Latency Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing

Vladimir Milicevic
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
vladimir.milicevic@carleton.ca

December 12, 2021

1 Abstract

Modern computing systems employ hundreds of processor cores which support simultaneous multithreading. Many language extensions for parallelism exist, such as Cilk and OpenMP. When running programs written in these parallel languages, the runtime system instantiates user-level threads which are mapped to the processors using a scheduler without prior knowledge of the threads. Some instructions may require significant time to complete, such as remote procedure calls or accessing databases from a disk. This creates a latent period whereby unused resources may be exploited to make progress on other work within the computation. In this work, a latency-hiding work-stealing scheduler is presented which allows idle processors waiting for high-latency instruction to steal tasks from busy processors and improve processor utilization while making progress on the overall work of parallel computations.

2 Introduction

Much prior work has focused on scheduling threads which do not incur latency penalties while operating, such as sending transactions over a network. While this simplifies previous parallel scheduling algorithms such as work stealing, this hinders performance as the scheduler runs threads to completion without any switching. This class of non-preemptive scheduler appears adequate when workloads are heavily computational such as traditional high-performance computing applications, where application threads rarely incur any latency penalties. This work extends the directed acyclic graph (dag) model of parallel computing to improve thread-level performance by accounting for the latency of operations that are greater than the unit-latency available.

3 Literature Review

Recent works in parallel scheduling have varied across dimensions between long and short-term scheduling along with coarse grain task schedulers and fine-grain thread scheduling. Many works in parallel scheduling have focused on fine-grain scheduling of hardware IO resources, such as scheduling jobs to fairly allocate IO bandwidth in network applications [6], other works in fine-grain short term scheduling studied effective schedulers for parallel computation on hierarchical caches [2]. Some works have aimed to capture system IO latencies within coarse-grain jobs schedulers to minimizing maximum flow time (makespan) [4] which effectively minimizes job latency. Others have moved towards implemented efficient algorithms for fine-grain scheduling uni-processor threads within a multi-processor system which incorporate inter-processor IO delays [7]. Some work has been done on scheduling unit-length tasks on asynchronous multi-processor systems [3], however both [7] and [3] do not consider variable-length instructions across multiple processors.

Other work aims to schedule coarse-grain long term jobs on parallel machines using game theory frameworks [5], work has been conducted in parallel job schedulers which batch threads during critical sections that access a common structure [1] to avoid hazards at the expense of blocking concurrent access by other threads.

Extensions such as Interactive Cilk (I-Cilk) [9] have been proposed for the popular mulithreaded parallel computing language Cilk. Extensions such as I-Cilk aim to allow programmers the ability to explicitly state priority for task-parallel code, and the runtime schedules of the computation in order to optimize response time for high-priority tasks. However this required the programmer to make assumptions about the state of the runtime, and would not allow the programmer to create schedules accounting for the overall system state during execution. For example, many modern multi-threaded multi-processor systems may co-schedule jobs that exhibit similar CPU or IO access leading to system bottlenecks,. Avoiding such behaviour requires the programmer to speculate on the complete system behaviour during execution. In general, these past works lack focus on fine-grain online scheduling of threads with non-unit execution time.

Many workloads which exploit the inherent parallelism of modern hardware such as data-center and database applications, are not solely computational but also display high communication usage patterns. Accessing remote storage or even invoke remote procedure calls on devices within a network. Access to these remote devices is often shown to incur various latency penalties while a thread waits for a response. Previous work in work-stealing scheduling algorithms do not account for the latency incurred by a particular thread. As such, this work presents a work-stealing algorithm [8] that hides latency by allowing threads to suspend and be swapped out by the preemptive scheduler to improve the performance of the underlying scheduling algorithm. Such techniques which allow the schedule to prompt the latency-bound thread and run another useful thread improve performance by hiding latency.

The use of a non-preemptive scheduler to schedule user-level threads differs from operating system schedulers, as the non-preemptive parallel schedulers studied here must control the order in which millions of fine-grain threads execute, while an OS scheduler seeks to hide the latency of course-grain jobs in the order of thousands. The non-preemptive latency

hiding work stealing model for schedulers explored in this work shows mathematically the behaviour of parallel threads that incur latency penalties, and develops a latency-hiding scheduling algorithm for such computation. The benefit of such a scheduler is the ability to allow threads to suspend, and be overtaken by other threads on the hardware.

4 Selected Work

The work by Muller et el.[8] extends the standard model of parallel computations as Directed Acyclic Graphs (DAGs, or dags) by allowing edges to have non-unit weights that represent the latency incurred by an instruction. General instructions such as integer or floating-point arithmetic operations incur no latency penalty, but IO operations or remote procedure calls represent non-unit weight edges. The span of the DAG accounts for the latency incurred on the critical path. The scheduling algorithm used in this work uses deques (double-ended queues) to model threads ready for execution. Different from prior work, this work uses multiple deques per-worker (almost, virtual-deques) which are switched out when appropriate. The introduced algorithm is online and requires no prior knowledge of the DAG or edge weights, but requires knowledge of which edges are non-unit weight. The notion of *Suspension Width* is introduced to indicate the number of non-unit edges on the critical path of a DAG.

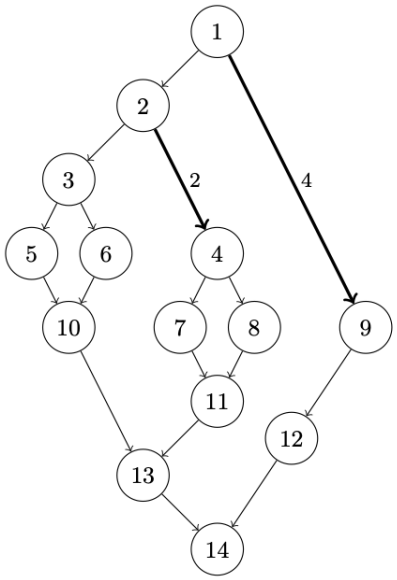

Figure 1: Instruction DAG including non-unit edges with suspension widths 2 and 4.

The offered runtime bound for a DAG with W work, S span, and $U \geq 1$ suspension width running on a P-processor system is: $O(\frac{W}{P} + SU(1 + logU))$. When the suspension width is 0 (i.e., all unit-weight edges) the algorithm achieves the bound $O(\frac{W}{P} + S)$ which is identical to standard work stealing.

5 Weighted Directed Acyclic Graph Model

The directed acyclic graphs (dag) are commonly used as a model for parallel computing, in which each parent is an instruction of a thread that may spawn one or more child nodes (subsequent instructions). This standard model is extended to include dags with weighted edges, allowing the model to capture instructions that incur non-unit latency by assigning them with heavy edges extending from their parent. From a multithreaded or parallel program, edges within a dag originate from three conditions: (1) the parent precedes the child in a single program thread, (2) the parent spawns a new thread whose first instruction is the child, or (3) the threads containing the parent and child synchronize, and serialize the instructions represented by the parent and child nodes. The model spawns child threads as the right-child of a parent, with the continuation of the parent thread portrayed in the left-child. This is a key feature for the non-preemptive scheduling algorithm.

We label a parent node as u, with its child represented with v. A parent and child are connected by an edge represented as (u, v, δ). Each edge is labeled with its latency, δ, where unity represents a 'light' edge that may be executed immediately. When $\delta > 1$ the edge is referred to as 'heavy' and requires a δ-step latency between initial execution of u and the use of results by v. In essence the connection between parent and child nodes $(u, v, \delta)|\delta > 1)$ results in v not being scheduled for execution for at least δ time steps.

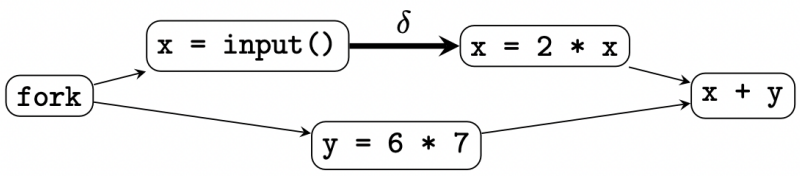

Figure 2: Example of a weighted dag.

This scheduling algorithm makes the following assumptions of the dag representing the parallel computation:

1. A dag has a single root with in-degree zero, and a final vertex with out-degree zero.
2. The out-degree of any vertex is at most two. Recall the model spawns child threads as the right-child, with the continuation of the thread portrayed in the left-child. This results in an instruction spawning, or synchronizing with, at most one thread.
3. A vertex v with heavy incoming edge has an in-degree of 1.
4. The dag is deterministic i.e., instruction latency has no temporal dependence (i.e., δ will remain constant no matter the order in which, or the time at which, any portion of the dag is discovered).

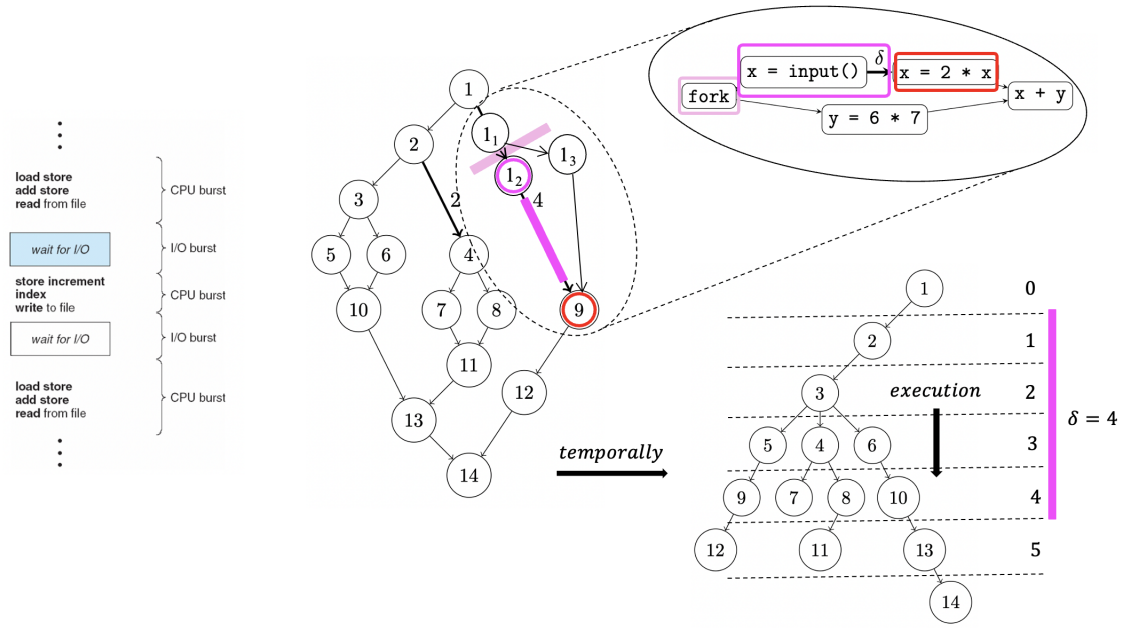

Figure 3: Example Workload, IO Bursts Incurring Non-unit Latency, Execution Graph Over Time.

The span of the weighted computation dag is the longest weighted path in the dag. When a dag has no weighted edges, this reduces to the traditional notion of span. The definition of work remains unchanged from the traditional model. The measure U is defined as the suspension width of a dag. More specifically, a parallel computation is represented by a dag $G = (V, E)$ with vertices V and Edges $E \subset V \times V \times N$. When $s$is the root vertex and t is the final vertex of G, the partitions $P = (S, T)$ of G where $S \uplus T = V$ such that S and T form a connected subdag of G such that $s \in S$ and $t \in T$. The suspension width of G is then defined as the maximum number of heavy edges that cross any partition, such that:

$$argmax_{(S,T)\in P}|\{(u, v, \delta) : u \in S, v \in T, (u, v, \delta) \in E|\delta > 1\}|$$

This measurement is relevant to scheduling weighted dags as it defines the maximum number of suspended instructions at any given time. The depth $d_G(v)$ of v in the original parallel computation dag G is the longest weighted path from the root of computation to v in G. Let the enabling span be the longest path in dag when unrolled temporally to include vertices where the Scheduler inserts completed vertices using its `pfor` capabilities. When a suspended vertex whose incoming edge has weight δ becomes ready, the targeted worker will not have continued processing deeper than δ levels in the computation dag, so the `pfor` vertex is inserted at a depth no more than factor $logU$ greater than $d_G(v)$, thus the enabling span is $O(S(1 + logI))$.

6 Scheduling Environment

The scheduling problem focused on in this work is the online scheduling problem for weighted dags, whereby a dag is scheduled as its structure is revealed during execution. The Latency Hiding Work Stealing (LHWS) Scheduler does not know the exact latency of non-unit length

instructions, however, can identify a 'heavy' edge as the dag structure is revealed.

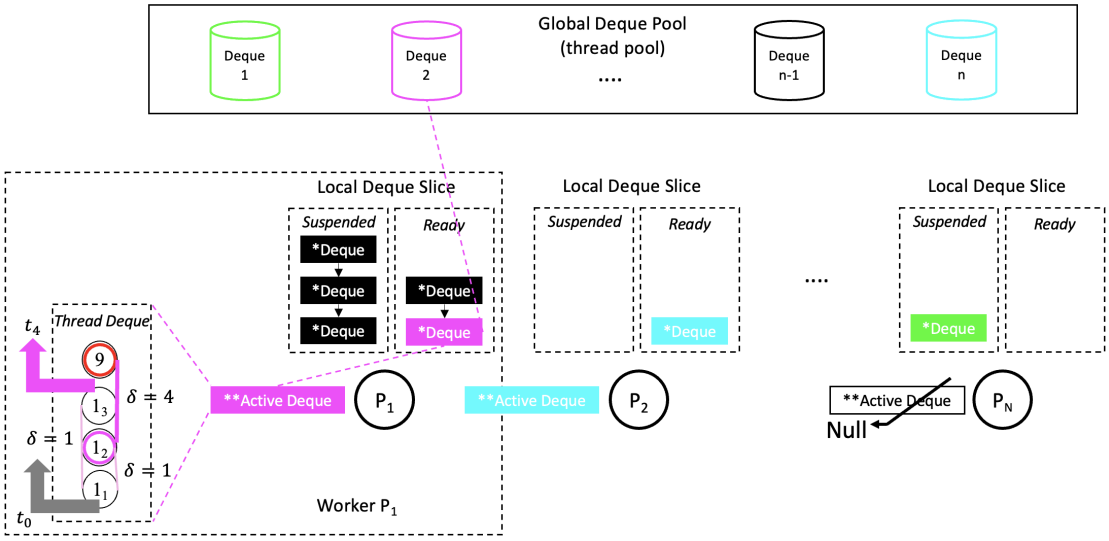

Figure 4: Abstract Scheduling Environment.

This variant of work stealing uses Double-ended Queues (Deques) to store vertices to be executed. Each worker owns a collection of deques which are split into two sets: a Ready set, and a Suspended set. The Suspended set contains empty deques that are targets to suspended vertices, while the Ready deques contain vertices ready for execution. One of the Ready Deques is assigned as the workers Active Deque.

All deques are pre-allocated in a Global Deque Pool, workers dereference pointers to these global deques in their local deque sets. The size of the Global Deque Pool is selected to allow each worker unfettered access to deques given a particular workload, while avoiding the need to use a growable array to house the Global Deque Pool, as this would incur copy and synchronization overhead. A global counter of total deques used is maintained, which bounds the random selection of deques during work stealing – the chosen deque may have been "freed" which results in a failure. Finally, the scheduler has a list of all resumed vertices since it's last invocation.

7 Scheduling Algorithm

Traditionally, work stealing schedulers use a single deque per worker, however, LHWS schedulers use multiple deques per worker. The worker has a single active deque; the worker attempts to pop work out of the bottom of its active deque. If successful, the worker executes the vertex and spawns up to two children, which are pushed onto the bottom of the active deque. If the vertex suspends (i.e., a heavy edge connects the executed parent node to any child) the vertex is linked to the active deque, and the worker returns to the deque to continue acquiring work.

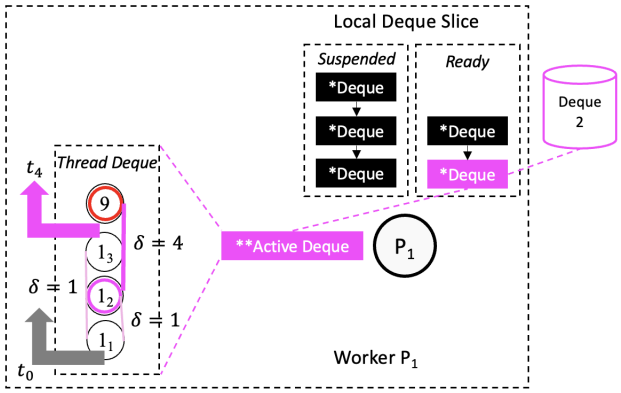

Figure 5: Abstraction of Worker P1 Local Resources, Deque 2 in Global Deque Pool.

If the worker returns to the deque and finds it empty, it checks the other deques it owns for work in a similar manner described above. If all the local deques are empty, the worker attempts to steal work from the global deque pool. To obtain work from the global deque pool, the worker selects a random deque and attempts to pop a vertex from the top. If successful, the worker inserts the stolen work into a new deque and makes the new deque its active deque.

The suspension of a vertex $v$from deque q triggers an additional step, whereby a callback `callback(v,q)` is installed to the vertex which, upon resumption, runs to add the vertex into a set of resumed vertices. The scheduler checks the resumed set, and when available partitions the resumed vertices by their deque. Once vertices resume after suspension for -unit steps, the scheduler evokes `addResumedVertices()` that executes a closure for all resumed vertices in a parallel for-loop belonging to the target deque. The closure allows the scheduler to push the resumed vertex onto the bottom of the target deque. It is assumed the listed operations take amortized constant time, and the execution of a parallel for-loop is possible in linear work and logarithmic span in the size of the set.

Work begins on a computation dag by setting one worker's `assignedVertex` to the root of the dag, and setting all workers `activeDeque` to an empty deque. When a worker has an assigned vertex, it takes the following actions during that round:

1. Execute the vertex, receive any children the vertex may spawn.
2. Call `handleChild()` on the right child, then to preserve task order (non-preemptive scheduling),
3. Call `addResumedVertices()`
4. Call `handleChild()` on the left child to preserve task order.

The LHWS Scheduling algorithm is a cooperative scheduler, and so does not pre-emptively reschedule tasks that suspend. Recall left children are the continuation of program task, while the execution of a right child node signifies the spawning of a new program thread.

```
Procedure callback(v, q)
   q.resumedVertices.add(v)
   q.suspendCtr--
   resumedDeques.add(q)

Procedure addResumedVertices() for (q in resumedDeques)
   for (u in q.resumedVertices)
      spawn thread[i](u, execute()) ; i++ ;
      v.add (u)
   sync thread
   q.pushBottom(v)
   readyDeques.add(q)
   resumedDeques.remove(q)
   q.resumedVertices.clear()

Procedure handleChild(v)
   q = activeDeque
   if (v.isSuspended)
      v.installCallback(callback(v, q))
      q.suspendCtr = q.suspendCtr++
   else
      q.pushBottom(v)

Procedure main() // Scheduler
   while (!computationDone)
   if (assignedVertex != NULL)
      (left, right) = assignedVertex.execute()
      if (right != NULL)
          handleChild(right)
      addResumedVertices()// Add Resumed Vertices, allows non-preemptive
          scheduling.
      if (left != NULL)
         handleChild(left)// Handle Left child, the continuation of current thread.
      assignedVertex = activeDeque.popBottom()
   else
      if (activeDeque.suspendCtr = 0)
             activeDeque.free()
             activeDeque = NULL
      // Try to resume a ready deque.
        new = readyDeques.removeAny()
      if (new != NULL)
          activeDeque = new
      else //Attempt to steal from Global Deque Pool.
          victim = randomDeque()
            assignedVertex = victim.popTop()
            if (assignedVertex != NULL)
                activeDeque = newDeque()
      addReadyVertices() // Resume Ready Vertices
      if (assignedVertex == NULL) // Null Vertex, try to find more work.
          assignedVertex = activeDeque.popBottom()
```

Table 1: LHWS Scheduler Psudeocode.

8 Results

A Distributed Map and Reduce example is considered, in which we want to map a function f(x) over exactly 5000 workers, then combine the resulting values. Latency is introduced into the computation by supposing each worker must fetch a value stored on a remote device. These results are compared against the original paper, and a work-stealing algorithm developed by Spoonhower et al. [10]. The results show the self-speedup curves for the LHWS scheduler and the standard work-stealing (WS) scheduler.

All results are shown relative to the one-processor run of the works-stealing scheduler. The time-step is iterated over 50ms, 5ms, and 1ms for each algorithm. The results show superlinear speedups compared to the standard work-stealing scheduler. This is expected with latency-hiding as the standard WS algorithm does not hide latency. A high value results in substantial speedup, and as reduces towards the latency of a complete round of scheduling, there is less benefit of hiding latency. This is due to the inability for latency hiding to be exploited during rounds where several vertices are suspended.

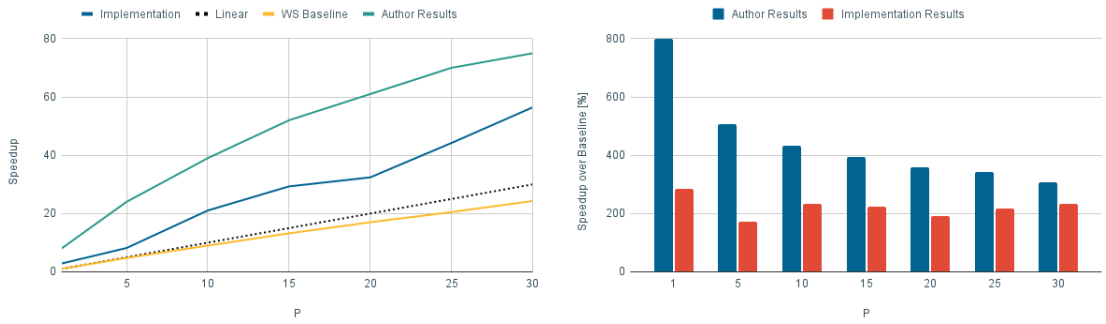

Figure 6: 50ms Heavy Edge Map-Reduce Results.

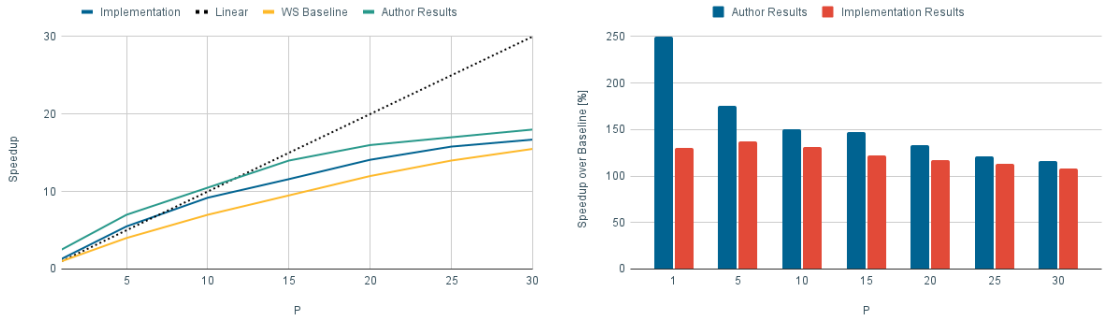

Figure 7: 5ms Heavy Edge Map-Reduce Results.

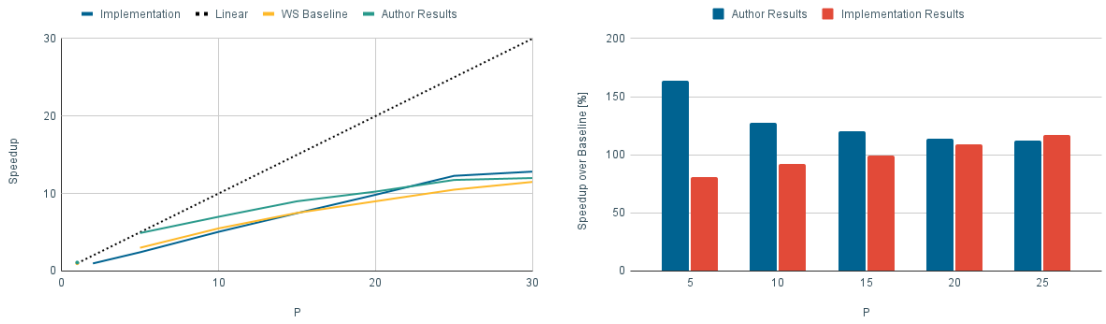

Figure 8: 1ms Heavy Edge Map-Reduce Results.

9 Conclusion

This work provides an extension to the tradition parallel computation directed acyclic graph (DAG) model to support non-unit latency incurring operation within the computation. A Latency Hiding Work Stealing Scheduler (LHWS) is shown which can efficiently schedule the latency incurring operations while continuing to provide high resource utilization. Salient improvements over normal Work-stealing (WS) scheduling algorithms include the ability to use multiple deques per-worker and not degrading performance when all operations are unity-delay.

References

[1] Kunal Agrawal, Jeremy T. Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, page 84–95, New York, NY, USA, 2014. Association for Computing Machinery.

[2] Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling parallelizable jobs online to minimize the maximum flow time. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 195–205, New York, NY, USA, 2016. Association for Computing Machinery.

[3] Michael A. Bender and Cynthia A. Phillips. Scheduling dags on asynchronous processors. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, page 35–45, New York, NY, USA, 2007. Association for Computing Machinery.

[4] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, page 355–366, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Leah Epstein and Elena Kleiman. Scheduling selfish jobs on multidimensional parallel machines. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms*

and Architectures, SPAA '14, page 108–117, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Ajay Gulati and Peter Varman. Lexicographic qos scheduling for parallel i/o. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, page 29–38, New York, NY, USA, 2005. Association for Computing Machinery.

[7] H. Jung, L. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, page 254–264, New York, NY, USA, 1989. Association for Computing Machinery.

[8] Stefan K. Muller and Umut A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 71–82, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. Priority scheduling for interactive applications. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 465–477, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Daniel John Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, USA, 2009. AAI3358062.