor and adjust concurrency levels in applications. Specifically in the case of scalability-limited STM applications, the dynamic concurrency control technique can automatically limit the concurrency to reduce execution time and improve resource utilisation. The performance of this approach was evaluated using an estimate of the total committed transaction throughput (transactions per interval, $tpi_{head} \approx tpi_T$), since an excessive or increasing number of conflicts in an STM environment leads to throughput loss, indicating that the level of allotted concurrency should be scaled back. In addition, maximising throughput is the goal rather than minimising conflicts, since minimising conflicts comes at the cost of low concurrency and higher execution times. The concurrency control technique, similar to TCP, maintains a *window*, representing the current level of

concurrency. This window grows at the beginning of execution, following a pattern similar to that of TCP slow start, with an initial exponential growth phase followed by a linear growth phase, and continues to adjust throughout the execution of the application, allowing for adaptation to any changes in concurrency levels.

STMs are a different paradigm than the one being examined in this thesis, in that STM applications do not use or maintain their own explicit locks. However, the work presented in this thesis still has the same objective of concurrency control and reducing execution time and follows the same high-level approach of using feedback mechanisms. Therefore, the successful use of feedback mechanisms within STMs for a similar purpose provides further motivation for exploring this approach in the context of explicit locking applications.

This thesis explores dynamically adjusting the available parallelism (threads) given during the execution of a parallel Boruvka's MST algorithm. The metric employed is failed lock acquisitions, where a rapidly growing number of failures indicates rising lock contention and the need for scaling back threads. This dynamic scaling framework is encapsulated within a library, and so does not require any toolchain modifications. While the Boruvka's MST algorithm is examined here, the dynamic scaling could potentially be applied to different algorithms, given they provide a failed lock acquisition count during execution.

CHAPTER 3
METHODOLOGY

The parallel Boruvka MST algorithm exhibits growing lock contention when it nears completion. That is, the amount of lock acquisition failures is increasing and the worker threads are no longer as productive because they are spending more time fighting over the shared resources. Therefore, the ability to scale back the number of allocated threads when encountering growing lock contention would be particularly useful in this scenario.

This thesis presents a dynamic scaling thread pool. The pool naturally has an initial set number of worker threads. However, the number of threads can increase or decrease seamlessly using a feedback mechanism driven by lock contention. The parallel Boruvka algorithm was chosen as a case study to evaluate the performance of the proposed work and was modified to provide the metric of lock contention via the number of lock acquisition failures.

3.1 Parallel Boruvka

The implementation of parallel Boruvka examined here works off of a shared queue of connected components. This queue is initialised with all the vertices of the graph, since the components will be built as the algorithm proceeds and the tree is formed. Each worker thread will attempt to dequeue a component and merge it with the associated component of its best outgoing edge.

This merge process requires the acquisition of two locks: one for the original component and one for the component it will be merged with. The acquisition of these locks is done using a tryLock call. If the lock is already being held, then the call will return false, rather than blocking until the lock is released. Once the components have been merged, then the resulting component is enqueued. The algorithm terminates once the queue is

Algorithm 1: Parallel Boruvka

```
function compute(Q):
    input: Q is the work queue
    output: T is computed MST
    while |Q| > 0 do
        let comp = dequeue(Q)
        if tryLock(comp) ≠ true then
            continue
        end
        if processed(comp) = true then
            unlock(comp)
            continue
        end
        let e = bestedge(comp)
        if e = ∅ then
            let T = comp
            unlock(comp)
            return T
        end
        let other = getother(e, comp)
        if processed(other) = true then
            unlock(other)
            unlock(comp)
            enqueue(Q, comp)
            continue
        end
        merge(comp, other)
        unlock(comp)
        unlock(other)
        enqueue(Q, comp)
    end
```

empty — which signals that the final minimum spanning tree has been created.

This implementation was scheduled onto a fixed number of worker threads in a simple manner, depicted by Algorithm 2. The threads themselves are created, with the algorithm as their target. Then, each thread begins executing. Finally, the main thread issues a join on each thread to wait until they all complete, before exiting out of the scheduling function.

Algorithm 2: Simple Scheduling Algorithm

function `schedule`(*n, Q, res, compute*)**:**
input: **n** is the number of threads, **Q** is the work queue, **res** is the result of the computation, **compute** is a function that computes the MST
let ts = $[t_1, t_2, ..., t_n]$
for *i = 0* $\rightarrow$ *n* **do**
t_i = *Thread(compute(Q, res))*
end
for *i = 0* $\rightarrow$ *n* **do**
start(t_i)
end
for *i = 0* $\rightarrow$ *n* **do**
join(t_i)
end
return

3.2 Dynamic Boruvka

The parallel implementation as-is cannot be used on top of the dynamic thread pool, because it does not currently provide any feedback to the runtime. However, the required modifications are relatively simple, on purpose: a large development time cost could potentially outweigh the performance benefits gained.

The metric that the runtime uses is lock contention, which is measured using the number of lock acquisition failures. Therefore, the main change to Algorithm 1 is keeping track of the number of times tryLock failures and passing this information to the runtime. Algorithm 3 contains the required modifications to provide the lock failure count to the runtime. This is facilitated using count which is an atomic integer reference that is shared by the algorithm and the underlying runtime. The runtime can then see the changes made by the algorithm and can proceed accordingly.

In addition, dynamic runtime needs to be able to tell the algorithm to quit and release any unfinished work. The thread itself cannot be "killed" directly in a safe manner, since the algorithm could be currently processing something and that work would then be lost, resulting in improper execution. This problem is therefore mitigated using an atomic boolean

shared between the runtime and the algorithm which tells the algorithm to quit. Once the algorithm quits, the worker thread that was running the algorithm dies, thereby scaling back the amount of available parallelism.

Algorithm 3: Parallel Boruvka with Failure Count and Quit

```
function compute(Q):
    input: Q is the work queue, count is an atomic integer, and quit is an atomic
     boolean
    output: T is computed MST
    while |Q| > 0 OR quit ≠ true do
        let comp = dequeue(Q)
        if tryLock(comp) ≠ true then
            count++
            continue
        end
        if processed(comp) = true then
            count++
            unlock(comp)
            continue
        end

        /* The rest of the algorithm is omitted, since it
           is the same as Algorithm 1                      */
    end
```

Internally, there is a quit signal for each worker thread, so that they can be controlled independently of each other. In addition, the runtime decides whether and when to turn on the quit signal, which is based on the current failure count. In cases where there is little lock contention, the quit signal may never be asserted, so thread scale back never occurs.

It is important to note that, the runtime itself only communicates with the algorithm via the lock failure count and the quit parameters. In addition, the runtime is not tied to the algorithm that is running on it: the only requirement is that the algorithm provides the required lock failure count and quits when the flag is true. That is to say, the runtime has not been specifically tailored towards the Boruvka's MST algorithm in any way.

CHAPTER 4

RESULTS

The sequential and standard parallel algorithm performances were tested using static graph resources, specifically three roadmap data graphs: Florida, Northeast USA, and Eastern USA, which were sourced from 9th DIMACS Implementation Challenge [18].

Within the execution of the program within a single JVM, the execution time was recorded over thirty trials, to smooth out any potential variations when computing the averages, which are listed in the tables below. In addition, System.gc() was called before each trial was started (before the start time was recorded), so as to encourage any garbage collection activity to occur now, rather than during the trial run, which could lead to certain trials having longer execution times.

4.1 Baseline Parallel Performance

Before testing out the proposed dynamic scaling framework, an initial baseline needs to be set between the sequential and parallel version of the Boruvka MST algorithm. The two algorithms were tested against the sample roadmap data with different thread allocations. The execution time and speedup were recorded for each run and are listed in the tables below:

Table 4.1: Sequential vs. Parallel - Florida

# of Threads	Sequential (ms)	Parallel (ms)	Speedup
4	1776.2	891.3	1.99
8	1792.2	673.6	2.66
12	1694.8	842.1	2.01
16	1714.3	812.0	2.11

Table 4.2: Sequential vs. Parallel - Northeast USA

# of Threads	Sequential (ms)	Parallel (ms)	Speedup
4	2982.0	1203.2	2.48
8	2971.0	917.0	3.23
12	2913.9	1142.5	2.55
16	2913.1	1177.8	2.47

Table 4.3: Sequential vs. Parallel - Eastern USA

# of Threads	Sequential (ms)	Parallel (ms)	Speedup
4	6710.8	2891.3	2.32
8	6807.4	2110.8	3.23
12	6755.3	2390.6	2.83
16	6813.2	2457.5	2.77

With the three different graphs and the different thread allocations, the parallel versions all showed a significant speedup over the sequential versions. However, for 12 and 16 threads, speedup is lower than that of 8 threads.

This reproduces the expected behaviour of performance loss when too much parallelism is statically allocated to the algorithm. Having 12 or 16 threads is likely yielding a great deal of lock contention towards completion, which is resulting in a performance degradation relative to the 8-thread allocated run.

4.2 Dynamic Scaling Performance

The special thread pool that was designed determines whether to scale the number of threads up or down based on a threshold. The initial threshold and adjustment policy will directly impact the execution time of the algorithm being tested. Therefore, to evaluate the performance and usefulness of the dynamic scaling, different initial thresholds and thread counts were used.

4.2.1 Static Threshold Policy

The first set of trials used a static threshold – one that is never adjusted during the entire algorithm execution. That is, the threshold stays the same regardless of the number of available threads in the pool. The static threshold is paired with a varying degree of starting threads, like the trials run in Section 4.1, to explore the interaction between number of threads and the threshold. The results of these trials are listed in the tables below:

Table 4.4: Static Threshold: 15 tryLock – Florida

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	1694.2	647.8	1625.6	0.40
12	1682.2	783.1	1656.4	0.47
16	1690.8	840.6	1652.2	0.51

Table 4.5: Static Threshold: 15 tryLock – Northeast USA

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	2825.7	853.6	2770.6	0.31
12	2958.4	1170.9	2832.3	0.41
16	2888.8	1059.4	2839.8	0.37

Table 4.6: Static Threshold: 15 tryLock – Eastern USA

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	6705.5	2107.5	6352.2	0.33
12	6780.9	2307.9	6350.3	0.36
16	6951.4	2406.3	6410.4	0.38

The static threshold of 15 tryLock failures was not successful at all. Most of the trials resulted in severe performance degradation relative to the standard parallel version. Keeping such a low threshold triggered aggressive scale-back, which prematurely cut the number of threads. Therefore, the performance became very similar to that of the sequential execution, since most of the execution was done by a single thread, as the other threads were shut down.

Table 4.7: Static Threshold: 125 tryLock – Florida

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	1791.2	588.2	550.7	1.07
12	1805.9	650.9	631.0	1.03
16	1877.0	705.3	675.3	1.04

Table 4.8: Static Threshold: 125 tryLock – Northeast USA

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	2667.8	862.9	822.7	1.05
12	2715.3	1193.5	1012.5	1.18
16	2733.9	1010.0	950.3	1.06

Table 4.9: Static Threshold: 125 tryLock – Eastern USA

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	6580.4	2115.9	2055.8	1.03
12	6875.2	2529.2	2217.3	1.14
16	7013.1	2437.6	2390.9	1.02

For the 125 tryLock failures static threshold, the results are much more promising than the low threshold tested before. All the trials resulted in a performance benefit between the dynamic and parallel versions. Some of the trials resulted in large performance gains ($> 10\%$), while others were not as exciting.

However, that is to be expected: the variance in the results is due to different input graphs. Different inputs will exhibit different structures of parallelism and lock contention. Therefore, the realised benefit from the dynamic approach will directly depend on this structure. Nevertheless, multiple trials yielding a significant performance benefit is reassurance that the approach is successful.

Table 4.10: Static Threshold: 1024 tryLock – Florida

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	1678.5	656.4	633.5	1.04
12	1705.2	790.8	826.3	0.96
16	1697.7	855.7	852.2	1.00

Table 4.11: Static Threshold: 1024 tryLock – Northeast USA

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	2736.7	861.3	870.3	0.99
12	3102.3	1175.8	1132.5	1.04
16	2870.2	1106.9	1139.7	0.97

Table 4.12: Static Threshold: 1024 tryLock – Eastern USA

# of Threads	Sequential (ms)	Parallel (ms)	Dynamic (ms)	Speedup (P/D)
8	6716.5	2113.7	2152.2	0.98
12	6882.8	2438.9	2453.2	0.99
16	6851.4	2455.6	2475.6	0.99

When the threshold is set very high, like in the case of 1024 tryLock failures, there is no significant performance benefit over the standard parallel approach. This is likely because the scaling policy became too conservative. In many cases, scaling never occurred since the threshold was too high. Therefore, it would be actually worse (albeit only slightly worse) than the standard parallel approach because there is an overhead to manage the feedback mechanism.

From these trials, the overall best threshold was the 125 tryLock failure count. Selecting a very small threshold resulted in an overly aggressive scaling which is detrimental to performance. Selecting a very large threshold resulted in very conservative scaling which amounted to no noticeable performance benefit or detriment over the standard parallel version. Hence, selecting a static threshold somewhere in the middle of both extremes yields a beneficial result: it strikes a balance between not scaling enough and scaling too quickly.

CHAPTER 5
CONCLUSION & FUTURE WORK

Lock contention complicates the task of effectively allocating parallelism resources to a particular job. For many problems with complex structures of parallelism, simply allocating a fixed number of threads is not productive after a certain point. Therefore, to extract additional performance benefit, a more nuanced approach must be taken to reduce the lock contention and its effects.

The dynamic scaling approach proposed here does just that. The metric of lock acquisition failures acts as a good approximation of the current level of lock contention that the algorithm is encountering. By using this metric to guide the feedback mechanism, the level of lock contention can be reduced by scaling back the number of threads at the appropriate time. The results of trials on the parallel Boruvka MST algorithm were significant performance benefits over the parallel version.

The major benefit of pursuing the library based approach for dynamic scaling is ease of use. The algorithm being run on the special thread pool library only needs to provide the metric of lock acquisition failure to the runtime. That is, no other changes are required and this change in of itself should be relatively trivial. This is particularly useful, since this library allows additional performance to be gained while putting in very little effort to port the algorithm to the framework.

While the current approach already accomplishes the task of reducing lock contention thereby reducing execution time, there are still many potential avenues to explore in the problem and approach explored within this work.

It would be interesting to explore shifting the scaling trigger from a separate thread to the actual worker threads. That is, design a specific method that the algorithm could call to perform a threshold check, rather than relying on a regular timer to check the status in the

background.

One issue in the current timer-based checking is that it does not adjust well to a rapid rate of increase in lock contention. Therefore, it is quite possible to have the current lock failure count be well over the threshold when the timer fires and the check occurs. Effectively, a balance needs to be struck between constant checking and long intervals.

Lastly, the dynamic scaling runtime could be extended to collect statistics. The number of times scaling-up or scaling-down operations performed and a view of lock acquisition failures over time would be useful metrics to capture, since they give a better view of the internal operation and conditions being experienced. An important additional statistic could be the energy impact of the runtime. Specifically, the energy delay product could be calculated based on the total execution time and the execution times for each of the active worker threads. This metric would be useful in evaluating the performance vs energy trade-off and tuning towards an optimal balance.

REFERENCES

[1] K. Agrawal, Y. He, and C. E. Leiserson, "Adaptive work stealing with parallelism feedback," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07, San Jose, California, USA: ACM, 2007, pp. 112–120, ISBN: 978-1-59593-602-8.

[2] A. Tzannes, G. C. Caragea, U. Vishkin, and R. Barua, "Lazy scheduling: A runtime adaptive scheduler for declarative parallelism," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 3, 10:1–10:51, Sep. 2014.

[3] G. Varisteas and M. Brorsson, "Palirria: Accurate on-line parallelism estimation for adaptive work-stealing," in *Proceedings of Programming Models and Applications on Multicores and Manycores*, ser. PMAM'14, Orlando, FL, USA: ACM, 2007, 120:120–120:131, ISBN: 978-1-4503-2657-5.

[4] D. J. Lawrie, *Kruskal's and Boruvka's Algorithms*, `http://www.cs.loyola.edu/~lawrie/CS302/S12/lecture/302-23.pdf`, 2012.

[5] N. R. Latha, G. Shyamala, and G. R. Prasad, "Exploring the parallel implementations of the three classical mst algorithms," in *2017 International Conference on Inventive Communication and Computational Technologies (ICICCT)*, 2017, pp. 340–346.

[6] A. Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *SIGPLAN Notices*, vol. 35, pp. 26–36, Jan. 2000.

[7] The OpenMP ARB, *OpenMP*, `https://www.openmp.org/`.

[8] The Khronos Group, *OpenCL*, `http://www.khronos.org/opencl/`.

[9] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[10] G. E. Blelloch, "Nesl: A nested data-parallel language. (version 3.1),," 1993.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[12] V. Cavé, Z. Budimlić, and V. Sarkar, "Comparing the usability of library vs. language approaches to task parallelism," in *Evaluation and Usability of Programming Lan-*

guages and Tools, ser. PLATEAU '10, Reno, Nevada: ACM, 2010, 9:1–9:6, ISBN: 978-1-4503-0547-1.

[13] Intel, *Intel Thread Building Blocks*, https://software.intel.com/en-us/intel-tbb/.

[14] Oracle, *Package java.util.concurrent*, https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/package-summary.html.

[15] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11, San Antonio, TX, USA: ACM, 2011, pp. 3–12, ISBN: 978-1-4503-0119-0.

[16] Intel, *Intel Cilk Plus*, https://www.cilkplus.org/.

[17] K. Ravichandran and S. Pande, "F2c2-stm: Flux-based feedback-driven concurrency control for stms," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 927–938.

[18] Center for Discrete Mathematics & Theoretical Computer Science, *9th DIMACS Implementation Challenge - Shortest Paths*, http://users.diag.uniroma1.it/challenge9/download.shtml/.