Multithreaded incremental solving for local search based metaheuristics with step chasing

Geoffrey De Smet · Tony Wauters

Received: date / Accepted: date

Abstract This work introduces a multithreaded solving methodology for local search based metaheuristics. It runs a single local search that spreads move evaluations across multiple threads. To preserve incremental score calculation (delta evaluation) capabilities, which are essential for the performance of local search methods, the child threads reproduce the step of the main thread in a method we named step chasing. The proposed method is implemented within OptaPlanner, a Java-based open source solver, and can thus be used by anyone. The effectiveness of the method is demonstrated using three metaheuristics (Tabu Search, Simulated Annealing, Late Acceptance) on four difficult combinatorial optimization problems: the nurse rostering problem, the vehicle routing problem, the curriculum course timetabling problem and the cloud balancing problem. Extensive experiments are performed using up to 16 threads with a total of 5550 runs, with significant speedups realized when more threads are available to the solver. All results are compared with a single threaded implementation, as well as a multi-walk approach. The greatest speedups take place with respect to the nurse rostering problem.

Keywords Parallel local search · Multithreading · Incremental solving · Metaheuristics · Open source

1 Introduction

Metaheuristics - and more specifically local search - are frequently used to solve difficult combinatorial optimization problems. In order to solve real-world

Geoffrey De Smet
Red Hat
E-mail: gds.geoffrey.de.smet@gmail.com

Tony Wauters
KU Leuven, Department of Computer Science, CODeS research group
E-mail: tony.wauters@kuleuven.be

problems and find high quality solutions, the speed at which these algorithms can calculate constraint and fitness scores is generally an important factor. The ability to accommodate large-scale problems is also crucial. A frequently applied technique for improving the scaling behaviour of local search is incremental score calculation (also known as delta evaluation). While this technique brings an important scaling factor, parallelization on multi-core machines and multi-node clouds enables further scaling. An opportunity which typically has been overlooked in the academic literature. Perhaps because combining parallelization and incremental score calculation poses some significant challenges.

Nevertheless, several works proposed a parallel local search method. [12] proposed multiple-walk and single walk parallel local search methods and applied it to the traveling salesman problem, the steiner tree problem and two scheduling problems. An overview of parallel metaheuristic strategies can be found in [5,1,11,6].

As highlighted by [2], most metaheuristic frameworks and libraries with parallelization focus on evolutionary algorithms. Therefore, very few frameworks support parallel local search or trajectory-based metaheuristics. Moreover, none of them support incremental score calculation.

This study introduces a parallel local search strategy with incremental score calculation, which is implemented in the open source constraint solver OptaPlanner [7]. We show that our parallelization strategy is effective for many local search algorithms (Hill Climbing, Tabu Search, Late Acceptance Hill Climbing, Simulated Annealing). Furthermore, we present benchmarks on four use cases (vehicle routing problem, nurse rostering problem, course scheduling and cloud balancing) resulting in a total of 37 datasets and 5550 benchmark runs.

The remainder of the paper is structured as follows. In Section 2 we discuss the primary implementation requirements for this research. The actual implemented method is discussed in Section 3. A thorough experimental investigation of the implemented method is provided in Section 4. Following this, Section 5 ends this paper with conclusions and directions for future work.

2 Requirements

Before we move on to the design of the implemented method, we must explore three important requirements which must be fulfilled in our implementation. These requirements are:

1. Incremental score calculation
2. Reproducible runs
3. Parallel computation

We will illustrate these requirements using the Nurse Rostering Problem (NRP) [4] as an example, an NP-hard problem which assigns shifts to nurses. Each shift must be assigned to exactly one nurse. Hard constraints typically include nurse conflicts, skill requirements and minimal rest periods. Soft constraints may include day off requests and illness affinity.

2.1 Incremental score calculation

A local search algorithm evaluates the neighborhood of proposed solutions at every iteration. It determines the quality of each proposed solution by calculating its score. In simple cases, a score is formed from two weighted numbers: one for the hard constraints (the feasibility check) and another for the soft constraints (the fitness function). Calculating these numbers is computationally expensive. For example, in the Nurse Rostering Problem, it requires detecting the number of nurse conflicts, and therefore every shift assignment must be compared with every other overlapping shift assignment to check if they are assigned to the same nurse. Given s shift assignments, this part of the score calculation requires $O(s^2)$ checks, and therefore scales quadratically. Some of the other hard or soft constraints are even more computationally expensive.

The local search algorithm triggers a score calculation for each move in its neighborhood until an acceptance criterion is met - at which point it applies the winning move and begins a new step to evaluate a new neighborhood. In the NRP, a simple change move assigns one shift to a different nurse. To calculate the score of the solution after applying such a move, we could calculate the score from scratch by iterating over all assignments. However, this is highly inefficient. Instead, we calculate it incrementally by determining the delta between the old and new score. For example in the NRP, the incremental score calculation of a move, that assigns one shift to a different nurse, need only compare that one shift assignment with every other overlapping shift assignment to determine the delta. This part of the score calculation now requires only $O(s)$ checks, instead of $O(s^2)$. This is an order of magnitude faster than non-incremental score calculation methods.

To achieve multithreaded solving, we cannot afford to forfeit incremental score calculation given that, in practice, the negative effects of being unable to calculate scores incrementally far outweigh any gain of parallel computation. For example, given a NRP with 5000 shift assignments and 250 nurses, incremental calculation of the nurse conflict constraint is up to 5000 times faster than non-incrementally. To make up for such a loss, a perfectly parallelized algorithm would require at least 5000 CPU's. From both a practical and economic perspective, this is clearly not an option.

Our implementation must therefore combine both incremental score calculation and multithreaded solving.

2.2 Reproducible runs

A constraint solver is reproducible if and only if running it twice yields the exact same solution (with the exact same score), given the exact same allocated CPU time (in the same manner). For local search and construction heuristic algorithms, this boils down to yielding the exact same solution at every step. A step is the outer iteration in these algorithms: each time they pick a winning move, it is a new step. Note that due to a difference of allocated CPU time,

the timing of each step between runs can vary and - given the same amount of time - a reproducible run still might not reach the same number of steps.

Most local search algorithms use a Pseudo Random Number Generator (PRNG), which influences reproducibility. For example, a PRNG breaks score ties in Tabu Search and influences move acceptance in Simulated Annealing. A single-threaded run can easily be made reproducible by using a single, seeded PRNG for all decisions. However, if a multi-threaded run uses a single PRNG across multiple threads, the concurrent calls on that PRNG suffer from congestion. Furthermore, we must somehow guarantee that all calls on that PRNG are always executed in the same order, given that re-ordering affects the step decision, causing the entire local search algorithm to go down a different path. As such, an efficient and reproducible multi-threaded run cannot use the same PRNG across multiple threads.

In practice, reproducibility is critical for any production solver: it allows programmers to debug their code during development as well as reproduce production issues on their own machines. Furthermore, in highly regulated enterprise environments, such as financial institutions, it enables the auditing of historic solver runs.

Our implementation of multithreaded solving must therefore not sacrifice reproducibility.

2.3 Parallel computation

Let us examine which parts of local search are suitable for parallelization. In order to improve performance over a single threaded solver, a multithreaded solver must parallelize at least parts of the algorithm. Given t threads and the same number of CPU cores, the performance of those algorithm parts increases by a factor of t, minus any overhead the multithreaded solving incurs.

Looking at the anatomy of a single threaded local search (Figure 1), there are several candidates to compute in parallel:

1. Select move: From the neighborhood, generate one move at a time, just in time. We could give each child thread a copy of the move selector (which generates the neighborhood on the fly). If we give each move selector its own PRNG (which is seeded by a common PRNG), they would consequently generate the same moves, thereby preserving reproducibility. However, we would need to force all child threads to generate and evaluate the exact same number of moves, meaning that the faster move selectors would have to wait for the slower ones. With a mix of fine and coarse grained moves, or varying CPU power per core (often the case in public clouds), the performance cost for enforcing an equal distribution of moves per child thread is unacceptable. Furthermore, move selection itself is usually inexpensive and thus we chose not to parallelize it in this work.
2. Calculate move score: Evaluating hard and soft constraints is computationally expensive given that all constraints must be evaluted. Even with incremental score calculation, evaluating the NRP's nurse conflict constraint

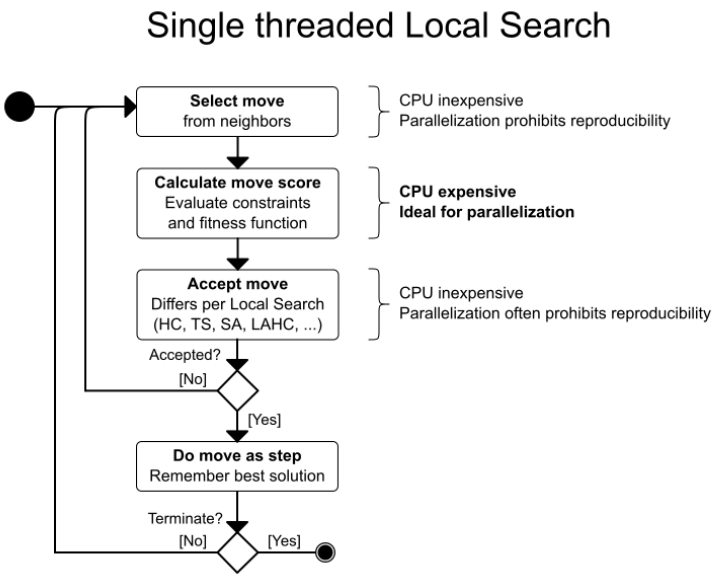

Fig. 1 Anatomy of a single threaded Local search

is still in $O(s)$ - which is still expensive because its nested in the move evaluation iteration, which is in turn nested in the step iteration. Yet each move can be scored in isolation from the others in the same step, so this is an ideal candidate for parallelization.

3. Accept move: In Hill Climbing, the acceptor simply checks if the score of a move improves upon the best score. Such an implementation can be easily parallelized. But in Simulated Annealing, the acceptance criterion uses the PRNG. Because we do not wish to enforce in advance which child thread evaluates which move (because, as explained earlier, this impairs efficiency), the acceptor must always use the same PRNG in the same order to guarantee reproducibility. Also, given that move acceptance itself is usually inexpensive, we choose not to parallelize it.

Our approach must therefore at least parallelize the score calculation of each move.

3 Method

We explain our method on local search in a general manner given that it works with respect to any local search algorithm (including Tabu Search [8], Simulated Annealing [10] and Late Acceptance Hill Climbing [3]), as shown by the implementation and benchmarks later in this paper.

The basic principle is parallelizing the score calculation by offloading it from the main solver thread (the parent thread) to n child threads, as shown in Figure 2.

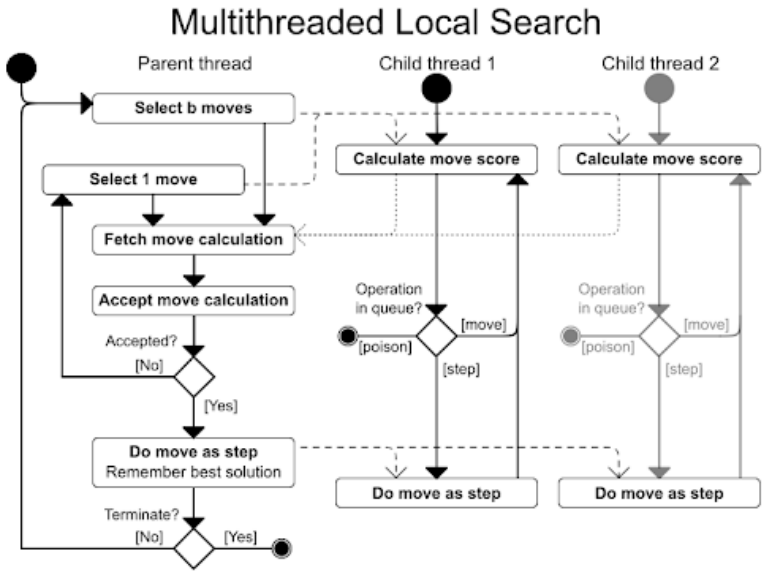

Fig. 2 Multithreaded local search

This redistribution has two noteworthy consequences:

- To evaluate multiple moves in parallel, multiple moves are active simultaneously.
- To preserve incremental score calculation, each child thread must have an isolated score calculation state that must be kept in sync with the steps on the parent thread.

3.1 Select multiple moves

One implication of calculating the score of two (or more) moves in parallel is that the second move must be generated before the calculation of the first move's score has finished. As explained earlier, we have chosen to generate all moves in the parent thread. Furthermore, the winning move of a step is also picked in the parent thread. So, given n child threads, we must at least generate n moves in the parent thread, before processing the resulting score of the first move in the same thread, to prevent the child threads from starving. All these generated moves for which the score has not been processed yet are considered active. For reproducibility, the number of active moves must be

deterministic. Generating a different number of moves, offsets the PRNG used in move selectors, causing non-reproducible runs. Furthermore, we must have more than n active moves given that a child thread can calculate multiple fine-grained moves while the parent thread is awaiting the result of a coarse-grained move from another child thread. Therefore we have b active moves, for which b is a multiple of n. There is a trade-off affecting the value of b: it must be large enough to satisfy the hunger of the child threads for maximum performance (especially with a mix of fine-grained and coarse-grained moves), but if b is too large then the generating of b moves at the beginning of each step iteration could negatively affect performance and memory usage. From our experiments, we found that setting b to 10 times n is a good default setting, where n is the number of child threads.

In the parent thread, we begin every step by selecting b moves from the neighborhoods, before receiving the first evaluated move from a child thread. Then, every time a move is received, we select a new one such there are always b active moves.

3.2 Move evaluation

The parent thread communicates with the child threads through an operation queue to send information and a result queue to receive information. The parent thread puts each selected move in the operation queue as a move calculation operation. The child threads continuously pulls these moves from that operation queue. When they do so, they calculate the score incrementally for that move and put the result in the result queue. The auxiliary data structures for such incremental score calculation are maintained by each child thread individually. Meanwhile, once the parent thread has generated b moves, it pulls the first move calculation result from the result queue. If that queue is empty, it waits until one of the child threads adds a result. After receiving a move, the parent thread accepts or rejects that move. Then it checks if an accepted move has won the step iteration. If at this point in time there is no winning step, it selects a new move.

This orchestration, shown in Figure 3, ensures that all child threads are kept busy evaluating moves, regardless of any variance concerning move evaluation duration.

Due to differences in move granularity, the score calculation results of each move come back out of order. For example, the result of the 4th generated move may arrive before the result of the 3rd generated move, especially if the 3rd move is more coarsely grained. For reproducibility, the parent thread orders the results back into their original order, on the fly. In this example, if the parent thread needs to process the 3rd move, but the 4th and 5th arrive first, it places those moves in a backlog and waits until the 3rd move arrives. Later, to return the 4th or 5th move, it first searches in the non-empty backlog before pulling a move from the result queue.

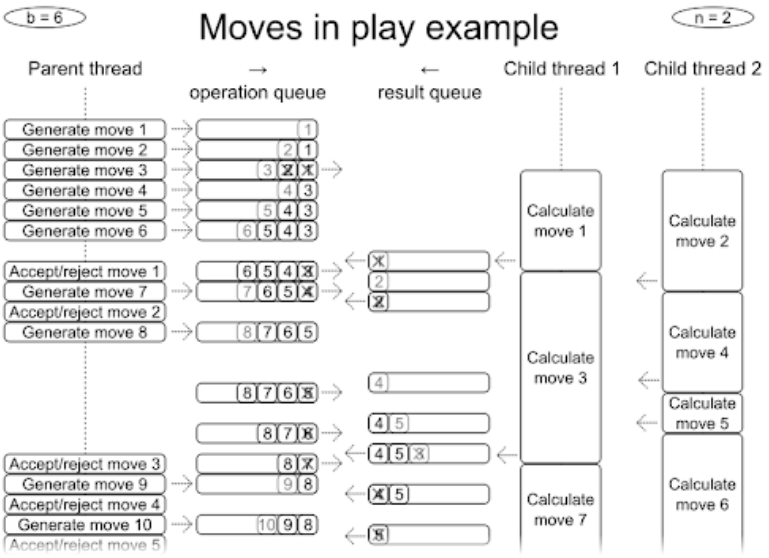

Fig. 3 Lifetime of active moves

The parent thread then accepts or rejects the retrieved move, depending on the acceptance criteria. If enough moves are accepted, it appoints the winning move of the current step. Both these decisions depend on the Local Search type and greatly affect the number of selected moves per step. For example, Tabu Search typically samples many non-tabu moves, which results in a slow stepping run. Simulated Annealing and Late Acceptance on the other hand, appoint the first accepted move, resulting in a fast stepping run - especially in the beginning.

If there is no winning move, the parent thread selects a new move and puts it in the operation queue to ensure there are always exactly b active moves, before repeating the process.

3.3 Step chasing

If there is a winning move, the local search advances: that move is applied on the working solution of the parent thread and the step ends. Before doing so, it clears the operation queue (which removes most of the b active moves except for about n active moves that already reached a child thread) and puts n step operations for that winning move in the operations queue, to sync the child threads. When a child thread pulls such a step operation it applies the winning move to keep their solution state in sync with the parent thread's state in an incremental manner (without cloning the entire solution state). After processing the step operation, the child thread also waits until all other

child threads have also pulled and processed their step operation. This is to prevent one child thread pulling two step operations of the same step from the operations queue, which would cause another child thread to not get any and go out of sync.

After applying the accepted move as a step, the parent thread also updates the best known solution if the new solution's score is better. Unless the termination criterion is reached, the process restarts and the parent thread generates b active moves. Then the parent thread continues as before: every time it receives an evaluated move, it generates a new active move. As for the old active moves that reached a child thread before the operations queue got cleared, they eventually end up in the result queue too and they are completely ignored when they finally arrive to the parent thread in the new step (based on their outdated step ID).

This orchestration, shown in Figure 4, ensures that every child thread has its own copy of an equivalent working solution, eventually in sync with the parent thread, to incrementally evaluates each move in isolation. They calculate the same score as the parent thread would, regardless of when and which child thread ends up evaluating a specific move.

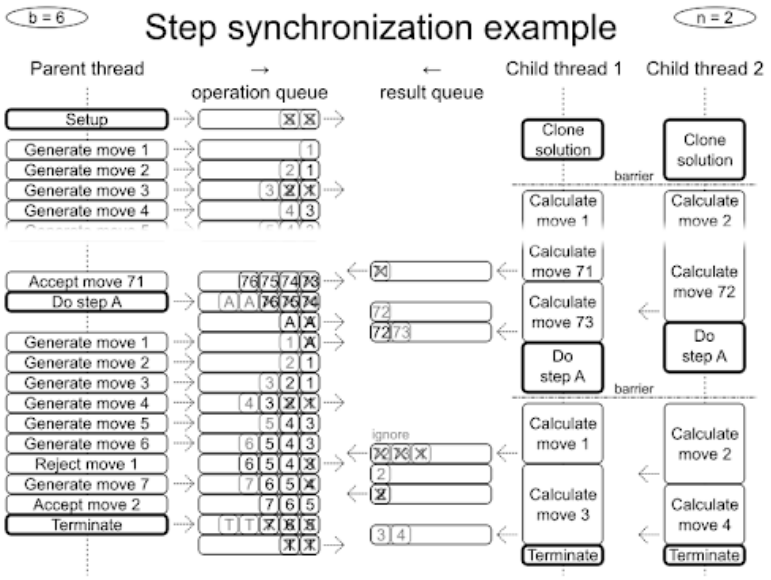

Fig. 4 Step synchronization procedure

If the termination criterion is reached, the parent thread clears the operation queue and puts n poison pill operations on the operation queue. When a child thread pulls such a poison pill, it terminates itself. Meanwhile the parent thread also terminates, returning the best solution encountered.

4 Experimental results

In order to evaluate the implemented parallelization method we ran a significant number of benchmarks on a range of optimization problems. The following problems, all of which are available as examples within OptaPlanner[1], are considered:

- Nurse rostering problem
- Vehicle routing problem
- Curriculum course timetabling problem
- Cloud balancing problem

All experiments[2] are performed on a dual Intel(R) Xeon(R) CPU E5-2660 v3 platform with 20 cores. Hyper Threading was disabled. A maximum heap size of 4GB RAM was allowed for the Java Virtual Machine (JDK 8), which was revealed to be sufficient for all experiments. OptaPlanner version 7.13.0.Final was used. Each algorithm configuration was run 10 times on each set of problem instances with different random seeds. As a termination condition, a fixed time of 300 seconds was used for all runs.

In the next sections we will first present the summarized results for each test case. Following this, Section 4.5 will provide a more thorough discussion of the results.

4.1 Nurse rostering problem results

In the nurse rostering problem, as defined by the first International Nurse Rostering Competition 2010 [9], shifts must be assigned to nurses while taking into account various constraints such as skill requirements, employee availability and unwanted shift patterns. We tested on the competition's medium, medium-late and medium-hidden[3] instances. Figure 5 shows the average speedups over all instances for Late Acceptance, Simulated Annealing and Tabu Search. For each algorithm the default version without multithreading is compared against a multithreaded version with step chasing using 2, 4, 8 and 16 threads. It is clear from the figure that the multithreaded solver offers good speedups compared to the default version, with increasing speedups up to 16 threads. The most significant speedups are reached by Tabu Search (which is up to 9.32 times faster when using 16 threads).

In addition to the realized speedups we also want to study the effect of parallelization on the quality of the solutions. Table 1 shows the comparison of solution quality (score) between a Single walk (1 thread), a multi-walk[4] (8

[1] OptaPlanner code and documentation available at: `https://www.optaplanner.org`

[2] A note on how to reproduce these results can be found at: `https://www.optaplanner.org/code/benchmarks.html`

[3] `https://www.kuleuven-kulak.be/nrpcompetition/instances-results`

[4] Since a real multi-walk is not implemented in OptaPlanner, the multi-walk results are defined by taking the best out of 8 independent walks.

independent walks/threads) and the multithreaded solver with step chasing (8 threads) on the nurse rostering problem. Scores are averaged over 10 runs. The columns **Δ single** and **Δ multi-walk** indicate the solution quality difference between the step chasing and the single, and between the step chasing and the multi-walk, respectively. The average results indicate that, when given 8 threads, multithreading with step chasing outperforms a single walk, and for Late Acceptance and Tabu Search also outperforms a multi-walk strategy.

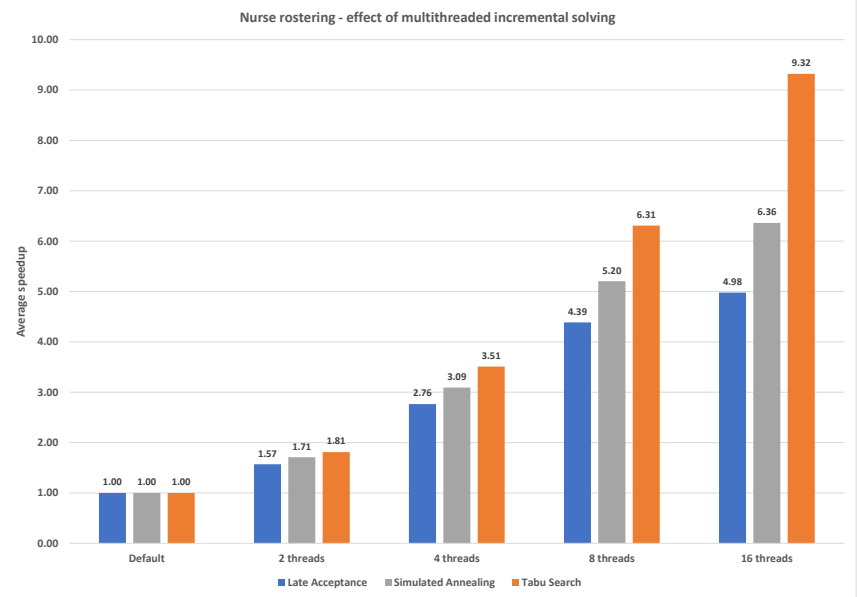

Fig. 5 Average speedups on the nurse rostering problem for Late Acceptance, Simulated Annealing and Tabu Search.

4.2 Vehicle routing problem results

In the vehicle routing problem, we tested on a set of instances from the vrp-rep benchmark website [5]: belgium-road-time-n50-k10, belgium-road-time-n100-k10, belgium-road-time-n500-k20, belgium-road-time-n1000-k20, belgium-road-time-n2750-k55.vrp. These include capacity constraints and realistic asymmetric distances calculated from OpenStreetMap. Figure 6 illustrates the average speedups over all instances for Late Acceptance, Simulated Annealing and Tabu Search on this problem. For each algorithm the default version without multithreading is again compared against a multithreaded version using 2, 4,

[5] `http://www.vrp-rep.org/datasets.html`

Alg.	Dataset	Single	Multi-walk 8T	Step chasing 8T	Δ single	Δ multi-walk
LA	medium01	346,3	334,0	334,9	3,29%	-0,27%
LA	medium02	347,1	333,0	334,4	3,66%	-0,42%
LA	medium03	334,8	328,0	326,0	2,63%	0,61%
LA	medium04	348,9	337,0	337,4	3,30%	-0,12%
LA	medium05	386,1	383,0	382,1	1,04%	0,23%
LA	medium_hint01	152,3	120,0	121,7	20,09%	-1,42%
LA	medium_hint02	268,9	190,0	160,0	40,50%	15,79%
LA	medium_hint03	314,7	187,0	190,6	39,43%	-1,93%
LA	medium_late01	330,6	242,0	247,2	25,23%	-2,15%
LA	medium_late02	102,2	99,0	96,1	5,97%	2,93%
LA	medium_late03	113,2	84,0	85,1	24,82%	-1,31%
LA	medium_late04	113,7	106,0	105,5	7,21%	0,47%
LA	medium_late05	365,8	221,0	225,6	38,33%	-2,08%
Late Acceptance average					**16,58%**	**0,80%**
SA	medium01	254,7	251,0	248,4	2,47%	1,04%
SA	medium02	253,7	248,0	248,4	2,09%	-0,16%
SA	medium03	250,6	244,0	244,3	2,51%	-0,12%
SA	medium04	251,6	246,0	245,0	2,62%	0,41%
SA	medium05	316,2	312,0	311,1	1,61%	0,29%
SA	medium_hint01	52,2	43,0	43,2	17,24%	-0,47%
SA	medium_hint02	103,6	95,0	97,1	6,27%	-2,21%
SA	medium_hint03	167,6	142,0	148,6	11,34%	-4,65%
SA	medium_late01	181,5	178,0	176,3	2,87%	0,96%
SA	medium_late02	36,1	25,0	27,5	23,82%	-10,00%
SA	medium_late03	41,7	35,0	36,7	11,99%	-4,86%
SA	medium_late04	47,4	42,0	40,8	13,92%	2,86%
SA	medium_late05	166,9	153,0	152,5	8,63%	0,33%
Simulated Annealing average					**8,26%**	**-1,28%**
TS	medium01	255,3	252,0	248,2	2,78%	1,51%
TS	medium02	252,1	248,0	245,7	2,54%	0,93%
TS	medium03	248,3	244,0	243,2	2,05%	0,33%
TS	medium04	253,1	248,0	246,8	2,49%	0,48%
TS	medium05	324,9	320,0	315,3	2,95%	1,47%
TS	medium_hint01	60,2	55,0	53,1	11,79%	3,45%
TS	medium_hint02	147,0	130,0	138,4	5,85%	-6,46%
TS	medium_hint03	203,1	189,0	185,0	8,91%	2,12%
TS	medium_late01	223,9	208,0	210,0	6,21%	-0,96%
TS	medium_late02	45,9	42,0	37,1	19,17%	11,67%
TS	medium_late03	51,2	49,0	45,1	11,91%	7,96%
TS	medium_late04	50,7	45,0	45,0	11,24%	0,00%
TS	medium_late05	218,8	198,0	200,1	8,55%	-1,06%
Tabu Search average					**7,42%**	**1,65%**

Table 1 Score comparison on the nurse rostering problem for single-walk, multi-walk (8 threads), and step chasing (8 threads). Lower scores indicate better solutions.

8 and 16 threads. The best results are obtained when using 4 threads. More significant speedups are obtained for larger instances, while the multithreaded version offers no benefit for the two smallest instances .

Table 2 shows the comparison of solution quality (score) between a Single walk (1 thread), a multi-walk (8 independent walks/threads) and the multi-threaded solver with step chasing (8 threads) on the vehicle routing problem. Scores are averaged over 10 runs. The columns **Δ single** and **Δ multi-walk** indicate the solution quality difference between the step chasing and the single, and between the step chasing and the multi-walk, respectively. Given the low speedup values on this problem when 8 threads are available, no significant benefit can be seen from the results. In fact, a multi-walk strategy performance better in most cases.

4.3 Curriculum course timetabling results

We also tested on the curriculum course timetabling problem. Lectures have to be assigned to rooms and periods, under constraints such as teacher conflicts, room capacity and curriculum compactness. Instances from the international

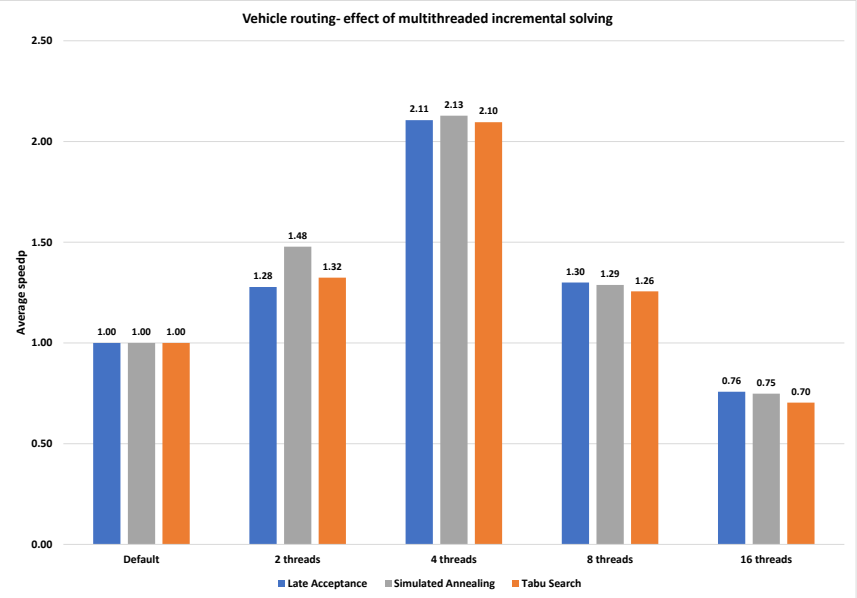

Fig. 6 Average speedups on the vehicle routing problem for Late Acceptance, Simulated Annealing and Tabu Search.

Alg.	Dataset	Single	Multi-walk 8T	Step chasing 8T	Δ single	Δ multi-walk
LA	belgium-road-time-n50-k10	114687,9	114320,2	114544,0	0,13%	-0,20%
LA	belgium-road-time-n100-k10	141631,7	140883,1	141667,2	-0,03%	-0,56%
LA	belgium-road-time-n500-k20	360078,3	356830,5	358769,3	0,36%	-0,54%
LA	belgium-road-time-n1000-k20	556317,0	555025,2	541285,1	2,70%	2,48%
LA	belgium-road-time-n2750-k55	1195007,5	1194608,9	1190600,6	0,37%	0,34%
Late Acceptance average					**0,71%**	**0,30%**
SA	belgium-road-time-n50-k10	117740,9	116871,7	117757,9	-0,01%	-0,76%
SA	belgium-road-time-n100-k10	152050,8	145775,7	152893,6	-0,55%	-4,88%
SA	belgium-road-time-n500-k20	374649,3	372377,2	375574,1	-0,25%	-0,86%
SA	belgium-road-time-n1000-k20	522698,7	512684,7	524798,4	-0,40%	-2,36%
SA	belgium-road-time-n2750-k55	1127755,5	1118702,3	1114100,7	1,21%	0,41%
Simulated Annealing average					**0,00%**	**-1,69%**
TS	belgium-road-time-n50-k10	118009,2	116340,7	117592,1	0,35%	-1,08%
TS	belgium-road-time-n100-k10	148411,5	144667,8	148509,0	-0,07%	-2,66%
TS	belgium-road-time-n500-k20	362010,5	353631,9	362017,5	0,00%	-2,37%
TS	belgium-road-time-n1000-k20	525642,8	521534,0	522372,1	0,62%	-0,16%
TS	belgium-road-time-n2750-k55	1200391,1	1200391,1	1200391,1	0,00%	0,00%
Tabu Search average					**0,18%**	**-1,25%**

Table 2 Score comparison on the vehicle routing problem for single-walk, multi-walk (8 threads), and step chasing (8 threads). Lower scores indicate better solutions.

Timetabling Competition 2007 Track 3 `http://www.cs.qub.ac.uk/itc2007/` are used.

Figure 7 shows the average speedups over all instances for Late Acceptance, Simulated Annealing and Tabu Search on this problem. For each algorithm the default version without multithreading is compared to a multithreaded version using 2, 4, 8 and 16 threads. The figure shows how good speedups are achieved when using 4 or more threads. However, no significant gains are realized when using more than 4 threads. Once again, Tabu Search benefits most from multithreaded incremental solving.

Table 3 shows the comparison of solution quality (score) between a Single walk (1 thread), a multi-walk (8 independent walks/threads) and the multithreaded solver with step chasing (8 threads) on the curriculum course timetabling problem. Scores are averaged over 10 runs. The columns **Δ single** and **Δ multi-walk** indicate the solution quality difference between the step chasing and the single, and between the step chasing and the multi-walk, respectively. It should be noted, that for this problem not all runs returned a feasible solution. Therefore, infeasible solutions are penalized with a penalty value of 100,000 for each hard constraint violation. On this problem, the multi-walk strategy is capable of finding more feasible solutions. However, when both strategies are able to find feasible solutions, the step chasing shows better results when Late Acceptance or Simulated Annealing is used.

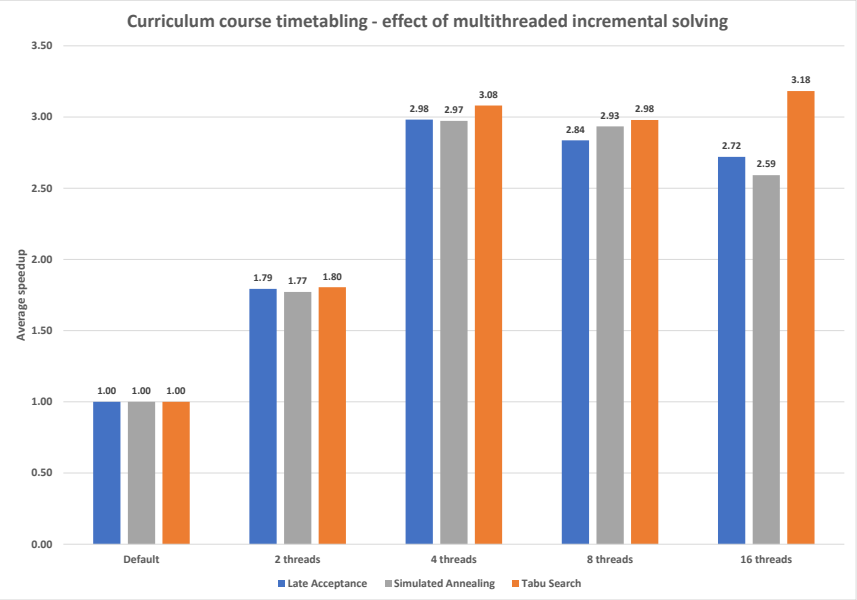

Fig. 7 Average speedups on the curriculum course timetabling problem for Late Acceptance, Simulated Annealing and Tabu Search.

4.4 Cloud balancing results

We have also tested the incremental solver on the cloud balancing problem. This problem concerns the assignment of computer processes to machines subject to CPU, RAM and network bandwidth constraints with the aim of reducing operating costs. The instances considered are 100computers-300processes, 200computers-600processes, 400computers-1200processes, 800computers-2400processes

Alg.	Dataset	Single	Multi-walk 8T	Step chasing 8T	Δ single	Δ multi-walk
LA	comp01	10,7	10,0	6,1	42,99%	39,00%
LA	comp02	214,5	191,0	199,9	6,81%	-4,66%
LA	comp03	209,5	191,0	188,7	9,93%	1,20%
LA	comp04	127,2	111,0	104,3	18,00%	6,04%
LA	comp05	250626,3	583,0	250654,5	-0,01%	-42893,91%
LA	comp06	244,3	222,0	181,2	25,83%	18,38%
LA	comp07	269,3	251,0	186,2	30,86%	25,82%
LA	comp08	147,7	130,0	120,5	18,42%	7,31%
LA	comp09	224,0	208,0	211,0	5,80%	-1,44%
LA	comp10	200,7	185,0	156,4	22,07%	15,46%
LA	comp11	1,1	0,0	0,0	100,00%	0,00%
LA	comp12	10533,3	499,0	534,8	94,92%	-7,17%
LA	comp13	163,7	154,0	147,0	10,20%	4,55%
LA	comp14	158,7	152,0	136,0	14,30%	10,53%
Late Acceptance average					**28,58%**	**-3055,64%**
SA	comp01	8,2	7,0	5,8	29,27%	17,14%
SA	comp02	40210,2	192,0	70184,3	-74,54%	-36454,32%
SA	comp03	205,5	178,0	197,3	3,99%	-10,84%
SA	comp04	99,9	92,0	79,2	20,72%	13,91%
SA	comp05	680749,7	600676,0	730764,1	-7,35%	-21,66%
SA	comp06	60183,0	167,0	30152,1	49,90%	-17955,15%
SA	comp07	184,2	167,0	125,7	31,76%	24,73%
SA	comp08	118,4	103,0	85,7	27,62%	16,80%
SA	comp09	187,2	174,0	176,0	5,98%	-1,15%
SA	comp10	147,0	135,0	113,7	22,65%	15,78%
SA	comp11	0,8	0,0	0,0	100,00%	0,00%
SA	comp12	30569,5	543,0	70568,6	-130,85%	-12896,06%
SA	comp13	126,6	120,0	106,4	15,96%	11,33%
SA	comp14	140,1	123,0	116,4	16,92%	5,37%
Simulated Annealing average					**8,00%**	**-4802,44%**
TS	comp01	7,8	6,0	5,8	25,64%	3,33%
TS	comp02	150242,3	233,0	160217,0	-6,64%	-68662,66%
TS	comp03	259,5	239,0	230,6	11,14%	3,51%
TS	comp04	129,3	105,0	114,0	11,83%	-8,57%
TS	comp05	730741,0	600693,0	790767,8	-8,21%	-31,64%
TS	comp06	120217,8	220,0	90185,4	24,98%	-40893,36%
TS	comp07	220,2	200,0	173,7	21,12%	13,15%
TS	comp08	149,9	137,0	135,8	9,41%	0,88%
TS	comp09	230,2	218,0	221,0	4,00%	-1,38%
TS	comp10	179,8	160,0	145,1	19,30%	9,31%
TS	comp11	10,2	1,0	1,9	81,37%	-90,00%
TS	comp12	120595,5	100564,0	100587,1	16,59%	-0,02%
TS	comp13	157,1	125,0	143,4	8,72%	-14,72%
TS	comp14	177,5	143,0	154,9	12,73%	-8,32%
Tabu Search average					**16,57%**	**-7834,32%**

Table 3 Score comparison on the curriculum course timetabling problem for single-walk, multi-walk (8 threads), and step chasing (8 threads). Lower scores indicate better solutions. Infeasible solutions are penalized with a penalty value of 100,000 for each hard constraint violation

and 1600computers-4800processes from optaplanner-examples. Figure 8 illustrates the average speedups over all instances when Late Acceptance, Simulated Annealing and Tabu Search are applied to this problem. For each algorithm the default version without multithreading is compared against a multithreaded version using 2, 4, 8 and 16 threads. The figure shows that the multithreaded incremental solver runs faster than the default version on this problem. The largest speedups are realized for the Tabu Search with 8 threads. However, further increasing the number of threads to 16 appears to have a negative impact on the speedup for Tabu Search.

Table 3 shows the comparison of solution quality (score) between a Single walk (1 thread), a multi-walk (8 independent walks/threads) and the multithreaded solver with step chasing (8 threads) on the cloud balancing problem. Scores are averaged over 10 runs. The columns **Δ single** and **Δ multi-walk** indicate the solution quality difference between the step chasing and the single, and between the step chasing and the multi-walk, respectively. Although a small benefit over a single walk can be observed, the step chasing speedups do

Alg.	Dataset	Single	Multi-walk 8T	Step chasing 8T	Δ single	Δ multi-walk
LA	100computers-300processes	111036,0	110650,0	110715,0	0,29%	-0,06%
LA	200computers-600processes	193061,0	191490,0	192793,0	0,14%	-0,68%
LA	400computers-1200processes	432172,0	429110,0	432230,0	-0,01%	-0,73%
LA	800computers-2400processes	890808,0	886520,0	889665,0	0,13%	-0,35%
LA	1600computers-4800processes	1788136,0	1783800,0	1784911,0	0,18%	-0,06%
Late Acceptance average					**0,14%**	**-0,38%**
SA	100computers-300processes	110335,0	109950,0	110142,0	0,17%	-0,17%
SA	200computers-600processes	192335,0	191370,0	192183,0	0,08%	-0,42%
SA	400computers-1200processes	429866,0	428300,0	430913,0	-0,24%	-0,61%
SA	800computers-2400processes	886034,0	884510,0	884609,0	0,16%	-0,01%
SA	1600computers-4800processes	1765000,0	1763950,0	1759957,0	0,29%	0,23%
Simulated Annealing average					**0,09%**	**-0,20%**
TS	100computers-300processes	110099,0	109410,0	108723,0	1,25%	0,63%
TS	200computers-600processes	191490,0	190530,0	190148,0	0,70%	0,20%
TS	400computers-1200processes	432013,0	429360,0	428019,0	0,92%	0,31%
TS	800computers-2400processes	895670,0	891230,0	881928,0	1,53%	1,04%
TS	1600computers-4800processes	1803831,0	1801300,0	1761139,0	2,37%	2,23%
Tabu Search average					**1,36%**	**0,88%**

Table 4 Score comparison on the cloud balancing problem for single-walk, multi-walk (8 threads), and step chasing (8 threads). Lower scores indicate better solutions.

not translate into significantly better solutions on this problem when compared to a multi-walk.

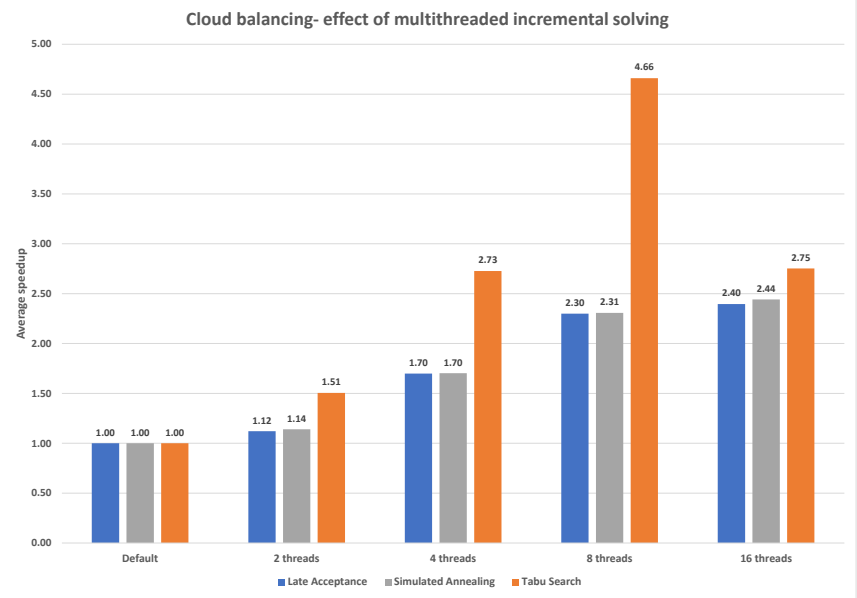

Fig. 8 Average speedups on the cloud balancing problem for Late Acceptance, Simulated Annealing and Tabu Search.

4.5 Results discussion

The results on the four tested benchmark problems show that the proposed multithreaded implementation offers significant speedups in general. However, the speedups depend upon the problem characteristics such as scale.

It can be seen that Tabu Search benefits more from the multithreaded implementation than Simulated Annealing and Late Acceptance. This can be attributed to the fact that Tabu Search typically evaluates more moves in each step than the other algorithms. This means that it performs fewer steps and therefore has less step chasing overhead.

When compared to a (non parallel) single-walk, the step chasing strategy results in significantly better solutions. However, when compared to a multi-walk strategy, which offers a more diverse search, step chasing does not always results in better solutions. In particular, when feasibility is difficult to achieve a multi-walk strategy seems to be the better choice.

5 Conclusion and future work

This paper introduced an effective multithreaded incremental solving method for metaheuristics using the concept of step chasing. This method was integrated into the OptaPlanner solver and compared against the default non-multithreaded metaheuristics on four difficult combinatorial optimization problems: the nurse rostering problem, the vehicle routing problem, the curriculum course timetabling problem and the cloud balancing problem. It shows significant speedups and better solutions.

An important requirement of the method are reproducible runs. This highly impacted architecture. Dropping this requirement could yield additional performance and scaling benefits. Future research is needed to quantify such benefits.

As it stands, experiments show diminishing returns as the child thread count increases or as move evaluation duration decreases. This is likely due to congestion in the operation and result queue. This leads to two potential improvements which ought to be addressed in future research:

- Ship multiple moves in bulk through the operation and result queues in order to decrease the frequency with which parent and child threads operate on it.
- Redesign an architecture in which the child threads do not share the same operation and result queue. For example, there could be one operation queue per child thread, such that only the parent thread and one child thread interact with the same queue (except for work stealing by other child threads which have already emptied their own operation queue). Alternatively, the result queue could be replaced by a reduce operation similar to that found in MapReduce.

Further research is also needed to determine how well this technique applies on other metaheuristics, such Genetic Algorithms, Particle Swarm Optimization and Ant Colony Optimization.

Acknowledgements Editorial consultation provided by Luke Connolly (KU Leuven).

References

1. Alba, E.: Parallel metaheuristics: a new class of algorithms, vol. 47. John Wiley & Sons (2005)
2. Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research **20**(1), 1–48 (2013)
3. Burke, E.K., Bykov, Y.: A late acceptance strategy in hill-climbing for exam timetabling problems. In: PATAT 2008 Conference, Montreal, Canada, pp. 1–7 (2008)
4. Burke, E.K., De Causmaecker, P., Berghe, G.V., Van Landeghem, H.: The state of the art of nurse rostering. J. of Scheduling **7**(6), 441–499 (2004). DOI 10.1023/B:JOSH.0000046076.75950.0b. URL `https://doi.org/10.1023/B:JOSH.0000046076.75950.0b`
5. Crainic, T.G., Toulouse, M.: Parallel strategies for meta-heuristics. In: Handbook of metaheuristics, pp. 475–513. Springer (2003)
6. Crainic, T.G., Toulouse, M.: Parallel meta-heuristics. In: Handbook of metaheuristics, pp. 497–541. Springer (2010)
7. De Smet, G., open source contributors: OptaPlanner User Guide (2006). URL `https://www.optaplanner.org`. OptaPlanner is an open source constraint solver in Java
8. Glover, F., Laguna, M.: Tabu search. In: Handbook of combinatorial optimization, pp. 2093–2229. Springer (1998)
9. Haspeslagh, S., De Causmaecker, P., Schaerf, A., Stølevik, M.: The first international nurse rostering competition 2010. Annals of Operations Research **218**(1), 221–236 (2014)
10. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)
11. Talbi, E.G.: Parallel combinatorial optimization, vol. 58. John Wiley & Sons (2006)
12. Verhoeven, M.G.A., Aarts, E.H.L.: Parallel local search. Journal of Heuristics **1**(1), 43–65 (1995). DOI 10.1007/BF02430365. URL `https://doi.org/10.1007/BF02430365`