# Parallel Flowshop in YewPar

Ignas Knizikevičius[1], Phil Trinder[1], Blair Archibald[✉1], and Jinghua Yan[2]

[1] School of Computing Science, University of Glasgow, Scotland
{phil.trinder, blair.archibald}@glasgow.ac.uk
[2] School of Computing, University of Utah, United States of America
jhyan@cs.utah.edu

**Abstract.** Parallelism may reduce the time to find exact solutions for many Operations Research (OR) problems, but parallelising combinatorial search is extremely challenging. YewPar is a new combinatorial search framework designed to allow domain specialists to benefit from parallelism by reusing sophisticated parallel search patterns.
This paper shows (1) that it is low effort to encode and parallelise a typical OR problem (Flowshop Scheduling FSP) in YewPar even for scalable clusters; (2) that the YewPar library makes it extremely easy to exploit three alternate FSP parallelisations; (3) that the YewPar FSP implementations are valid, and have sequential performance comparable with a published algorithm; and (4) provides a systematic performance evaluation of the three parallel FSP versions on 10 standard FSP instances with up to 240 workers on a Beowulf cluster.

## 1 Introduction

Flowshop Scheduling (FSP) is a classic OR problem: $n$ jobs are to be processed on $m$ machines with the processing time of any job on any machine specified. The goal is to find a sequence of jobs that has the least makespan, or time to complete the jobs. Optimal FSP solutions can be found by exploring the entire search space, but this is computationally expensive. Approximation strategies reduce runtime by exploring only part of the search space and, despite being unable to guarantee optimality, are commonly used.

The runtimes of exact combinatorial search problems like FSP can be reduced (1) by using Branch and Bound (B&B) techniques that dynamically explore the search space and discard (prune) branches that cannot contain solutions; and (2) by exploiting parallelism. However parallelising exact combinatorial search is extremely challenging due to the huge and highly irregular search trees, the need to preserve search order heuristics, and pruning that dynamically alters the workload.

YewPar [5] is a new combinatorial search framework designed to allow domain specialists to benefit from parallelism by reusing sophisticated parallel search patterns (subsection 2.1). As a performance baseline we develop a direct exact sequential FSP solver in C++.

The research contributions of this paper are as follows, and **the key results are summarised in section 6**.

1. **We show that minimal effort is required to encode a typical OR problem (FSP) in YewPar.** The direct and sequential YewPar solvers implement a standard FSP algorithm, and both require around 390 source lines of code (SLOC), much of which is shared. Although a simple Travelling Salesperson is reported in [3], FSP is the first non-trivial OR application implemented using YewPar (section 3).

2. **We show that YewPar makes it extremely easy to experiment with three sophisticated search parallelisations on shared and distributed memory architectures.** An alternate parallelisation simply entails a few lines of code that configure an appropriate YewPar skeleton, and the selection of suitable parameters. While *StackStealing* [10] and *DepthBounded* search coordinations [15] have been widely used, we report *the first use of Budget parallel search coordination for FSP* (subsection 4.4).

3. **We validate the solvers using 100+ standard instances; and baseline the sequential performance** against the recent LL solver [10] showing comparable performance. YewPar is on average 5% faster than LL, but 9% slower than the direct FSP solver (section 4).

4. **We systematically measure the performance of the three parallel FSP searches on 10 Taillard instances with up to 240 workers** on a Beowulf cluster. We show that YewPar increases the number of instances that can be practically solved, e.g. one instance runtime falls from 2.4 days to 16 minutes. Good absolute speedups (up to 297×) are achieved for large instances (sequential runtimes between 4 and 58 hours). Of the three parallelisations *Budget* delivers the best performance, with mean speedup over 6 large Taillard instances of 246× (section 5).

## 2 Background

Job-Shop scheduling optimises the scheduling of $n$ jobs onto $m$ machines while maintaining job precedence constraints. In Flowshop scheduling the precedence constraints are identical for all jobs and the goal is to minimise the *makespan*—the time to complete all jobs. There are a variety of solution techniques [2] and here we seek exact solutions that both identify a solution *and* prove that no better solution exists. One exact technique uses a branch-and-bound search of the tree of all possible solutions, with a bounding function that eliminates sub-trees that cannot contain an improved solution. Each search tree node represents a (partial) schedule, and the tree is never fully manifest in memory: it is *generated* during the search.

Parallelising the search can reduce runtimes by concurrently exploring subtrees. While there is huge scope for parallelism, parallel tree search is far from trivial. We need to determine *how* and *when* a tree should be split; subtree sizes vary hugely and work must be balanced between cores; global knowledge, e.g. new bounds, must be shared. The parallel coordination must be added to the search code, and is often so intrusive that only a single parallelisation is developed. Moreover parallelisation typically targets the less-challenging shared-memory architectures, but this limits scalability to around 100 cores.

## 2.1 YewPar

YewPar is the *"the first general purpose scalable parallel framework for exact combinatorial search"* [5] and is designed to make it easier to parallelise search applications. YewPar offers reusable generic parallel search patterns (algorithmic skeletons), that only require user parameterisation with sequential functions. It has been used to parallelise 7 different search applications using 3 different parallelism techniques [5]. Other general search frameworks include TaskWork [13] and MaLLBa [1], but these support only a single search coordination.

To separate search tree generation from parallelism (tree exploration) a YewPar user specifies their search application as a *Lazy Node Generator* (LNG). A LNG specifies how, for a given node, a heuristically ordered list of child nodes is created. Lazy generation minimises memory overheads and avoids generating subtrees that are pruned.

## 2.2 Search Coordinations

Search coordinations specify parallel work generation and distribution strategies. YewPar currently supports four well-known search coordinations as follows. Detailed specifications of the YewPar search coordinations, including a parallel operational semantics for each, are available in [5].

**Sequential** performs a depth-first search from the root of the search tree. It executes in a single thread and is useful for performance baselining and debugging LNG implementations.

**Depth-Bounded** creates a new task for any search tree node higher than a user-specified cut-off depth $d_{cutoff}$ with tasks shared using locally-biased distributed random work-stealing. The intuition is that tasks created near the root of the search tree will undertake significant searches, i.e. have long runtimes. Selecting a depth enables the user to control the number of tasks created.

**Budget** creates new tasks to be stolen every time a search task executes *budget* back-tracks. The intuition is that search tasks that have a long runtime can usefully spawn other search tasks to help (the task has not instantly been pruned for example). Selecting a budget enables the user to control the size of the search tasks.

Budget is similar in spirit to restarting searches, but instead of starting the search afresh after a timeout (a budget), it creates new tasks that continue the search from the (top-of the) current subtree. Unlike a restart these new tasks do not necessarily execute immediately, as they may be queued waiting for a worker to become available. Parallelising searches using restarts is an ongoing research focus, e.g. [4].

**Stack-Stealing** is a dynamic work generation approach where idle worker-threads request work directly from other workers (aka. work stealing). On receipt of a work-stealing request a worker may either respond that it has no work available, or send a node the search tree as close to the root as possible. That is, it also assumes that tasks created near the root of the search tree have long runtimes, and hence are worth communicating.

### 2.3 Other Parallel FSP searches

Many parallel FSP searches are handcrafted, in contrast our impementation uses the general purpose YewPar framework. A master-worker paradigm is common where tasks and new knowledge are stored on a *centralised* master and distributed dynamically to workers, e.g. [7,15]. To overcome the centralisation bottleneck decentralised approaches use work-stealing, not unlike the StackStealing coordination, e.g. [21,20].

Some recent permutation FSP implementations have obtained excellent performance on GPUs, sometimes exploiting $10^4$ GPU cores e.g. [9]. Some of these approaches exploit a specific *Integer-Vector-Matrix* (IVM) data-structure to enable efficient parallelism both on CPUs and hybrid CPU/GPU architectures, e.g. [10,11]. Compared with YewPar, GPU and hybrid CPU/GPU programming requires significantly more developer effort.

Crucially previous parallelisations are similar to DepthBounded and StackStealing search coordinations, but *we are not aware of any prior parallelisation using a Budget coordination.*

## 3 FSP Implementations

We engineer two entirely standard FSP solvers for comparison purposes: a *direct* sequential C++ implementation, and a YewPar lazy node generator that supports sequential, and three parallel searches: one for each coordination. Our implementations [14] are branch-and-bound and use the common approach of maintaining two partial schedules $\sigma_1$ (initial schedule) and $\sigma_2$ (final schedule) [16,11]. They are competitive rather than state-of-the-art FSP solvers.

Branching adds an unscheduled job (chosen in ascending numerical order) $j$ to either $\sigma_1$ or $\sigma_2$. The choice of $\sigma_1$ or $\sigma_2$ is based on the *Alternate* rule [11] that adds $j$ to $\sigma_1$ if the current tree depth is odd, else $\sigma_2$. We use the simple *One Machine Bound* of Ignall and Schrage [12] that has low runtime at the cost of some tightness.

Finally, we use the NEH approximation algorithm [17] to compute an initial upper bound before search.

**Listing 1.1.** YewPar StackStealing FSP Search

```
sol = StackStealing<GenNode, Optimisation, BoundFunction<bound_func>,
    ObjectiveComparison<std::less<unsigned>>>::search(space, root, params);
```

**Listing 1.2.** YewPar Budget FSP Search

```
params.backtrackBudget = opts["backtrack-budget"].as<unsigned>();
sol = Budget<GenNode, Optimisation, BoundFunction<bound_func>,
    ObjectiveComparison<std::less<unsigned>>>::search(space, root, params);
```

### 3.1 Comparing YewPar and Direct Solver Programs

The direct implementation requires some 370 source lines of code (SLOC) [14], as counted with CLOC. In comparison, the YewPar Sequential implementation requires 390 SLOC [14]. It is easy to refactor an existing search to a Lazy Node Generator as much of the code, e.g. bounding functions is shared. Parallelising a YewPar search entails specifying reusable serialisation functions (30 SLOC). Thereafter specifying an alternate parallel search requires only a few lines of code, and Listings 1.1 and 1.2 show the YewPar stack stealing and budget FSP parallelisations.

## 4 Validation, Baselining and Parameter Tuning

### 4.1 Experimental setup

Experiments are run on a Beowulf cluster consisting of 16 nodes (256 cores). Each node uses Ubuntu 14.04.3 LTS, has 2 Intel Xeon E5-2640v2 2GHz CPU (no hyper-threading), 64GB of RAM, and a 10Gb Ethernet Interconnect. For YewPar each node dedicates one core to management tasks meaning that each node has 15 worker threads (workers) doing sub-tree search.

Performance analysis of parallel searches is notoriously difficult due to the non-determinism caused by pruning, finding alternate valid solutions, and random work-stealing. These lead to performance anomalies [8] that manifest as slowdowns or superlinear speedups. We control for this by investigating multiple FSP instances, running each experiment multiple times, and reporting cumulative statistics.

**Table 1.** 10 Taillard validation examples, and baselining YewPar against the direct and LL solvers [10].

| Instance | Make-span | Direct solver runtime(s) | YewPar runtime(s) | YewPar:Direct Slowdown | LL runtime(s) | YewPar:LL Slowdown |
|---|---|---|---|---|---|---|
| Ta21 | 2297 | 96590 | 104928 | 8.6% | 24489 | 328.5% |
| Ta22 | 2099 | 5039 | 5479 | 8.7% | 11758 | -53.4% |
| Ta23 | 2326 | 192641 | 208976 | 8.5% | 79322 | 163.5% |
| Ta24 | 2223 | 12732 | 13758 | 8.1% | 19753 | -30.3% |
| Ta25 | 2291 | 16425 | 17963 | 9.4% | 25332 | -29.1% |
| Ta26 | 2226 | 32744 | 35302 | 7.8% | 34562 | 2.1% |
| Ta27 | 2273 | 39013 | 42421 | 8.7% | 28295 | 49.9% |
| Ta28 | 2200 | 1022 | 1119 | 9.5% | 4569 | -75.5% |
| Ta29 | 2237 | 1546 | 1688 | 9.2% | 3674 | -54.1% |
| Ta30 | 2178 | 1139 | 1237 | 8.6% | 898 | 37.8% |
| Geometric mean | | | | 8.7% | | -4.72% |

## 4.2 Validation

The "correctness" of the direct and YewPar FSP solvers is validated by comparing their makespans with published results for 107 well-known instances including the Carlier instances [6], the first thirty Taillard instances [18], and VRF instances [19] up to 20 jobs, 15 machines, excluding one instance with runtime over 12 hours [14]. Table 1 illustrates some Taillard instances (21-30), that have significant runtime and published performance results [10]. For all instances the direct and YewPar solvers visits an equal number of search tree nodes, showing that the search algorithms are nearly identical.

## 4.3 Sequential Performance Baselining

Columns 3, 4 and 5 of Table 1 report the direct and YewPar solver runtimes, and the YewPar slowdown. As YewPar provides reusable search encodings we expect it to have some overheads compared with a search-specific solver like the direct solver [5]. For these 10 instances the observed slowdown varies from 7.8% to 9.5% with a geometric mean of 8.7%.

To relate YewPar performance with an existing algorithm Columns 6 and 7 of Table 1 compare the runtimes of sequential YewPar with a recent linked-list (LL) solver [10] for Taillard instances 21-30 [18]. The hardware/OS platforms are similar, specifically the core has the same 2GHz clock frequency (on a pair of 8-core Sandy Bridge E5-2650 processors), uses 32 GB of memory, and CentOS 6.5 Linux [10].

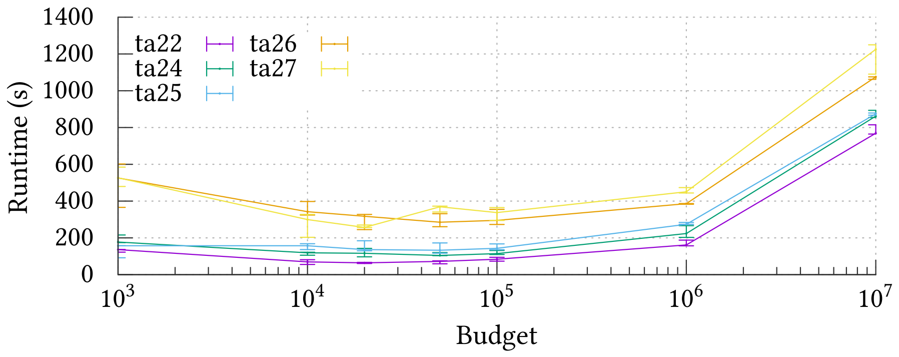

**Fig. 1.** Budget parameter sweep: runtime on 120 workers

As expected the comparative runtimes vary greatly: exactly half of the YewPar runtimes are shorter than LL. The differences are almost certainly due to the solvers using different bounding operators: YewPar uses a computationally cheap *One Machine Bound* (section 3) while LL uses a more expensive, but tighter *Two Machine Bound* [10]. A One Machine Bound may extend runtime if it fails to prune significant subtrees, but being computationally inexpensive may reduce runtimes in searches where tight bounds are less important.

As before we expect that the generic YewPar solver will incur overheads compared with a search-specific solver like LL. YewPar slowdowns vary from -75.5% to 328.5%. The geometric mean slowdown for these search instances is -4.72%, and we conclude that the sequential performance of YewPar solver is comparable to state-of-the-art solvers.

### 4.4 Parallel Search Parameter Tuning

Two YewPar parallel search coordinations require the selection of suitable parameter values, specifically the depth at which to spawn tasks in a *DepthBounded* search, and the number of backtracks that a search task should perform in a *Budget* search before generating new search tasks. We determine suitable values from a parameter sweep using Taillard instances with sequential runtimes between 1 and 12 hours. To account for random work stealing the instances are measured three times on 120 workers, and we plot lower, median, and upper values.

The *budget* parameter sweep initially considered exponentially increasing values between $10^3$ and $10^7$, as shown in Figure 1. Runtimes fall between budgets $10^3$ - $10^4$, and increase between $10^5$ - $10^7$. Additional measurements at $2 \times 10^4$ and $5 \times 10^4$ reveal small differences, and we select a budget of $5 \times 10^4$ for the evaluation in section 5. The sweep for the *depth* is similar, and identifies depth 5 as providing the best performance for these searches.

## 5 Comparing Three Parallel FSP Searches

A key benefit of the YewPar framework is that experimenting with alternative parallel coordinations is easy, requiring limited code changes (Section 3.1). This is essential as previous work [3] shows that there is no best coordination for different search applications.

This section compares the performance of *StackStealing*, *DepthBounded* and *Budget* YewPar FSP searches. We again select the Ta21-30 instances, and group them based on the sequential YewPar runtimes: *small* instances complete within 3.5 hours, and *large* within 60 hours on our cluster. As is typical for parallel workloads, the *small* instances have least scope to scale as runtimes are already low on small numbers of cores, so we initially focus on the large Taillard instances, i.e. Ta21–27 excluding Ta22. The performance of the three parallel search coordinations is shown in Figure 2. Speedups are relative to sequential YewPar runtime, and we plot ideal speedup as a dashed line. Runtimes and speedups are shown on log-log plots, with error bars representing the min/max measured runtimes.

### 5.1 StackStealing Evaluation (top plots)

While StackStealing requires no parameter tuning, it does not scale particularly well and quickly diverges from linear speedup. It is likely that many tasks are

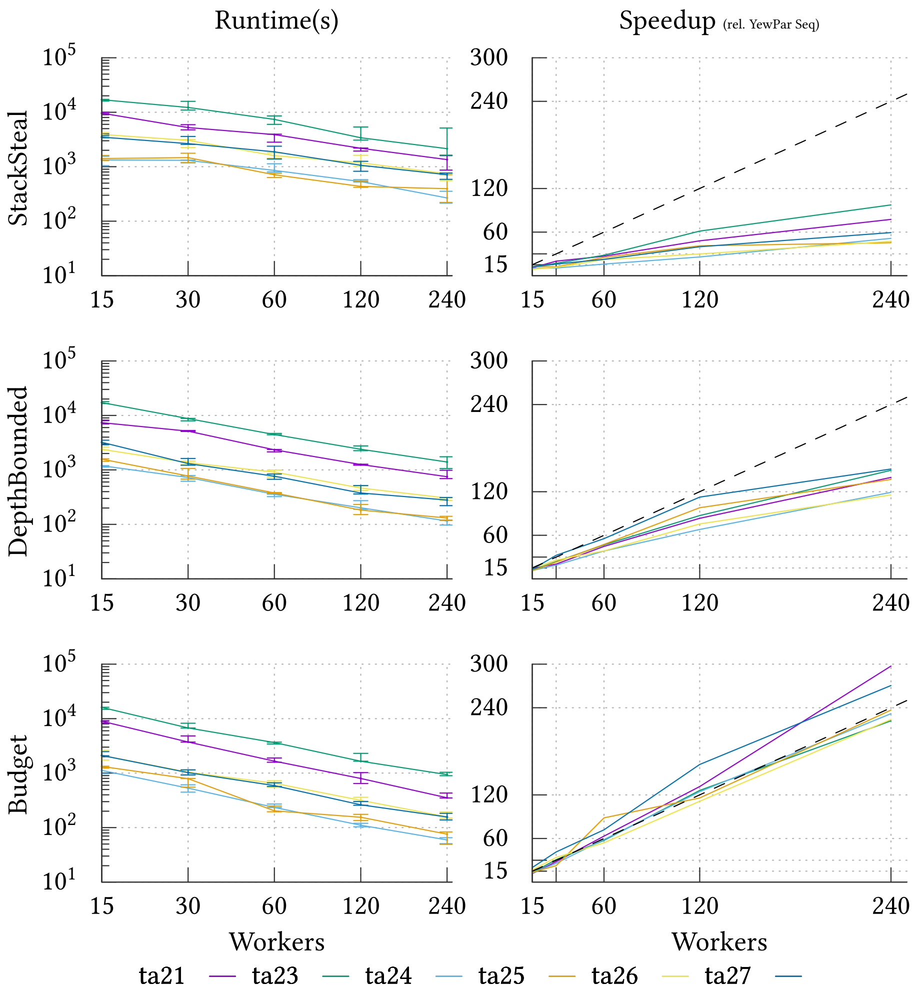

**Fig. 2.** Runtime and Speedups for 3 parallel flowshop searches: StackStealing, Depth-Bounded and Budget.

pruned early and the hardest subtrees remain on a single cluster node (effectively giving 15 workers). As steals are random it seems that remote workers don't locate hard tasks.

The range between minimum and maximum runtimes is often large (note log scale), as expected due to random work-stealing. Despite these issues, without exploiting a large cluster and without parameter tuning, runtimes are significantly reduced. For example the median runtime of Ta23 falls from 4.7 hours on 15 workers (58 hours on a single worker) to less than an hour on 240 workers.

### 5.2 DepthBounded Evaluation (middle plots)

The DepthBounded searches use the cutoff depth $d_{cutoff} = 5$ determined to be effective (subsection 4.4). DepthBounded search achieves good speedups for all instances, with no apparent limit to the scaling. As DepthBounded spawns new tasks only as a task at depth $< d_{cutoff}$ is *executed*, a single steal may generate multiple tasks improving load balance, and hence speedups. This is especially true for the relatively low $d_{cutoff}$ values used here. This is also likely why the runtime ranges are relatively small compared to StackStealing despite random work distribution.

### 5.3 Budget Evaluation (bottom plots)

The searches use a budget of $5 \times 10^4$ backtracks that was determined to be effective (subsection 4.4). Speedups are excellent: near linear in all cases, with some super-linear speedups due to knowledge transfer and speculation. We believe that Budget performs so well because it is able to generate only tasks that contain large amounts of work. Clearly this task size heuristic is more effective than the heuristics used in the other coordinations, e.g. depth. As for DepthBounded the range of runtimes is small despite random work distribution.

**Table 2.** Comparing parallel search coordinations on **Small** FSP instances with 15 or 240 workers. Smaller speedups than for larger instances, and Budget outperforms DepthBounded and StackStealing (runtimes in seconds).

| | Sequential | StackStealing | | | DepthBounded | | | Budget | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | runtime | runtime | | spdup | runtime | | spdup | runtime | | spdup |
| Instance | 1 | 15 | 240 | 240 | 15 | 240 | 240 | 15 | 240 | 240 |
| Ta22 | 5479 | 642 | 133 | 41 | 590 | 68 | 81 | 463 | 55 | 100 |
| Ta28 | 1119 | 102 | 31 | 36 | 252 | 19 | 59 | 80 | 12 | 93 |
| Ta29 | 1688 | 179 | 17 | 99 | 330 | 32 | 53 | 112 | 19 | 89 |
| Ta30 | 1237 | 142 | 20 | 62 | 281 | 22 | 56 | 95 | 19 | 65 |
| Geom. mean | | | | 55 | | | 61 | | | 86 |

### 5.4 Small Search Instances

While large search instances provide the best parallel performance, YewPar can also be applied to smaller search instances. Table 2 summarises the performance of the three parallel search coordinations for the four small Taillard instances, i.e. with sequential runtimes less that 3.5 hours. It shows runtimes in seconds and speedups relative to sequential YewPar runtime with 15 and 240 workers. 15 is the number of workers on a single shared-memory cluster node, and 240 workers is the maximum number of workers on the 16-node cluster.

Runtimes on small numbers of cores are already short, e.g. around 10 minutes on 15 cores, and hence the speedups are lower than for larger instances. Nevertheless all three search coordinations achieve mean speedups in excess of 50 for these 4 instances.

**Table 3.** Comparing parallel search coordinations on **Large** FSP instances with 15 or 240 workers. Budget outperforms DepthBounded and StackStealing (runtimes in seconds).

| | Sequential | StackStealing | | | DepthBounded | | | Budget | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | runtime | runtime | | spdup | runtime | | spdup | runtime | | spdup |
| Instance | 1 | 15 | 240 | 240 | 15 | 240 | 240 | 15 | 240 | 240 |
| Ta21 | 104928 | 9620 | 1354 | 78 | 7359 | 751 | 140 | 8988 | 353 | 297 |
| Ta23 | 208976 | 17009 | 2145 | 97 | 17251 | 1397 | 150 | 16141 | 941 | 222 |
| Ta24 | 13758 | 1331 | 268 | 51 | 1196 | 116 | 119 | 1122 | 59 | 233 |
| Ta25 | 17963 | 1415 | 395 | 46 | 1579 | 131 | 137 | 1325 | 76 | 236 |
| Ta26 | 35302 | 3914 | 745 | 47 | 2393 | 305 | 116 | 2117 | 158 | 223 |
| Ta27 | 42421 | 3501 | 715 | 59 | 3219 | 280 | 152 | 2112 | 157 | 270 |
| Geom. mean | | | | 61 | | | 135 | | | 246 |

### 5.5 Parallel Coordination Comparison

Table 3 summarises the performance of the three parallel search coordinations for the 6 large FSP instances. We compare the performance of the search coordinations considering both large and small FSP instances (Table 2).

For both large and small instances Budget outperforms the other coordinations, with significantly reduced runtimes, and the highest speedups. For example the geometric mean speedup for the large instances is more than 4 times that of StackStealing: 245× compared with 61×. For the four smaller instances it provides mean speedups over 80×. Indeed the speedups for 2 of the 10 instances are super-linear with a maximum speedup of 297×.

*DepthBounded* is next best, for the larger instances it consistently provides speedups over 115× and a mean speedup of 135×, and for the smaller instances speedups over 60×. *StackStealing* provides the worst performance with not a single instance reaching 100× speedup, and the mean for the 6 larger instances being 61×.

## 6 Conclusions

We report the first YewPar encoding of a non-trivial OR problem: a standard FlowShop Scheduling solver. The encoding is low effort: around 390 source lines of code (SLOC), compared with 370 for a direct FSP implementation (subsection 3.1). Alternate parallelisations simply entail (1) some reusable serialisation (30 SLOC), and (2) configuring an appropriate skeleton from the YewPar library: a few lines of code and the selection of suitable parameters, e.g. compare Listings 1.1 and 1.2. Moreover the *Budget* parallel search coordination is a first for FSP.

Our solvers are validated with 107 standard (Carlier, Taillard, and VRF) FSP instances. We baseline the sequential performance of YewPar against the LL state-of-the-art solver [10] on a similar hardware/OS platform. There is considerable variation in the runtimes of the Ta21 - Ta30 searches (Columns 4&6 of Table 1) as the solvers use different bounding operators. The performance of LL and YewPar is comparable for the searches: YewPar is faster in 5 out of 10 instances, and on average 5% faster. There is, however, a performance penalty for YewPar's generality: a mean 8.7% slowdown compared with the direct solver for the same 10 Taillard instances (Columns 3,4&5 of Table 1).

Systematic performance measurements of the three parallel FSP searches on a Beowulf cluster with 240 workers shows that YewPar increases the number of instances that can practically be solved, e.g. the runtime of Ta21 falls from 2.4 days to 16 minutes. YewPar also provides useful speedups for small search instances, e.g. mean speedups of over 50× in Table 2. Of the three search coordinations, the novel *Budget* coordination performs best on both large and small instances. For example on the six larger Taillard instances it consistently provides absolute speedups of over 220× and a geometric mean speedup of 246×. *DepthBounded* is next best, for the larger instances it consistently provides speedups over 115× and a mean speedup of 135×. Although it requires no prior parameter tuning, *StackStealing* provides the worst performance with not a single instance reaching 100× speedup, and the mean for the 6 larger instances being 61× (Tables 2 and 3 in subsection 5.5).

**Future work** may compare the parallel performance implications of alternate bounding functions, e.g. the *Two Machine Bound* [10], and this can be done without any parallelism code changes. There are many very large FSP instances that require weeks to compute sequentially, and it would be interesting to explore whether these representative OR problems could be practically solved by deploying YewPar on a mid-scale HPC (around 10K cores).

## References

1. Enrique Alba, Francisco Almeida, Maria J. Blesa, J. Cabeza, Carlos Cotta, Manuel Díaz, Isabel Dorta, Joaquim Gabarró, Coromoto León, J. Luna, Luz Marina Moreno,

C. Pablos, Jordi Petit, Angélica Rojas, and Fatos Xhafa. MALLBA: A Library of Skeletons for Combinatorial Optimisation (Research Note). In *Euro-Par 2002, Parallel Processing, $8^{th}$ International Euro-Par Conference, Proceedings*, pages 927–932, 2002. `doi:10.1007/3-540-45706-2_132`.

2. Ali Allahverdi. A survey of scheduling problems with no-wait in process. *European Journal of Operational Research*, 255(3):665–686, 2016. `doi:10.1016/j.ejor.2016.05.036`.
3. Blair Archibald. *Algorithmic skeletons for exact combinatorial search at scale*. PhD thesis, University of Glasgow, UK, 2018. URL: `https://theses.gla.ac.uk/31000/`.
4. Blair Archibald, Fraser Dunlop, Ruth Hoffmann, Ciaran McCreesh, Patrick Prosser, and James Trimble. Sequential and parallel solution-biased search for subgraph algorithms. In Louis-Martin Rousseau and Kostas Stergiou, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, volume 11494 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2019. `doi:10.1007/978-3-030-19212-9\_2`.
5. Blair Archibald, Patrick Maier, Rob Stewart, and Phil Trinder. Yewpar: skeletons for exact combinatorial search. In Rajiv Gupta and Xipeng Shen, editors, *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2020*, pages 292–307. ACM, 2020. `doi:10.1145/3332466.3374537`.
6. Jacques Carlier. Ordonnancements à contraintes disjonctives. *Rairo-operations Research*, 12(4):333–350, 1978.
7. Imen Chakroun and Nouredine Melab. HB&B@GRID: An heterogeneous grid-enabled branch and bound algorithm. In *International Conference on High Performance Computing & Simulation, HPCS 2016, 2016*, pages 697–704. IEEE, 2016. `doi:10.1109/HPCSim.2016.7568403`.
8. A. de Bruin, G.A.P. Kindervater, and H.W.J.M. Trienekens. Asynchronous parallel branch and bound and anomalies. In Afonso Ferreira and José Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *Lecture Notes in Computer Science*, pages 363–377. Springer Berlin Heidelberg, 1995. `doi:10.1007/3-540-60321-2_29`.
9. Jan Gmys. Solving large permutation flow-shop scheduling problems on gpu-accelerated supercomputers. arXiv preprint arXiv:2012.0951, 2020.
10. Jan Gmys, Rudi Leroy, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. Work stealing with private integer-vector-matrix data structure for multi-core branch-and-bound algorithms. *Concurrency Computation Practice and Experience*, 28(18):4463–4484, 2016. `doi:10.1002/cpe.3771`.
11. Jan Gmys, Mohand Mezmaz, Nouredine Melab, and Daniel Tuyttens. A computationally efficient branch-and-bound algorithm for the permutation flow-shop scheduling problem. *European Journal of Operational Research*, 284(3):814–833, 2020. `doi:10.1016/j.ejor.2020.01.039`.
12. Edward Ignall and Linus Schrage. Application of the branch and bound technique to some flow-shop scheduling problems. *Operations Research*, 13:400–412. `doi:10.1287/opre.13.3.400`.
13. Stefan Kehrer and Wolfgang Blochinger. Development and operation of elastic parallel tree search applications using TASKWORK. In *Cloud Computing and Services Science - 9th International Conference, CLOSER 2019, Revised Selected Papers*, volume 1218 of *Communications in Computer and Information Science*, pages 42–65. Springer, 2019. `doi:10.1007/978-3-030-49432-2\_3`.

14. Ignas Knizikevičius, Phil Trinder, and Blair Archibald. Low effort parallel scalable flowshop in yewpar [code and data repository], 2021. doi:10.5281/zenodo.5646543.
15. Samia Kouki, Mohamed Jemni, and Talel Ladhari. Scalable distributed branch and bound for the permutation flow shop problem. In *Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2013*, pages 503–508. IEEE, 2013. doi:10.1109/3PGCIC.2013.86.
16. Talel Ladhari and Mohamed Haouari. A computational study of the permutation flow shop problem based on a tight lower bound. *Computers and Operations Research*, 32:1831–1847, 2005. doi:10.1016/j.cor.2003.12.001.
17. Muhammad Nawaz, E Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11:91–95, 1983. doi:10.1016/0305-0483(83)90088-9.
18. Eric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993. doi:10.1016/0377-2217(93)90182-M.
19. Eva Vallada, Rubén Ruiz, and Jose M. Framiñan. New hard benchmark for flowshop scheduling problems minimising makespan. *Eur. J. Oper. Res.*, 240(3):666–677, 2015. doi:10.1016/j.ejor.2014.07.033.
20. Trong-Tuan Vu and Bilel Derbel. Parallel branch-and-bound in multi-core multi-cpu multi-gpu heterogeneous environments. *Future Generation Computer Systems*, 56:95–109, 2016. doi:10.1016/j.future.2015.10.009.
21. Trong-Tuan Vu, Bilel Derbel, Ali Asim, Ahcène Bendjoudi, and Nouredine Melab. Overlay-centric load balancing: Applications to UTS and B&B. In *2012 IEEE International Conference on Cluster Computing, CLUSTER 2012*, pages 382–390. IEEE Computer Society, 2012. doi:10.1109/CLUSTER.2012.17.