Parallel Beam Search for Combinatorial Optimization (Extended Abstract)

Nikolaus Frohner[1]*, Jan Gmys[2], Nouredine Melab[2], Günther R. Raidl[1], El-ghazali Talbi[2]

[1]Institute of Logic and Computation, TU Wien, Vienna, Austria
[2]University of Lille, Inria Lille - Nord Europe, BONUS, France
{nfrohner, raidl}@ac.tuwien.ac.at, jan.gmys@inria.fr, {nouredine.melab, el-ghazali.talbi}@univ-lille.fr

Introduction

Over the last decade, the increase in computational power can be largely attributed to parallelization, while single-threaded performance tends to saturate. Clusters with hundreds of nodes equipped with multiple CPUs and GPUs consisting of thousands of cores become available to more and more researchers all over the world. With semiconductor geometries still shrinking we can expect the core density to still grow even further over the next years.

Accordingly, solution approaches to difficult combinatorial optimization problems were designed and implemented to benefit from these technological transitions. One solution paradigm is to formulate a problem recursively and model the solution process as traversal of a search tree, for instance as done by classic branch-and-bound (B&B) algorithms or A* search. Many frameworks for parallel tree search have been proposed, for instance in Xu et al. (2005) and Archibald et al. (2020), but the focus so far lied more on exact approaches. We seek to contribute to bridging this gap and propose a heuristic parallel tree exploration approach, namely *parallel beam search*. Beam search is a well-known search method where a tree is traversed layer-wise keeping a bounded number of nodes in each layer, resulting in a polynomial number of nodes to be evaluated—a truncated breadth-first-search.

Beam search has been used to construct high-quality feasible solutions in the context of branch-and-bound, standalone, or combined in a hybrid setting with an improvement heuristic like local search. Quite recently, strong results have been obtained on difficult scheduling problems, see, e.g., Libralesso et al. (2021) on Permutation Flow Shop Scheduling (PFSP) and Frohner, Neumann, and Raidl (2020) on the Traveling Tournament Problem (TTP). The most crucial parts are always the evaluation of the nodes, the *guidance*, and the *beam width*, limiting the maximum width of the search tree.

We propose a framework for combinatorial optimization problems which admit a recursive formulation, i.e., where there exists the notion of partial solutions with a corresponding state which we can evaluate to guide our search. While at first focusing on CPU execution models, we already have heap-less systems with preallocated memory regions in mind to pave the way for GPU implementations. Parallelization is performed on an intra-node level with shared memory, where the work in form of the beam's node expansions and transitions to successors is split evenly among threads. While this is a rather straightforward data parallelism approach, the implementation details to achieve high parallel efficiency over a broad set of problems are non-trivial and the focus of our research.

We provide a work-in-progress implementation published on GitHub[1] in the Julia programming language and show the concrete implementations for two well-known NP-hard optimization problems, namely Permutation Flow Shop Scheduling (PFSP) with flowtime objective and the Traveling Tournament Problem (TTP), based on the state-of-the-art approaches from the literature. In the remainder of this extended abstract, we give a concise description of the algorithm and first preliminary results on parallelization of the PFSP, where we observe a parallel efficiency with 32 cores of over 95% on sufficiently large problem instances and beam widths, resulting in a speed-up of over $30\times$.

More details of our approach including results on the TTP will be provided in a related full publication. Future work is concerned with the parallelization of state duplicate checking and dominance filtering (see Blum, Blesa, and Lopez-Ibanez (2009)) and to study implementations for more problems within our framework, for instance the maximum independent set problem and string problems.

Parallel Beam Search

In this section, we present the algorithmic details of our framework and also implementation details which we observed are important to achieve higher parallel efficiency in Julia. The main idea of beam search is to traverse a search graph layer by layer and keep in every layer the β most promising nodes and therefore consider many partial solutions in parallel. If the evaluation function, also called guidance, to rank the node runs in polynomial time, the whole construction algorithm does. Often the nodes carry additional state information to facilitate an incremental evaluation of its successors. Relevant parts determining the runtime are said evaluation of successor nodes, determining the

*Corresponding author
Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]https://github.com/nfrohner/parbeam

Algorithm 1: Data-Parallel Beam Search Algorithm

Input: instance C, auxiliary data A, beam width β, guidance function b, beams $Q_{\mathrm{up}}, Q_{\mathrm{down}}$, successor region Q_{S}
Output: feasible schedule $\mathbf{d}$

$\mathbf{d}^{\mathrm{best}} = (), Q \leftarrow Q_{\mathrm{up}}, Q_{\mathrm{next}} \leftarrow Q_{\mathrm{down}}, n_{\mathrm{Q}} \leftarrow 1$;
initialize root node in $Q[1]$;
for $l \leftarrow 1$ **to** $l_{\max}(C)$ **do**
  $n_{\mathrm{S}} \leftarrow$ create-successors(C, A, n_{Q}, Q, Q_{S}, β, b);
  $n_{\mathrm{Q}} \leftarrow$ process-successors(C, l, n_{S}, Q_{S}, Q, Q_{next});
  if $n_{\mathrm{Q}} = 0$ **then**
    `// no surving successor`
    return $\mathbf{d}^{\mathrm{best}}$;
  end
  $\mathbf{d}^{\mathrm{best}} \leftarrow$ check-for-terminals(C, Q_{next}, n_{Q}, $\mathbf{d}^{\mathrm{best}}$);
  $Q, Q_{\mathrm{next}} \leftarrow$flip-beams($Q_{\mathrm{up}}$, Q_{down});
end
return $\mathbf{d}^{\mathrm{best}}$;

β best ones, and performing the actual transitions to the next layer.

We make use of a fixed-length representation of solutions and static memory layout, which allows to greatly reduce heap operations and therefore the work of the garbage collector in Julia.[2] Each incremental successor information is stored in a predefined slot of its preallocated memory region, depending on the index of its parent in the beam. The storage is potentially sparse since fewer than the maximum number of successors might be stored due to constraints, depending on the parent state. The storage of the nodes in the beam is dense due to their comparably larger memory footprint.

In Algorithm 1, we give a high-level description of the intra-node parallel beam search procedure. It receives the preprocessed instance C and corresponding auxiliary data A as input, along with preallocated memory regions for the nodes and the successors $Q_{\mathrm{up}}, Q_{\mathrm{down}}, Q_{\mathrm{S}}$, respectively, and search parameters beam width β, and the guidance function b. The algorithm returns the best found solution or the empty solution, if no solution was found, which can happen for constrained problems. The parallelized functions *create-successors* and *process-successors* call functions themselves to evaluate successor nodes and make the actual transitions on concrete data types, which is the problem-specific part that has to be implemented by the user of the framework. Both functions return the number of successor information entries evaluated and the number of actually created successors after the processing. The selection of the β best successors is performed by a parallel histogram approach over the nodes' values. We flip between two preallocated beams, each represents alternatively the current or the next layer.

To distribute the work across threads, we make use of the Julia *Threads* module, which allows loop parallelization with static scheduling of consecutive chunks of data to threads. A data parallel pattern is employed, where each thread writes into its own private memory region and, if necessary, these are combined in a final sequential step to achieve the desired result, e.g., to calculate the minimum and maximum over an array of numbers. A pitfall is *false sharing*, where threads interfere with each other due to overlapping cache lines of seemingly independent memory regions. Furthermore, Julia stores arrays in column-major ordering, hence for two dimensional arrays, the thread dimension has to be second and a safety buffer of a cache line length is added between the data.[3] We observed that the binning of the histogram step would otherwise not parallelize at all with even slightly increasing runtimes when multithreaded.

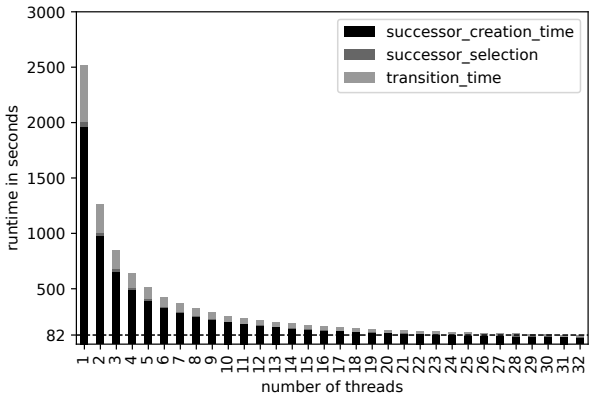

Figure 1: Actual distribution of the runtime over the most relevant algorithmic parts, successor creation, selection, and transition for PFSP parallel beam search on a VFR instance with 100 jobs, 40 machines, and a beam width of 640 000.

Preliminary Results

We implemented a single-shot version of the iterative widening beam search approach by Libralesso et al. (2021) for the flowtime variant of the PFSP within our framework. The achieved results over the famous Taillard benchmark (Taillard 1993) are shown to be competitive (Libralesso et al. 2021) with their introduced guidance which combines a layer-dependent weighted sum of the machines' idle times and the costs-so-far.

We ran experiments on a machine with a single-socket AMD EPYC 7642 CPU using up to 32 cores and uniform memory access on 512 GiB of RAM.[4] In Figure 1 we see the runtime (w/o Julia's startup) distribution over the number of threads for an example instance of the more recent VFR benchmark (Vallada, Ruiz, and Framinan 2015) with 100 jobs, 40 machines, and a beam width of 640 000. The largest portion of work, the successor creation, admits a high parallel efficiency of 97%, followed by the transition parallel efficiency of also 97%. The selection of the β best drops down to 55%, but itself is only a small part of the runtime resulting into the total parallel efficiency of 95% with a related speed-up of over $30\times$.

[2] https://docs.julialang.org/en/v1/manual/performance-tips/

[3] https://juliafolds.github.io/data-parallelism/

[4] https://www.grid5000.fr/w/Lyon:Hardware#neowise

Acknowledgements

This project is partially funded by the Doctoral Program "Vienna Graduate School on Computational Optimization", Austrian Science Foundation (FWF) Project No. W1260-N35. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr)

References

Archibald, B.; Maier, P.; Stewart, R.; and Trinder, P. 2020. YewPar: skeletons for exact combinatorial search. In *PPoPP '20: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 292–307. Association for Computing Machinery.

Blum, C.; Blesa, M. J.; and Lopez-Ibanez, M. 2009. Beam search for the longest common subsequence problem. *Computers & Operations Research*, 36(12): 3178–3186.

Frohner, N.; Neumann, B.; and Raidl, G. R. 2020. A Beam Search Approach to the Traveling Tournament Problem. In Paquete, L.; and Zarges, C., eds., *Evolutionary Computation in Combinatorial Optimization – 20th European Conference, EvoCOP 2020, Held as Part of EvoStar 2020*, volume 12102 of *LNCS*, 67–82. Springer.

Libralesso, L.; Focke, P. A.; Secardin, A.; and Jost, V. 2021. Iterative beam search algorithms for the permutation flowshop. *European Journal of Operational Research*. In press.

Taillard, E. 1993. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2): 278–285.

Vallada, E.; Ruiz, R.; and Framinan, J. M. 2015. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240(3): 666–677.

Xu, Y.; Ralphs, T. K.; Ladányi, L.; and Saltzman, M. J. 2005. Alps: A framework for implementing parallel tree search algorithms. In *The next wave in computing, optimization, and decision technologies*, 319–334. Springer.