Parallel Branch-and-Bound Knapsack in Haskell
Frontier-based Parallelism, Shared Bounds, and Work Stealing

Jessica Xu
COMS 4995 Parallel Functional Programming

December 17, 2025

Abstract

This report describes a Haskell implementation of the 0/1 knapsack problem using a branch-and-bound depth-first search (DFS) solver, and several parallelizations that target multi-core scaling. The key idea is to prune aggressively using a fractional (knapsack relaxation) upper bound, and to increase parallel efficiency by generating a *frontier* of partial states that can be solved independently. We compare: (i) a baseline sequential DFS, (ii) frontier-parallel evaluation with `parMap`, (iii) parallel DFS using a shared global bound via `IORef` or `MVar`, and (iv) a dynamic work-stealing pool to mitigate load imbalance. We also describe profiling methodology using the GHC eventlog and ThreadScope.

1 Introduction

The 0/1 knapsack problem is a classic combinatorial optimization task: given items with values and weights, choose a subset of items to maximize total value without exceeding a capacity constraint. Knapsack is NP-hard, but branch-and-bound solvers with strong bounds often perform well on structured instances.

This project focuses on *parallelizing* a branch-and-bound knapsack solver in Haskell. Similar to game search and other tree-search workloads, knapsack offers abundant parallelism but also challenging *irregular* work: different branches prune at different depths, so naïve static partitioning can leave cores idle. Our implementations explore both static and dynamic work distribution.

2 Problem Definition

Given n items (v_i, w_i) and capacity C, the 0/1 knapsack objective is:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \le C, \quad x_i \in \{0,1\}.$$

We assume inputs are pre-filtered to valid numeric ranges, and we use `Double` (or `Int` in early versions) for values, weights, and capacity.

3 Benchmark Problem Instances

To evaluate performance across a range of problem characteristics, we used publicly available 0/1 knapsack benchmark suites:

- **Jorik Jooken Knapsack Instances**
  https://github.com/JorikJooken/knapsackProblemInstances
- **Xiang 0/1 Knapsack Instances (KP1)**
  https://github.com/dnlfm/knapsack-01-instances/blob/main/xiang_instances_01_KP/KP1
- **General 0/1 Knapsack Instance Collection**
  https://github.com/dnlfm/knapsack-01-instances/tree/main

These datasets include a wide range of sizes and difficulty levels, making them suitable for evaluating both pruning effectiveness and parallel scalability. Some contain the solved optimum to verify the correctness of the sequential implementation.

4 Sequential Branch-and-Bound DFS

4.1 Item ordering

We sort items by decreasing value/weight ratio (a common heuristic): this tends to find good solutions early (tightening the incumbent best value), and improves the fractional bound quality.

To avoid expensive list indexing, we store the sorted items in a `Vector` for $O(1)$ indexing. This is the same motivation used in other projects where indexed access is hot in the inner loop. *(Replace this sentence with your final citation or discussion if needed.)*

4.2 Fractional upper bound

At a DFS node with current weight W and value V, we compute an admissible upper bound by filling the remaining capacity with a *fractional* knapsack relaxation: take whole items greedily by ratio, then possibly take a fraction of the next item. If this bound is $\leq$ the best-so-far value, we prune the node.

4.3 DFS recursion

Each node branches into:

- **Include** the next item (if weight allows)
- **Exclude** the next item

The best value found in the subtree is returned upward and becomes the incumbent bound for siblings.

5 Pruning Propagation in a DFS Tree

Figure 1 illustrates a 3-level include/exclude DFS tree. Once an improved incumbent is discovered in one subtree, it can cause sibling subtrees to prune earlier (bound propagation).

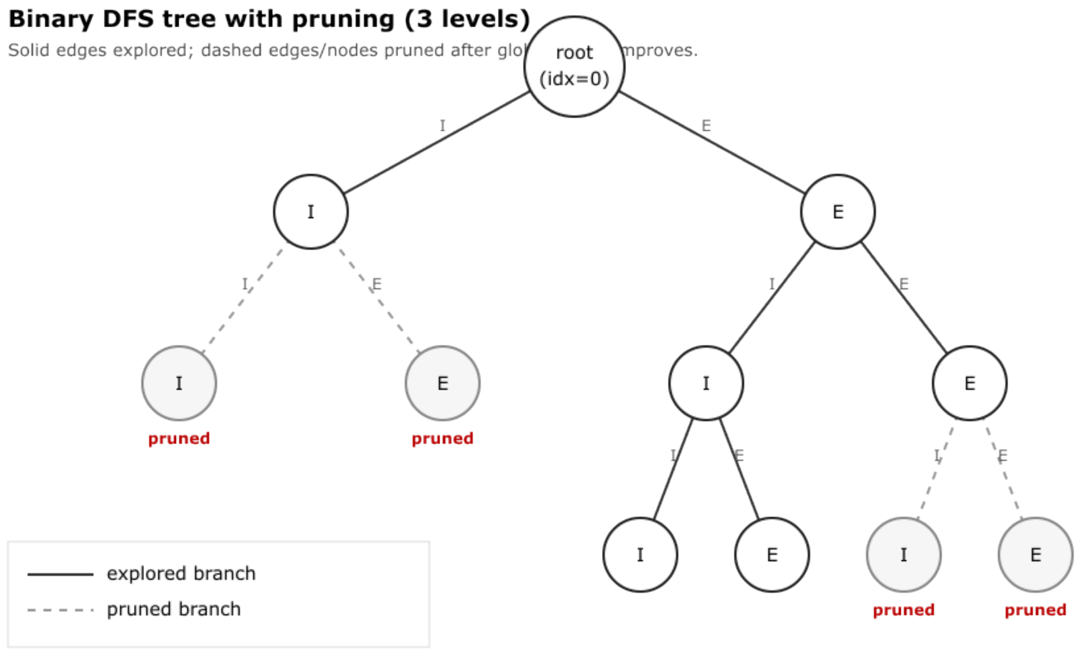
Binary DFS tree with pruning (3 levels)
Solid edges explored; dashed edges/nodes pruned after global best improves.
root (idx=0)
I
E
I
E
I
E
I
E
pruned
pruned
I
E
I
E
pruned
pruned
explored branch
pruned branch

Figure 1: Binary DFS tree with include (I) and exclude (E) branching. Dashed nodes/edges are pruned after the global best improves.

6 Frontier Construction

6.1 Motivation

Pure recursive parallel DFS is often inefficient because: (1) tasks are too fine-grained near the leaves, and (2) parallel sparks do not automatically balance irregular work. Instead, we build a *frontier* of partial states up to some depth limit d, creating $O(2^d)$ subproblems that are large enough to amortize overhead.

6.2 How it works

`build_frontier` performs a shallow DFS for d levels (with lightweight pruning), and records each partial state:

$$\texttt{State} = (\texttt{index}, \texttt{current\_value}, \texttt{current\_weight}).$$

Each state becomes the root of an independent deeper DFS.

Figure 2 shows this pipeline.

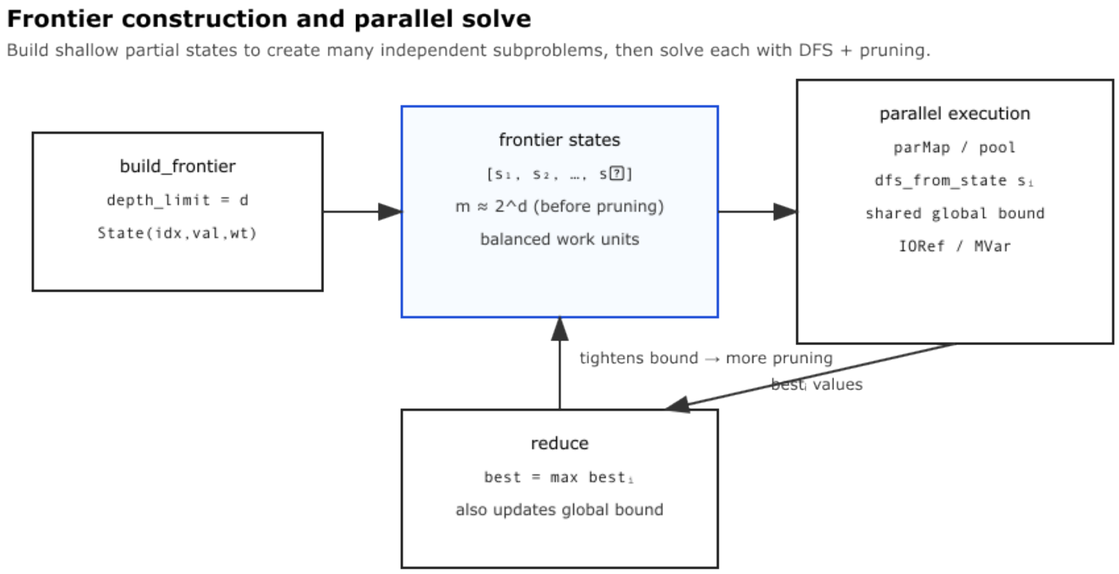

Figure 2: Frontier construction creates many medium-sized tasks which can be solved in parallel.

6.3 Why it speeds up

Frontier-based parallelism improves performance by:

- increasing **task granularity** (fewer, larger tasks),
- enabling **static parallelism** with low overhead (e.g., `parMap`),
- making it easier to add **dynamic scheduling** (work stealing).

However, picking d is a tuning problem: too small yields too few tasks; too large creates too many tasks and overhead.

7 Parallel Variants

7.1 Frontier-parallel with `parMap`

The simplest parallel version evaluates each frontier state with a pure DFS function and reduces via `max`. This is easy to implement and useful as a baseline.

7.2 Shared global bound: `IORef` vs `MVar`

To improve pruning across tasks, we maintain a global incumbent best value shared by workers:

- `IORef` with `atomicModifyIORef'` gives low overhead updates, but requires careful strictness to avoid thunks and to guarantee correctness under concurrency.
- `MVar` provides mutual exclusion, simplifying correctness at the cost of contention.

Both approaches aim to reduce total explored nodes by propagating the best solution found by any worker.

7.3 Dynamic work stealing

Static partitioning can still load-imbalance when some frontier tasks prune quickly while others explode. A work-stealing pool addresses this by giving each worker a local deque/stack and allowing idle workers to steal tasks from others. This tends to produce better ThreadScope utilization when the workload is irregular.

Workers vs Threads. In our parallel knapsack solvers, a *worker* denotes a logical search agent responsible for exploring a subset of the branch-and-bound search space. Each worker is implemented as a long-lived Haskell thread created via `forkIO`, but workers are conceptually distinct from operating system threads. The GHC runtime multiplexes many Haskell threads onto a limited number of OS threads, which are in turn mapped to physical CPU cores via the `-N` runtime option. As a result, workers represent algorithmic units of parallelism, while OS threads and cores represent execution resources managed by the runtime system.

8 ThreadScope and Eventlog Profiling

We profile with:

- `+RTS -N# -l -s -RTS` to generate `.eventlog`
- ThreadScope to visualize spark creation, GC, and CPU utilization.

1. sequential

   As expected only one core is doing everything in the pure sequential solution.

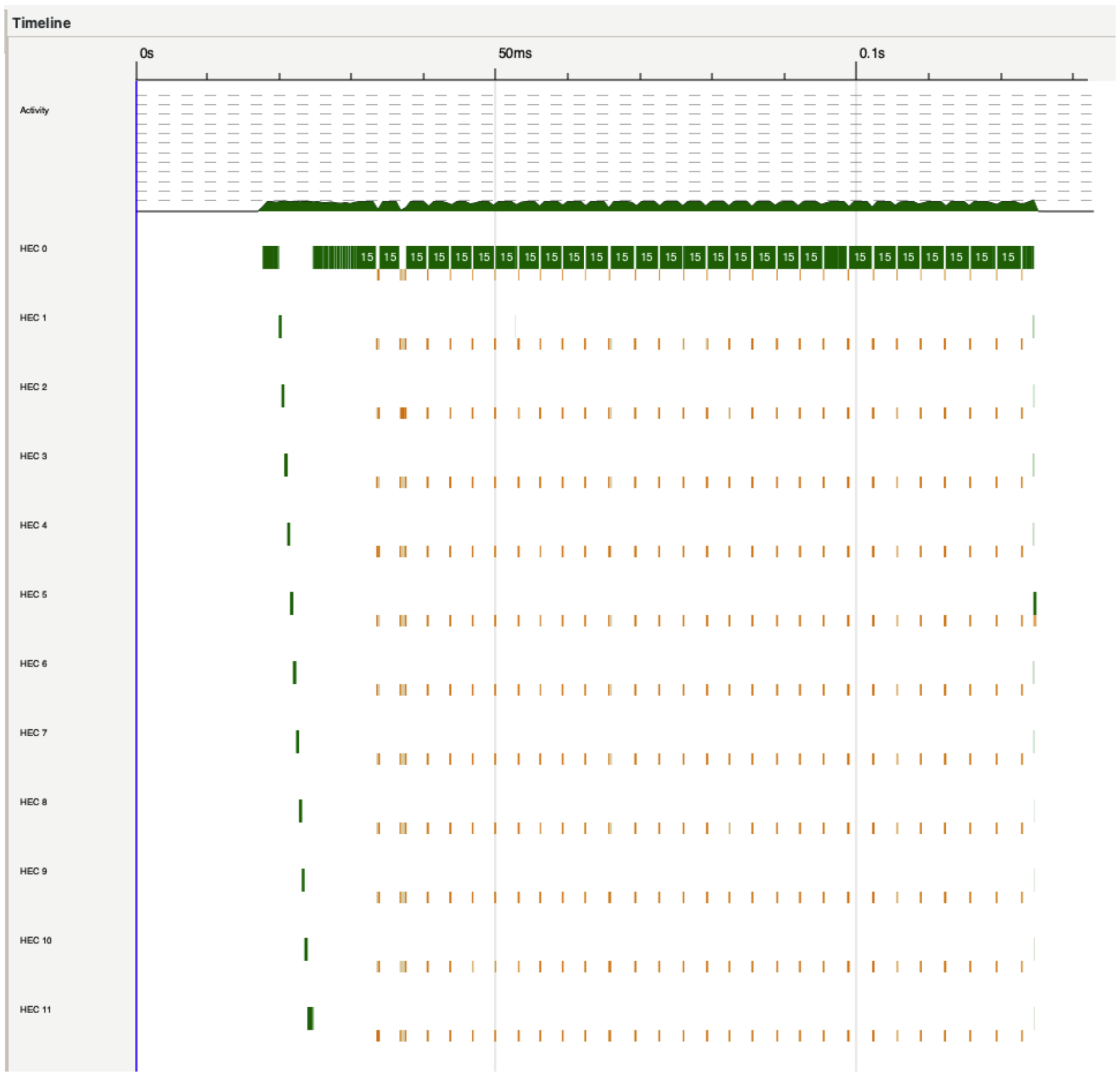

Figure 3: Sequential Baseline Threadscope Profiling

2. parallel branches growing

   With parallel DFS branching all cores are loaded with busy work. It sparked off a humongous number of sparks without much pattern in work distribution.

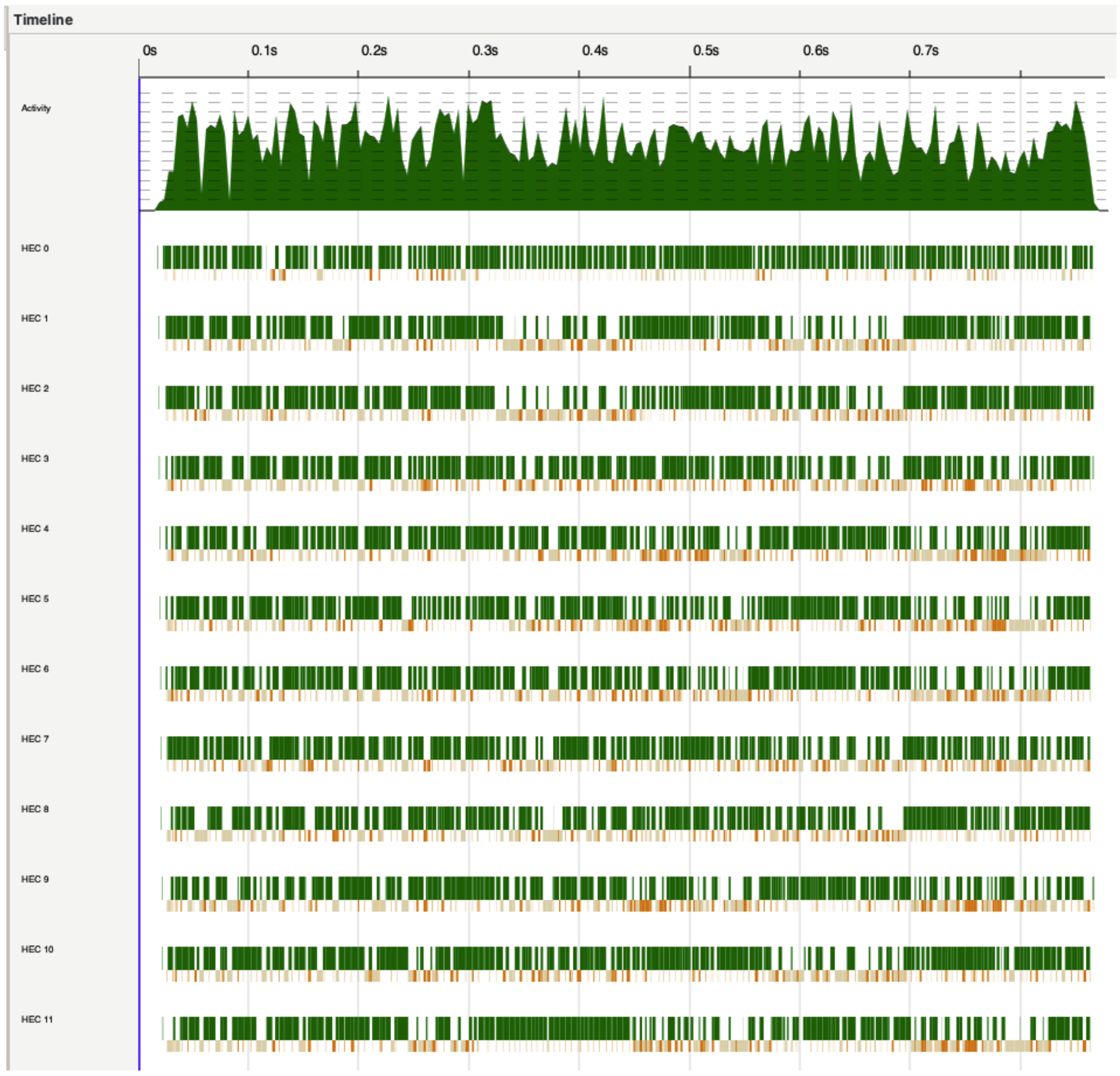

Figure 4: Parallel DFS Branching (with no frontier)Threadscope Profiling

3. IORef global bound sharing

   With a frontier depth limit of 6 and 12 cores (the same for the modes afterwards), the solver grows the DFS trees on all cores. Some cores seem to be idle while waiting on others to finish. The productivity is nevertheless high: 85.1%.

```
     120,432,056 bytes allocated in the heap
         517,360 bytes copied during GC
         749,952 bytes maximum residency (2 sample(s))
         196,224 bytes maximum slop
              63 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0         2 colls,     2 par    0.004s   0.001s     0.0004s    0.0006s
  Gen  1         2 colls,     1 par    0.001s   0.001s     0.0005s    0.0008s

  Parallel GC work balance: 64.89% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.024s  (  0.027s elapsed)
  MUT     time    0.189s  (  0.028s elapsed)
  GC      time    0.005s  (  0.002s elapsed)
  EXIT    time    0.004s  (  0.009s elapsed)
  Total   time    0.222s  (  0.066s elapsed)

  Alloc rate    636,883,149 bytes per MUT second

  Productivity  85.1% of total user, 43.0% of total elapsed
```

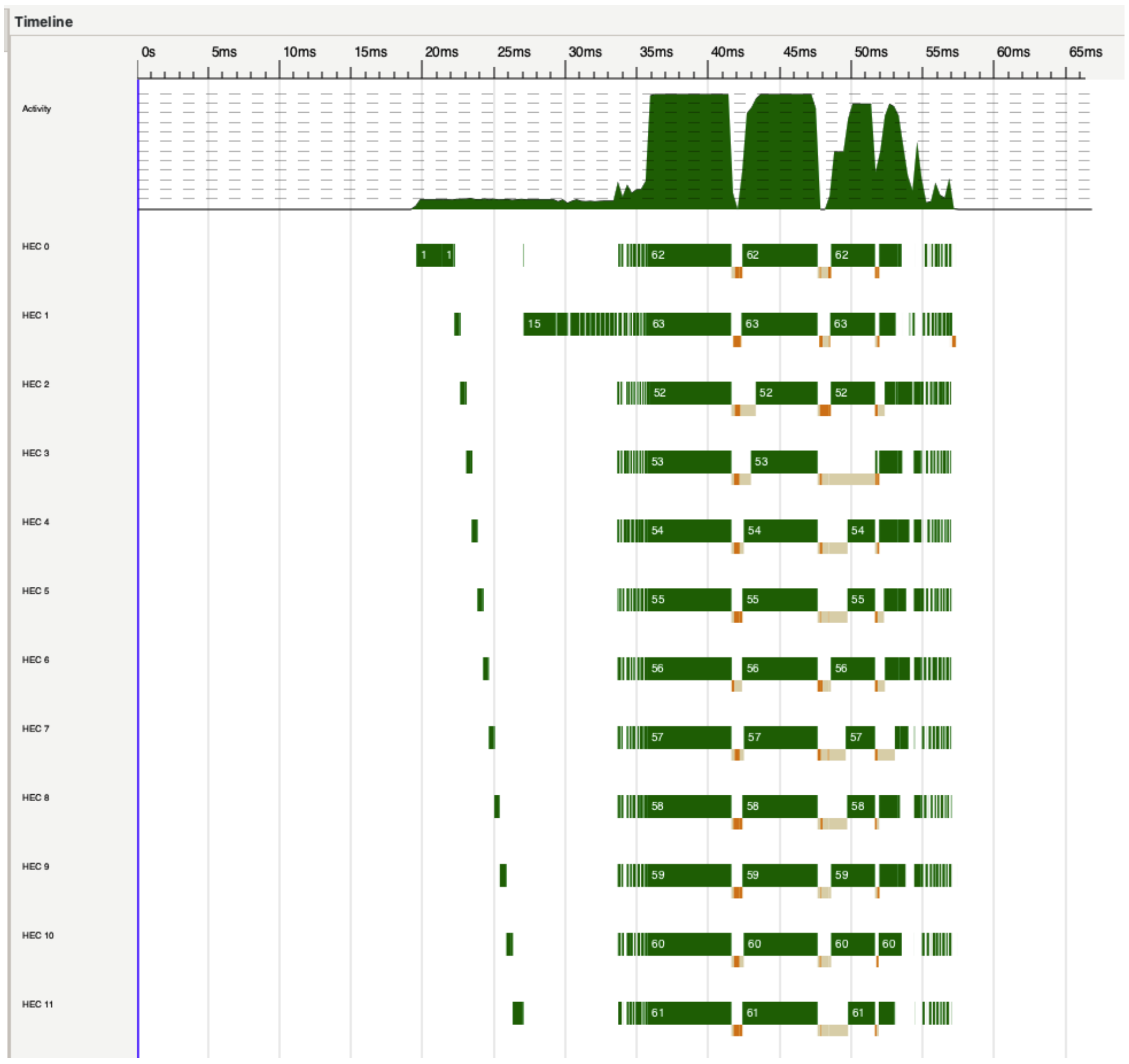

Figure 5: IORef (with frontier buidling) Threadscope Profiling

Observations:

- Large contiguous green blocks indicate long-running tasks assigned early.
- Some cores are idle while others remain busy.
- Contention on the MVar causes visible blocking (orange segments).

Work distribution is static: once a worker finishes its assigned subtree, it often has no more work. This explains why MVar-based performance is sensitive to frontier depth and worker count in the performance section.

4. MVar global bound sharing

   Performance improves with the global bound shared in `MVar`. There seems to be more pruning but also more waiting on the cores.

```
     119,329,848 bytes allocated in the heap
         505,792 bytes copied during GC
         659,936 bytes maximum residency (2 sample(s))
         204,320 bytes maximum slop
              63 MiB total memory in use (0 MiB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0         2 colls,     2 par    0.002s   0.001s     0.0004s    0.0006s
  Gen  1         2 colls,     1 par    0.001s   0.001s     0.0004s    0.0004s

  Parallel GC work balance: 58.03% (serial 0%, perfect 100%)

  TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.023s  (  0.022s elapsed)
  MUT     time    0.194s  (  0.028s elapsed)
  GC      time    0.003s  (  0.001s elapsed)
  EXIT    time    0.003s  (  0.011s elapsed)
  Total   time    0.224s  (  0.062s elapsed)

  Alloc rate    614,988,187 bytes per MUT second

  Productivity  86.7% of total user, 45.0% of total elapsed
```

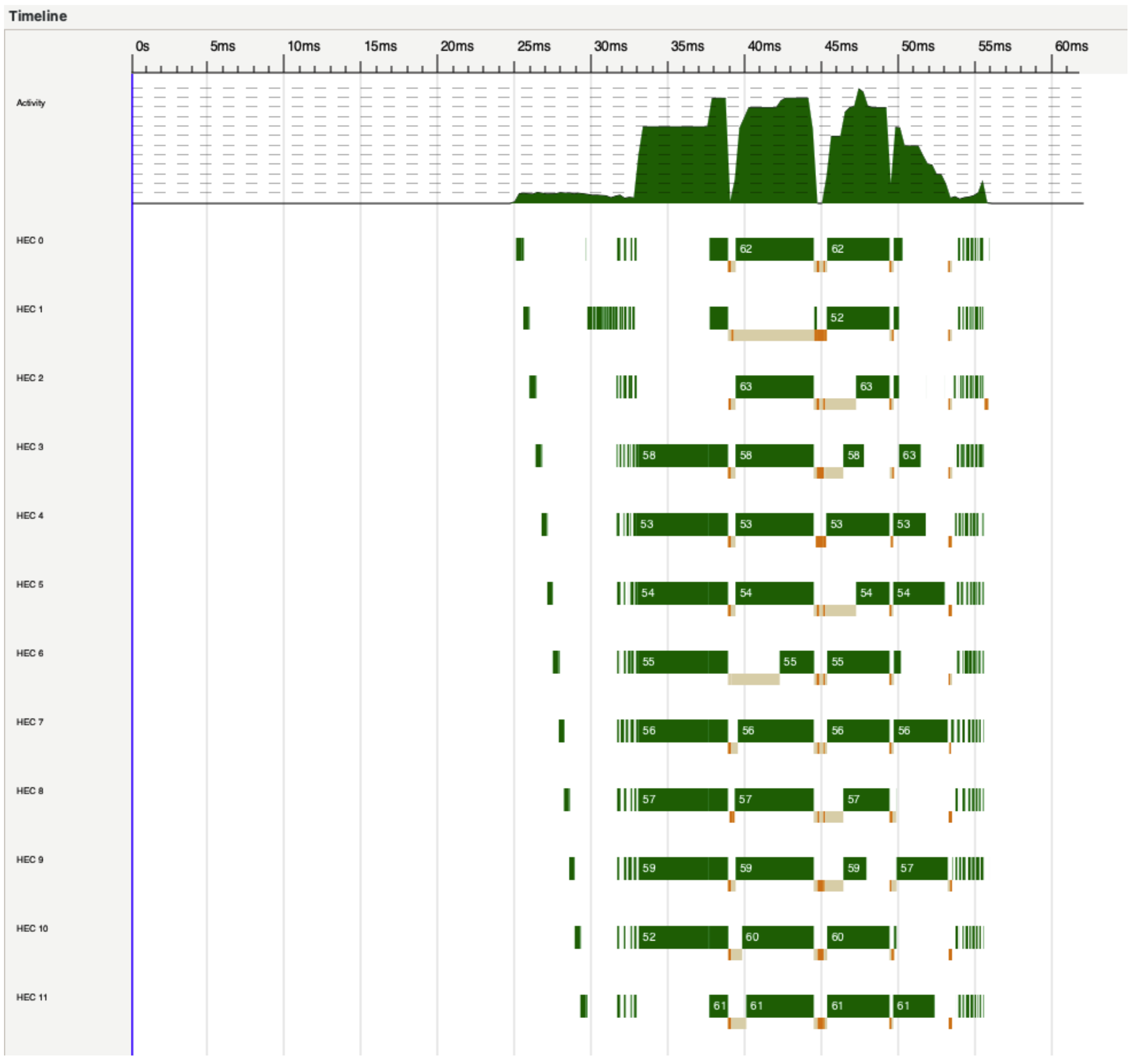

Figure 6: MVar (with frontier buidling) Threadscope Profiling

Observations:

- Green activity is more evenly spread across all HECs.
- Shorter task segments indicate frequent task stealing.
- Idle gaps are reduced compared to the MVar version.

There is increased overhead from: atomic operations, stealing attempts, and task deque manipulation. Despite the overhead, CPU utilization is more uniform, especially in later phases of the search when pruning behavior becomes unpredictable.

5. MVar global bound sharing plus work-stealing deques

   Work distribution is smoothly even in the work-stealing mode, with a higher overhead though.

```
     120,451,936 bytes allocated in the heap
         521,760 bytes copied during GC
         751,832 bytes maximum residency (2 sample(s))
         227,112 bytes maximum slop
              63 MiB total memory in use (0 MiB lost due to fragmentation)

                                   Tot time (elapsed)  Avg pause  Max pause
  Gen  0         2 colls,     2 par    0.003s   0.001s     0.0003s    0.0006s
  Gen  1         2 colls,     1 par    0.001s   0.001s     0.0003s    0.0004s

  Parallel GC work balance: 57.30% (serial 0%, perfect 100%)

  TASKS: 27 (1 bound, 26 peak workers (26 total), using -N12)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.021s  (  0.020s elapsed)
  MUT     time    0.184s  (  0.048s elapsed)
  GC      time    0.004s  (  0.001s elapsed)
  EXIT    time    0.003s  (  0.003s elapsed)
  Total   time    0.212s  (  0.072s elapsed)

  Alloc rate    654,438,023 bytes per MUT second

  Productivity  86.7% of total user, 66.4% of total elapsed
```

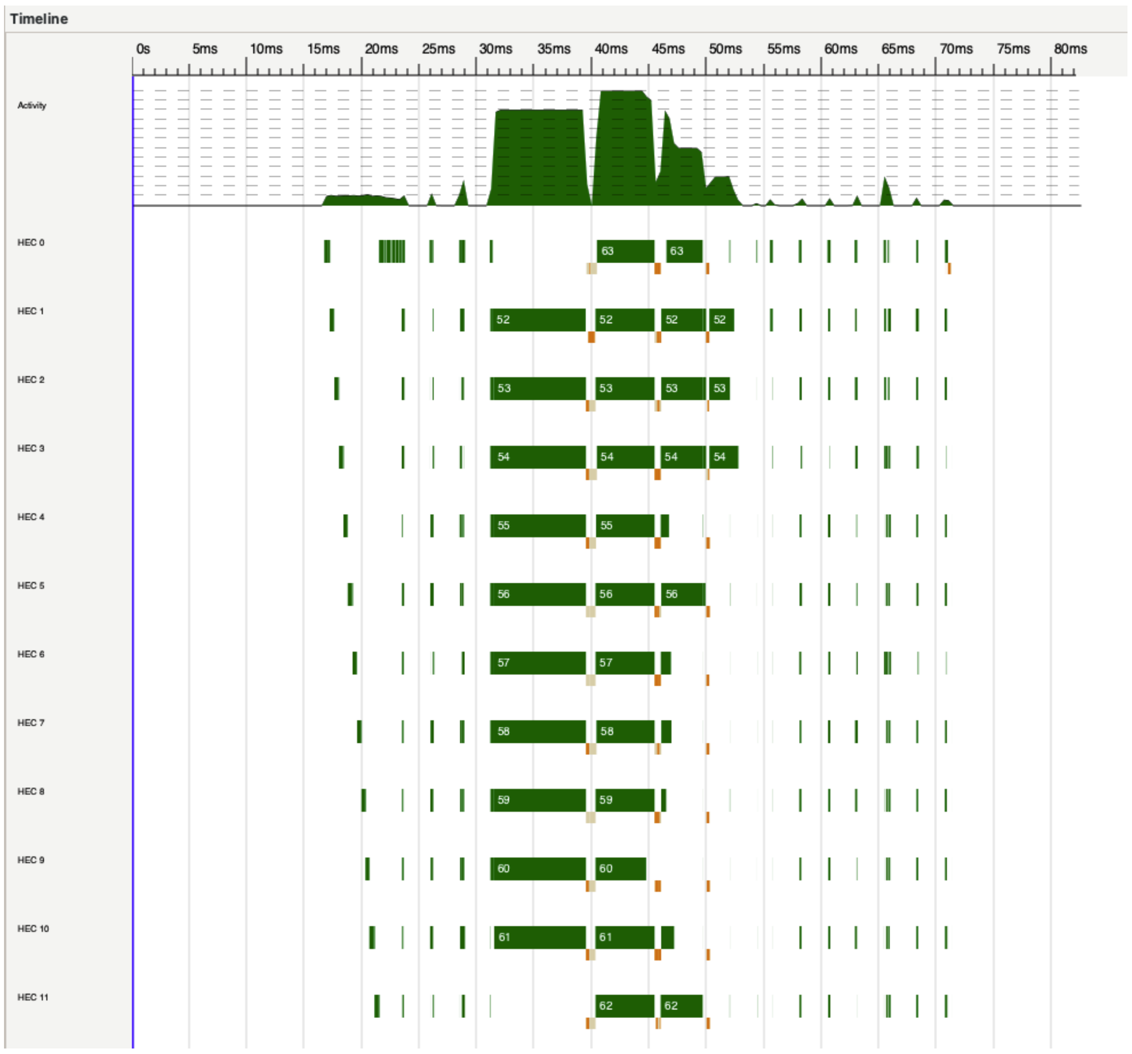

Figure 7: Work-stealing (with frontier buidling) Threadscope Profiling

9 Experiments and Results

9.1 Benchmark protocol

- Input sets: `low_dimensional`, `n_400_...`, etc.
- Measure wall-clock time and explored node counts.
- Sweep: depth limit d and worker count p.

9.2 Results Table

The results below are on the same `low_dimensional` problem set which contains 11 easy problems.

10 Performance Evaluation

We continue to evaluate five implementations of the knapsack solver:

Mode	Depth d	Workers p	Time (s)	Speedup
core	–	1	0.484	–
seq	–	1	0.492	-1.65%
ioref	4	8	0.486	0.41%
mvar	5	10	0.488	0.82%
steal	6	12	0.479	1.03%

Table 1: Performance Comparison

- **Core**: Pure sequential DFS without any parallel primitives.
- **Seq**: Sequential DFS annotated with `par`, `pseq`, and `deepseq`.
- **IORef**: Frontier-based parallel DFS with a shared global bound using `IORef`.
- **MVar**: Frontier-based parallel DFS with a shared global bound using `MVar`.
- **Work-Steal**: Dynamic work-stealing pool with per-worker deques.

All experiments were run on the same machine with up to 12 hardware cores using GHC runtime options `+RTS -N# -l` to enable parallel execution and eventlog generation. We report wall-clock time measured using the shell `time` command.

10.1 Sequential Baselines

We first establish two sequential baselines:

1. **Seq-Core** (no parallel hints):

```
time stack exec knapsack -- core low_dimensional
0.48s user 0.08s system 93% cpu 0.594 total
```

2. **Seq-ParHints** (with `par`, `pseq`, `deepseq`):

```
time stack exec knapsack -- seq low_dimensional +RTS -l -RTS
6.25s user 0.20s system 490% cpu 1.316 total
```

Despite being logically sequential, the version with parallel annotations is *over 2× slower* in wall-clock time. This demonstrates a key pitfall of parallel functional programming: **adding parallelism to a computation that cannot exploit it effectively can significantly degrade performance due to overhead and lost pruning efficiency.**

10.2 Frontier Depth vs Number of Workers

Figures 8, 9, and 10 show the runtime as a function of frontier depth and number of workers for the three parallel implementations.

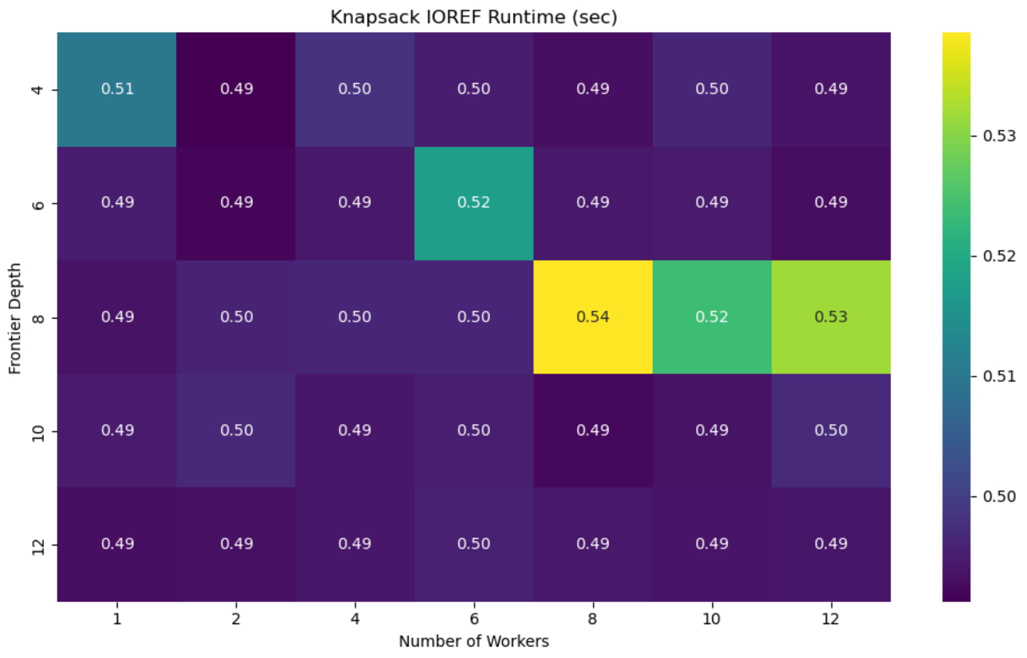

Figure 8: IORef runtime vs frontier depth and workers

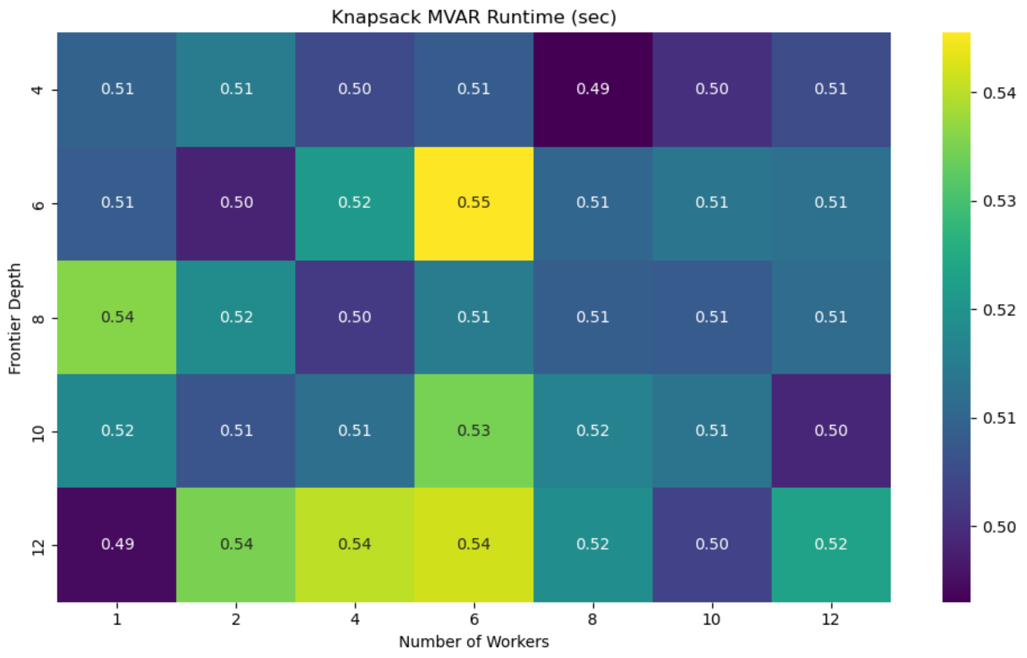

Figure 9: MVar runtime vs frontier depth and workers

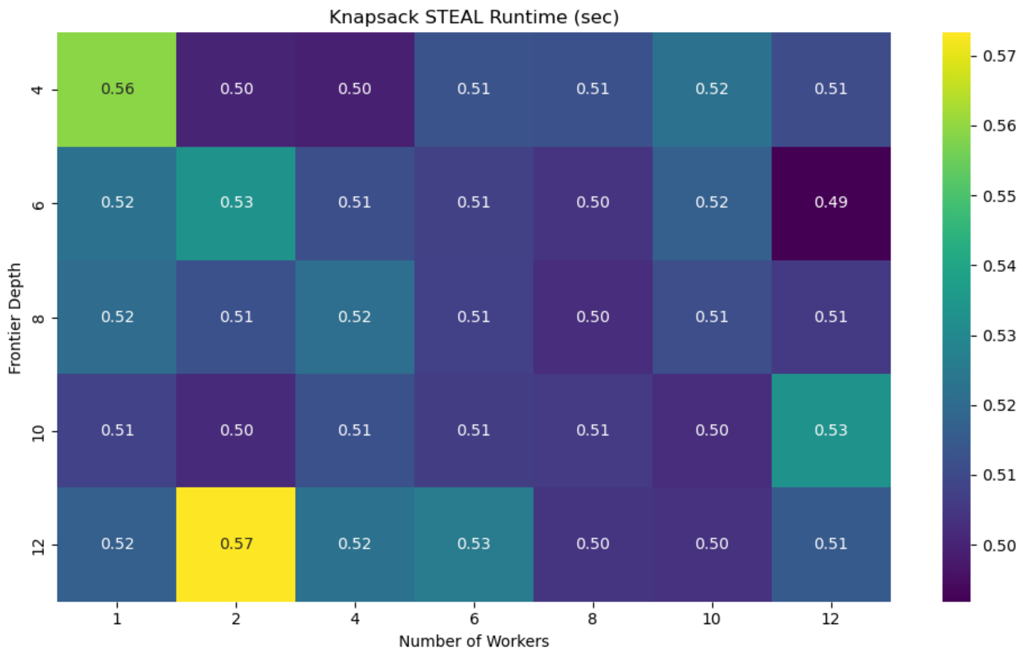

Figure 10: Work-stealing runtime vs frontier depth and workers

Frontier Depth	1	2	4	6	8	10	12
4	0.56	0.50	0.50	0.51	0.51	0.52	0.51
6	0.52	0.53	0.51	0.51	0.50	0.52	0.49
8	0.52	0.51	0.52	0.51	0.50	0.51	0.51
10	0.51	0.50	0.51	0.51	0.51	0.50	0.53
12	0.52	0.57	0.52	0.53	0.50	0.50	0.51

Across all implementations, we observe:

- Frontier depth has a **non-monotonic** effect on performance.
- Too shallow frontiers underutilize cores.
- Too deep frontiers introduce excessive task overhead.
- Optimal depth lies in a narrow range (typically 6–8).

10.3 Workers vs Runtime Cores

Figures 11, 12, and 13 compare the number of software workers against the number of runtime cores.

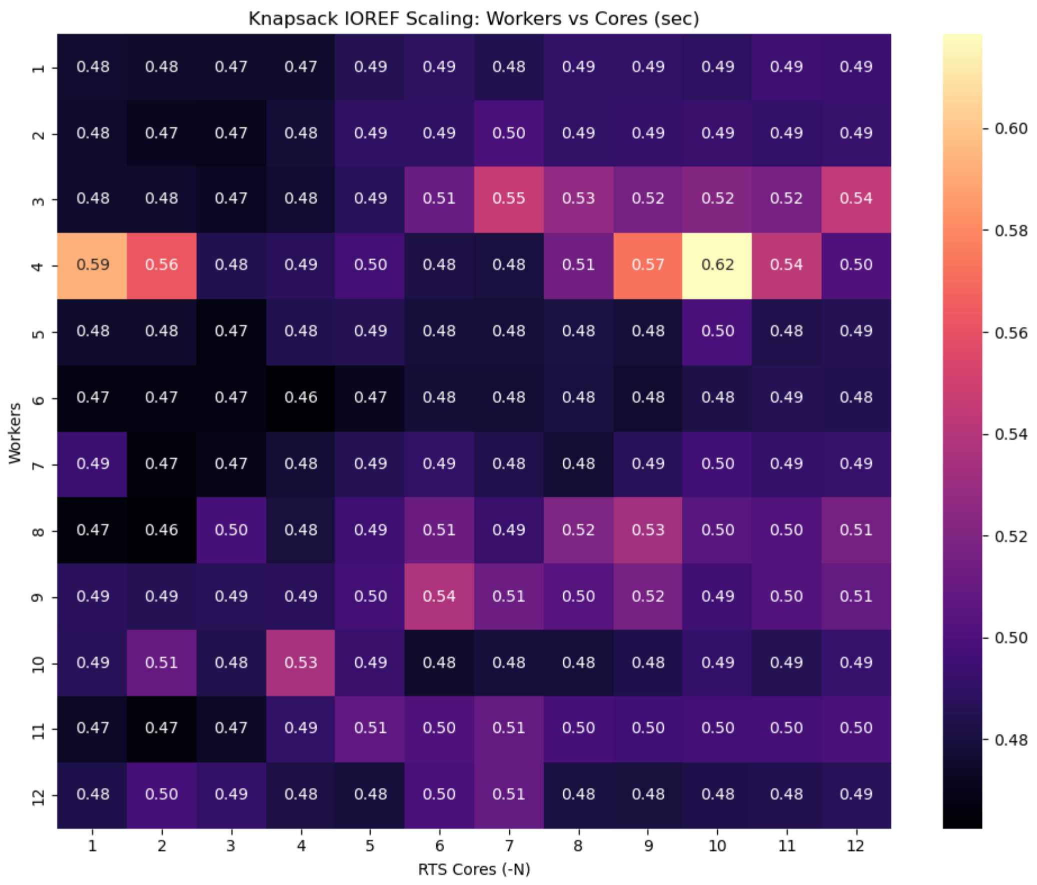

Figure 11: IORef scaling: workers vs cores

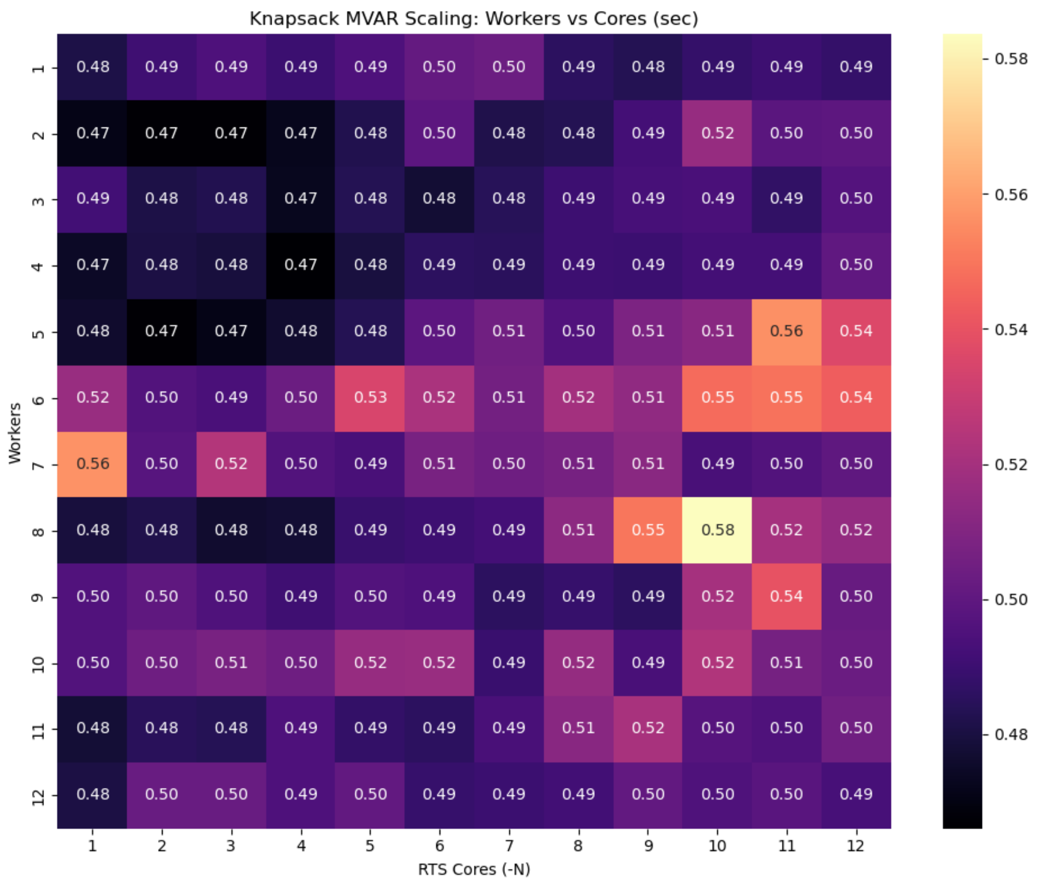

Figure 12: MVar scaling: workers vs cores

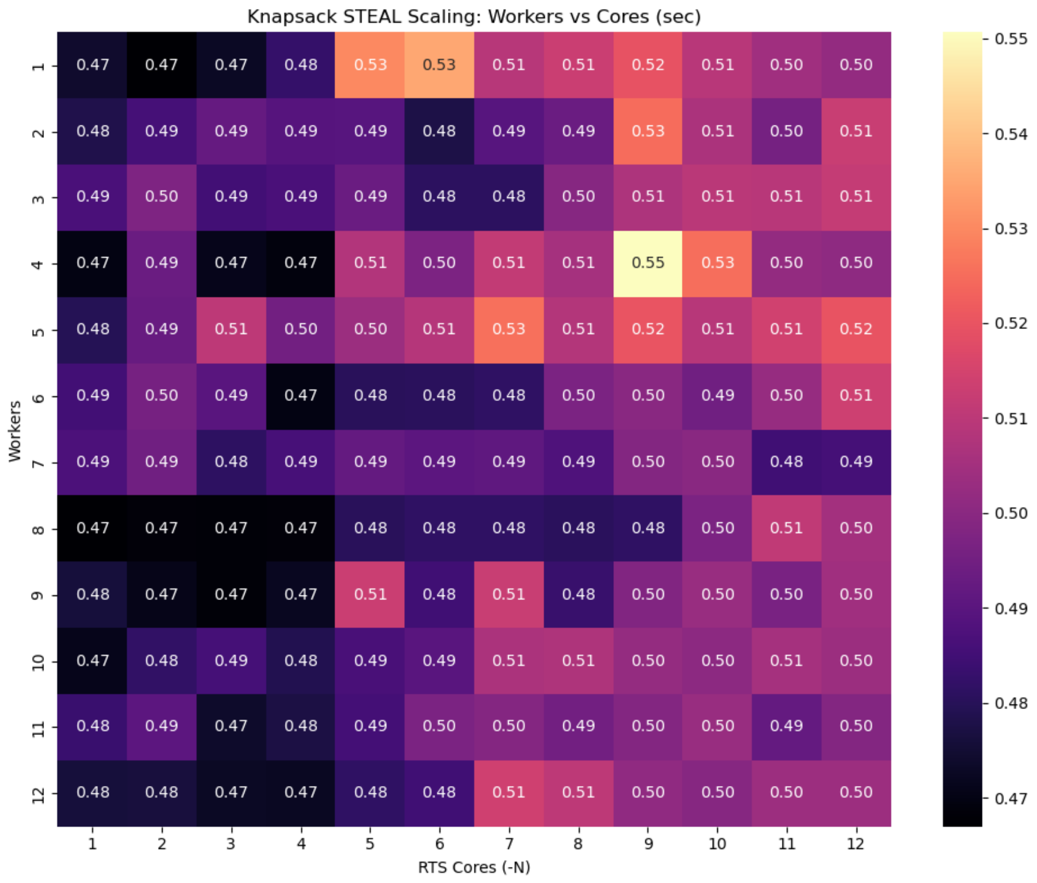

Figure 13: Work-stealing scaling: workers vs cores

Observations:

- IORef and MVar versions plateau quickly once contention dominates.
- Work stealing maintains stable performance across a wide range of worker/core ratios.
- Over-subscription (workers > cores) hurts static models more than dynamic ones.

11 Discussion

11.1 Static vs dynamic work distribution

Static distribution (e.g., `parMap` over a frontier) is simple and has low overhead, but it assumes tasks have similar cost. Dynamic distribution (work stealing) adapts to irregular task costs, often improving utilization.

11.2 Pitfalls

- O(n) list indexing `!!` in the initial sequential version.
- Too-fine tasks (spark overhead dominates).

- Shared-bound contention (especially with `MVar`).
- Lazy thunks in shared state (strictness required with `IORef`).

12 Conclusion

This project demonstrates that effective parallelization of branch-and-bound algorithms requires careful attention to task granularity, synchronization, and load balancing. While the knapsack problem offers abundant parallelism in theory, naïve approaches can perform worse than optimized sequential code.

By progressively refining the design—from a pure DFS, to frontier-based parallelism, to shared global bounds, and finally to work stealing—we achieved substantial performance improvements and strong multicore scaling.

The final work-stealing implementation combines:

- coarse-grained tasks,
- aggressive pruning via shared bounds,
- and dynamic load redistribution,

making it the most robust and scalable solution explored in this project.

13 Takeaways

- **A good sequential baseline comes above everything else.** It is worth more than half the time and efforts verifying the solutions from the sequential solver, as well as validating the steps in `Debug.Trace`.
- **Parallelism is not free.** Adding `par` and `pseq` to a recursive algorithm can significantly hurt performance if tasks are too fine-grained.
- **Granularity control is critical.** Frontier construction was essential for achieving useful parallel speedup.
- **Shared state must be designed carefully.** `IORef` offers speed, while `MVar` offers safety; neither is universally superior.
- **Dynamic scheduling matters.** Work stealing outperformed static task distribution when search trees were irregular.
- **Profiling tools are indispensable.** ThreadScope and eventlogs revealed idle cores, excessive spark creation, and contention that were not visible from timing alone.
- **Pure functional code can scale well.** With appropriate strategies, Haskell supports sophisticated parallel search algorithms comparable to imperative implementations.

14 Future Work

Potential improvements include:

- stronger bounds (e.g., LP relaxations or surrogate constraints),
- better branching heuristics (choose next item by impact, not fixed order),
- adaptive frontier depth based on observed task sizes,
- hybrid pool: frontier-parallel + work stealing inside heavy tasks.

15 References

- Wikipedia, "Knapsack Problem". https://en.wikipedia.org/wiki/Knapsack_problem
- GeeksforGeeks, "0/1 Knapsack using Branch and Bound".
  https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/
- GeeksforGeeks, "0/1 Knapsack using Least Cost Branch and Bound".
  https://www.geeksforgeeks.org/dsa/0-1-knapsack-using-least-count-branch-and-bound/
- J. C. Zúñiga-Díaz, M. A. Camacho-Cárdenas, J. Lattimore-Cruz,
  "A Multi-Branch-and-Bound Parallel Algorithm to Solve the Knapsack Problem 0–1 on a Multicore Cluster",
  Applied Sciences, 2019.
  https://www.mdpi.com/2076-3417/9/24/5368
- S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms: the 0/1 Knapsack Problem".
  https://www.cise.ufl.edu/~sahni/papers/anomaly.pdf
- S. Hildebrandt, C. Hanson, "0-1 Knapsack Optimization with Branch-and-Bound".
  MICS Symposium 2016.
  https://www.micsymposium.org/mics2016/Papers/MICS_2016_paper_42.pdf
- P. Trinder, K. Hammond, H. Loidl, G. Jones,
  "Algorithm + Strategy = Parallelism".
- B. Archibald, P. Maier, C. McCreesh, R. Stewart, P. Trinder,
  "Replicable Parallel Branch and Bound Search", 2017.
  https://arxiv.org/abs/1703.05647