Declarative Decomposition and Dispatching for Large-Scale Job-Shop Scheduling*

Giacomo Da Col and Erich C. Teppan

Alpen-Adria-Universität Klagenfurt,
Universitätsstr. 65-67, 9020 Klagenfurt, Austria
{giacomo.da;erich.teppan}@aau.at

Abstract. Job-shop scheduling problems constitute a big challenge in nowadays industrial manufacturing environments. Because of the size of realistic problem instances, applied methods can only afford low computational costs. Furthermore, because of highly dynamic production regimes, adaptability is an absolute must. In state-of-the-art production factories the large-scale problem instances are split into subinstances, and greedy dispatching rules are applied to decide which job operation is to be loaded next on a machine. In this paper we propose a novel scheduling approach inspired by those hand-crafted scheduling routines. Our approach builds on problem decomposition for keeping computational costs low, dispatching rules for effectiveness and declarative programming for high adaptability and maintainability. We present first results proving the concept of our novel scheduling approach based on a new large-scale job-shop benchmark with proven optimal solutions.

Keywords: job-shop scheduling, problem decomposition, declarative programming

1 Introduction

The scheduling of jobs [6] is an important task in almost all production systems in order to optimize various objectives such as resource consumption, tardiness, or flow time. In general, jobs are structured into operations which must be allocated to resources such that various manufacturing constraints are satisfied and in addition an objective function is minimized. The job-shop scheduling problem (JSP) is among the most famous $\mathcal{NP}$-hard [12] and longest studied combinatorial problems (e.g. [4],). In this paper we focus on the makespan as the most classic optimization criterium. The makespan is the time needed for completing all job operations. Thus, the JSP can be defined as follows:

- Given is a set $M = \{1, \ldots, m\}$ of machines and a set $J = \{1, \ldots, j\}$ of jobs.

* The research for this paper was conducted in the scope of the project *Heuristic Intelligence (HINT)* in cooperation with Infineon Technologies Austria AG and Siemens AG Österreich funded by the Austrian research fund FFG under grant 840242. Authors are given in alphabetical order and contributed equally to this paper.

- Each job $j \in J$ consists of a sequence of operations $Ops_j = \{j_1, \dots, j_{l_j}\}$ whereby j_{l_j} is the last operation of job j.
  Practically, jobs can be interpreted as products and operations can be interpreted as their production steps.
  With respect to a job j and its operation j_i (with $1 < i < l_j$), the operation j_{i+1} is called successor and the operation j_{i-1} is called predecessor.
- Each operation j_i has an operation length $length_{j_i} \in \mathbb{N}$.
- Each operation j_i is assigned to a machine $m_{j_i} \in M$ by which it is processed.
- A (consistent and complete) schedule consists of a starting time $start_{j_i}$ for each operation j_i such that:
  - An operation's successor starts after the operation has been finished, i.e. with respect to a job j and the operations j_i and j_{i+1}:
    - $start_{j_{i+1}} \geq start_{j_i} + length_{j_i}$
  - Operations processed by the same machine are non-overlapping, i.e. with respect to two operations $j_x \neq k_y$ with $m_{j_x} = m_{k_y}$:
    - $start_{j_x} \neq start_{k_y}$
    - $start_{j_x} < start_{k_y} \rightarrow start_{j_x} + length_{j_x} \leq start_{k_y}$
- Makespan, i.e. the time period needed for processing all operations, is minimized, i.e.:
  - $\max_{j \in J, j_i \in Ops_j} \{start_{j_i} + length_{j_i}\} \rightarrow min$

The flexible job-shop scheduling problem (FJSP) is a variant of the JSP where for an operation there is a set of candidate machines that can process the operation, i.e. for each operation j_i there is a set $s_{j_i} \subseteq M$. Out of s_{j_i} there is one machine $m_{j_i} \in s_{j_i}$ which actually processes the operation. Hence, there is no predefined operation-to-machine assignment and the mapping of operations to machines is part of the problem.

In terms of complexity, the JSP is $\mathcal{NP}$-hard [11] and as it is a special case of the FJSP, also the FJSP is NP-hard. Thus, computing optimal solutions is only feasible for small problem instances.

Driven by the demands of the semiconductor industry, our general aim is the design of practically applicable algorithms for job-shop scheduling problems for domains comprising hundreds of machines and thousands of jobs and job operations. The size of such large-scale job-shop scheduling problem instances go beyond the usual benchmarks. For example, in the well-known job-shop benchmarks of [8] problem instances comprise not more than 50 jobs and 20 machines, i.e. the maximum branching factor in a corresponding search problem is 50.

In the case of our project partner Infineon Austria Technologies, common problem instances for a weekly workload are of the order of 10^4 operations on 10^2 machines in the back-end, i.e. where the products are made ready for shipping, and 10^5 operations on 10^3 machines in the front-end, i.e. where the chips are actually produced. The branching factors even go beyond the order of 10^3. On the one hand, this is due to the large number of jobs. On the other hand, this is because there are typically multiple machines equipped to perform a certain job

operation, i.e. it is a flexible job-shop. Apparently, calculating optimal schedules for such large-scale problems are typically far out of reach for such domains.

State-of-the-art search algorithms for JSP and FJSP are local search methods like tabu search [17] or large-neighborhood search [19]. However, without equipping the search approaches with sophisticated domain specific heuristics the performance is quite limited [7, 1, 9]. Furthermore for large-scale problems, problem decomposition is absolutely needed [5]. When applying problem decomposition, a problem instance is partitioned into subinstances which are solved independently. The subsolutions are then combined again to form the overall solution.

Daily scheduling routines in semi-conductor factories like those of our project partner also build on problem decomposition on the one hand and heuristics on the other hand for producing scheduling solutions. The problem decomposition is hereby realized by partitioning the machines into so called workcenters. Each workcenter is responsible only for a fraction of operation types, i.e. job operations are assigned to workcenters rather than actual machines. The production plan is made for each workcenter independently based on dispatching rules, a widely employed state-of-the-art technique for dealing with large and complex scheduling problems in nowadays manufacturing environments [15].

Dispatching rules are greedy heuristics for step-wise deciding which is the operation to be processed next by a machine. Simple examples for dispatching rules are first-come-first-serve, i.e. the preference of the longest waiting operation, or shortest-job-first. One of the most effective dispatching rules for minimizing the makespan is the most-total-work-remaining (MTWR) rule [18]. According to MTWR, the next operation to be dispatched belongs to a job such that the sum of lengths of all remaining operations is maximal. One big advantage of dispatching rules is that they can be computed typically in linear time. Another big advantage is their flexibility. Dispatching rules can be changed, adapted or combined easily in order to react on changing order situations as well as changing product portfolios. This is crucial for modern manufacturing regimes like mass customization [10], just-in-time or lean production [16]. Moreover, almost every real-world scheduling problem has specific constraints regarding the manufacturing processes.

Consequently, in order to be successfully applied in real-life production environments, an approach for automatic decomposition and dispatching must be easily adaptable. In particular, it must be possible to implement different decomposition methods and dispatching rules in a compact but well-maintainable form.

One way of achieving high adaptability and maintainability is declarative programming. Declarative programming approaches such as answer set and constraint programming [13, 2] provide high-level language representation features. Encodings specify the problem to be solved rather than how it is to be solved, which leads to short and well-maintainable code. A general problem solver is then responsible for finding a consistent solution for the encoded problem. As a consequence, declarative approaches have already been often applied to scheduling

problems (e.g. [3]). However, for large-scale problems, their direct applicability is limited. For example, incremental scheduling problem instances (a variant of FJSP) used in the answer set programming competition [1] do not comprise more than 120 job operations on max 7 machines. Also constraint programming approaches cannot be directly applied on problems like investigated in this paper. This is due to the fact that otherwise well-applicable global constraints [2] possess quadratic complexities which, in the light of instances comprising up to 10000 job operations and 100 machines, can already be too much.

In this paper we present a novel scheduling approach building on declarative programming in order to benefit from the high-level and compact knowledge representation and combine it with problem decomposition and dispatching rules in order to have the best of the different worlds: adaptability and extendability, low computational costs and effectiveness.

The remainder of the paper is structured as follows: In the next section we present a novel scheduling approach in which it is possible to program decomposition and scheduling rules by means of declarative programming. We describe the architecture and also a first prototype following this architecture. In Section 3 we describe a new large-scale benchmark of JSP and FJSP instances. The instances are patterned on scheduling problems of our project partner Infineon Austria Technologies and comprise up to more than 2000 jobs with up to more than 10000 operations in total to be scheduled on up to 100 machines. The new benchmark instances also have the feature of proven minimal makespans. In Section 4 we present first experimental results clearly proving the new concept. Section 5 concludes the paper.

2 A Novel Scheduling Approach

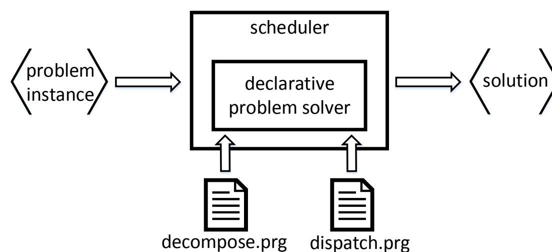

Fig. 1. General architecture of the scheduler

[1] https://www.mat.unical.it/aspcomp2014/
[2] http://sofdem.github.io/gccat/

Architecture Figure 1 shows the general architecture of our scheduling approach. Conforming to our proposed architecture, a scheduler is build upon a general purpose declarative problem solver. This can be - but is not limited to - constraint solvers like Gecode[3] or Jacop[4], answer set solvers like Clingo[5] or hybrids like proposed in [2, 14].

Two declarative programs written in the language of the solver are responsible to define the scheduling behavior. In particular, *decompose.prg* specifies the strategy for the problem decomposition, i.e. the method for splitting the problem instance into subinstances, which are then treated independently. How those subinstances are processed is determined by a dispatching rule defined in *dispatch.prg*.

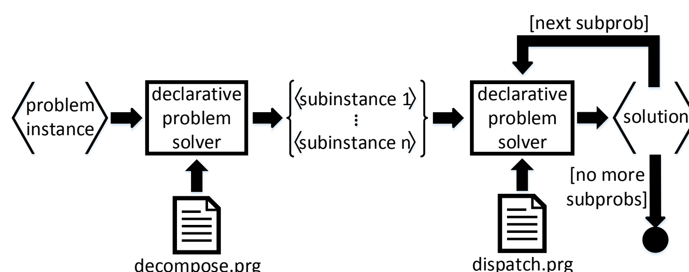

Fig. 2. Data flow of the scheduling process

Figure 2 shows the data flow of the scheduling process. First, the solver combines the instance with the decomposition program, that divides it into a number of subinstances. Both the number of subinstances and the decomposition strategy are determined by *decompose.prg*. Once the subinstances have been created, they are processed sequentially by the declarative solver following the dispatching rule defined in *dispatch.prg*. The dispatching process is incremental. Thus, the solution of one subinstance is forwarded as an additional input to the next subinstance in line until no more subinstances are left. The overall solution is the composition of all the subsolutions.

Prototype We build a prototype in order to provide a first proof of concept. Our prototype scheduler is implemented in Java incorporating ASCASS, a constraint answer set programming (CASP) solver [20]. ASCASS follows the idea given in [2], i.e. answer set programming (ASP) is used for specification of constraint satisfaction problems (CSPs). The CSPs are solved by the constraint solver Jacop. CASP approaches have proven to be highly effective for problems with very large domains, like industrial sized scheduling problems [3]. This is due to the

[3] http://www.gecode.org/
[4] http://jacop.osolpro.com/
[5] http://potassco.sourceforge.net/

combination of the high-level knowledge representation features of ASP and the possibility of stating the variable domains as intervals by means of constraint variables. Since the scheduling problems usually present large domains for the time representation, CASP suits well our needs.

In our prototype the decomposition strategy implemented in *decompose.prg* splits the input based on the predefined machine and workcenter information. This means that every subinstance only comprises a single machine of each workcenter. This makes sure that it is possible to process every operation type in every subinstance. In order to balance the workload, the set of jobs is equally divided among the subinstances. Concerning the dispatching rule, we apply the most-total-work-remaining (MTWR) rule. In our implementation, this rule is expressed as a logic predicate specifying the operation priorities in a recursive manner, i.e.

```
opPriority(J,L):-
  op(J), opLength(J,L), not precedes(J,_).
opPriority(J,P2+L):-
  op(J), op(J2), precedes(J,J2), opPriority(J2,P2), opLength(J,L).
```

The declarative solver (ASCASS) processes the operations conforming the stated priorities, behaving in accordance to the MTWR rule[6].

Example In the following we describe the behavior of our prototype on a small sample instance. Take the following FJSP instance, represented as logic facts:

```
machine(1).machine(2).machine(3).machine(4).
machineWorkcenter(1,1).machineWorkcenter(2,1).
machineWorkcenter(3,2).machineWorkcenter(4,2).
%job 1
op(11).opLength(11,2).opWorkcenter(11,2).precedes(11,12).
op(12).opLength(12,3).opWorkcenter(12,1).
%job 2
op(21).opLength(21,3).opWorkcenter(21,1).precedes(21,22).
op(22).opLength(22,2).opWorkcenter(22,2).precedes(22,23).
op(23).opLength(23,4).opWorkcenter(23,1).
%job 3
op(31).opLength(31,8).opWorkcenter(31,1).precedes(31,32).
op(32).opLength(32,2).opWorkcenter(32,2).
%job 4
op(41).opLength(41,2).opWorkcenter(41,1).precedes(41,42)
op(42).opLength(42,1).opWorkcenter(42,2).precedes(42,43).
op(43).opLength(43,7).opWorkcenter(43,1).
```

The above code snippet encodes:

- four machines organized in two workcenters

[6] For space reasons it is not possible to go further into the implementation details of the prototype. The interested reader can find further information on programming in ASCASS and our prototype at http://isbi.aau.at/HINT

– four jobs, whereby jobs 1 and 3 consist of two operations and job 2 and 4 consist of three operations, whereby
– every operation has a defined operation length, and the *precedes* predicate indicates the operation ordering within a job.

The decomposition method used in our prototype splits the sample input in two subinstances. Each of the subinstances comprises one machine per workcenter. The jobs are equally distributed among the subinstances, such that the first one contains jobs 1 and 2, and the second contains job 3 and 4.
For the first subinstance, the dispatch program produces the following operation

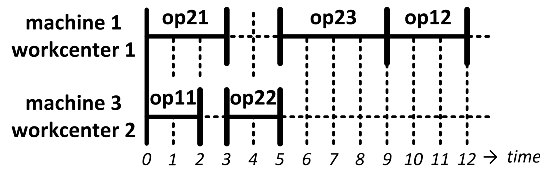

Fig. 3. Partial scheduling solution of the sample instance

priorities conforming to MTWR: op12=3, op11=5; op23=4, op22=6, op21=9. The operations are handled compliant with the priorities in descending order, i.e. highest priority operations first. Consequently, the partial solution in Figure 3 is produced. The overall solution is composed by this subsolution and the subsolution of the second subinstance.

3 Benchmark

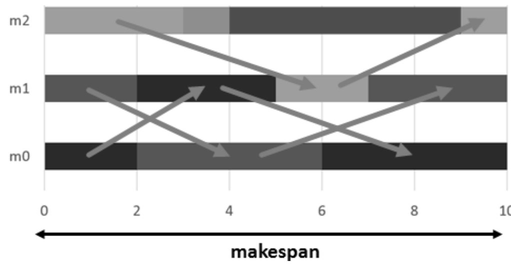

Fig. 4. Principle of instance generation

For evaluation purposes we produced test instances of realistic sizes which are on the one hand patterned on problems of our project partner Infineon

Austria Technologies and on the other hand have the advantage of proven minimal makespans. This is accomplished by first producing an optimal solution without machine idle times for a given number of operations (#ops) to be scheduled on a given number of machines (#machines) and the minimal makespan. This is done by randomly partitioning the machines' time continuum $[0..makespan \times \#machines - 1]$ into number of operations many partitions whereby each partition corresponds to the processing period of an operation. Consequently, the minimal makespan, average operation length (avg(opLength)), number of operations (#ops) and number of machines (#machines) relate conforming to $makespan = \frac{\#ops \times avg(opLength)}{\#machines}$. Based on such a partitioning, successor relations are randomly generated. Each operation gets at most one successor and/or predecessor such that the successor's starting time is greater than the predecessor's finishing time. Figure 4 shows the principle for 11 operations (5 jobs) on 3 machines and a minimal makespan of 10.

We applied two different procedures for generating random successor relations based on the pre-calculated solution:

1. For each operation *op* (in random order) define as successor *suc* a random operation such that
   - *suc* is not on the same machine as *op*.
   - *suc* starts later than *op* ends.
   - *suc* is not yet a successor of another operation.
   - If no such *suc* exists *op* has no successor.
2. For each operation *op* (in random order) define as successor *suc* an operation such that
   - *suc* is not on the same machine as *op*.
   - *suc* starts later than *op* ends.
   - *suc* is not yet a successor of another operation and
   - the time between *op* ends and *suc* starts is minimal.
   - In case that there are multiple possible successors, a random one is chosen.
   - If no such *suc* exists *op* has no successor.

The two different generating approaches result in benchmark instances which are different in nature: (1) produces many jobs consisting of a small number of operations. We refer to this set of instances as 'short-jobs' (SJ). On the contrary, (2) produces fewer jobs but with a larger number of operations per job. We refer to this set of instances as 'long-jobs' (LJ). For each machine/operations setting, i.e. 10 machines / 100 operations, 10 machines / 1000 operations, 100 machines / 1000 operations and 100 machines / 10000 operations we produced three basic instances (files). Each of the basic instances stands for a set of instances, depending whether the instance is interpreted as JSP or FJSP and which flexibility factor is assumed in the FJSP. A flexibility of 1 stands for the JSP, i.e. all machines are different to each other. A flexibility of 2 means that each operation can be processed by two machines, i.e. there is always two machines of the same type. For 10 machines we produced instances for the flexibility factors 1 (i.e. the

JSP), 2, 5 and 10 (i.e. all machines can process all operations). For 100 machines we produced instances for the flexibility factors 1, 2, 5, 10, 20, 50 and 100. This results in 66 instances for the long-jobs benchmark and the same number of instances for the short-jobs benchmark. All instances have a minimal makespan of 600000, which, interpreted in seconds, constitutes roughly one week. Tables 1 and 2 show the statistics for the benchmark. The whole benchmark is available online at http://isbi.aau.at/hint/scheduling-prototype.

	#machines/#ops	file	numJobs	minNumOps	maxNumOps	avgNumOps
LJ	10/100	1	12	4	12	8.3
LJ	10/100	2	14	3	13	7.1
LJ	10/100	3	14	4	11	7.1
LJ	10/1000	1	18	10	75	55.6
LJ	10/1000	2	18	1	79	55.6
LJ	10/1000	3	17	2	83	58.8
LJ	100/1000	1	102	2	16	9.8
LJ	100/1000	2	103	4	16	9.7
LJ	100/1000	3	102	2	17	9.8
LJ	100/10000	1	102	68	134	98.0
LJ	100/10000	2	103	71	122	97.1
LJ	100/10000	3	103	5	133	97.1
SJ	10/100	1	32	1	5	3.1
SJ	10/100	2	33	1	5	3
SJ	10/100	3	32	1	7	3.1
SJ	10/1000	1	245	2	11	4.1
SJ	10/1000	2	232	1	9	4.3
SJ	10/1000	3	236	2	10	4.2
SJ	100/1000	1	285	1	8	3.5
SJ	100/1000	2	285	1	8	3.5
SJ	100/1000	3	286	1	7	3.5
SJ	100/10000	1	2174	2	15	4.6
SJ	100/10000	2	2153	2	13	4.6
SJ	100/10000	3	2161	2	14	4.6

Table 1. Benchmark statistics: operations per job

4 Experiment

The main goal of our experiment is to provide a first proof of concept of our novel scheduling approach. In particular, we want to investigate the impact of problem decomposition on our large-scale scheduling benchmark. The purpose of this experiment is not to investigate the performance of different dispatching rules. Instead, we want to show that dispatching rules in general can be successfully exploited in the context of declarative programming and problem decomposition.

4.1 Setup

We tested our prototype described in Section 2 against the benchmark instances described in Section 3. All the instances are solved once with the decomposition approach, also described in Section 2, and once without decomposition. In both cases, we use the most total work remaining (MTWR) dispatching rule. The

	#machines/#ops	file	minOpLength	maxOpLength	avgOpLength
LJ	10/100	1	109	306357	60000
LJ	10/100	2	187	369450	60000
LJ	10/100	3	105	338068	60000
LJ	10/1000	1	11	41650	6000
LJ	10/1000	2	2	57111	6000
LJ	10/1000	3	8	45096	6000
LJ	100/1000	1	48	558806	60000
LJ	100/1000	2	144	464127	60000
LJ	100/1000	3	15	432530	60000
LJ	100/10000	1	2	54454	6000
LJ	100/10000	2	1	64436	6000
LJ	100/10000	3	2	59748	6000
SJ	10/100	1	52	403461	60000
SJ	10/100	2	1951	307065	60000
SJ	10/100	3	414	249757	60000
SJ	10/1000	1	4	46271	6000
SJ	10/1000	2	20	39801	6000
SJ	10/1000	3	9	54007	6000
SJ	100/1000	1	61	484415	60000
SJ	100/1000	2	116	384096	60000
SJ	100/1000	3	3	531413	60000
SJ	100/10000	1	1	64636	6000
SJ	100/10000	2	1	52669	6000
SJ	100/10000	3	1	59361	6000

Table 2. Benchmark statistics: operation lengths

experiment was conducted on a system with Intel i7-3930K CPU (3.20GHz), 64 Gb of RAM. The timeout for the computation of the complete schedule for each instance was set to 4000 seconds. This time frame would allow a frequent recalculation in a weekly or bi-weekly scheduling scenario.

4.2 Results

	#mach/#ops	JSP	FJSP-2	FJSP-5	FJSP-10	FJSP-20	FJSP-50	FJSP-100	total
no dec	10/100	100%	100%	100%	0%	-	-	-	75%
dec	10/100	100%	100%	100%	100%	-	-	-	100%
no dec	10/1000	100%	100%	0%	0%	-	-	-	50%
dec	10/1000	100%	100%	100%	100%	-	-	-	100%
no dec	100/1000	100%	100%	66.7%	0%	0%	0%	0%	38%
dec	100/1000	100%	100%	100%	100%	100%	100%	100%	100%
no dec	100/10000	83.3%	0%	0%	0%	0%	0%	0%	11.9%
dec	100/10000	83.3%	100%	100%	100%	100%	100%	100%	97.6%
no dec	total	95.8%	75%	58.4%	0%	0%	0%	0%	38.6%
dec	total	95.8%	100%	100%	100%	100%	100%	100%	99.2%

Table 3. Comparison of percentage of solved instances of the approaches with and without decomposition

Table 3 shows the percentage of solved instances for the different machines/operation settings. The first column distinguishes between the approaches with decomposition (dec) and without decomposition (no dec). The results are given grouped by the different flexibility factors, i.e. the number of possible machines

for each operation. It gets obvious that the sizes of the job-shop in terms of number of machines/operations as a big impact in the number of solved instances. In particular, less instances can be solved without producing a timeout in larger job-shops. On the other hand, also the increase of the flexibility factor negatively impacts on the number of solved instances. A particular result is the case with flexibility of 5, where in the approach without decomposition it was possible to solve 66.7% of the 100/1000 instances, but 0% of the 10/1000 instances. The reason behind this behavior is the ratio between the number of operations and the number of machines. In the case with 100 machines and 1000 operations the ratio is 10, meaning that in a perfectly balanced schedule, where all the machines get the same amount of operations, there are 10 operations per machine. In the case of 10 machines and 1000 operations the ratio is 100 operations per machine. An increased number of operations per machine leads to a longer constraint propagation in the general declarative problem solver.

Concerning the instances with 10 machines and 1000 operations, 12 out of 24 (50%) were solved. In the set of instances with 100 machines and 1000 operations, the number of solved instances was 16 out of 42 (38%). Note, the varying number of total instances (24 vs. 42) is due to the varying number of machines in the different settings. A greater number of machines naturally allows more flexibility factors. The total number of solved instances of the approach without decomposition is 51 out of 132, which corresponds to 38.6% of the benchmark.

In comparison, the decomposition approach showed better results. With respect to the JSP, the results are the same as in the approach without decomposition, because the JSP corresponds to a FJSP with flexibility of 1. Consequently, no decomposition occurs, as the workcenters cannot be split any further. In all the other cases the decomposition plays a crucial role in the solvability, since it was possible to solve all the FJSP instances with any size of the problem. The only unsolved case is a JSP instance with 100 machines and 10000 operations, where the decomposition is not applied. Overall it was possible to solve 131 out of 132 (99.2%) instances, proving the effectiveness of the decomposition approach. Table 4 shows the makespans produced by MTWR without decomposition (no

	#mach/#ops	**JSP**	**FJSP-2**	**FJSP-5**	**FJSP-10**	**FJSP-20**	**FJSP-50**	**FJSP-100**
no dec	10/100	855350	712685	645226	t/o	-	-	-
dec	10/100	855350	1067370	1093303	1014521	-	-	-
no dec	10/1000	780589	696931	t/o	t/o	-	-	-
dec	10/1000	780589	995531	1083530	995953	-	-	-
no dec	100/1000	933775	754580	670931	t/o	t/o	t/o	t/o
dec	100/1000	933775	1170844	1426045	1451335	1686282	1418501	1193235
no dec	100/10000	767278	t/o	t/o	t/o	t/o	t/o	t/o
dec	100/10000	767278	1035810	1225529	1295864	1353059	1365883	1236629

Table 4. Comparison of the makespan results (in seconds) of the two approaches

dec) and with decomposition (dec) respectively. The term *t/o* indicates where the instances reached the timeout. It is to be noticed that, when it is possible to find a solution, the approach without decomposition presents a lower makespan.

This is due to the fact that the decomposition program used in our experiment aims to distribute the job equally among the subinstances. However, the number of operations per job varies in the benchmark. Thus, an equal number of jobs in the subinstances does not necessarily correspond to an equal number of operations. Consequently, the total workload to be processed in the subinstances is not perfectly balanced. Opportune tuning of the decomposition program may lead to better results in terms of makespan. This will be investigated in future work.

	#mach/#ops	**JSP**	**FJSP-2**	**FJSP-5**	**FJSP-10**	**FJSP-20**	**FJSP-50**	**FJSP-100**
no dec	10/100	0.1	0.1	2.1	t/o	-	-	-
dec	10/100	0.1	0.2	0.2	0.2	-	-	-
no dec	10/1000	86.3	436	t/o	t/o	-	-	-
dec	10/1000	86.3	91.6	76.0	57.1	-	-	-
no dec	100/1000	1.3	2.1	839.5	t/o	t/o	t/o	t/o
dec	100/1000	1.3	1.1	0.9	0.9	0.9	0.9	0.9
no dec	100/10000	2611.8	t/o	t/o	t/o	t/o	t/o	t/o
dec	100/10000	2611.8	2963	1970.5	1516.5	1133.7	1130.7	449

Table 5. Comparison of the runtime results (in seconds) of the two approaches

Finally, Table 5 shows the actual runtimes for solution calculations without decomposition (no dec) and with decomposition (dec) respectively. The decomposition approach was capable to solve the instances significantly quicker than without decomposition.

5 Conclusion

We presented a novel scheduling approach targeted to large job-shop and flexible job-shop problems. The approach combines the high level knowledge representation of declarative programming with problem decomposition and dispatching rules. We introduced a new large-scale scheduling benchmark of instances comprising up to 10000 job operations to be scheduled on up to 100 machines and with proven optimal solutions. We provided a first evaluation for a prototype implementation of our approach. The results clearly prove the concept of our scheduling method. In particular, with this method it was possible to find schedules for all the large-scale instances of the benchmark before a predefined timeout occured, while without decomposition, many instances could not be solved. Summarizing, we can conclude that the combination of declarative methods, problem decomposition and dispatching rules can successfully be applied in real-world large-scale scheduling domains.

References

1. Azi, N., Gendreau, M., Potvin, J.Y.: A dynamic vehicle routing problem with multiple delivery routes. Annals of Operations Research 199(1), 103–112 (2012)

2. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In: ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09) (2009)
3. Balduccini, M.: Logic Programming and Nonmonotonic Reasoning: 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings, chap. Industrial-Size Scheduling with ASP+CP, pp. 284–296. Springer Berlin Heidelberg (2011)
4. Bellman, R.: Mathematical aspects of scheduling theory. SIAM Jour of App Math 4, 168–205 (1956)
5. Bent, R., Van Hentenryck, P.: Spatial, temporal, and hybrid decompositions for large-scale vehicle routing with time windows. In: Cohen, D. (ed.) Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP 2010). pp. 99–113. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
6. Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Weglarz, J.: Handbook on Scheduling: Models and Methods for Advanced Planning (International Handbooks on Information Systems). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
7. Carchrae, T., Beck, J.C.: Principles for the design of large neighborhood search. Journal of Mathematical Modelling and Algorithms 8(3), 245–270 (2009)
8. Demirkol, E., Mehta, S., Uzsoy, R.: Benchmarks for shop scheduling problems. European Journal of Operational Research 109(1), 137 – 141 (1998), http://www.sciencedirect.com/science/article/pii/S0377221797000192
9. Easton, T., Singireddy, A.: A large neighborhood search heuristic for the longest common subsequence problem. Journal of Heuristics 14(3), 271–283 (2008)
10. Fogliatto, F.S., Da Silveira, G.J.C., Borenstein, D.: The mass customization decade: An updated review of the literature. International Journal of Production Economics 138(1), 14–25 (2012)
11. Garey, M.R.: The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research 1(2), 117–129 (1976)
12. Garey, M.R., Johnson, D.S.: Computers and Intractability. A Guide to the Theory of NP-Completeness. W. H. Freeman and Company (1979)
13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)
14. Gebser, M., Ostrowski, M., Schaub, T.: Constraint Answer Set Solving, pp. 235–249. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
15. Hildebrandt, T., Goswami, D., Freitag, M.: Large-scale simulation-based optimization of semiconductor dispatching rules. In: Proceedings of the 2014 Winter Simulation Conference. pp. 2580–2590. WSC '14, IEEE Press, Piscataway, NJ, USA (2014), http://dl.acm.org/citation.cfm?id=2693848.2694175
16. Holweg, M.: The genealogy of lean production. Journal of Operations Management 25(2), 420–437 (2007), special Issue Evolution of the Field of Operations Management SI/ Special Issue Organisation Theory and Supply Chain Management
17. Hurink, J., Jurisch, B., Thole, M.: Tabu search for the job-shop scheduling problem with multi-purpose machines. Operations-Research-Spektrum 15(4), 205–215 (1994)
18. Kaban, A.K., Othman, Z., Rohmah, D.S.: Comparison of dispatching rules in job-shop scheduling problem using simulation: a case study. International Journal of Simulation Modelling 11(3), 129–140 (2012)

19. Pacino, D., Van Hentenryck, P.: Large neighborhood search and adaptive randomized decompositions for flexible jobshop scheduling. International Joint Conference on Artificial Intelligence. Proceedings (2011)
20. Teppan, E.C., Friedrich, G.: Heuristic constraint answer set programming. In: Proceedings of the 6th International Workshop on Combinations of Intelligent Methods and Applications (CIMA16 at ECAI16) (2016)