Scheduler Augmentation

A Lightweight, Customizable, Low-Cost Profiling Technique for Fork-Join Parallel Programs

Sam Westrick*
s.westrick@nyu.edu
New York University
New York, New York, USA

Darshan Dinesh Kumar*
dd3888@nyu.edu
New York University
New York, New York, USA

Seong-Heon Jung*
jung.s@nyu.edu
New York University
New York, New York, USA

Abstract

For performance analysis, optimization, and debugging, it is often useful to abstract away the low-level details of how a parallel program is scheduled, and instead focus on the logical structure of the program's execution in the form of a *computation graph* (a.k.a. "the dag"). However, existing tools for dynamic computation graph analysis are tightly integrated with specific compilers and run-time systems, making them difficult to use in conjunction with lightweight scheduling libraries.

We propose ***scheduler augmentation*** to address this gap. Our approach is to add a *vertex interface* to the scheduler, enabling it to explicitly track vertices in the computation graph. Users then provide their own definitions of vertex-local data and specify how this data should be updated, locally, at forks and joins. We show that scheduler augmentation can encode a wide variety of interesting features, including work and span profiling, space profiling, and granularity analysis. We specifically consider two case studies: (1) identifying subcomputations that are too fine-grained to benefit from parallelism, and (2) optimizing an application's worst-case memory high-water mark.

We describe how to implement scheduler augmentation as an extension to standard work-stealing schedulers, and implement our approach in ParlayLib, a lightweight scheduling library for C++. Notably, our implementation requires no changes to the compiler. In an evaluation across 30 benchmarks from PBBS and ParlayLib, we show that typical augmentations have very little cost: we measure on average less than 3% overhead across all processor counts (between $P = 1$ and 80), and less than 10% for 27 of the 30 benchmarks. Altogether, our results show that scheduler augmentation is an attractive option for supporting a wide variety of user-customizable dag-based features, without significant overhead or maintenance burden.

CCS Concepts

• Software and its engineering → Parallel programming languages; **Software performance**; **Dynamic analysis**; **• Theory of computation → Shared memory algorithms**.

*All authors contributed equally.

This work is licensed under a Creative Commons Attribution 4.0 International License.
SPAA '26, London, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2761-0/2026/07
https://doi.org/10.1145/3816782.3819212

Keywords

work-stealing, scheduling, fork-join, parallelism, lightweight libraries, profiling

ACM Reference Format:
Sam Westrick, Darshan Dinesh Kumar, and Seong-Heon Jung. 2026. Scheduler Augmentation: A Lightweight, Customizable, Low-Cost Profiling Technique for Fork-Join Parallel Programs. In *38th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '26), July 06–10, 2026, London, United Kingdom.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3816782.3819212

1 Introduction

Numerous languages and libraries provide a scheduler to dynamically map parallel tasks onto processors, enabling the programmer to focus on algorithmic concerns and express parallelism at a higher level of abstraction. In this general approach, it is possible for a programmer to debug and optimize their program in a ***schedule-oblivious*** manner: rather than considering the low-level details of how tasks are scheduled onto processors, they instead focus on the logical structure of the program's execution, often in the form of a ***computation graph*** or "***dag***."

The unifying abstraction of the computation graph has fostered a rich ecosystem of schedule-oblivious analysis tools and programming features. The CilkView [17] and CilkProf [30] tools can make accurate predictions about the running times and scalability of parallel executions using data collected from just a single execution trace, essentially by overlaying profiling information onto the computation graph. Similar techniques have also been developed for space profiling, such as in CilkMem [19] (which can predict the high-water mark space consumption of a P-processor execution) and to provide a space-cost semantics for pure functional languages [34]. There is a large body of work on leveraging computation graphs to detect determinacy races [7, 14, 18, 24, 37, 39, 40, 44], as well as closely related properties such as memory entanglement [42]. Finally, we note that there are interesting programming features that are closely integrated with computation graphs, such as deterministic parallel random number generators [22, 35]. Quite a few of these prior works internally use a schedule-oblivious vertex labeling scheme such as offset-span [24], English-Hebrew [26], DePa [43], or pedigrees [22].

Across all of these prior works, we observe a common underlying mechanism: during execution, the system explicitly builds or incrementally observes the elements of the computation graph. The work we present in this paper is motivated by the desire to abstract this underlying mechanism into a general and reusable technique, which can be easily integrated across a wide variety of languages and libraries.

One especially important motivation is to bring dag-based analysis and debugging techniques to *lightweight* parallel libraries, such as ParlayLib [8], Intel Threading Building Blocks [41], the Rust Rayon library [28], the Java Fork/Join framework [21], `domainslib` in multicore OCaml [13, 33], etc. The advantage of these libraries is that they require no special compiler support, and can easily be integrated into a larger project, even just to parallelize a single key subroutine. However, as far as we know, no existing lightweight library provides support for incremental observation or maintenance of the computation graph during execution. The reason is two-fold.

(1) Incremental maintenance of a computation graph seems to require significant modifications to the scheduler, which could impact performance and maintainability. Such modifications are generally not desirable in a lightweight library, unless the modifications are small, maintainable, and can be easily deactivated when not in use.
(2) Prior work has generally focused on full-fledged compilers and run-time systems, such as Cilk/OpenCilk [9, 16, 31], OpenMP [27], X10 [12], Habanero Java [11], and many others. In these settings, it is possible for the compiler to support specific features and user-customizable instrumentation (for example, using instrumentation hooks provided by OpenCilk [31]). Such support is not available in lightweight libraries, which are designed for compilers without built-in support for structured parallelism.

Scheduler Augmentation. In this paper, we present a solution in the form of a general technique called ***scheduler augmentation***. Our approach enables the programmer to "augment" the scheduler with custom functions to observe the computation graph during execution. These augmentations can easily be deactivated or left uninstantiated, resulting in zero overhead by default. When in use, the overhead of augmentation is localized to primitives that the scheduler already exposes, and is therefore easy to predict. Furthermore, because programmers are already careful to amortize the overheads of scheduling (through granularity control and other techniques), we find that typical augmentations incur very little overhead in practice, even without any changes to existing application code.

We focus in particular on fork-join computation graphs. In our approach, the programmer defines a custom notion of *vertex-local data* and specifies how this data is updated at forks and joins, and at the start and stop of every vertex. The scheduler then automatically calls the programmer-defined functions to incrementally maintain the computation graph as directed by the programmer.

To demonstrate the expressiveness of scheduler augmentation, we discuss how a wide variety of schedule-oblivious features can be implemented only by specifying a custom notion of vertex-local data. We present a number of examples, and especially focus on two case studies in detail: a granularity analysis technique (Section 3), and a space profiling technique (Section 4). In both cases, scheduler augmentation enables the programmer to quickly and efficiently identify performance issues at the level of the computation graph (as opposed to a specific schedule of the computation graph).

Scheduler augmentation can be implemented by making reasonably small and maintainable changes to well-known scheduling techniques. We specifically consider work stealing [5, 10], and show how to modify standard work-stealing implementations to enable augmentation. We implement scheduler augmentation in the ParlayLib C++ library [8], and in this setting, we measure the overhead of typical augmentations across 30 benchmarks from ParlayLib and the Problem-Based Benchmark Suite [8]. Our results show very little overhead in practice (less than 3% on average, for typical lightweight augmentations) and almost no impact on scalability. All of our source code is available online,[1] including our modifications to ParlayLib and our experiments.

Altogether, we argue that scheduler augmentation is an effective and practical technique for extending the functionality of lightweight schedulers and libraries. We encourage the maintainers of lightweight parallel libraries to consider officially supporting scheduler augmentation as a low-maintenance and (essentially) zero-cost option to support a wide variety of user-customizable schedule-oblivious features.

2 Scheduler Augmentation

Fork-Join and Computation Graphs. We consider fork-join computations written in terms of a single primitive: `pardo(f,g)`. A call to `pardo` (1) spawns two child tasks to execute the function calls `f()` and `g()` in parallel, (2) waits for both child tasks to return, and then (3) resumes execution of the caller. Spawning the two child tasks is called a ***fork***, and the corresponding synchronization point (waiting for both children to complete) is called a ***join***. Calls to `pardo` can be freely nested, often recursively. Under the hood, a scheduler dynamically manages tasks as they are spawned and assigns these tasks to processors.

Computations in this style can be modeled as directed acylic graphs ("***dags***") called ***computation graphs***. Vertices in the dag represent computations that execute sequentially, uninterrupted by any forks or joins, and edges indicate scheduling dependencies between these computations. The graphs produced by fork-join computations can be inductively defined in terms of sequential and parallel composition of subgraphs, and are commonly referred to as ***series-parallel*** (SP) graphs. Every such graph has a single ***source*** vertex and a single ***sink*** vertex, which could be equal (in the singleton case).

Scheduler Augmentation. To observe the computation graph during execution, the programmer provides a ***vertex definition*** which satisfies the vertex interface shown in Figure 1. Specifically, the programmer defines a vertex type, `V`, and four methods on this type:

`v.start()`: called when the vertex `v` begins execution.
`v.stop()`: called when `v` finishes its execution due to either a fork, join, or program termination.
`v.fork(&vl,&vr)`: called when a vertex `v` forks, creating two child vertices `vl` and `vr`.
`v.join(&vl,&vr,&vc)`: called when the two vertices `vl` and `vr` join into a new vertex `vc` for the resumed caller. The vertex `v` is the vertex of the corresponding fork.

The series-parallel structure matches each `v.fork(&vl,&vr)` call with a corresponding `v.join(&vl_end,&vr_end,&vc)` call. The vertices `vl_end` and `vr_end` are the sinks of the two child tasks;

[1] https://github.com/nyu-parcour/scheduler-augmentation

```
class V {
  // ... omitted: constructors, etc.
  void start();
  void stop();
  void fork(V* left, V* right);
  void join(V* left, V* right, V* afterJoin);
}
```

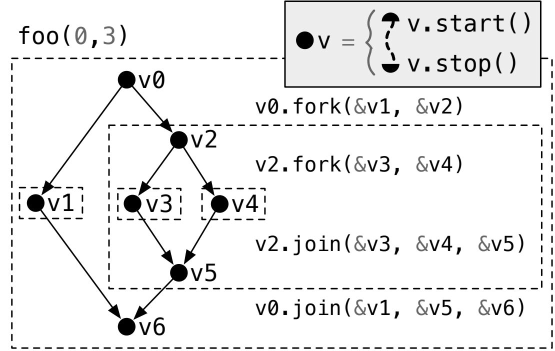
Figure 1: Vertex interface. Custom vertex types, provided by the client, must adhere to this interface.

```
template <typename LF, typename RF>
void pardo(LF&& left_func, RF&& right_func);

template <typename F>
V augment(F&& func); // indicate region of interest

// can be called inside augmented region
V* getCurrentVtx();
```

Figure 2: Augmented scheduler interface. Functions `augment` and `getCurrentVtx` are our additions. The `pardo` function is the main fork-join primitive, and remains unchanged.




```
void foo(int lo, int hi) {
 if (hi-lo <= 1) { ...; return; }
 ...; int mid = lo + (hi-lo)/2;
 pardo([&]{ foo(lo, mid); },
       [&]{ foo(mid, hi); });
}
```

```
V v6 = augment([&]{
  // v0 implicitly here
  foo(0,3);
  // now at v6
});
```

Figure 3: Scheduler augmentation example, showing an execution of `foo(0,3)`. All omitted code is sequential (no calls to `pardo`). Dashed boxes correspond to nested calls to `foo()`.

they may or may not be equal to `vl` and `vr`, depending on whether or not the children performed any nested calls to `pardo`.

The scheduler implicitly maintains and updates vertices, guaranteeing that all user code executes "at" some current vertex `v` sandwiched between calls to `v.start()` and `v.stop()`. This facilitates lightweight profiling, for example, by starting and stopping a timer for each vertex. The callbacks to `v.fork()` and `v.join()` can be used by the programmer to track the evolution of the computation graph during execution, which has a number of interesting use-cases that we develop further in Sections 2.1, 3, and 4. Often, all augmentation data can be efficiently updated only at forks and joins, allowing for this data to remain local to vertices and resulting in low cost for a wide variety of augmentations. We also include the `augment` utility function which the programmer may wrap around a region of interest. This initializes a fresh vertex before executing the region and returns the final vertex when the region finishes execution. Within the region, the programmer may also use `getCurrentVtx()` to access vertices at runtime.

Example. Figure 3 illustrates scheduler augmentation through a small example function `foo(lo,hi)` which finds a midpoint `mid` between `lo` and `hi`, and recursively executes `foo(lo,mid)` and `foo(mid,hi)` in parallel, with an appropriate base case. The example calls `foo(0,3)`, resulting in two splits, first at `mid=1` and then (on the right-hand side) at `mid=2`. Altogether, executing `foo(0,3)` results in 7 vertices, labeled `v0`-`v6`, as shown in Figure 3. The scheduler automatically creates these vertices and calls the corresponding `start`, `stop`, `fork`, and `join` methods throughout execution.

2.1 Use-Cases

Scheduler augmentation has a wide variety of potential applications, each of which can be implemented by providing a custom definition of the vertex type and associated methods. We list a few of these applications here, and we note that many other use-cases are possible as well. We consider two applications in detail in Section 3 and Section 4, and we leave the exploration of other applications to future work.

Vertex Labelling. Vertex labelling is one of the core components of dag-based analyses. Many vertex labelling strategies can be implemented with scheduler augmentation, with local updates to the labels at forks and joins. For example, offset-span labels [24] can be implemented by maintaining a list L of (o, s) pairs at each vertex. Similarly, the pedigree scheme of Leiserson et al. [22] can be implemented with a list of rank counters at each vertex, and DePa labels [43] can be implemented with a depth counter and a bit-vector at each vertex.

Computation Graph Visualization. Computation graph visualization, for example as utilized by Spoonhower et al. [34] and Muller [25], can be implemented by emitting a trace of the graph edges during execution. The graph edges can be specified locally at forks and joins using one of the vertex labelling strategies described above.

Source-Level Profiling of Work and Span. Lightweight profiling for work and span (as well as derived metrics such as parallelism) can be accomplished by associating a source-level timer with each vertex as well as the running total work and span, performing the appropriate sums and maxes at joins.

Burdened Span Analysis. The *burdened dag model* of He et al. [17] can be implemented similarly to work and span profiling as sketched above. At each join-point, the burdened span can be calculated as $s_{\text{join}} = s_p + \max(s_l, s_r) + B$ where s_p is the span of the parent, s_l and s_r the spans of the left and right children, and B is the chosen burden constant. We note that in comparison to instruction-level work and span profiling, this approach is more coarse-grained and less precise because it relies only on source-level timers. However, the results can nevertheless be quite useful in practice.

```
class GrainAnalysisVertex {
  double w = 0.0;  int64_t f = 0;  time_t t;  int p;
  void setPhase(int phase){ p = phase; }
  void start() { t = now(); }
  void stop() { w += (now() - t); }
  void fork(V* l, V* r) { f++; l->p = r->p = p; }
  void join(V* l, V* r, V* c) {
    c->p = p;
    c->w = w + l->w + r->w;
    c->f = f + l->f + r->f;
    if (c->w > THRESHOLD) writeLog(p, c->w, c->f);
  }
}
```

Figure 4: Augmentation to perform granularity analysis by tracking the work (w) and number of forks (f) in every sub-dag. The "phase" identifier (p) is useful for our case study and is discussed in Section 3.1.

3 Application: Granularity Analysis

We present a granularity analysis technique that leverages scheduler augmentation to identify subcomputations that are too fine-grained and thus could benefit from coarsening or pruning. Identifying such subcomputations can be difficult because they may be deeply buried within a complex codebase, for example, hidden within a library that the programmer did not author. Our technique provides a way to quickly discover these subcomputations, without any major changes to the code and with minimal effort by the programmer.

Classifying sub-dags by w/f. The key idea behind this granularity analysis technique is to collect information about every sub-dag of the computation graph, and classify these sub-dags by how coarse-grained (or fine-grained) they are. In particular, we propose classifying sub-dags by the ratio w/f, where w is the amount of work performed within the sub-dag, and f is the number of forks within the sub-dag. Intuitively, this ratio can then be compared against the (known) cost of a single call to `pardo`, which we call τ.[2] If $w/f \gg \tau$, then the sub-dag is sufficiently granularity-controlled; if $w/f \approx \tau$ (say, within the same order-of-magnitude), then the sub-dag is too fine-grained.

Calculating w and f for each sub-dag using scheduler augmentation is relatively straightforward, as shown in Figure 4. We associate with each vertex four fields: a work counter w, a fork counter f, a timestamp t, and a "phase" identifier p (which is useful for our case study and is discussed in Section 3.1, below). Each vertex tracks its own work using a timer in the `start` and `stop` methods, and updates the w and f counters appropriately at forks and joins. At each join point, we calculate the ratio w/f and append this information to a log, if the ratio exceeds some threshold (to amortize the cost of logging itself). After the computation finishes, we can then analyze the log to identify sub-dags that have small w/f ratios, and are thus too fine-grained.

[2] On our machine, we measure $\tau \approx 50$ns. See Section 6.3 for details.

3.1 Case Study: Parallel Range Query

To demonstrate the effectiveness of this technique, we apply it to the `rangeQuery2d/parallelPlaneSweep` benchmark from the Problem-Based Benchmark Suite [4]. Our investigation of this benchmark was motivated by an observation that the performance of the benchmark appeared to be highly sensitive to small changes to the scheduler, which hints at a granularity control issue. As part of this process, we developed a visualization technique to help analyze the results of the granularity analysis, which we present here.

The PBBS parallel plane sweep benchmark. The high-level structure of the algorithm consists of three phases. (1) First, it builds one range-tree at each x-coordinate by sweeping across the points. The range trees are provided by the PAM library [36]. (2) Second, it performs the queries in parallel (specifically by binary searching to identify the relevant left and right range trees, splitting these trees at the top and bottom query range, and taking the difference in sizes). (3) Finally, it deallocates all of the range trees, which triggers a reference-counting garbage collector within PAM. The benchmark waits for the GC to finish before returning.

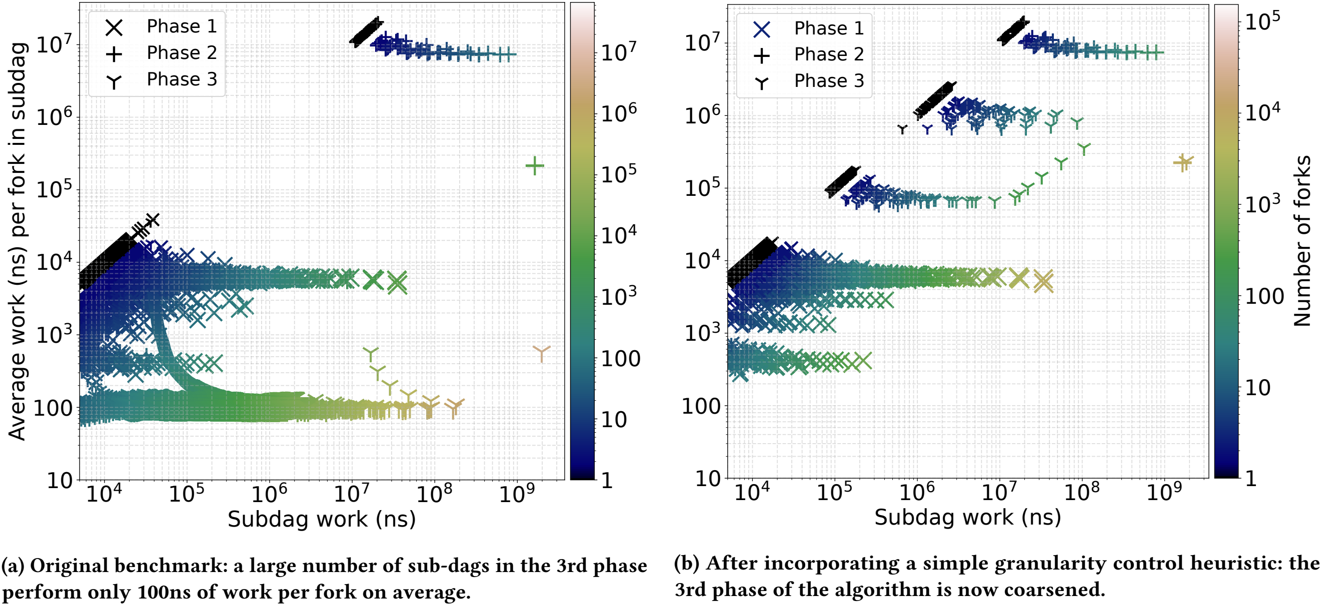
Observing a potential granularity control issue. In our experiments, we noticed that this benchmark was highly sensitive to the cost of the `pardo` function. In particular, artificially inflating the cost of `pardo` by a small amount caused a significant slowdown ($\approx 20\%$) in the overall runtime. This led us to suspect that the benchmark contains a significant subcomputation that is too fine-grained. However, it is not immediately obvious where to look to find the offending subcomputation. Most of the complexity of the benchmark is offloaded to the PAM library, which handles parallel traversals and manipulations of the range tree data structures.

Applying granularity analysis. To investigate, we applied our granularity analysis technique, which in this case consists of (1) augmenting the scheduler with the vertex definitions shown in Figure 4, (2) adding three lines of code to the benchmark source, calling `getCurrentVtx->setPhase()` immediately before the start of each phase of the algorithm, to help track which phase each vertex appears within, and finally (3) compiling and running the benchmark. This produces a trace of entries (p, w, f), indicating that in phase p of the algorithm there was a sub-dag with work w and number of forks f.

To analyze the resulting trace, we can plot the entries (p, w, f) on a scatter plot, as shown in Figure 5a. Here, we plot each sub-dag at position $(w, w/f)$, where w and f are (respectively) work and number of forks performed within the sub-dag. Such a plot has the following characteristics:

- **The granularity of each sub-dag** is represented by its y-coordinate: higher is coarser, and lower is finer.
- **The impact of each sub-dag** on the overall runtime of the computation is represented by its x-coordinate: larger is more impactful, smaller is less impactful.

Immediately, we observe a cluster (near the bottom of the plot) of sub-dags that are very fine-grained, performing only approximately 100ns of work per fork on average. In comparison to the cost of a fork ($\tau \approx 50$ns), this ratio of $w/f \approx 100$ns suggests that



(a) Original benchmark: a large number of sub-dags in the 3rd phase perform only 100ns of work per fork on average.

(b) After incorporating a simple granularity control heuristic: the 3rd phase of the algorithm is now coarsened.

Figure 5: Using scheduler augmentation for granularity analysis of every sub-dag in the PBBS `rangeQuery2d/parallelPlaneSweep` benchmark on 1M points. We measure the work w and number of forks f within each sub-dag, and plot it at $(w, w/f)$.

as much as 1/3 of the total cost of this portion of the benchmark can be attributed to the overhead of task creation, management, and synchronization. In other words, the benchmark contains a significant subcomputation that is too fine-grained. By tracking the phase of each vertex, we can easily see that all of the fine-grained sub-dags appear to be located somewhere within the 3rd phase of the algorithm. This analysis immediately directs us towards the PAM GC implementation.

Investigating and optimizing the PAM GC. Looking at the PAM GC call in the benchmark source leads to a recursive function in the PAM source code, called `decrement_recursive`. This function deallocates a PAM tree node by decrementing an associated reference count; if the reference count hits zero, it frees the node and then recursively deallocates the node's two children in parallel (using `pardo`). To control for granularity, PAM uses a heuristic: the recursive calls are executed in parallel only if the size (i.e., number of nodes within) the left[3] child is above some constant threshold. This constant threshold guarantees that the cost of the parallelism is amortized only if the reference counts of the descendant nodes are all 1. If any descendant node has a reference count larger than 1, the recursive call at that node will only perform a single decrement and then immediately exit, which is not enough work to amortize the call to `pardo`. To address this, we adjusted the heuristic to check the reference counts of the children and grandchildren of the current node, and only execute the recursive calls in parallel if all of these reference counts are 1. This guarantees that each call to `pardo` is amortized against at least a modest amount of work, to process the immediate neighborhood below the current node.

[3]PAM trees are balanced, so the choice of left versus right is arbitrary.

Confirming the optimization. After applying the optimization described above, we then reran our granularity analysis. The resulting plot is shown in Figure 5b. We can see that the cluster of fine-grained sub-dags has been completely eliminated, and the new w/f ratio for the 3rd phase of the algorithm has increased substantially, up to $100\mu s$ or more, which is plenty of work to amortize the cost of the parallelism. This optimized version of the benchmark appears in our performance evaluation (Section 6).

4 Application: Space Profiling

We present a space profiling technique based on scheduler augmentation that can predict the worst-case peak memory consumption of a parallel execution, regardless of the number of processors. The idea is to identify the worst-case prefix of the computation graph that maximizes the amount of memory allocated but not yet freed. As we will show here, this can be computed on-the-fly using $O(1)$ operations at each vertex of the computation graph.

Space Graph. Our space profiling technique is based on a ***space graph***, which subdivides each vertex of the computation graph into a chain of one or more ***space vertices*** and annotates each space vertex with the amount of memory allocated or freed at that vertex. In the space graph, we can identify a prefix (i.e., a valid partial schedule) of the graph that maximizes the memory allocated but not yet freed by the (space) vertices in the prefix. This prefix is essentially a snapshot of the computation in the worst possible schedule (in terms of space usage). We define S_∞ to be the total amount of memory in use by this worst-case prefix, and refer to it as the ***worst-case parallel high-water mark***. Intuitively, S_∞ is the maximum memory consumption on an arbitrary (possibly infinite) number of processors.

```
class SpaceVertex {
  int64_t δ = 0;  uint64_t sinf = 0;
  void join(V* vl, V* vr, V* vc) {
    vc->sinf = max(sinf, δ + vl->sinf + vr->sinf);
    vc->δ = δ + vl->δ + vr->δ;
  }
  void alloc(size_t sz) {
    δ += sz;
    sinf = max(sinf, δ);
  }
  void dealloc(size_t sz) { δ -= sz; }
}
```

Figure 6: Augmentation to track space usage in terms of S_∞.

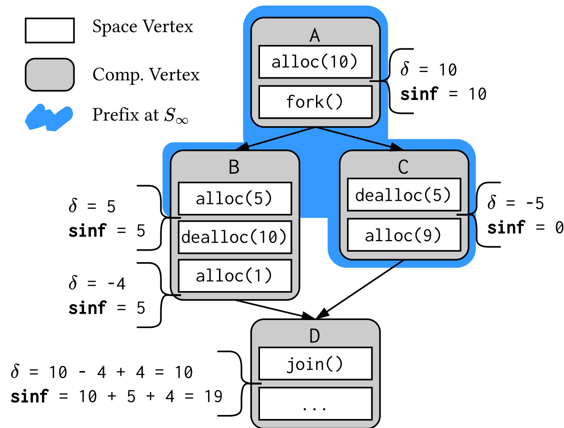


Figure 7: Example space graph with S_∞ calculation.

Using scheduler augmentation combined with instrumentation of memory allocations and deallocations, we can simulate the space graph. In particular, we add vertex functions (named `alloc` and `dealloc` in Figure 6) and assume that the memory allocator calls these functions whenever memory is allocated or freed.

Figure 6 gives the specific definitions needed to track S_∞. The δ field maintains the net memory allocation of the vertex so far. This value can be negative as vertices may free memory that was allocated by another vertex. The `sinf` field holds the S_∞ value of the immediately enclosing sub-dag. At joins, we add the local S_∞ of each subgraph and the net allocations prior to the fork, updating S_∞ if the calculated sum is greater than the previous S_∞. This corresponds to identifying the worst-case prefix on the left and right independently and taking their sum.

Example. An example of the space graph and corresponding space profile is shown in Figure 7. Computation vertices (rounded, gray boxes) are subdivided into one or more space vertices (white boxes), and each space vertex is annotated with some number of bytes allocated or deallocated. Vertex `A` heap allocates 10 bytes of memory before forking subtask `B` and `C`. Vertex `B` and `C` start with $\delta = 0$ and `sinf` $= 0$. Each vertex updates its values on-the-fly. In the case of `B`, the δ changes at every allocation/deallocation, but its `sinf` value changes only once at `alloc(5)`. At the join, `D` combines the values from `A`, `B`, `C`. For δ, `D` simply sums the δ values from every vertex and for `sinf`, it sums δ(`A`) and the `sinf`s of `B` and `C`.

Formalization. We have formalized the relationship between the space graph and computation graph and have proven that the `SpaceVertex` definitions correctly track S_∞ (Theorem 4.1, below). For brevity, we only present a sketch of the proof here, and the full proof is given in the Appendix.The proof inducts over the structure of the series-parallel computation graph. We first establish a closed-form formula for S_∞ in each of the three structural cases: singleton graphs, parallel compositions, and series compositions. For instance, in a parallel composition with source u, sink v, and children L, R,

$$S_\infty(G) = \max(h(u), \delta(u) + S_\infty(L) + S_\infty(R), \delta(u) + \Delta L + \Delta R + h(v))$$

where $h(\cdot)$ is the local high-water mark (max prefix sum of a single vertex's allocations), $\delta(v)$ is the net memory allocated by a computation vertex v, and ΔH denotes the sum of the net memory allocated across all vertices in a subgraph. The three cases correspond to the worst-case prefix terminating within u, within the children, or within v. The inductive step then shows that the `alloc`, `dealloc`, and `join` methods maintain the invariant that each vertex's `delta`, and `sinf` fields match ΔG and $S_\infty(G)$ upon completion of the enclosing subgraph.

Theorem 4.1. *Let G be a fork-join computation graph and v be its sink vertex, augmented with* `SpaceVertex` *definitions. The vertex v satisfies v*`.delta` $= \Delta G$ *and v*`.sinf` $= S_\infty(G)$.

```
1  seq<int> qh(seq<Point> &pts, seq<int> idxs,
2             Point l, Point r) {
3    int midIdx = findMid(l, r); Point mid = pts[midIdx];
4    seq<int> lIdxs = filter(idxs, aboveLine(l, mid));
5    seq<int> rIdxs = filter(idxs, aboveLine(mid, r));
6    idxs.clear();
7    seq<int> lRec, rRec;
8    pardo([&]{lRec = qh(pts, std::move(lIdxs), l, mid);},
9          [&]{rRec = qh(pts, std::move(rIdxs), mid, r);});
10   seq<seq<int>> nested = {lRec, intseq(1, midIdx), rRec};
11   return flatten(nested);
12 }
```

Figure 8: ParlayLib's quickhull implementation. We study the space impact of the highlighted line in Section 4.1.

4.1 Case Study: Quickhull

We employ space profiling to evaluate a subtle space optimization in ParlayLib's quickhull implementation. We specifically isolate and analyze intermediate allocations made within the subroutine for finding the upper hull.

Quickhull is a divide-and-conquer algorithm used to find the convex semihull of a set of points. ParlayLib provides a fairly standard C++ implementation of quickhull (Figure 8). It takes as input an array of points, a line (given by points `l` and `r`), and an array containing the indices of the points above the line. The algorithm first identifies a `mid`, the point furthest from the line to act as a new vertex in the hull. Then, it filters the indices array and produces two new arrays: the indices of the points above the line from `l` to `mid` and the indices of the points above the line from `mid` to `r`. It calls itself recursively in parallel via `pardo` and combines the results.

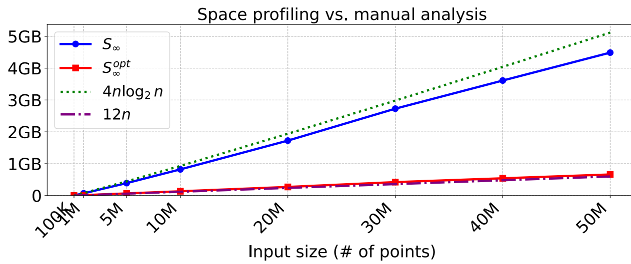


Figure 9: Comparing the S_∞ values calculated by space profiling against S_∞ as predicted by manual analysis. Solid lines represent results from space profiling.

This implementation includes a subtle but crucial space optimization: clearing the `idxs` sequence *before* the recursive calls (line 6). Without this operation, the `qh` function incurs $O(n \log n)$ worst-case HWM where n is the size of the `idxs` sequence. To understand why, note how without line 6, the `idxs` sequence would persist during the recursive calls. In the pathological case where all input points are on the hull, the recurrence formula for the worst-case HWM of `qh` becomes $S_\infty(n) = Cn + 2S\left(\frac{n-1}{2}\right)$. Accounting for $C = 4$ bytes per index, this comes out to $S_\infty(n) \approx 4n \log_2 n$ (ignoring smaller order terms).

Dropping the `idxs` sequence with `clear` before the `pardo` removes the Cn term from the recurrence, and shifts the peak memory footprint to after the call to `pardo`, resulting in a worst-case HWM that is linear in the size of the input. A rough analysis yields $12m$ bytes, where m is the size of the hull: `lRec` and `rRec` together take up $4m$ bytes, `nested` deeply copies `lRec` and `rRec` ($+4m$), and the return value takes up another $4m$ bytes. In the case that all input points are part of the hull, ($m = n$), so we expect $S_\infty^{opt}(n) \approx 12n$.

Confirming the Optimization. To verify whether the optimization actually achieves its stated goal, we run two versions of ParlayLib's quickhull algorithm[4] — one optimized and one de-optimized — with the `SpaceVertex` augmentation. To induce an execution where all points end up on the hull, the input points are placed uniformly on a unit circle. We use 8 different input sizes (100K–50M points) and run the experiments with the settings described in Section 6.1.

Figure 9 compares the S_∞ values calculated through augmentation against the S_∞ values we derived by hand above ($4n \log n$, $12n$). In both cases, we observe that space profiling matches our analysis very closely. This strongly suggests that the optimization indeed reduces S_∞ from $O(n \log n)$ to linear, at least for hull-heavy inputs. As a cherry on top, we report that the optimization translates to more efficient memory use in practice; on $P = 80$ processors, the observed high-water mark of quickhull at 50M points decreased significantly from ~740MB to ~350MB.

5 Implementation

We present our implementation of scheduler augmentation, based on ParlayLib's work-stealing scheduler. We first describe how to extend the work-stealing algorithm, and then present details of our specific changes to ParlayLib.

[4]We patch a bug in the original that led to duplicated points in the hull.

Background. In work-stealing, computations run on a set of ***workers***, each with a local double-ended task queue (***deque***) whose ends are referred to as the *top* and *bottom*. At every moment, each worker is either working on a ***current task***, or it is ***stealing*** (and itself has an empty deque). When stealing, the worker repeatedly attempts to steal a task from the top of other workers' deques.

Whenever a worker's current task executes a fork, the worker creates the two child tasks, pushes one onto its deque, assigns the other as its current task, and begins executing it.

Whenever a worker completes a task, it checks the status of the sibling. If the sibling is at the bottom of the worker's own deque, the worker pops it and executes it. If the sibling is not at the bottom of the deque, there are two cases: (1) the sibling has already been completed, or (2) the sibling is currently being executed by another worker. In the first case, the worker proceeds with the task corresponding to the computation after the join-point. In the second case, the worker switches to stealing. Note that in the second case, the worker's deque must be empty. This upholds the invariant that stealing workers always have empty deques.

5.1 Augmented Work-Stealing

To incorporate scheduler augmentation into work-stealing, it is necessary to keep track of the vertices of the computation graph and how they map onto tasks. To do so, we equip each task with a ***latest vertex***. We say that the ***current vertex*** of a worker is the latest vertex of its current task. The augmented work-stealing algorithm proceeds as follows:

Fork: At current vertex `v`, if the worker encounters a fork, it first calls `v->stop()`, and then executes `v->fork(l, r)` with fresh vertices `l` and `r`. It creates a new task for the left and assigns `l` as its latest vertex, and similarly for the right with `r`. It pushes the right task to the bottom of its deque, and begins executing the left task.

Completing a task: At current vertex `v`, if the worker completes a task, it first calls `v->stop()` to mark the end of the current vertex. It then checks the status of the sibling. If the sibling has already completed, the worker proceeds with a Join (see below). If the sibling is at the bottom of the worker's own deque, the worker pops it, calls `start()` on the sibling's latest vertex, sets the sibling as its current task, and begins executing the task. Otherwise, the worker begins stealing.

Join: At current vertex `v` with a completed sibling vertex `u`, the worker identifies the latest vertex `p` of the parent task. The worker executes `p->join(v, u, p')` with a fresh vertex `p'`, sets the latest vertex of the parent task to `p'`, calls `p'->start()`, and resumes the parent task (executing the computation after the join).

Steal: When a worker steals a task, it sets its current vertex to the latest vertex `v` of the acquired task, executes `v->start()`, and then begins executing the task.

5.2 Extending ParlayLib

We now walk through how we concretely implement scheduler augmentation in ParlayLib, with details shown in Figure 10. Our implementation follows the high-level design described in Section 5.1, and integrates these into the `pardo` function of ParlayLib.

We track each worker's current vertex by a thread local vertex pointer (line 1), since ParlayLib allots exactly one `pthread` per

```
1 static thread_local Vertex* currV;
2 template<typename L, typename R>
3 void pardo(L&& lf, R&& rf) {
4   currV->stop();
5   Vertex* parentV = currV;
6   Vertex leftV, rightV;
7   WorkStealingJob rightTask = makeJob(rf, &rightV);
8   parentV->fork(&leftV, &rightV);
9   deques[myWorkerId()].pushBot(&rightTask);
10  currV = &leftV; leftV.start();
11  lf(); // subtasks may overwrite leftV/rightV
12  leftV.stop();
13  // check if job was stolen
14  if (deques[myWorkerId()].tryPopBot()) {
15    // unstolen case
16    currV = &rightV; rightV.start();
17    rf();
18    rightV.stop();
19  }
20  else {
21    // stolen case
22    waitUntil(rightTask.finished());
23  }
24  Vertex afterV;
25  parentV->join(&leftV, &rightV, &afterV);
26  *parentV = std::move(afterV);
27  currV = parentV; currV->start();
28 }
```

Figure 10: The `parlay::pardo` function with augmentation. Our modifications highlighted.

worker. Additionally, we add a vertex pointer field to ParlayLib's task object (`WorkStealingJob`) for storing its latest vertex.

The vertices themselves reside in the stack of `pardo`. At a fork, we create the initial vertices for the left and right subtasks on the stack of the parent task (the caller of `pardo`). The subtasks then reuse this address when updating its latest vertex. We see this in the synchronization process. First, a new `afterV` vertex is created (line 24). After calling `join` on `parentV`, we `std::move afterV` into the address of `parentV`, effectively overwriting it (line 26).

The safety of this implementation hinges on two implementation details of ParlayLib. Contrary to other frameworks, ParlayLib does not expose fork and join primitives, instead handling them both inside the `pardo` function. This ensures the stack frame of `pardo` strictly outlives the child tasks and makes their vertices safe to allocate on the stack. The root vertex is the only exception, as it is created at program initialization (and hence before `pardo`). So, we manually create and free the root vertex at scheduler construction and destruction.

6 Evaluation

In this section, we address the question: *What is the performance impact of scheduler augmentation?* To answer this question, we compare our augmented scheduler (Section 5) against the original, unmodified ParlayLib scheduler on 30 benchmarks from the

```
class EvaluationVertex {
  double w = 0.0, s = 0.0; int64_t f = 0; time_t t;
  void start() { t = now(); }
  void stop() { double e = now()-t; w += e; s += e; }
  void fork(V* l, V* r) { f++; }
  void join(V* l, V* r, V* c) {
    c->w = w + l->w + r->w;
    c->s = s + max(l->s, r->s);
    c->f = f + l->f + r->f;
  }
}
```

Figure 11: A representative lightweight augmentation measuring work (w), span (s) and number of forks (f), used throughout the evaluation in Section 6.

Problem-Based Benchmark Suite (PBBS) [4] and ParlayLib [8]. We focus on a representative vertex definition which is designed to resemble typical (lightweight) augmentations, and measure the overhead of augmentation across a range of processor counts.

The results demonstrate that the augmented scheduler has low overhead, with an average of 2.6% (geomean) overhead due to augmentation across all benchmarks and processor counts. Our experiments also demonstrate that augmentation has little impact on scalability, with overheads staying relatively constant across processor counts for almost all benchmarks.

6.1 Experimental Setup

We performed all our experiments on an 80-core x86_64 Chameleon bare metal instance [20] with two 2.30GHz Intel Xeon (40-core) Platinum 8380 CPUs with 256 GiB of RAM running Ubuntu 24.04.2 LTS. We disabled hyperthreading for all the experiments and use the terms core and processor interchangeably. All the benchmarks were compiled with GCC 13.3.0 and run with `numactl -i all`. We use ParlayLib's default memory configuration and disable ParlayLib's elasticity feature. To measure timings, we run the benchmarks in a loop back-to-back, excluding 3s of warmup runs and then take the average of 20 runs. We exclude loading or generating the input from measurement.

We perform our evaluation on 22 PBBS benchmarks and 8 ParlayLib benchmarks, covering a range of problem domains. We used the input generators provided by both benchmark suites, in particular using the R-MAT model for graph inputs, the Kuzmin distribution for 2D point sets, and uniformly distributed inputs for 1D sequence-based benchmarks, unless indicated otherwise. Full details are provided in the Appendix.

Vertex Configuration. Throughout this section we use the vertex definition shown in Figure 11. This vertex is designed to resemble typical "lightweight" augmentations: each vertex is on the order of tens of bytes (specifically 32 here), each vertex method executes a few arithmetic operations, and each start and stop queries the system clock (`std::chrono::high_resolution_clock`). We include the timers because these are especially useful for profiling, and we imagine many augmentations will require them. In this section, we do not focus on the specific data collected by this vertex definition,

Suite	Short Name	Benchmark (Input Size)	P=1		P=40		P=80	
			Unaug (s)	Aug (%)	Unaug (s)	Aug (%)	Unaug (s)	Aug (%)
PBBS	intSort	integerSort/parallelRadixSort (n=100M)	1.4575	+8.42%	0.0348	+1.28%	0.0249	+1.27%
	cmpSort	comparisonSort/sampleSort (n=100M)	6.7736	+0.48%	0.1719	+1.49%	0.0952	+1.30%
	rmDup	removeDuplicates/parlayhash (n=100M)	2.5617	-0.72%	0.0712	-1.01%	0.0449	+0.33%
	hist	histogram/parallel (n=100M)	1.2248	+0.88%	0.0361	-0.04%	0.0241	+2.33%
	wordCnt	wordCounts/histogram (n=250M)	5.6066	-1.04%	0.1462	+1.07%	0.0772	+0.97%
	invIdx	invertedIndex/parallel (n=250M)	8.2371	+1.23%	0.2306	+1.90%	0.1277	+1.56%
	suffArr	suffixArray/parallelRange (n=34M)	9.6545	+1.41%	0.2990	+1.33%	0.1852	+1.60%
	lrs	longestRepeatedSubstring/doubling (n=34M)	12.1652	+4.12%	0.3838	+3.88%	0.2418	+3.45%
	classDT	classify/decisionTree (covtype.data)	7.2779	+0.07%	0.3544	+1.91%	0.2237	+2.19%
	msf	minSpanningForest/parallelFilterKruskal (V=16M, E=192M)	17.8063	+1.83%	0.6324	+4.75%	0.4338	+9.23%
	sf	spanningForest/ndST (V=16M, E=192M)	4.6902	-9.39%	0.1860	-1.90%	0.1302	+0.32%
	bfs	breadthFirstSearch/backForwardBFS (V=16M, E=192M)	1.9045	-0.13%	0.0582	+1.15%	0.0318	+0.56%
	maxMatch	maximalMatching/incrementalMatching (V=20M, E=200M)	6.7890	+0.66%	0.2440	+1.42%	0.1447	+1.52%
	mis	maximalIndependentSet/incrementalMIS (V=16M, E=192M)	0.8441	-0.48%	0.0319	+5.42%	0.0212	+7.22%
	nn	nearestNeighbors/octTree (n=10M)	5.5899	+2.30%	0.1768	+1.82%	0.1026	+2.62%
	rayCast	rayCast/kdTree (n=1,087,716)	7.1378	+2.39%	0.2568	+2.79%	0.1498	+3.23%
	cvxHull	convexHull/quickHull (n=100M)	1.3678	+8.14%	0.0529	+4.82%	0.0423	+3.15%
	delTri	delaunayTriangulation/incrementalDelaunay (n=10M)	41.6850	-1.75%	1.5513	+5.45%	0.9849	+4.40%
	delRef	delaunayRefine/incrementalRefine (n=5M)	40.0717	+3.45%	1.7021	+6.11%	1.1754	+6.50%
	nBody	nBody/parallelCK (n=1M)	21.0450	-0.08%	0.5952	+0.21%	0.3374	+0.15%
	rq2d	rangeQuery2d/parallelPlaneSweep (n=10M)	34.2357	+22.31%	1.5487	+13.71%	1.2133	+8.01%
	rq2d_ours	** rangeQuery2d/parallelPlaneSweep_ours (n=10M)	33.0361	-1.27%	1.5036	+0.11%	1.1886	-0.73%
ParlayLib	ldd	graph/low_diameter_decomposition (V=10M, E=200M)	1.0845	-0.17%	0.0427	+2.19%	0.0277	+1.93%
	triCnt	graph/triangle_count (V=800K, E=16M)	1.9667	+10.48%	0.0554	+10.30%	0.0298	+12.10%
	karatsuba	numerical/karatsuba (n=10M)	1.2285	+2.93%	0.0333	+3.10%	0.0185	+2.79%
	mcss	numerical/mcss (n=1000M)	1.1716	+0.09%	0.0328	+0.06%	0.0224	-0.11%
	primes	numerical/primes (n=800M)	3.5518	-5.49%	0.0983	-6.30%	0.0604	-6.27%
	mergeSort	sorting/mergesort (n=10M)	1.2148	+6.03%	0.0323	+6.96%	0.0198	+3.25%
	quickSort	sorting/quicksort (n=10M)	1.5726	+1.24%	0.0625	+7.64%	0.0380	+5.44%
	huffman	string/huffman_tree (n=3M)	1.0168	+1.87%	0.0318	+2.35%	0.0195	+2.03%
Geometric Mean (All 30 Benchmarks)				**+1.86%**		**+2.73%**		**+2.70%**

Table 1: Overhead of augmentation with the `EvaluationVertex` for $P = 1, 40$ and 80. Execution times in seconds. The %-differences are relative to the unaugmented ParlayLib. The `rq2d_ours` benchmark is the `rq2d` with our modifications discussed in Section 6.2.

and instead focus on the performance characteristics of the augmented scheduler under this definition. However, we do note that this vertex measures work, span, and the number of forks.

6.2 Overhead and Scalability

In this section, we investigate the overheads and scalability of scheduler augmentation. Table 1 captures the execution time overhead introduced by augmentation compared to the unaugmented ParlayLib scheduler across our two benchmark suites. In Figure 12, we plot the overhead ratio (augmented vs. unaugmented execution time) across the full range of processor counts ($P = 1$ to 80).

Overhead of Augmentation. We observe an average (geomean) overhead of less than 3% across all benchmarks and processor counts and 27 out of the 30 benchmarks exhibit less than 10% overhead across all processor counts. Only one benchmark, `rq2d`, exceeds a threshold of 15%. This is due to granularity control as shown in Section 3.1, and we discuss its performance in more detail below. We consider these overheads to be acceptable, especially because we envision scheduler augmentation as primarily an analysis technique (as opposed to an "always-on" technique for production use).

Impact on Scalability. Examining the plots in Figure 12, a "flat" curve indicates that augmentation incurs a consistent overhead across all processor counts — in other words, the augmentation matches the scalability of the baseline. The majority of benchmarks exhibit a flat overhead profile, but we do bring attention to the few outliers and explain their behavior. The `msf` and `mis` benchmarks exhibit a noticeable increase in overhead as P increases. This increase in overhead is primarily due to adaptive parallelism strategies within the benchmarks themselves, which adjust their number of spawned tasks based on P. For example, in `mis`, the total number of forks increases by an order of magnitude between sequential and parallel runs, from 46,927 at $P = 1$ to 582,663 at $P = 80$. In contrast, benchmarks with deterministic task decomposition strategies exhibit flatter overhead profiles, as the number of forks (and therefore the overheads of augmentation) stay constant across processor counts. For example, the `karatsuba` benchmark follows a deterministic divide-and-conquer task decomposition, and has almost perfectly flat overhead profile; we note that, in this benchmark, we have measured that the number of forks stays nearly constant across processor counts.

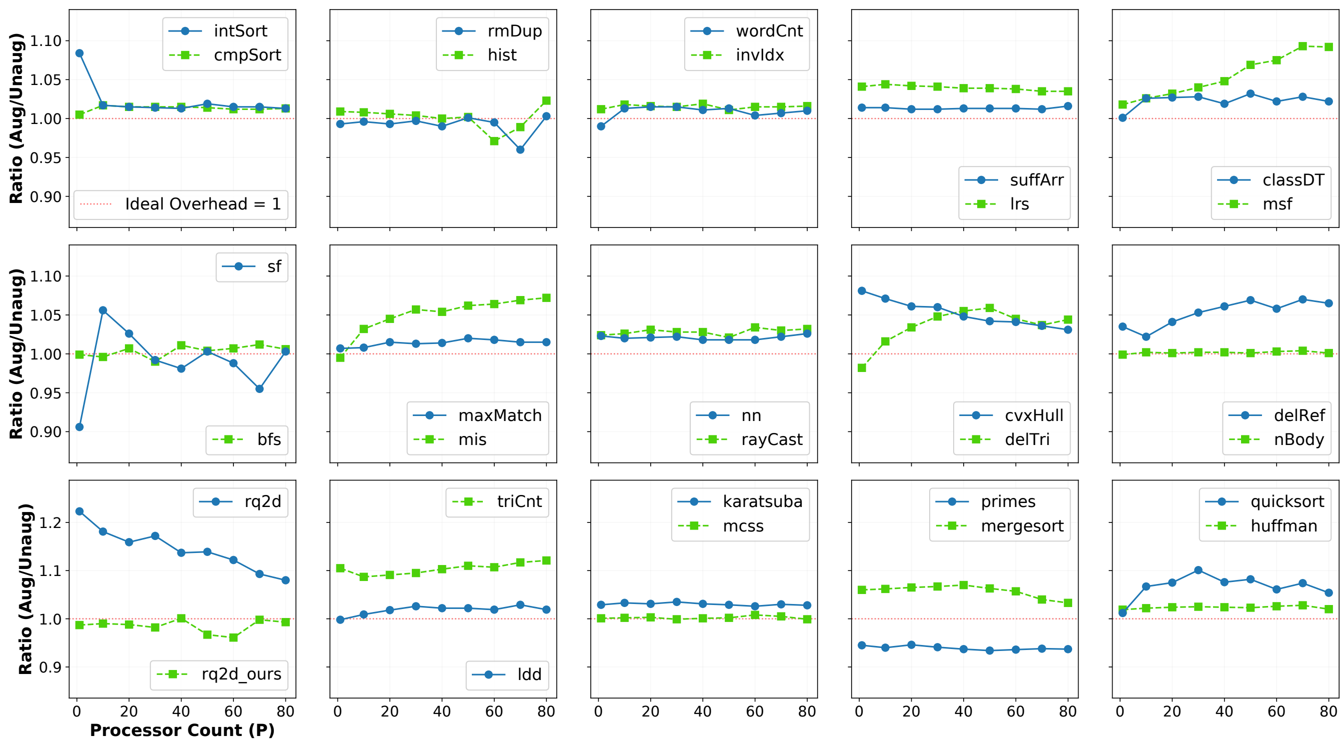


Figure 12: Overhead of augmentation with the `EvaluationVertex` relative to the original, unaugmented ParlayLib. The x-axis ranges from $P = 1$ to 80 at increments of 10. The y-axis scale is consistent within each row.

Re-tuning granularity in `rq2d`. All benchmarks we consider utilize granularity control parameters that were tuned (by the original benchmark authors) to amortize the overhead of the original, unaugmented scheduler. Typically, such granularity tuning is a manual minimization process: the smallest possible grain size is chosen which still amortizes the overhead of scheduling to ensure the maximum amount of parallelism is still available. This process naturally overfits to a particular scheduler, and can make the performance of a benchmark sensitive to any change in the underlying scheduler.

The `rq2d` benchmark in particular exhibits high sensitivity to the overheads introduced by augmentation, with as much as a 22.3% penalty at $P = 1$. This benchmark was the focus of our granularity analysis case study (Section 3.1), which details the specific issue and how we addressed it. The re-tuned benchmark is present in our evaluation, named `rq2d_ours`, and exhibits significantly lower overheads across all processor counts without any loss in raw measured performance. In fact, the absolute timings in Table 1 reveal that `rq2d_ours` is slightly faster than `rq2d` at all processor counts under both schedulers.

That being said, it is interesting to note that on the original `rq2d` benchmark, the overheads of augmentation steadily decrease as the processor count increases, indicating that the overhead of augmentation itself is highly parallelizable. In particular, `rq2d` is primarily memory-bound, with only 28x self-speedup at $P = 80$; as the benchmark approaches a memory wall on higher core counts, the relative overhead of augmentation decreases.

6.3 Predicting the Cost of Augmentation

In this section, we demonstrate that it is possible to predict the overhead of augmentation (at least for augmentations with $O(1)$ cost for each operation) through a simple experiment. Our findings show that (a) the unaugmented cost of a `pardo` is approximately 50ns, and that (b) the `EvaluationVertex` adds an overhead of approximately 250ns per `pardo`. By counting the number of forks within a particular benchmark, we can then estimate the overhead of augmentation.

To measure the cost of `pardo`, we use a small synthetic benchmark which calls `pardo` recursively up to a configurable depth D, essentially traversing a full binary tree of tasks where the tasks themselves perform nearly zero work. The total number of executed `pardo`s is $2^D - 1$, and the cost of a single `pardo` can therefore be estimated by dividing the total execution time by this amount. Running this experiment with the unaugmented scheduler yields a measurement of approximately 50ns per `pardo`. With the augmented scheduler (using the vertex in Figure 11), we measure a cost of approximately 300ns per `pardo`.

Given the cost of augmentation per `pardo`, we can predict the overhead of a particular benchmark by multiplying the number of `pardo`s in the benchmark by the overhead per `pardo`. For example, the `karatsuba` benchmark executes 189,394 `pardo`s, and therefore we can predict an overhead of approximately 47ms (= 189,394 × 250ns) due to augmentation. Relative to the unaugmented execution time of 1.23s at $P = 1$, this offers a prediction of 3.8% overhead

due to augmentation, and our empirical measurements confirm an overhead of approximately 3%.

7 Related Work

Profiling Parallel Programs. One application of scheduler augmentation is as a source-level profiling technique, which has natural limitations in terms of precision and overhead, especially for time-based metrics (see discussion in [3]). Nevertheless, it can be a useful tool for quickly obtaining useful insights about performance. Specialized tools for particular languages can offer more precision, such as CilkView [17] and CilkProf [30]. Both tools perform incremental maintenance of the computation graph by dynamically updating local information at each executed fork/spawn and join/sync, which is similar to our approach. Basso et al. [6] considered profiling in a setting with more expressive fork-join tasks (in particular, features such as task cancellation). There has also been a large body of work on HPCToolkit [3], which offers a powerful tool suite for profiling parallel programs based on sampling and binary analysis. In the context of this prior work, one benefit of scheduler augmentation is that it enables basic graph-based profiling in languages and environments where such tools are not available.

OpenCilk Instrumentation. The OpenCilk compiler infrastructure provides a user-customizable instrumentation framework [31], closely integrated with the Tapir intermediate representation [32]. Using this framework, the user can instrument `spawn`s and `sync`s and insert custom data structures into call-stack frames. Although the interface is lower-level than the vertex interface we present in this paper, we note that many of the graph-based analyses that are possible with scheduler augmentation are also expressible as instrumentations within this framework.

Scheduler-based Metrics. Collecting performance metrics directly from the scheduler (such as idleness, steal counts, etc.) can be useful for pinpointing particular bottlenecks, especially scheduler overheads and insufficient parallelism [1, 38]. Lifflander et al. [23] develop a general technique for collecting traces of scheduler events in work-stealing schedulers, which bears some resemblance to our approach, except that they seek to incorporate schedule-specific details into trace, whereas we seek to abstract away the schedule and instead focus on the underlying computation graph.

Space Profiling. CilkMem [19] offers a powerful tool based on compiler instrumentation [29] to upper bound the memory usage of p-processor Cilk executions. Their approach is based on a notion of a space-annotated dag very similar to the space graph we use in Section 4. In principle, their algorithms could be applied within the scheduler augmentation framework by emitting the full space graph and then separately analyzing it. We note also that the S_∞ analysis we present could easily and efficiently be incorporated into CilkMem. Space profiling has also been explored in the context of functional languages [34], primarily for the purpose of cost semantics.

Lightweight Scheduling Libraries. Our implementation is built on top of ParlayLib [8], a C++ library offering a parallel scheduler and fundamental parallel primitives. Similar "library-only" aproaches are available across many languages (we name a few examples in Section 1). All of these libraries feature schedulers based on standard techniques, especially work-stealing [5, 10]. We believe that scheduler augmentation will be applicable across all of these libraries, with only small modifications to the scheduler implementation, and we are planning to investigate this in future work.

Cilk Hyperobjects. The implementation of Cilk hyperobjects [15] bears some resemblance to scheduler augmentation: each hyperobject maintains a small amount of state across workers, and this state is combined (monoidally) at join-points, automatically. In some sense, the functionality of hyperobjects can be thought of as a form of scheduler augmentation, by viewing the hyperobject state as vertex-local data and observing that vertex joining rules can be derived from the corresponding monoid of the hyperobject.

8 Conclusion and Future Work

We present *scheduler augmentation*, a technique that enables the programmer to observe the computation graph of a parallel execution by providing vertex-local data definitions to the scheduler. The technique is applicable even in library-only approaches, with no special compiler support. Our results show that scheduler augmentation can have very low overhead and can be easily customized for a variety of analyses. In this paper, we showcase two applications in particular: granularity analysis and space profiling.

In future work, we plan to explore more use-cases for scheduler augmentation and also extend the technique beyond the two-way fork primitive we focus on in this paper. In particular, we are interested in adding explicit support for parallel loops, as well as exploring primitives beyond fork-join. We believe the dag calculus [2] offers a promising path forward, as it provides a small set of primitives capable of expressing a wide variety of computation graph "shapes". By adapting the update rules of scheduler augmentation to mirror the dag calculus primitives, our approach could be extended to other parallel programming models such as futures or async-finish.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed [20] supported by the National Science Foundation.

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2017. Parallel Work Inflation, Memory Effects, and their Empirical Analysis. *CoRR* abs/1709.03767 (2017). arXiv:1709.03767 http://arxiv.org/abs/1709.03767

[2] Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016. Dag-calculus: a calculus for parallel computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 18–32. doi:10.1145/2951913.2951946

[3] Laksono Adhianto, S. Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. 2010. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.* 22, 6 (2010), 685–701. doi:10.1002/CPE.1553

[4] Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 445–447. doi:10.1145/3503221.3508422

[5] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. Association for Computing Machinery, New York, NY, USA, 119–129. doi:10.1145/277651.277678

[6] Matteo Basso, Eduardo Rosales, Filippo Schiavio, Andrea Rosà, and Walter Binder. 2022. Accurate Fork-Join Profiling on the Java Virtual Machine. In *Euro-Par 2022: Parallel Processing - 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22-26, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13440)*, José Cano and Phil Trinder (Eds.). Springer, 35–50. doi:10.1007/978-3-031-12597-3_3

[7] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, Phillip B. Gibbons and Micah Adler (Eds.). ACM, 133–144. doi:10.1145/1007912.1007933

[8] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, Christian Scheideler and Michael Spear (Eds.). ACM, 507–509. doi:10.1145/3350755.3400254

[9] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel Distributed Comput.* 37, 1 (1996), 55–69. doi:10.1006/JPDC.1996.0107

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. doi:10.1145/324133.324234

[11] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*, Christian W. Probst and Christian Wimmer (Eds.). ACM, 51–61. doi:10.1145/2093157.2093165

[12] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 519–538. doi:10.1145/1094811.1094852

[13] Domainslib Developers. 2026. *domainslib: Parallel Programming over Domains for Multicore OCaml.* https://github.com/ocaml-multicore/domainslib

[14] Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory Comput. Syst.* 32, 3 (1999), 301–326. doi:10.1007/S002240000120

[15] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, Friedhelm Meyer auf der Heide and Michael A. Bender (Eds.). ACM, 79–90. doi:10.1145/1583991.1584017

[16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, Jack W. Davidson, Keith D. Cooper, and A. Michael Berman (Eds.). ACM, 212–223. doi:10.1145/277650.277725

[17] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview scalability analyzer. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, Friedhelm Meyer auf der Heide and Cynthia A. Phillips (Eds.). ACM, 145–156. doi:10.1145/1810479.1810509

[18] Feiyang Jin, Lechen Yu, Tiago Cogumbreiro, Jun Shirako, and Vivek Sarkar. 2023. Dynamic Determinacy Race Detection for Task-Parallel Programs with Promises. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, Seattle, Washington, United States, July 17-21, 2023 (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:30. doi:10.4230/LIPICS.ECOOP.2023.13

[19] Tim Kaler, William Kuszmaul, Tao B. Schardl, and Daniele Vettorel. 2020. Cilkmem: Algorithms for Analyzing the Memory High-Water Mark of Fork-Join Parallel Programs. In *1st Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Salt Lake City, UT, USA, January 8, 2020*, Bruce M. Maggs (Ed.). SIAM, 162–176. doi:10.1137/1.9781611976021.12

[20] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[21] Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000*, Dennis Gannon and Piyush Mehrotra (Eds.). ACM, 36–43. doi:10.1145/337449.337465

[22] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. 2012. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, J. Ramanujam and P. Sadayappan (Eds.). ACM, 193–204. doi:10.1145/2145816.2145841

[23] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kalé. 2013. Steal Tree: low-overhead tracing of work stealing schedulers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 507–518. doi:10.1145/2491956.2462193

[24] John M. Mellor-Crummey. 1991. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, Joanne L. Martin (Ed.). ACM, 24–33. doi:10.1145/125826.125861

[25] Stefan K. Muller. 2022. Static prediction of parallel computation graphs. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. doi:10.1145/3498708

[26] Itzhak Nudler and Larry Rudolph. 1986. Tools for the efficient development of efficient parallel programs. In *Proceedings of the first Israeli conference on computer systems engineering*. 4–1.

[27] OpenMP. 2025. *OpenMP: The OpenMP API specification for parallel programming.* https://www.openmp.org/

[28] The Rust Project Developers. 2025. *Rayon.* https://github.com/rayon-rs/rayon

[29] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. 2018. The CSI Framework for Compiler-Inserted Program Instrumentation. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2018, Irvine, CA, USA, June 18-22, 2018*, Konstantinos Psounis, Aditya Akella, and Adam Wierman (Eds.). ACM, 100–102. doi:10.1145/3219617.3219657

[30] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The Cilkprof Scalability Profiler. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, Guy E. Blelloch and Kunal Agrawal (Eds.). ACM, 89–100. doi:10.1145/2755573.2755603

[31] Tao B. Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 189–203. doi:10.1145/3572848.3577509

[32] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2019. Tapir: Embedding Recursive Fork-join Parallelism into LLVM's Intermediate Representation. *ACM Trans. Parallel Comput.* 6, 4, Article 19 (Dec. 2019), 33 pages. doi:10.1145/3365655

[33] K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30. doi:10.1145/3408995

[34] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2010. Space profiling for parallel functional programs. *J. Funct. Program.* 20, 5-6 (2010), 417–461. doi:10.1017/S0956796810000146

[35] Guy L. Steele Jr., Doug Lea, and Christine H. Flood. 2014. Fast splittable pseudorandom number generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 453–472. doi:10.1145/2660193.2660195

[36] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 290–304. doi:10.1145/3178487.3178509

[37] Rishi Surendran and Vivek Sarkar. 2016. Dynamic Determinacy Race Detection for Task Parallelism with Futures. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 10012)*, Yliès Falcone and César Sánchez (Eds.). Springer, 368–385. doi:10.1007/978-3-319-46982-9_23

[38] Nathan R. Tallent and John M. Mellor-Crummey. 2009. Identifying Performance Bottlenecks in Work-Stealing Computations. *Computer* 42, 11 (2009), 44–50. doi:10.1109/MC.2009.396

[39] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, Christian Scheideler and Seth Gilbert (Eds.). ACM, 83–94. doi:10.1145/2935764.2935801

[40] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2019. Efficient race detection with futures. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 340–354. doi:10.1145/3293883.3295732

[41] UXL Foundation. 2025. *oneAPI Threading Building Blocks.* https://uxlfoundation.github.io/oneTBB/

[42] Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (aug 2022), 32 pages. doi:10.1145/3547646
[43] Sam Westrick, Larry Wang, and Umut A. Acar. 2022. DePa: Simple, Provably Efficient, and Practical Order Maintenance for Task Parallelism. *CoRR* abs/2204.14168 (2022). arXiv:2204.14168 doi:10.48550/ARXIV.2204.14168
[44] Yifan Xu, Kunal Agrawal, and I-Ting Angelina Lee. 2021. Efficient Parallel Determinacy Race Detection for Structured Futures. In *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, Kunal Agrawal and Yossi Azar (Eds.). ACM, 398–409. doi:10.1145/3409964.3461815

```
class SpaceVertex {
  int64_t δ = 0; uint64_t s1 = 0, sinf = 0;
  void join(V* vl, V* vr, V* vc) {
    vc->s1 = max(s1, δ + vl->s1, δ + vl->δ + vr->s1);
    vc->sinf = max(sinf, δ + vl->sinf + vr->sinf);
    vc->δ = δ + vl->δ + vr->δ;
  }
  void alloc(size_t sz) {
    δ += sz;
    s1 = max(s1, δ);
    sinf = max(sinf, δ);
  }
  void dealloc(size_t sz) { δ -= sz; }
}
```

Figure 13: Augmentation to track space usage in terms of S_1 and S_∞.

A Proof for the SpaceVertex Augmentation

Figure 13 presents an extended definition of the SpaceVertex with the additional capability to track the sequential high-water mark. Furthermore, we consider the series-parallel graph construction where upon series composition, the earlier graph's sink becomes the source of the later graph.

For a space vertex v, we define $\delta(v)$ to be the memory allocated by v. If v deallocated, $\delta(v) < 0$ and if v is a fork or join vertex, $\delta(v) = 0$. As a shorthand, we also define $\Delta T = \sum_{v \in T} \delta(v)$ where T is a set of vertices.

A set of vertices U is a ***prefix*** (also called an *initial segment*) of G if and only if for every $v \in U$, the ancesters of v are also members of U. A ***schedule*** $\vec{U}$ over G is the ordered sequence of prefixes $U_1, U_2, \cdots, U_n$ where $U_i \subset U_{i+1}$. Each U_i represents vertices that have been completed at timestep i.

A.1 Mapping to Computation Graphs

Let G be a space graph with corresponding computation graph G'. We introduce, for the purposes of the proof, the function f, which given a computation vertex $v' \in G'$, retrieves the corresponding chain of space vertices of v'.

Because the space graph subdivides a computation vertex into multiple vertices, a prefix $U \subseteq G$ may represent transient states where a computation vertex has only partially completed. To address this gap, we introduce *Done* to represent computation vertices that have to completion and *Running* to represent those that have only partially executed.

Definition A.1. Let the G' be the computation graph representing the space graph G. Let U be a prefix of G.

$$Done(U) = \{v' \in G' \mid f(v') \subseteq U\}$$
$$Running(U) = \{w' \in G' \mid f(w') \cap U \neq \emptyset \wedge f(w') \not\subseteq U\}$$

Additionally, we extend the δ function to computation vertices and introduce the notion of a local high-water mark (HWM).

Definition A.2. Let $v' \in G'$.

$$\delta(v') = \sum_{v \in f(v')} \delta(v)$$

and the local high-water mark of v' is

$$h(v') = \text{max prefix sum of } f(v')$$

The local HWM represents the peak memory footprint of a single vertex.

High-Water Mark. We generalize HWM to a schedule and then the entire space graph.

Definition A.3. Let $\vec{U}$ be a schedule over the space graph G. The ***high-water mark*** (HWM) of schedule $\vec{U}$ is

$$HWM(\vec{U}) = \max_{i=1,2,\cdots,n} \Delta U_i$$

The HWM represents the peak memory consumption for a specific schedule; it captures the behavior of an individual execution, but not of the program.

Definition A.4. The ***worst-case HWM*** of the space graph G on is

$$S_\infty(G) = \max\left\{HWM(\vec{U}) \mid \vec{U} \text{ is a schedule over } G\right\}$$

On the other hand, the worst-case HWM considers all possible ways to execute the graph, strongly upper-bounding the space consumption of the program regardless of the schedule.

Corollary A.5. *Let $\mathcal{I}(G)$ be the set of all prefixes of G.*

$$S_\infty(G) = \max_{U \in \mathcal{I}(G)} \Delta U$$

Corollary A.6.

$$S_\infty(G) \geq \Delta G$$

A.2 Computing S_1 and S_∞

Now, we present formulae for deriving the S_1 and S_∞ values of a computation graph. We approach this inductively, starting with the singleton graph.

Lemma A.7 (Singleton Graph). *Let $G' = \{v\}$ be a singleton computation graph. Then, $S_\infty(G') = S_1(G') = h(v')$.*

Lemma A.8 (S_∞ Parallel Composition). *Let G' be a parallel composition of G'_L, G'_R with source v and sink w. Then*

$$S_\infty(G') = \max\begin{cases} h(v) & \text{(1a)} \\ \delta(v) + S_\infty(G'_L) + S_\infty(G'_R), & \text{(1b)} \\ \delta(v) + \Delta G'_L + \Delta G'_R + h(w), & \text{(1c)} \end{cases}$$

Proof. By Corollary A.5, $S_\infty(G')$ is the maximum ΔU over all prefixes U of G'. We case-split on the structure of U.

(a) *U does not extend past v.* Then U contains only a prefix of v's space vertices, so $\Delta U \leq h(v)$.
(b) *U extends into G_L and/or G_R but does not reach w.* Then U contributes $\delta(v)$ from the completed source, plus prefixes $U_L \subseteq G_L$ and $U_R \subseteq G_R$. Thus $\Delta U = \delta(v) + \Delta U_L + \Delta U_R$. Since U_L and U_R are independent prefixes, this is maximized at $\delta(v) + S_\infty(G'_L) + S_\infty(G'_R)$.
(c) *U reaches w.* Both G_L and G_R must be fully completed for w to become ready, so $\Delta U = \delta(v) + \Delta G'_L + \Delta G'_R + \Delta U_w$ where U_w is a prefix of w's space vertices. This is maximized when $\Delta U_w = h(w)$.

Taking the maximum over all three cases gives the stated formula. □

LEMMA A.9 (S_1 PARALLEL COMPOSITION). *Let G' be a parallel composition of G'_L, G'_R with source v and sink w. Then*

$$S_1(G') = \max \begin{cases} h(v), & \text{(2a)} \\ \delta(v) + S_1(G'_L), & \text{(2b)} \\ \delta(v) + \Delta G'_L + S_1(G'_R), & \text{(2c)} \\ \delta(v) + \Delta G'_L + \Delta G'_R + h(w) & \text{(2d)} \end{cases}$$

PROOF. In a sequential left-to-right schedule, the execution first completes v's space vertices, then all of G_L, then all of G_R, then w's space vertices. We case-split based on when the peak occurs.

(a) *During v:* The peak is at most $h(v)$.
(b) *During G_L:* The accumulated memory is $\delta(v)$ plus G_L's contributions, yielding at most $\delta(v) + S_1(G'_L)$.
(c) *During G_R:* Since we consider a sequential left-to-right schedule, G_L must have fully completed prior to the start of G_R. So, the baseline is $\delta(v) + \Delta G'_L$, yielding at most $\delta(v) + \Delta G'_L + S_1(G'_R)$.
(d) *During w:* Both children are done, giving baseline $\delta(v) + \Delta G'_L + \Delta G'_R$, yielding at most $\delta(v) + \Delta G'_L + \Delta G'_R + h(w)$.

Taking the maximum gives the stated formula. □

LEMMA A.10 (SERIES COMPOSITION). *Let G' be a series composition of G'_1, G'_2 sharing vertex u. Then for $p \in \{1, \infty\}$,*

$$S_p(G') = \max\left(S_p(G'_1), \Delta(G'_1 \setminus \{u\}) + S_p(G'_2)\right)$$

PROOF. Any prefix U of G is either a prefix of G_1, yielding the first argument to max, or it is the entire G_1 (excluding the shared vertex u to avoid double-counting) combined with a prefix of G_2, yielding the second argument to max. □

THEOREM A.11. *Let G be a fork-join computation graph and v be its sink vertex, augmented with* `SpaceVertex` *definitions. The vertex v satisfies*

$$v.\texttt{delta} = \Delta G$$
$$v.\texttt{s1} = S_1(G)$$
$$v.\texttt{sinf} = S_\infty(G)$$

PROOF. We prove by structural induction on the series-parallel dag G.

Base case. Let G be the singleton graph with vertex v.

Net memory. The `alloc` and `dealloc` methods update the `delta` field cumulatively. Therefore, at the end of the computation, `delta` contains the sum of the δ values of its internal space vertices. Definition A.2 then gives $v.delta = \Delta G$.

Parallel & Sequential HWM. Since the `alloc` method updates `s1` and `sinf` at every call by taking the larger amongst the new `delta` value and the old `s1/sinf`, $v.s1 = v.sinf = h(v)$. Then, by Lemma A.7, $v.s1 = v.sinf = S_\infty(G) = S_1(G)$.

Parallel Composition. Assume G is a parallel composition of subgraphs G_L and G_R with source u and sink v. Let G_L and G_R have sink v_L and v_R respectively. We first show that at the moment of joining, the `join` method correctly calculates the desired value. Then, we argue that this invariant is upheld by subsequent allocations and deallocations in v.

Net memory. Immediately after joining, $u.delta = v.delta + v_L.delta + v_R.delta$. By induction, $v_L.delta = \Delta G_L$ and $v_R.delta = \Delta G_R$. Thus, at the moment of the join, $v.delta$ correctly reflects the net memory allocated by the computation until that point. This invariant is preserved up until v is completed. Therefore, upon completion of G, $v.delta = \Delta G$.

Parallel HWM. The `join` method sets

$$v.sinf = \max\left(u.sinf, u.delta + v_L.sinf + v_R.sinf\right).$$

By the inductive hypothesis, $v_L.sinf = S_\infty(G_L)$ and $v_R.sinf = S_\infty(G_R)$. By Lemma A.7, $u.sinf = S_\infty(\{u\}) = h(u)$. Substituting these values, we have

$$v.sinf = \max\left(h(u), \delta(u) + S_\infty(G_L) + S_\infty(G_R)\right)$$

and $v.delta = \delta(u) + \Delta G_L + \Delta G_R$. Since no allocations have yet occurred in v at this point, $h(v) = 0$. By Corollary A.6,

$$\begin{aligned} &\delta(u) + \Delta G_L + \Delta G_R + h(v) \\ &= \delta(u) + \Delta G_L + \Delta G_R \\ &\le \delta(u) + S_\infty(G_L) + S_\infty(G_R) \end{aligned}$$

Therefore, at join time, case (1b) does not dominate the other two terms.

As subsequent allocations and deallocations occur in v, the `alloc` method updates $v.sinf \leftarrow \max(v.sinf,\ v.delta)$. Starting from

$$v.delta = \delta(u) + \Delta G_L + \Delta G_R$$

the maximum value $v.delta$ attains over v's own space vertices is $\delta(u) + \Delta G_L + \Delta G_R + h(v)$. Therefore, upon completion,

$$v.sinf = \max\Big(\underbrace{h(u)}_{(1a)}, \underbrace{\delta(u) + S_\infty(G_L) + S_\infty(G_R)}_{(1b)}, \underbrace{\delta(u) + \Delta G_L + \Delta G_R + h(v)}_{(1c)}\Big)$$

matching (1a)–(1c) of Lemma A.8.

Sequential HWM. The `join` method sets

$$v.s1 = \max\left(u.s1,\ \ u.delta + v_L.s1,\ \ u.delta + v_L.delta + v_R.s1\right).$$

By induction, $v_L.s1 = S_1(G_L)$, $v_R.s1 = S_1(G_R)$, and $v_L.delta = \Delta G_L$. By Lemma A.7, $u.s1 = h(u)$. Thus, immediately after the join,

$$v.s1 = \max\left(h(u), \delta(u) + S_1(G_L), \delta(u) + \Delta G_L + S_1(G_R)\right).$$

By the same argument as for S_∞, subsequent allocations at v contribute the additional term (2d): $\delta(u) + \Delta G_L + \Delta G_R + h(v)$. Upon completion,

$$v.s1 = \max\Big(\underbrace{h(u)}_{(2a)}, \underbrace{\delta(u) + S_1(G_L)}_{(2b)}, \underbrace{\delta(u) + \Delta G_L + S_1(G_R)}_{(2c)}, \underbrace{\delta(u) + \Delta G_L + \Delta G_R + h(v)}_{(2d)}\Big)$$

matching (2a)–(2d) of Lemma A.9.

Suite	Short Name	Benchmark	Input Size	Input Details
PBBS	intSort	integerSort/parallelRadixSort	$n = 100$M	randomSeq_100M_int: n ints generated uniformly at random in the range [0:n)
	cmpSort	comparisonSort/sampleSort	$n = 100$M	randomSeq_100M_double: n doubles generated uniformly at random in the range [0:1]
	rmDup	removeDuplicates/parlayhash	$n = 100$M	randomSeq_100M_int: n ints generated uniformly at random in the range [0:n)
	hist	histogram/parallel	$n = 100$M	randomSeq_100M_int: n ints generated uniformly at random in the range [0:n)
	wordCnt	wordCounts/histogram	$n = 250$M	wikipedia250M.txt: A sample from wikipedia's xml source files
	invIdx	invertedIndex/parallel	$n = 250$M	wikipedia250M.txt: A sample from wikipedia's xml source files
	suffArr	suffixArray/parallelRange	$n = 34$M	chr22.dna: A DNA sequence consisting only of the characters C,G,C,A,N
	lrs	longestRepeatedSubstring/doubling	$n = 34$M	chr22.dna: A DNA sequence consisting only of the characters C,G,C,A,N
	classDT	classify/decisionTree	covtype.data	covtype.data: A data set used to predict the cover type of forests from cartographic variables
	msf	minSpanningForest/parallelFilterKruskal	$V = 16$M, $E = 192$M	rMatGraph_WE_12_16000000: Weighted undirected RMAT graph with parameters (.2,.125,.125,.55)
	sf	spanningForest/ndST	$V = 16$M, $E = 192$M	rMatGraph_E_12_16000000: Undirected RMAT graph with parameters (.2,.125,.125,.55)
	bfs	breadthFirstSearch/backForwardBFS	$V = 16$M, $E = 192$M	rMatGraph_J_12_16000000: Directed RMAT graph with parameters (.2,.125,.125,.55)
	maxMatch	maximalMatching/incrementalMatching	$V = 20$M, $E = 200$M	rMatGraph_E_10_20000000: Undirected RMAT graph with parameters (.2,.125,.125,.55)
	mis	maximalIndependentSet/incrementalMIS	$V = 16$M, $E = 192$M	rMatGraph_JR_12_16000000: Undirected RMAT graph with parameters (.2,.125,.125,.55)
	nn	nearestNeighbors/octTree	$n = 10$M	2Dkuzmin_10M: n 2D points chosen at random from the Kuzmin distribution
	rayCast	rayCast/kdTree	$n = 1,087,716$	happyTriangles happyRays: The "happy Buddha" mesh from the Stanford 3D Scanning Repository
	cvxHull	convexHull/quickHull	$n = 100$M	2Dkuzmin_100000000: n 2D points chosen at random from the Kuzmin distribution
	delTri	delaunayTriangulation/incrementalDelaunay	$n = 10$M	2Dkuzmin_10M: n 2D points chosen at random from the Kuzmin distribution
	delRef	delaunayRefine/incrementalRefine	$n = 5$M	2DkuzminDelaunay_5000000: n 2D points chosen at random from the Kuzmin distribution
	nBody	nBody/parallelCK	$n = 1$M	3DonSphere_1000000: n 3D points chosen uniformly at random on the surface of a unit sphere
	rq2d	rangeQuery2d/parallelPlaneSweep	$n = 10$M	2Dkuzmin_10M: n 2D points chosen at random from the Kuzmin distribution
	rq2d_ours	rangeQuery2d/parallelPlaneSweep	$n = 10$M	2Dkuzmin_10M: n 2D points chosen at random from the Kuzmin distribution
ParlayLib	ldd	graph/low_diameter_decomposition	$V = 10$M, $E = 200$M	Graph generated using the RMAT model
	triCnt	graph/triangle_count	$V = 800$K, $E = 16$M	Graph generated using the RMAT model
	karatsuba	numerical/karatsuba	$n = 10$M	2 Bigint Numbers generated independently and uniformly at random in the range [0, 2^{n-1}]
	mcss	numerical/mcss	$n = 1,000$M	n integers generated uniformly at random in the range [-100, 100]
	primes	numerical/primes	$n = 800$M	n is the limit of primes to generate
	mergeSort	sorting/mergesort	$n = 10$M	n longs generated uniformly at random in the range [0, n-1]
	quickSort	sorting/quicksort	$n = 10$M	n longs generated uniformly at random in the range [0, n-1]
	huffman	string/huffman_tree	$n = 3$M	Set of n discrete symbols whose probabilities are drawn from a randomized Harmonic distribution

Table 2: Description of the PBBS and ParlayLib benchmarks used in Section 6.

Series Composition. Assume G is a series composition of G_1 and G_2 sharing vertex u, where v is the sink of G (and of G_2). By the inductive hypothesis, upon completion of G_1, $u.delta = \Delta G_1$, $u.s1 = S_1(G_1)$, and $u.sinf = S_\infty(G_1)$. Execution then continues with G_2, which has u as its source.

Net memory. The `delta` field accumulates all net allocations throughout G. Since every vertex in G is processed exactly once, $v.delta = \Delta G$ upon completion.

Parallel & Sequential HWM. After G_1 completes, $u.s_p = S_p(G_1)$ by induction. Since the algorithm updates s_p only via max, $v.s_p \geq S_p(G_1)$ at all subsequent times.

In the combined execution, u's space vertices are processed during G_1 and not revisited during G_2. After u completes, `delta` $= \Delta G_1 = \Delta(G_1 \setminus \{u\}) + \delta(u)$, whereas standalone G_2 would have `delta` $= \delta(u)$ at the same point. Since subsequent operations modify `delta` additively, the offset $\Delta(G_1 \setminus \{u\})$ persists, so every post-u peak in the combined execution exceeds its standalone counterpart by exactly $\Delta(G_1 \setminus \{u\})$. Letting M denote the maximum post-u peak in standalone G_2, the algorithm gives

$$v.s_p = \max\left(S_p(G_1),\ \Delta(G_1 \setminus \{u\}) + M\right)$$

Since $S_p(G_2) = \max(h(u), M)$ and $\Delta(G_1 \setminus \{u\}) + h(u) \leq S_p(G_1)$ (the prefix of G_1 in which $G_1 \setminus \{u\}$ is complete and u is at its local peak attains exactly this cost), replacing M with $S_p(G_2)$ does not change the outer max:

$$v.s_p = \max\left(S_p(G_1),\ \Delta(G_1 \setminus \{u\}) + S_p(G_2)\right)$$

matching Lemma A.10. □

B Benchmark Details

Table 2 presents details of the benchmarks used in our evaluation, including specifications of the inputs.