GPGPU-based Parallel Algorithms for Scheduling Against Due Date

Abhishek Awasthi, Jörg Lässig, Jens Leuschner
Department of Computer Science,
University of Applied Sciences Zittau/Görlitz,
Görlitz, Saxony, Germany
Email: {aawasthi,jlaessig,jleuschner}@eadgroup.org

Thomas Weise
School of Computer Science and Technology,
University of Science and Technology of China,
Hefei, Anhui, China
Email: tweise@ustc.edu.cn

***Abstract*—We present an in-depth analysis and implementation of parallel programming on two NP-hard combinatorial optimization problems, namely, the Common Due-Date (CDD) problem and the Unrestricted CDD with Controllable Processing Times (UCDDCP). The CDD and UCDDCP require scheduling and sequencing a certain number of jobs with different processing times on a single machine against a common due-date. The goal is to minimize the total weighted penalty incurred due to earliness or tardiness of the jobs and the penalty due to the compression of the processing times of the jobs. In the UCDDCP, the processing time of a job can be reduced by letting the machine work at a faster pace, which, however, comes at a (*compression penalty*) cost per time unit. Optimization for both is carried out by hybrid algorithms, composed of a metaheuristic that creates good job sequences and an $O(n)$ algorithm which finds the optimal completion times for the all the jobs in such sequences created by the metaheuristic algorithms. We investigate both Simulated Annealing (SA) and a Discrete Particle Swarm Algorithm (DPSO) for this purpose. Parallel versions of both algorithms are implemented based on CUDA™. Experiments are carried out on the benchmark instances provided in the OR-library and executed on a Nvidia® graphics processing unit. We find that the parallel SA algorithm performs very well while obtaining speedups of 100× within a deviation of two percent compared to the best known solutions. Furthermore, our parallel algorithms also improve the best known solution values for several benchmark instances.**

This is a preview version of this article [20] (see page 11 for the reference). It is posted here for your personal use and not for redistribution. The final publication and definite version is available from IEEE Computer Society (who hold the copyright) at http://www.ieee.org/. See also http://dx.doi.org/10.1109/IPDPSW.2016.66.

I. Introduction

Scheduling and sequencing are decision making processes to determine the order of the processing of jobs and the time at which each job should start, on one or more machines. In other words, they involve allocation of resources over time to perform a set of tasks. The requirement of scheduling occurs in everyday situations. Most of these scheduling problems are NP-hard. Due to the unavailability of exact deterministic algorithms to solve these problems, they are mostly dealt with metaheuristic optimization algorithms. The size of the search space and the complexity of these problems both reduce the ability of these algorithms to obtain optimal or near optimal solutions. In the last twenty years, there has been a rapid development in the utilization of Graphical Processing Units (GPUs) for their massive computational capability. Initially GPUs were designed to compute hardware-assisted bitmap operations to assist the display and usability of graphical operating systems. However, they can now be used to perform complex scientific computations, which has led to the term GPGPU (*i.e.*, General Purpose Computing on Graphic Processing Units). For a range of algorithms, the highly parallel structure of GPUs makes them more effective than CPUs. In the field of combinatorial optimization, GPUs have been successfully utilized, yielding several folds of speed-ups and better solutions.

Chakroun *et al.* propose a branch and bound algorithm to solve large NP-hard combinatorial optimization problems on GPUs [1]. They perform experiments on the flow shop scheduling problem and speed-ups of up to 160 are achieved compared to the corresponding CPU implementations. Spampinato and Elster propose a revised simplex algorithm for CPUs and GPUs to solve linear programming problems [2]. This approach uses modern CUDA libraries and the ATLAS library to solve linear programs with up to 2000 variables. The GPU version of this approach is about 2.5 times faster. GPUs have also been successfully used to solve combinatorial optimization problems with metaheuristic algorithms. Some of these approaches evaluate the solutions in parallel on the GPU, while the others outsource some computations to it or perform the full computation on the GPU. Luong *et al.* introduce a parallel local search algorithm using the GPU [3]. They perform computational studies on the Traveling Salesman Problem, the Quadratic Assignment Problem and the Permuted Perceptrons Problem while showing significant speed-ups in comparison to serial CPU implementations. In 2010 Luong *et al.* also investigate the parallelization of large neighborhood local search algorithms and experiments on binary problems offer speed-ups of up to 25 times [4]. Czapiński and Barnes propose a GPU based parallel tabu search algorithm for the Permutation Flowshop Scheduling Problem and compute the solutions 89 times faster than the CPU [5]. Ferreiro *et al.* proposed some strategies to parallelize Simulated Annealing metaheuristic and tested them on several benchmark problems [12]. Tsutsui and Fujimoto implement a parallel evolutionary algorithm to compute the Quadratic Assignment Problem on a GPU and obtain the results 12 times faster than the corresponding CPU implementation [6].

In this research work, we investigate GPU parallelization using CUDA on two NP-hard scheduling problems, the Common Due-Date problem (CDD) and the Unrestricted Common Due-Date problem with Controllable Processing Times (UCDDCP). For both the problems, any given sequence is optimized to its minimum weighted penalties by polynomial algorithms and the best job sequence is obtained with parallel

versions of Simulated Annealing (SA) and Discrete Particle Swarm Optimization (DPSO). The realization and development of these parallel metaheuristics on a GPU using CUDA is been explained in detail. The presented algorithms are optimized both in their performance and memory usage by using the Nvidia CUDA profiler. Later on, we present the results obtained via these parallel approaches for the benchmark problem instances and compare the quality of the solutions and runtimes between CPU and GPU implementations.

II. Problem Formulation

The CDD problem consists of scheduling and sequencing a fixed number of jobs on a single machine against a common due date. Each job possesses a processing time along with the earliness and tardiness penalties, which it incurs if it is scheduled before or after the due date, respectively. In the Common Due-Date problem with Controllable Processing Times (CDDCP), there is an additional compression penalty associated with all the jobs, which is incurred if the processing time of the jobs is reduced. Here, the reduction of the processing time of a job essentially means that the job is processed faster than its usual processing time, by the machine. In doing so, the machine needs to operate at rather extreme pace, consuming resources such a fuel. Due to this reason, a penalty per unit time is associated with each job in case it is processed faster, in other words, when the processing time is reduced. This penalty is termed as the compression penalty of the job. In this paper, we deal with the *Unrestricted* Common Due-Date problem with Controllable Processing Times (UCDDCP), where the due-date is always greater than or equal to the sum of the processing times of all the jobs. These problems can be mathematically formulated as below. Let,
n = number of jobs,
C_i = completion time of job i,
P_i = actual processing time for job i, $\forall i = 1, 2, \ldots, n$,
d = common due-date,
M_i = minimum processing time for job i,
E_i = earliness of job i, where $E_i = \max\{0, d - C_i\}$,
T_i = tardiness of job i, where $T_i = \max\{0, C_i - d\}$,
X_i = reduction in the processing time of job i, $X_i \leq P_i - M_i$,
α_i = earliness penalty per time unit for any job i,
β_i = tardiness penalty per time unit for any job i,
γ_i = compression penalty per time unit for any job i.

In the above formulation, C_i and X_i are the decision variables of the problem. The objective functions for the CDD and UCDDCP can then be written as

$$\text{CDD: } \min \sum_{i=1}^{n} (\alpha_i \cdot E_i + \beta_i \cdot T_i) \tag{1}$$

$$\text{UCDDCP: } \min \sum_{i=1}^{n} (\alpha_i \cdot E_i + \beta_i \cdot T_i + \gamma_i \cdot X_i) \tag{2}$$

III. Two-Layered Approach

We now present our overall parallel approach to optimize the CDD and UCDDCP problems. The idea behind our approach is to break the integer programming formulation of the NP-hard problems in two parts, *i.e.*, (i) finding a good (near optimal) job sequence and (ii) finding the optimal values of the completion times C_i for all the jobs in this job sequence. The job sequences are optimized by the GPGPU-parallelized metaheuristics. For each candidate sequence, they solve the sub-problem (ii) as linear program by applying specialized deterministic algorithms. These deterministic algorithms do not need to be parallelized as they have a polynomial runtime. To get a clear picture of this strategy, we need to look at the integer programming formulation for one of these problems. First, we present the 0-1 integer programming formulation for the UCDDCP problem:

$$\text{Minimize } \sum_{i=1}^{n} (\alpha_i \cdot E_i + \beta_i \cdot T_i + \gamma_i \cdot X_i)$$

subject to,

$$\begin{aligned}
&E_i \geq d - C_i, && i = 1, \ldots, n,\\
&T_i \geq C_i - d, && i = 1, \ldots, n,\\
&X_i \leq P_i - M_i, && i = 1, \ldots, n,\\
&C_i \geq P_i - X_i + C_j - G \cdot \delta_{ij}, && i = 1, \ldots, n-1,\\
& && j = i+1, \ldots, n,\\
&C_j \geq P_j - X_j + C_i - G \cdot (1 - \delta_{ij}), && i = 1, \ldots, n-1,\\
& && j = i+1, \ldots, n,\\
&C_i \geq P_i - X_i, && i = 1, \ldots, n,\\
&E_i, T_i, X_i \geq 0, && i = 1, \ldots, n,\\
&\boldsymbol{\delta_{ij} \in \{0, 1\}}, && i = 1, \ldots, n-1,\\
& && j = i+1, \ldots, n.
\end{aligned}$$

The variables have the same meaning as explained in Section II, except for G and δ_{ij}. G is some very large positive number and δ_{ij} is the decision variable with $\delta_{ij} \in \{0, 1\}$, $i = 1, 2, \ldots, n-1, j = i+1, \ldots, n$. We have $\delta_{ij} = 1$ if job i precedes job j in the sequence (not necessarily right before it) and vice-versa. The sole purpose of this binary decision variable is to find the optimal job sequence. Also notice, that there are several possible feasible sets of values for δ_{ij}. Any such a set of values of $delta_{ij}$ corresponds to a job sequence. Once it has been generated, the above formulation will become a linear programming formulation for those $delta_{ij}$ values. This linear program basically solves for the optimal completion times and the reduction of the processing times of all the jobs in that particular job sequence. This linear programming problem is polynomially solvable. Thus we can utilize the above strategy to break our NP-hard problems in two parts. One part deals in finding the completion times (C_i) of the jobs for any given job sequence. And the second part, utilizes the GPGPU parallelized metaheuristic algorithms to efficiently search for the optimal/best job sequence.

IV. Algorithm for a Given Job Sequence

LP solvers are quite slow when run iteratively on some general heuristic algorithm. We therefore developed faster specialized polynomial algorithms for the specific LPs, to gain from the above mentioned strategy. Lässig *et al.* [7] and Awasthi *et al.* [8] have recently worked on this strategy and developed linear algorithms for any given job sequence for the CDD and UCDDCP, or in other words, specialized linear algorithms for the resulting linear programming of the fixed δ_{ij} values. Due to lack of space, we only present the explanation of the algorithms and their illustrations, without proving their correctness. We urge the readers to refer to these research works for the proof of correctness of these algorithms.

A. Linear Algorithm for CDD job sequence

We now present the main idea for solving the single machine CDD problem for a given job sequence. The intuition for the approach is based on two properties presented in [9], [10] and the theorem provided by Lässig *et al.* [7] which is an extension to the property presented by Cheng [11] for the CDD with symmetric earliness/tardiness penalty. Cheng and Kahlbacher [9] prove that an optimal solution of a CDD has no machine idle time between any two jobs. Hall *et al.* [10] prove that for every instance of the CDD, there exists an optimal solution where either the first job starts at time $t = 0$ or one job completes at the due date ($C_i = d$), or both. Besides, Lässig *et al.* prove the following theorem regarding the position of the due-date in the optimal schedule for any job sequence [7].

Theorem 1. *If the optimal due-date position in any given job sequence of the CDD lies between C_{r-1} and C_r,* i.e.*, $C_{r-1} < d \leq C_r$, then the following relations hold for the two cases*
Case 1: If $C_{r-1} < d < C_r$

i) $\sum_{i=1}^{k-1} \alpha_i \leq \sum_{i=k}^{n} \beta_i,\ k = 1, 2, 3, \ldots, r$.

Case 2: If $C_r = d$

i) $\sum_{i=k+1}^{n} \beta_i \leq \sum_{i=1}^{k} \alpha_i,\ k = r, r+1, \ldots, n$ *and*

ii) $\sum_{i=1}^{k-1} \alpha_i \leq \sum_{i=k}^{n} \beta_i,\ k = 1, 2, 3, \ldots, r$.

Theorem 1 states that the difference in the sum of the tardiness and earliness penalties changes sign before and after the optimal due-date position, if the optimal solution has the due-date position at the completion time of a job. Hence, it is evident that to achieve the optimal solution we must first start scheduling the jobs from time $t = 0$. If the sum of the tardiness penalties is greater than the sum of the earliness penalties then we know that this initialization is the optimal solution. The reason is clear from Case 1 of Theorem 1, which basically states that the sum of the earliness penalties will be less than or equal to the sum of the tardiness penalties for maximum value of k. Evidently, the jobs can not be shifted to the left any further and hence the maximum value of the k will occur for the initial schedule with the first job starting at time $t = 0$. However, if the sum of the tardiness penalties is less than or equal to the sum of earliness penalties, then we shift all the jobs towards increasing completion times by placing the due-date position at the end of the completion times of jobs sequentially as long as the second property of Theorem 1 Case 2 holds. This procedure is continued until the sum of the tardiness penalties remains less than or equal to the sum of the earliness penalties.

1) Illustration of CDD Algorithm: In this Section we discuss our linear algorithm for CDD problem proposed in [7] with the help of an illustrative example consisting of $n = 5$ jobs. The data for our illustrative examples for CDD and UCDDCP is given in Table I. For the CDD problem, we do not need parameters M_i and γ_i. These parameters are later used to explain the UCDDCP illustration. We optimize the given sequence of jobs J where $J_i = i,\ i = 1, 2, \ldots, 5$. There are five jobs to be processed against a common due-date (d) of 16. The objective is to minimize Equation (1). We first initialize the completion times of all the jobs ($C_i,\ i = 1, 2, , \ldots, n$), such that $C_i = \sum_{k=1}^{i} P_k$ as shown in Figure 1. Hence, we have $C_i = \{6, 11, 13, 17, 21\}$. The first job starts processing at time $t = 0$ and the following jobs are processed one after the other, without any machine idle time. The due-date position lies in between the completion times of job 3 and 4.

TABLE I. THE DATA FOR THE EXEMPLARY CASE. THE PARAMETERS POSSESS THE SAME MEANING AS EXPLAINED IN SECTION II.

i	P_i	M_i	α_i	β_i	γ_i
1	6	5	7	9	5
2	5	5	9	5	4
3	2	2	6	4	3
4	4	3	9	3	2
5	4	3	3	2	1

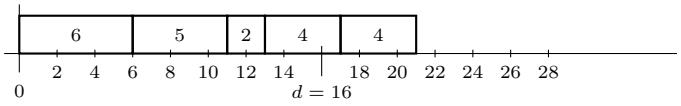

Fig. 1. Initialization of the schedule with the first job starting at time $t = 0$ and the remaining jobs following with no machine idle time.

In the next step we compute the vector $DT_i = C_i - d$, which gives us, $DT_i = \{-10, -5, -3, 1, 5\}$. Notice that DT_i fetches us the earliness or tardiness values for any job i. A negative value indicates its earliness, while the positive value indicates its tardiness. We then calculate the maximum index τ for $DT_i \leq 0$ or in other words, maximum index of the job which is either early or finishes at the due-date. With $\tau \neq 0$, we calculate the sum of the earliness penalties (pe) and the tardiness penalties (pl) of the jobs. Hence, initially $pe = 22$ and $pl = 5$. In the next step we shift all the jobs by DT_τ to check if the property (ii) of Case 2 in Theorem 1 holds. After a right shift of 3 units, we have $DT_i = \{-7, -2, 0, 4, 8\}$ and the 3rd job finishes at the due date, as shown in Figure 2.

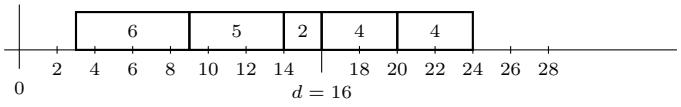

Fig. 2. Schedule with the completion time of job 3 lying at the due-date, after the right shift of all the jobs by 3 units.

Since this schedule still satisfies $pl < pe$, we again shift the jobs to the right, this time by the processing time of job 3. This also means that the 3rd job will now be tardy implying that the sum of the earliness penalty (pe) will reduce by $\alpha_\tau = 6$ and the sum of the tardiness penalties will increase by $\beta_\tau = 4$. Hence, we have $pe = 16$ and $pl = 9$. Figure 3 shows the schedule after the second right shift of the jobs, with job 2 finishing at the due-date. This process of sequential right shift is continued as long as the $pl < pe$. The final schedule obtained is shown in Figure 3, as for any further shift property (ii) of Case 2 in Theorem 1 does not hold. Finally, we calculate the earliness and tardiness of each job and multiply them with their corresponding penalty. The optimal solution found for this job sequence is thus 81.

B. Linear Algorithm for UCDDCP job sequence

We now present and illustrate the $O(n)$ algorithm, to optimize a job sequence of UCDDCP, given by [8]. Since we deal with the unrestricted case, we take the due date $d \geq \sum_{i=1}^{n} P_i$. The minimum processing time of a job is the

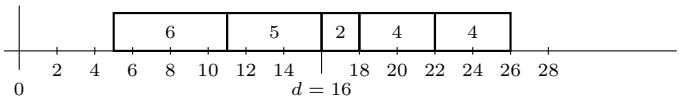

Fig. 3. Schedule with the completion time of job 2 lying at the due-date, after an additional right shift of all the jobs by 2 units.

time it takes to complete, if processed fast. The compression penalty is the penalty per unit time associated with each job when the processing time of the job is reduced.

Before explaining the algorithm, we present two properties for the UCDDCP problem, proved by Awasthi *et al.* [8].

Property 1. *If the due-date position in the optimal schedule of unrestricted case of the CDD lies at the completion time of some job r, then its position remains unchanged for the controllable case of the unrestricted CDD problem.*

Property 2. *If controlling the processing times fetches a better solution, then the compression of the processing times should be to their minimum value.*

The idea of the algorithm for the UCDDCP is to first optimize the sequence for the CDD problem and then compress the jobs towards the due date. Figure 4 shows the optimal schedule for the CDD problem with the second job completing at the due date. As Property 1 suggests, the position of the due date will remain unchanged for the UCDDCP problem. This in turn means that if the compression of the jobs yield a better solution, they necessarily have to be compressed towards the due date (indicated by the arrows in Figure 4).

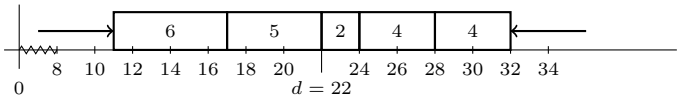

Fig. 4. Optimal CDD schedule with arrows indicating possible compression of the jobs towards the due-date.

Property 2 shows that compression of the jobs should be to their minimum value, if it fetches an improvement. Hence it gets clear that if required, the processing time of the jobs has to be reduced to their minimum possible value, indicated by M_i in Table I. Ultimately, we only need to determine which jobs should be reduced in terms of the processing times. To determine that, we start from the last job of the sequence.

1) Illustration of UCDDCP Algorithm: We take due date as $d = 22$ ($\geq \sum_{i=1}^{n} P_i$) and first consider job 5 which is tardy. The compression penalty of this job is 1, while the tardiness penalty is 2. Hence, a reduction of the processing time from $P_5 = 4$ to $M_5 = 3$ will increase the compression penalty by $X_5 \cdot \gamma_5$ (where, $X_5 = P_5 - M_5$) but reduce the tardiness penalty by $X_5 \cdot \beta_5$ (as the job is compressed towards the due date). Since $\beta_5 > \gamma_5$, reducing the processing time of job 5 fetches us an overall improvement in the penalty by $X_5 \cdot (\beta_5 - \gamma_5) = 1$. The schedule after this compression is shown in Figure 5.

In the next step, we move to the second last job, job 4 in this case. Job 4 is reducible by 1 time unit. Since there should not be any machine idle time in between jobs, observe that reducing job 4 towards the due date will reduce the tardiness of job 4 as well as job 5. Hence, a reduction will offer an

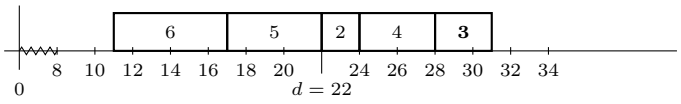

Fig. 5. Schedule with the reduction of job 5 to its minimum value of 3 time units.

improvement by $X_4 \cdot (\beta_4 + \beta_5 - \gamma_4)$. Since, $\beta_4 + \beta_5 - \gamma_4 = 3$, reduction of job 4 gives us a further improvement of 3 cost units, as shown in Figure 6.

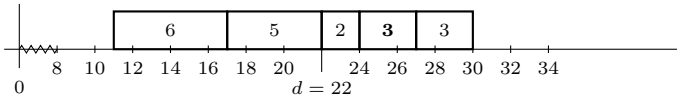

Fig. 6. Schedule with the reduction of job 4 to its minimum value of 3 time units.

This procedure is iterated over all the jobs. The procedure for all the tardy jobs remains the same as just explained. The jobs which are early or the one which finishes at the due date have to be dealt with in the opposite manner. A job is reduced if the compression penalty is less than the sum of the earliness penalties of all its preceding jobs. In our case, any further reduction will not improve the solution any further than Figure 6 with the optimal penalty cost of 77 for the considered job sequence.

V. Parallel Approach

There are several strategies to parallelize SA which are described by Ferreiro *et al.* [12]. The first strategy is an application dependent parallelization, where the operands of the objective function are divided into multiple processors. This approach is not applicable here since the operands of our objective functions occur sequentially, *i.e.*, each operand needs to wait for its preceding operand to complete. The second strategy is a domain decomposition, where the search space is divided into several parts and each processor searches for the best solution on its own sub-domain while sharing its results to other processors. The drawback of this strategy is the enormous size of the search space itself, and it becomes ineffective for a job size of 50 or more. Another approach uses multiple Markov chains to parallelize SA. This strategy executes multiple Markov chains asynchronously. After a certain period or at the end of the process, the processors communicate their results to each other. Depending on the number of communications, this strategy is classified by Ferreiro *et al.* [12] into *Asynchronous* and *Synchronous* simulated annealing.

A. Asynchronous Simulated Annealing

The *asynchronous* SA basically executes a large number of independent SA algorithms simultaneously *i.e.*, each processor performs separate SA asynchronously. When all algorithm instances are finished, a reduce operation is used to select the best result among all the processors. The initial configuration for the algorithm can be the same or different for all chains [12]. Figure 7 shows the asynchronous approach where ω processors execute SA simultaneously. Each chain uses t iterations, where each iteration reduces the temperature T_0 exponentially by the factor μ, where $0 < \mu < 1$. At the end a reduction operation is used to select the best solution.

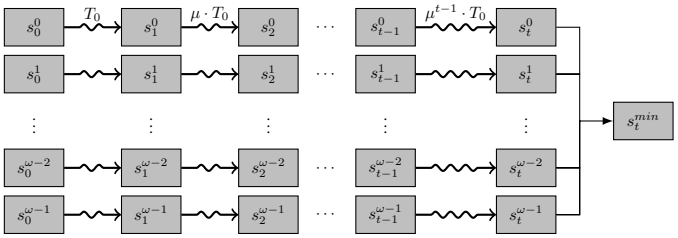

Fig. 7. Schematic representation of the asynchronous approach of parallel simulated annealing algorithm as suggested by Ferreiro *et al.* [12].

B. Synchronous Simulated Annealing

The *synchronous* version of SA starts in each processor with a random initial state s^i, where $i = 0, 1 \ldots \omega - 1$, and ω is the number of processors. In the next step, a Markov chain of a constant length M is simulated on each processor at a constant temperature. When all the processors have finished, they offer their final states s^i_j , where $i = 0, 1 \ldots, \omega - 1$, and $j = 1, 2, \ldots, t$. Again, a reduction operation is performed over the best solution of all the processors to obtain s^{min}_j at the end of the Markov chain iterations. For the next iteration of the Markov chain, s^{min}_j (which is the state with minimal cost for the objective function after the jth iteration) is selected as the initial state on all the processors at the next temperature level. Ferreiro *et al.* [12] claim that the exchange of the states and results can be very intensive in terms of the runtime. Figure 8 shows the synchronous approach with ω processors for t iterations of SA at each temperature level.

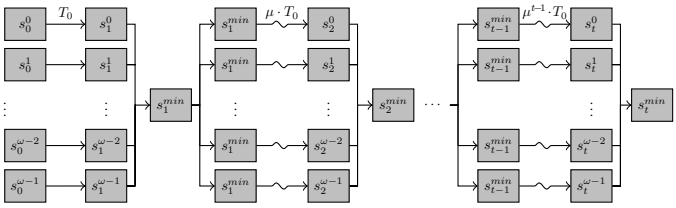

Fig. 8. Schematic representation of the synchronous approach of parallel simulated annealing algorithm as suggested by Ferreiro *et al.* [12].

VI. GPU Based Simulated Annealing

We now explain our parallel implementation of the Asynchronous Simulated Annealing algorithm [12]. The reason for choosing the asynchronous version over the synchronous SA is due to the premature convergence of the latter approach, examined from our experimental analysis. SA implemented on each CUDA thread involves the standard metropolis acceptance criterion and the exponential cooling schedule, as shown in Algorithm 1. The initial temperature T_0 is taken as the standard deviation of fitness values of 5000 different job sequences, generated randomly. This value for the initial temperature has been suggested by [13]. The exponential cooling rate of 0.88 has been adopted in this work, which is inferred from our experiments over a range of cooling rates. The neighborhood of any individual (*i.e.* job sequence) is generated by a perturbation mechanism, with a perturbation size of $Pert$. After every 10 SA iterations, $Pert$ number of jobs are selected at random from the current sequence and shuffled using the *Fisher Yates* algorithm, provided in [14].

Algorithm 1: The core Simulated Annealing algorithm running in each CUDA thread.

```
s ← s_0
T ← T_0
E ← Fitness(s)
while (i ≤ #Iterations) do
    s_new ← Neighbour(s)
    E_new ← Fitness(s_new)
    if exp((E − E_new)/T) ≥ rand(0, 1) then
        s ← s_new
        E ← E_new
    T ← T · μ
    i ← i + 1
Return s
```

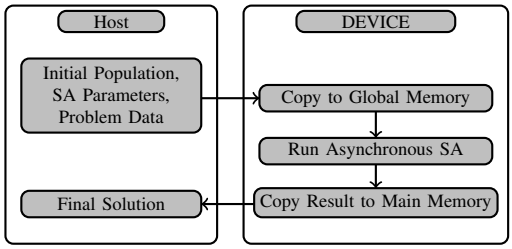

Fig. 9. Schematic representation of data transfer between the host and device. The data is transferred two times, back-and-forth, while the SA iterations are performed by the device.

The parallelization of the SA is initiated by allocating the number of threads and blocks on the GPU. CUDA offers three dimensional grids and blocks in (x, y, z) directions. The grid configuration G can be written as (g_x, g_y, g_z) and the block configuration B as (b_x, b_y, b_z). The grid configuration G implies that there are g_x, g_y and g_z number of blocks in x, y and z directions, respectively. Likewise, B configuration for the blocks implies b_x, b_y, b_z threads in the three dimensions. Let N be the total ensemble size and N_B be the block size, then a grid size of $\lceil N/N_B \rceil$ is allocated in the device for the parallel runs of the algorithms. In our work, we consider linear configurations for both the grid and the blocks, with $G = (\lceil N/N_B \rceil, 1, 1)$ and $B = (N_B, 1, 1)$, to avoid *race-conditions*. Henceforth, the initial job sequences are copied to the GPU global memory, along with the earliness, tardiness penalties and the processing times of the jobs. The due date d and the number of jobs n are transferred to the constant memory of the device to benefit from its broadcast mechanism. For the UCDDCP, the minimum processing times and the compression penalties are also copied to the GPU. Figure 9 shows the data transfer mechanism from the host to device and vice-versa. We then launch four different kernels, one after the other to calculate the *i)* fitness function, *ii)* perturbation, *iii)* SA acceptance, and *iv)* the best solution. Figure 10 shows these kernels in CUDA standard double *braket* notation. The CUDA threads are depicted as T1, T2, *etc*, implying that each thread is running the same algorithm in parallel.

A. Fitness Function Kernel

The fitness kernel first copies the earliness and tardiness cost per unit of time, inside the shared memory of a block, because this memory has shorter latency than global memory. This is one reason, why we consider linear configuration for

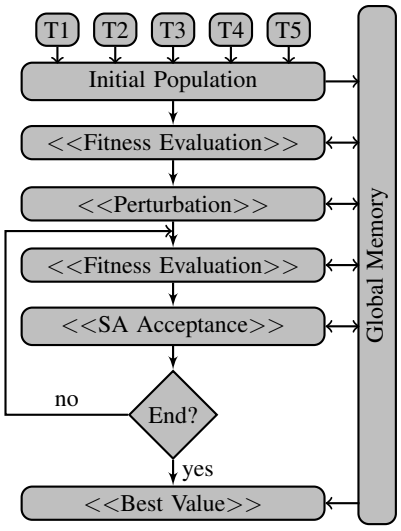

Fig. 10. Flow chart of the parallel Asynchronous Simulated Annealing

the grid G and block size B, in a single dimension only. Otherwise there would be many threads writing to the same address in shared memory which can in turn invoke the *race-conditions*. After writing to the shared memory, the kernel synchronizes the current block. This must be done because the warps within the block can be at different positions of the program depending on their scheduling. This synchronization ensures that all the write operations on the shared memory are finished before reading them. Otherwise, a thread would have the chance to read from an address where no thread has written. The processing times of the jobs are not cached because there are only a few reads from it inside the fitness function. Next, the fitness value for the job sequence in each thread uses the linear algorithms illustrated in Section IV, provided by [7] and [8] for CDD and UCDDCP, respectively.

B. Perturbation Kernel

The neighborhood of any job sequence is calculated by applying the *Fisher Yates* algorithm to a part of the parent job sequence. A sub-sequence of size $Pert = 4$ is selected from the parent job sequence and then the Fisher Yates algorithm is implemented on this sub-sequence while retaining the position of other jobs in the sequence. The random numbers required for the acceptance criterion are generated using the *cuRand* library of CUDA. Since *cuRand* provides only integer values, a normalization is carried out to obtain a floating point value in $[0, 1]$. After creating a new permutation for each thread, the fitness kernel is launched again to evaluate the solution for each newly created job sequence.

C. Acceptance Kernel

Henceforth, the acceptance kernel is launched, which simply checks if the given solution should be accepted or not, depending on the standard metropolis acceptance criterion of the SA algorithm. The random numbers in this kernel are again created using the *cuRand library*.

D. Reduction Kernel

The last kernel which is launched is the reduction kernel to find the minimum value among all the threads. The minimal value among all the threads is calculated by performing an atomic minimization function. The atomic function performs its operations inside the L2-Cache, which provides a good performance although the full process results in a sequential execution order. After invoking all these kernels, there should be a synchronization of the device, because all kernel calls are asynchronous and inside a queue. Hence, the synchronization operation is performed by the CPU. Through this operation, the CPU waits until the GPU finishes processing. At the end of the number of iterations, the global best solution is copied back to the host.

We now present a short explanation and the implementation of the core discrete particle swarm optimization algorithm. The parallel implementation of the DPSO algorithm on the GPU is carried out in the asynchronous manner, as explained for the SA. In the forthcoming section, we then present and compare our results for our GPGPU utilized parallelization of both SA and DPSO with the CPU implementations.

VII. Discrete Particle Swarm Optimization (DPSO)

Due to the lack of space and to avoid redundancy, we only explain the DPSO algorithm. Apart from the core aspect of the algorithm, the parallelization approach remains the same as for SA. Algorithm 2 provides the pseudo code for the DPSO implemented in this work, based on Pan *et al.* [15].

Algorithm 2: The core Discrete Particle Swarm Optimization Algorithm implemented.

```
Initialize Population
Evaluate fitness-function
while (i ≤ #Iterations) do
    find particles' best
    find swarms best
    Update particles' position
    Evaluate fitness-function
    i ← i + 1
Return Best Particle
```

Since, the traditional Particle Swarm Optimization [16] is designed to work on real valued positions, we use DPSO for our scheduling problems. Pan *et al.* have previously used DPSO on the no-wait flow shop problem [15]. DPSO contains an adjusted method to update the particles position, based on discrete job permutations. The updated method includes particles' position ($p_i(t)$), its best position ($p_i^b(t)$) and the swarms best position ($g(t)$) [15]. The new position $p_i(t+1)$ of the particle is given by

$$p_i(t+1) = c_2 \oplus F_3\left(c_1 \oplus F_2\left(w \oplus F_1\left(p_i(t)\right), p_i^b(t)\right), g(t)\right) \ . \quad (3)$$

In the above equation, operator $\oplus$ in any clause $x' = c \oplus f(x)$ means, operate function f on x with a probability of c, *i.e.* $x' = f(x)$, if $rand(0,1) < c$ and $x' = x$, if $rand(0,1) > c$. The first component of the update mechanism in Equation (3) is the particles velocity given by $\lambda_i(t+1) = w \oplus F_1(p_i(t))$, where F_1 represents a swap operator which selects two different jobs in the sequence ($p_i(t)$) randomly and swaps their

position in the job sequence, with a probability of w. The second component is given by $\delta_i(t+1) = c_1 \oplus F_2(\lambda_i(t), p_i^b(t))$ and represents the particles cognition part, where F_2 is a one-point crossover operator with a probability of c_1. The last component is the particles social part, representing the orientation on the swarm behaviour. The third component results in the new position of a particle and can be defined as $X_i(t+1) = c_2 \oplus F_3(\delta_i(t), g(t))$, where F_3 is a two point crossover.

VIII. Results

In this section, we present our results for the SA and DPSO parallelization on the GPU for the CDD and UCDDCP problems. We compare the two algorithms for both the problems and present the results for the benchmark instances. The benchmark instances for the CDD problem have been obtained from the OR-library [17] and Awasthi *et al.* provide the problem instances for UCDDCP in [8]. The runtime of the presented GPU based metaheuristics is influenced by the number of generations and the number of GPU threads which perform the linear algorithms for the job sequence. Figure 11 shows the influence of both the parameters for parallel Asynchronous SA on the UCDDCP problem. It is evident that increasing the number of generations or threads increases the runtime considerably. However, to achieve a good solution quality in a relatively short time, one needs to keep a balance between these two parameters and avoid invoking several serial processing of the blocks by running a very large number of iterations. Increasing the number of threads (population size) also increases the runtime of the algorithm, since a SM is limited to the number of threads per block it can operate on, simultaneously. This implies that loading several threads within a block results in serial processing of the blocks through the SM. On the other hand, increasing the block size offers less registers which a thread can use.

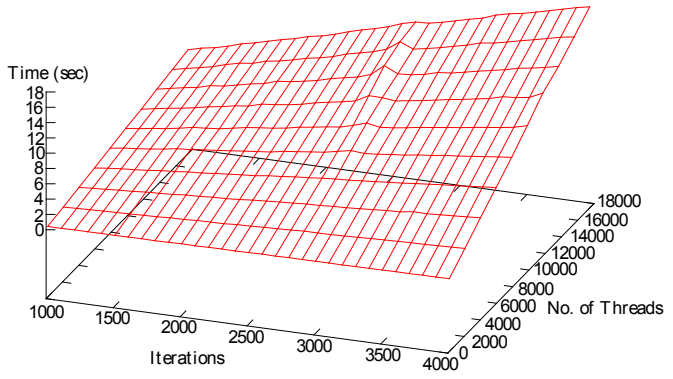

Fig. 11. Runtime in seconds for the parallel fitness function evaluations of the UCDDCP problem, with respect to the number of threads (population size) and the number of generations.

The theoretical limit for the number threads in one block of the Kepler device we use is 1024. However, after several experimental evaluations we observe that the best results for both the problems are achieved with a block size of 192. Selecting the number of grids and the number of iterations, is a rather complex task and is usually problem dependent, due to the above mentioned trade-off between the number of iterations and the number of threads. Having a high value for these parameters decrease the speed but on the other hand it fetches better results. Hence, after testing our approach on several experimental values, we choose to present our results for the two best configurations, which result in best speed-ups compared to the results provided by [7] and [8]. In both the cases, the grid size is kept at a constant value of four. This is not a high value considering the GPU device we use, but the results obtained are of excellent quality with a high speed-up, in comparison to the CPU runtimes. Hence, the total number of threads, which is also the population size is equal to 768, with 4 blocks each with 192 threads. We test SA and DPSO both for 1000 and 5000 iterations. The cooling factor for the Simulated Annealing is kept at 0.88 with an exponential cooling schedule and the perturbation size was taken as four, for all the problem instances. The implementation of the parallel algorithm was carried out on a GeForce GT 560M device, with 2 GB graphics card memory on a host CPU of 32 GB RAM with Intel-Xeon 2.4 GHz processor. Before presenting the results, we first explain some parameters used in the analysis of our results.

$$
\begin{aligned}
\text{SA}_{1000} &= \text{SA algorithm with 1000 iterations,}\\
\text{SA}_{5000} &= \text{SA algorithm with 5000 iterations,}\\
\text{DPSO}_{1000} &= \text{DPSO algorithm with 1000 iterations,}\\
\text{DPSO}_{5000} &= \text{DPSO algorithm with 5000 iterations,}\\
Z &= \text{Solution obtained with our parallel approach for any benchmark instance,}\\
Z_{best} &= \text{Solution obtained with the CPU version algorithms by [7] and [8],}\\
\%\Delta &= \text{Percentage Deviation of the GPU results with that of CPU, where } \%\Delta = \frac{(Z - Z_{best})}{Z_{best}} \cdot 100.
\end{aligned}
$$

A. Results for the CDD

We now present our results for the Common Due Date problem obtained with our parallel approaches. Table II presents the average percentage deviation ($\%\Delta$) for the CDD problem relative to the sequential implementation in Lässig *et al.* [7]. The percentage deviation shown in the table is the average over 40 different instances for each job size. The graphical representation of these percentage deviations is shown in Figure 12 as a bar chart.

TABLE II. Average percentage deviation of our approaches for each problem size for CDD, relative to Lässig *et al.* [7].

Jobs	SA_{1000}	SA_{5000}	DPSO_{1000}	DPSO_{5000}
10	0.159	0	0	0
20	0.793	0.392	0.141	0.033
50	0.442	0.243	0.652	0.146
100	0.386	0.307	2.048	0.463
200	0.437	0.388	4.854	1.148
500	0.734	0.354	15.562	3.807
1000	1.904	0.401	32.376	9.342

As can be seen from the table and the bar chart, SA performs extremely well keeping the percentage deviation for SA within two percent, while the quality of the results obtained by DPSO deteriorates as the number of jobs increase. The figure also shows, that for smaller instances up to 50 jobs, the DPSO version performs only marginally better than SA, in terms of the solution quality. The exact $\%\Delta$ values for the

bar chart can be found in Table II. The maximum $\%\Delta$ values for SA_{1000} and SA_{5000} are 1.9 and 0.4 percent, respectively. However, the percentage deviation for DPSO increases several folds, reaching the values of 32 and 9 percent, respectively, for 1000 and 5000 iterations. DPSO computes better values than SA only till job size of 50, but for higher instances it does not converge to better solutions than SA.

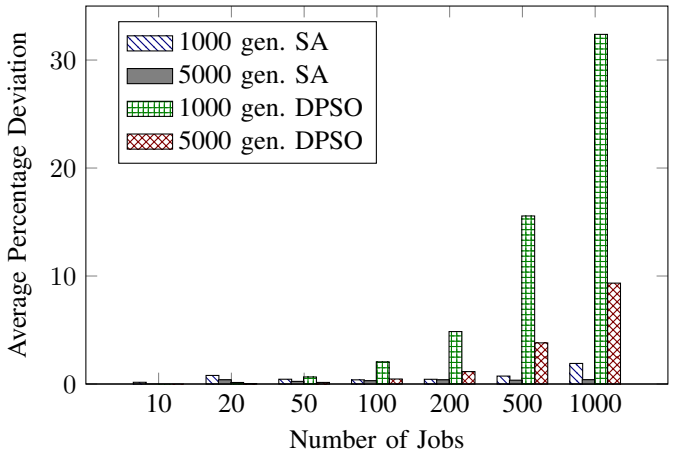

Fig. 12. Comparative average percentage deviation of our four parallel algorithms relative to the best known solutions of [7] for the CDD problem.

Among all the four approaches, SA_{5000} performs the best and fetches us a superior solution quality with average percentage deviation of less than 0.5 percent, for any instance. Moreover, comparing the runtimes of the CPU implementations by Lässig *et al.* [7] and Biskup and Feldmann [18], we observe that the speed-ups obtained by our parallel algorithms certainly prove their worth, as shown in Table III and Figure 13. Although the speed-up values for small instances are not so significant, the solutions for higher instance sizes of 50, 100, 200, 500 and 1000 are encouraging. Table III depicts the speed-ups of our four approaches with previous two CPU implementations by [7] and [18]. For SA_{1000}, our speed-up values in comparison to [18] are 227, 264, 619, 1137, 1971 and 3214 for problem sizes of 20 to 1000, respectively. We would like to mention here that the speed-ups calculated in this work are the ratio of the time required for the corresponding CPU implementation to the total runtime of our parallel algorithms incorporating all the memory transfers between the host and the device. Considering the fact that our solution quality is well within 2 percent of the best known solutions, these values of speed-ups show the effectiveness of GPU computing. The speed-ups with the very recent work of [7] are 40, 47, 94 and 111, for problem sizes of 100 to 1000, respectively. Considering the level of percentage deviation and the speed-ups obtained, we reckon that DPSO does not perform as efficient as the Simulated Annealing. Figure 14 shows the runtime plot of the SA and the DPSO algorithm with 1000 and 5000 generations for the CDD, along with the CPU rum-time of [7]. As can be seen the runtime for SA_{1000} is better for each given input size. SA_{5000} usually takes 5 times as much runtime as SA_{1000}. For an input size of 1000 jobs the SA_{5000} algorithm runs for about 17.26 seconds whereas the CPU version takes 379.36 seconds, offering a speed-up of 21, with an average percentage gap of just 0.4 percent. The table also shows that

TABLE III. OBTAINED SPEED-UPS OF THE PARALLEL ALGORITHMS FOR THE CDD PROBLEM RELATIVE TO [7] AND [18].

Jobs	SA_{1000}		SA_{5000}		$DPSO_{1000}$		$DPSO_{5000}$	
	[7]	[18]	[7]	[18]	[7]	[18]	[7]	[18]
10	1.9	4.7	0.5	1.3	1.2	2.9	0.5	1.2
20	3.8	227.6	1.1	65.4	1.9	113.8	0.6	36.7
50	11.8	264.5	2.9	65.1	4.8	107.7	1.2	28.0
100	40.6	619.3	9.2	141.7	12.7	195.1	3.0	46.6
200	47.7	1137.1	10.4	248.7	14.2	338.7	3.1	75.6
500	94.7	1971.4	19.7	410.2	23.6	492.2	5.4	113.5
1000	111.2	3214.8	21.9	635.1	24.6	711.8	5.6	164.2

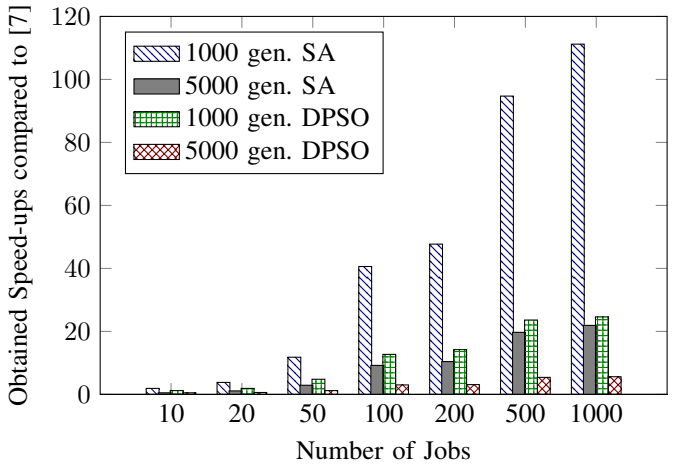

Fig. 13. Graphical representation of the obtained speed-ups of the parallel algorithms for the CDD problem relative to [7].

increasing the number of generations by a factor of five also increases the runtime by a factor about five, as expected. The runtime plot also shows that the DPSO algorithm is slower than SA, with number of iterations. Since we have the same CUDA thread counts and the number of iterations, it is evident from Table II and Figure 12 that Simulated Annealing outperforms the Discrete Particle Swarm Optimization algorithm.

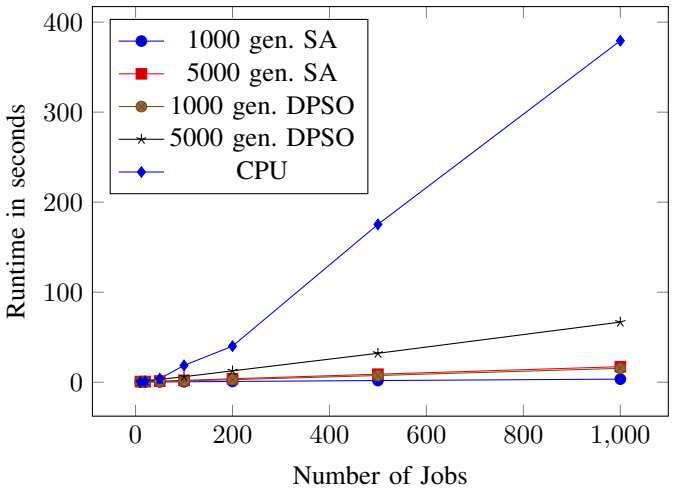

Fig. 14. Plot of the runtimes of the four parallel approaches and the CPU implementation of Lässig *et al.* [7] for the CDD problem.

B. Results for the UCDDCP

We now present our results for the Unrestricted Common Due-Date Problem with Controllable Processing Times (UCDDCP) using the parallel SA and DPSO with 1000 and 5000 generations, each. Figure 15 shows the relative percentage deviation of the GPU results in comparison to the CPU based algorithm for the UCDDCP by Awasthi *et al.* [8]. The negative values mean that the results obtained by these parallel algorithms are better than the best known solution provided by [8]. The exact percentage deviations for all the jobs can be found in Table IV. As in the case for CDD, we again

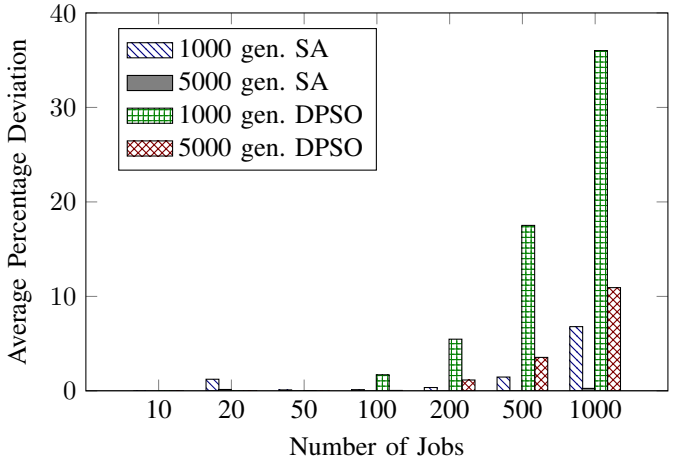

Fig. 15. Comparative average percentage deviation of our four parallel algorithms relative to the best known solutions of [8] for the UCDDCP problem.

TABLE IV. AVERAGE PERCENTAGE DEVIATION OF OUR APPROACHES FOR EACH PROBLEM SIZE FOR UCDDCP, RELATIVE TO AWASTHI *et al.* [8].

Jobs	SA_{1000}	SA_{5000}	$DPSO_{1000}$	$DPSO_{5000}$
10	0	0	0	0
20	1.233	0.151	-0.094	-0.083
50	0.105	-0.142	0.005	-0.382
100	0.131	-0.191	1.705	0.048
200	0.356	-0.136	5.472	1.153
500	1.465	-0.777	17.514	3.544
1000	6.801	0.265	36.015	10.928

observe that DPSO computes worse results from job size 100 and above, compared to SA. Table IV show that both versions of DPSO obtain better results than SA for input sizes of 20 and 50 jobs. For 1000 jobs the deviation for $DPSO_{1000}$ is 36 percent, while that of SA_{1000} is less than 7 percent. The $DPSO_{5000}$ performs better than $DPSO_{1000}$, but for 500 and 1000 jobs its deviation relative to SA is quite high. Simulated Annealing again achieves better results for higher instances of 100 jobs and above. Figure 15 shows that SA with 5000 generations perform much better than with 1000 generations with 25 times better solution value for 1000 jobs. Table IV shows that SA with 5000 generations obtains better results than the best known solution, for all the problem instances, except for 1000 jobs, where $\%\Delta$ is still just 0.26. Figure 16 shows the average runtime of the presented algorithms, each for 1000 and 5000 generations and their comparison with [8]. SA version with 1000 generations requires only 0.67 seconds for 50 jobs which is about 3.7 times faster than the CPU

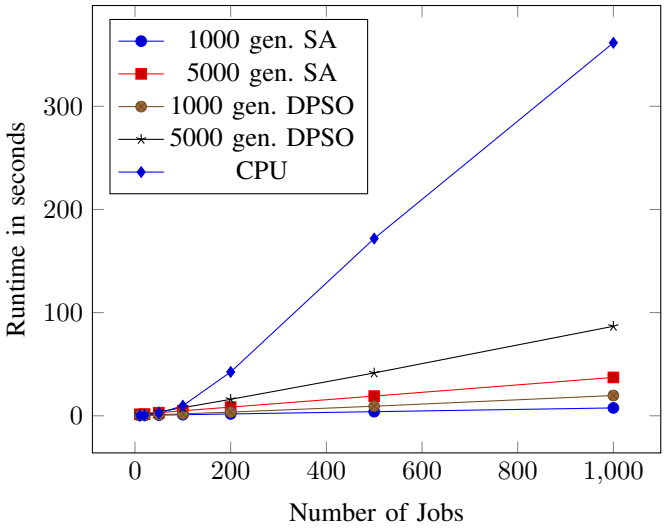

Fig. 16. Plot of the runtimes of the four parallel approaches and the CPU implementation of Lässig *et al.* [7] for the UCDDCP problem.

TABLE V. OBTAINED SPEED-UPS OF THE PARALLEL ALGORITHMS FOR THE UCDDCP PROBLEM RELATIVE TO AWASTHI *et al.* [8].

Jobs	SA_{1000}	SA_{5000}	$DPSO_{1000}$	$DPSO_{5000}$
10	0.459	0.119	0.436	0.189
20	1.225	0.289	1.043	0.327
50	3.701	0.841	2.480	0.642
100	9.226	2.012	5.229	1.247
200	23.600	5.039	11.866	2.662
500	43.060	8.981	18.494	4.138
1000	47.383	9.721	18.38	4.167

version. For 100 jobs the speed-up is about 9 times and for an input size of 1000 jobs the speed-up is about 47.4 times faster with an average deviation of 6.8 percent. With the 5000 generation version of SA, the quality of the results is much better, but the time needed is a bit more, however still much faster than the CPU. The speed-ups for this case are obtained only for input sizes 100 and more, as is clear from Table V. The values of speed-ups are about 2, 11, 18 and 19 percent for 100 jobs and above, respectively. Our experimental results

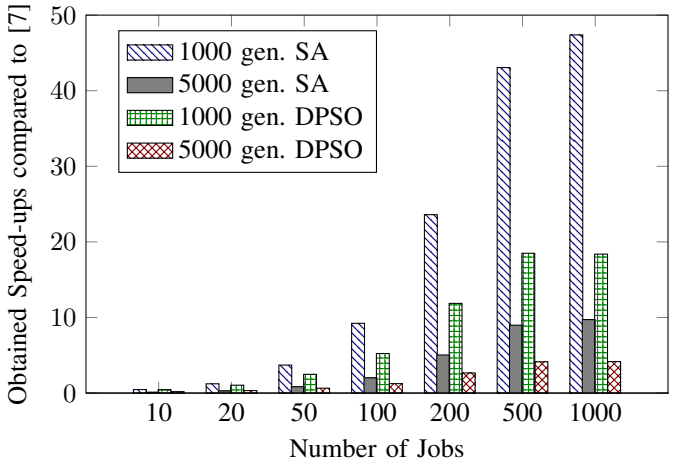

Fig. 17. Graphical representation of the obtained speed-ups of the parallel algorithms for the UCDDCP problem relative to [8].

certainly conclude that GPU parallelization is very powerful and efficient. Another observation which cannot be overlooked is that GPU technology has proven to be worthwhile only for large instances as shown in [4] and [1]. However, we show that with an efficient strategy for data transfer and algorithm parameters, high level of speed-ups are also possible for small instances. Concluding, our results and comparisons show that the best solutions can be achieved with SA and 5000 generations. To improve the values for 20 and 50 jobs, DPSO can be used but for larger problem instances DPSO does not work well with the given parameters. Reason for this could be that SA is an *intensification* oriented metaheuristic which searches intensively on a promising part of the domain, where as the DPSO is a diversification oriented metaheuristic which works more scattered [19].

IX. Conclusion

This work presents an efficient parallelization of the Simulated Annealing algorithm for the Common Due Date (CDD) problem and the Unrestricted CDD with Controllable Processing Times. We utilize the 2-layered approach to break up the NP-hard problem in two components to parallelize the metaheuristic algorithms. Henceforth, two strategies for parallelizing the SA algorithm are explained, based on Ferreiro *et al.* [12]. Later on in the paper, we focus on exhaustively explaining our parallel SA algorithm and its exact implementation. The NP-hard problems which are covered in this work are the CDD and the UCDDCP. We effectively use the polynomial algorithms provided in recent works of Lässig *et al.* [7] and Awasthi *et al.* [8], to optimize the given sequences for both these problems and to develop the parallel metaheuristic algorithms. Section VI describes how the SA metaheuristic algorithm is mapped on to the CUDA programming model. Finally, we present the evaluations of our parallel strategies for the two NP-hard scheduling problems. The algorithms are implemented over the benchmark instances provided in the OR-library [17] and by Awasthi *et al.* [8]. The efficiency of our parallel Simulated Annealing algorithm is proven by the comparison of our results with the previous CPU implementations, as well as the parallel DPSO algorithm on the same GPU architecture.

Not only do we obtain high speed-ups, our parallel algorithms also provide improvements to the best known solution values for several benchmark instances. It is evident from our implementation that parallel DPSO does not perform as well as the parallel SA, at least for the studied problems. The speed-ups obtained with SA are massive compared to the very recent work of [7] and [8]. The speed-up values obtained are of the order of 100 and 50, even for a relatively small problem instance of 1000 jobs. DPSO is not just slow compared to the SA but it is also not able to find solutions of high quality, compared to the parallel SA. Future works in this area should also examine the utilization of the texture memory of the GPU to make use of its spatial cache. Another observation is that the development of GPU based algorithms is simplified fundamentally with CUDA in comparison to solutions in the past where data had to be mapped as images to be processed by a GPU. In conclusion, it can be said with high confidence that it is worth to invest more effort in investigating the GPU parallelization for NP-hard combinatorial optimization problems.

Acknowledgement

The research project was promoted and funded by the European Union and the Free State of Saxony, Germany. Dr. Weise is supported by the Fundamental Research Funds for the Central Universities.

References

[1] I. Chakroun, N. Melab, M. Mezmaz, and D. Tuyttens, "Combining multi-core and GPU computing for solving combinatorial optimization problems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1563–1577, 2013.

[2] D. G. Spampinato and A. C. Elster, "Linear optimization on modern GPUs," in *IEEE International Symposium on Parallel & Distributed Processing, (IPDPS)*, 2009, pp. 1–8.

[3] T. V. Luong, N. Melab, and E. G. Talbi, "GPU computing for parallel local search metaheuristic algorithms," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 173–185, 2013.

[4] ——, "Large neighborhood local search optimization on graphics processing units," in *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 2010, pp. 1–8.

[5] M. Czapiński and S. Barnes, "Tabu search with two approaches to parallel flowshop evaluation on CUDA platform," *Journal of Parallel and Distributed Computing*, vol. 71, no. 6, pp. 802–811, 2011.

[6] S. Tsutsui and N. Fujimoto, "Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study," in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*. ACM, 2009, pp. 2523–2530.

[7] J. Lässig, A. Awasthi, and O. Kramer, "Common due-date problem: Linear algorithm for a given job sequence," in *17th IEEE International Conferences on Computational Science and Engineering (CSE)*, 2014, pp. 97–104.

[8] A. Awasthi, J. Lässig, and O. Kramer, "Un-restricted common due-date problem with controllable processing times: Linear algorithm for a given job sequence," in *17th International Conference on Enterprise Information Systems (ICEIS)*, 2015, pp. 526–534.

[9] T. C. E. Cheng and H. G. Kahlbacher, "A proof for the longest-job-first policy in one-machine scheduling," *Naval Research Logistics (NRL)*, vol. 38, no. 5, pp. 715–720, 1991.

[10] N. G. Hall, W. Kubiak, and S. P. Sethi, "Earliness–tardiness scheduling problems, ii: deviation of completion times about a restrictive common due date," *Operations Research*, vol. 39, no. 5, pp. 847–856, 1991.

[11] T. C. E. Cheng, "Optimal due-date assignment and sequencing in a single machine shop," *Applied Mathematics Letters*, vol. 2, no. 1, pp. 21–24, 1989.

[12] A. M. Ferreiro, J. A. García, J. G. López-Salas, and C. Vázquez, "An efficient implementation of parallel simulated annealing algorithm in GPUs," *Journal of Global Optimization*, vol. 57, no. 3, pp. 863–890, 2013.

[13] P. Salamon, P. Sibani, and R. Frost, *Facts, Conjectures, and Improvements for Simulated Annealing*. Society for Industrial and Applied Mathematics, 2002.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[15] Q. K. Pan, M. F. Tasgetiren, and Y. C. Liang, "A discrete particle swarm optimization algorithm for the no-wait flowshop scheduling problem," *Computers & Operations Research*, vol. 35, no. 9, pp. 2807 – 2839, 2008, part Special Issue: Bio-inspired Methods in Combinatorial Optimization.

[16] C. Blum, R. Chiong, M. Clerc, K. D. Jong, Z. Michalewicz, F. Neri, and T. Weise, *Variants of Evolutionary Algorithms for Real-World Applications*, 1st ed. Springer-Verlag Berlin Heidelberg, 2012, ch. Evolutionary Optimization, pp. 1–29.

[17] J. E. Beasley, "OR-library: Distributing test problems by electronic mail," *Journal of the Operational Research Society*, vol. 41, no. 11, pp. 1069–1072, 1990.

[18] M. Feldmann and D. Biskup, “Single-machine scheduling for minimizing earliness and tardiness penalties by meta-heuristic approaches,” *Computers & Industrial Engineering*, vol. 44, no. 2, pp. 307–323, 2003.

[19] C. Blum and A. Roli, “Metaheuristics in combinatorial optimization: Overview and conceptual comparison,” *ACM Comput. Surv.*, vol. 35, no. 3, pp. 268–308, 2003.

[20] A. Awashi, J. Lässig, J. Leuschner, and T. Weise, “GPGPU-based Parallel Algorithms for Scheduling Against Due Date,” Proceedings of 6th IEEE Workshop on Parallel Computing and Optimization (PCO’16), May 23-27, 2016, Chicago, IL, USA, pages 766–775, Chicago, IL, USA: IEEE Computer Society, ISBN: 978-1-5090-3682-0. doi:10.1109/IPDPSW.2016.66


```
@inproceedings{ALLW2016GBPAFSADD,
author    = {Abhishek Awashi, Jrg Lssig, Jens Leuschner, and Thomas Weise},
title     = {GPGPU-based Parallel Algorithms for Scheduling Against Due Date},
booktitle = {Proceedings of 6th IEEE Workshop on Parallel Computing and Optimization (PCO'16)},
publisher = {IEEE Computer Society},
address   = {Piscataway, NJ, USA},
location  = {Chicago, IL, USA},
year      = {2016},
month     = may # {~23-27, },
pages     = {766--775},
isbn      = {978-1-5090-3682-0},
},
```