Resolving Classical Concurrency Problems Using Adaptive Conflictless Scheduling

Mateusz Smolinski
Lodz University of Technology
Institute of Information Technology
Lodz, Poland
Email: mateusz.smolinski@p.lodz.pl

***Abstract*—Classical concurrency problems define environments for task processing with high contention of shared resources. The adaptive conflictless scheduling is an alternative to existing synchronization mechanisms used in known solutions of concurrency problem to provide parallel tasks processing without resource conflicts. Used in this paper scheduling concept eliminates deadlock between tasks that are caused by the wrong allocation of shared resources, regardless of the specification of task environment. The adaptive conflictless scheduling, in opposition to other solutions dedicated to specific problem, is a universal approach and can be used to solve different concurrency problems. This article presents versatility of applications of adaptive conflictless scheduling by its usage in two classical concurrency problems: readers and writers and dining philosophers. Analysis of scheduling rules, when conflictless scheduling is applied, shows lack of task starvation in both classical concurrency problems. If conflictless scheduling is used, there is no need to use any other synchronization mechanisms for control tasks access to groups of shared resources. In presented approach only the programmer has to separate tasks and for each of them has to establish and report all required shared resources. Then tasks synchronization in accessing resource groups will be performed according to the adaptive conflictless schedule, which is the sequence of conflictless schedules. The concept of conflictless scheduling bases on efficient resource conflict detection between tasks by its resource identifiers and delayed execution of conflicted tasks using many FIFO queues assigned individually to each group of shared resources. Task queues are emptied based on the conflictless schedule, which is calculated every time according to specific state of the task processing environment.**

***Index Terms*—Resource conflict as outlier, mutual exclusion, deadlock avoidance, cooperative concurrency control, conflictless scheduling.**

I. Introduction

In modern computer architectures the number of processing units increases. Efficient usage of these processing units requires to maximize the number of tasks that are performed in parallel. The efficient utilization of processing units concerns various software environments like virtualization infrastructures, operating systems, OLTP (OnLine Transactions Processing) and OLAP (OnLine Analytical Processing) systems, multithreading applications, DBMS (DataBase Management System), HPC (High Performance Computing) environments and many others. According to the characteristic of task processing environment software developer can indicate many tasks as processes, threads, tasklets, transactions or as a sequences of operations e.t.c. If indicated tasks are independent, then their executions can be performed in parallel without synchronization. If tasks use shared resources, additional measures are required to synchronize their executions and then creation of correct program becomes a complex issue. It is not enough to define each task boundaries, programmer has to take into account the complex dependencies between tasks and includes them in execution paths.

978-1-5090-5795-5/17/$31.00

One of popular relationships between tasks indicated in software development is the usage of shared resource during their parallel processing. If two parallel executed tasks require to use the same instance of shared resource and at least one of them perform write operation on this resource then conflict occurs [1], [2]. Resource conflicts are as oultier in task processing environment, because they change task execution order from parallel to sequential [3]. Many occurrences of resource conflict can cause the tasks deadlock, which is the worst situation in task processing because all tasks participating in deadlock have no progress in execution. Concurrency programming offers many techniques and mechanisms to synchronize task accesses to shared resources to eliminate conflict occurrences. The programmer can use coordinated or competitive concurrency to synchronize executions of many tasks that all required resources are used exclusively. In all cases programmer is responsible for the selection of appropriate methods of tasks synchronization and their correct usage for the encountered concurrency problem.

As alternative the programmer can apply the new scheduling concept, which bases on mutual exclusion in temporary allocation each of global resource group to one of tasks to eliminate resource conflicts occurrences. The execution plan that guarantees no resource conflicts can be established using conflictless scheduling, that is alternative to other concurrency solutions that use 2PL (Two Phase Locking) [4], [5]. An adaptive conflictless schedule can be used in environments where tasks execution time is not known in advance and the number of shared resources is limited. To prepare adaptive conflictless schedule a dedicated model of task resources representation and additional data structures like task classes or conflict array are required [6], [7]. This approach was extended by Duraj, where in conflictless scheduling association rules were used [8]. Unlike other concurrency solutions, proposed adaptive

conflictless scheduling can be used in solving different concurrency problems. The conflictless scheduling can be used only in task processing environments that meets all assumptions presented in next chapter.

A. Environment Assumptions for Task Processing without Resource Conflicts

The conflictless task scheduling is dedicated to task processing environments with high contention and limited number of reusable, shared resources. In program code each task is defined by sequence of operation and input data set. In conflictless task scheduling this task definition is regularized as the shortest sequence of operations with a group of global resources required to perform those operations [7]. For example task can be defined as special code block in program, which sets the order of operations and all its required global resources that are used by other tasks. Software developer only needs to select each task boundaries and all its required reusable shared resources. The conflictless schedule fixes execution order for conflicted tasks and in this way assures mutual exclusion between tasks in access to reusable shared resources. Each time when task is ready to execution, it has to be requested and in this moment, all shared resources required by task have to be known and this set of resources have to be constant. Also each finish of task execution has to be reported, even if it is terminated as a result of an processing error. It is assumed that the execution of each requested task can be delayed, each conflicted task is suspended in one of FIFO queues. However in conflictless scheduling for each suspended task is guaranteed that its execution starts in finite time. Another important assumption is lack of task priorities. The task execution time does not have to be known in advanced to use coordination by conflictless scheduling.

B. Conflictless Task Scheduling

In conflictless scheduling each task has dedicated representation of all its global resources. Used task resources representation provides rapid conflict detection between tasks. For each task the global resources representation includes two binary resources identifiers IRW and IR. Each bit in those binary identifiers represents other global resource. Binary identifier IRW_i represents all global resources used by task t_i that are read (not changes resources state) or written (changes resources state) and IR_i represents all global resources used by task t_i that are only read.

There is no resource conflict between tasks t_i and t_j if $IRW_i and IRW_j = 0$. And at least one resource conflict between tasks t_i and t_j occurs, onlyif$(IRW_i and IRW_j) xor (IR_i and IR_j) \neq 0$, where: IRW_i, IR_i binary resources identifiers of task t_i, IRW_j, IR_j binary resources identifiers of task t_j.

Also additional structures like task classes and conflict array are required to effective prepare conflictless schedule according to situation like finish execution one of active tasks or a request a new task for execution. The task class groups requested tasks that have the same value of binary resource identifiers IRW and IR. Therefore tasks belonging to the same class has identical set of required global resources and use them in the same read-only or only-write or read-write operations. Each requested task has to be assigned to one of task class, when task class is not exist the new one is created. The conflict array M^n stores all detected resource conflicts between existed task classes and is a representation of Wait For Graph for task classes. For each new requested task, which has a class assigned, verification of resource conflict existence with any active task is possible. Therefore for any active task its class is denoted as active and set of all active classes is maintained in conflict array. If new requested task has no resource conflict with any other active task, than its execution is started immediately. Otherwise the task is located in class FIFO (First In First Out) queue, to which it belongs, and waiting there for until its execution starts [7].

The creation of conflictless schedule requires to prepare an adaptive conflictless schedule for each situation, when one of active task finish its executions. Each conflictless schedule determines task classes that FIFO queues can be emptying. According to class definition and values of its binary resource identifiers some of them release only the oldest waiting task and other class can release all waiting tasks from queues [7]. The adaptive conflictless schedule S^n is prepared to specific environment state and is worthless in other states of task processing environment. In $n-th$ point in time environment state includes set of active task R^n that are executed and all tasks waiting for execution in C_k^n class FIFO queue where $k = \{1, ..., N^n\}$, N^n represents number of task classes in $n-th$ point in time. Each finish of execution for one of active tasks determines next points in logical time. Next $(n+1)-th$ point in time is never known in advance, because task execution time cannot be determined earlier.

The number of prepared adaptive conflictless schedules for $n+1$ point in time is determined by number of active task set (R^n). When one of active tasks finish execution, then only one of prepared adaptive conflictless schedules will be used. This is due to one of environment assumptions, the task execution time is not known in advance. Also any new requested task can change previously prepared adaptive conflictless schedules S_k^n, $k = \{1, ..., N^n\}$. It is important to note, that adaptive conflictless schedules have to be prepared often and efficiently (without any additional delay). For this purposes it is recommended to prepare conflictless schedules in isolated computing environment i.e. using GPGPU (General Purpose computing on Graphic Processing Unit). This is alternative usage of GPU, which is often used among others in HPC or for efficient graph processing, schedule preparation for transaction processing [9], [10], [11].

The algorithm for preparation conflictless schedule assures task fairness and liveness. Lack of task priorities in conflictless scheduling cause that requested tasks are treated identically. To prevent occurrence of the task starvation problem in conflictless scheduling each task besides of two binary resource identifiers IRW and IR has assigned a timestamp of logical time. The conflictless schedule preparation includes checking

of logical time to determine the execution order for waiting conflicted tasks. According to this rule the oldest conflicted task will be executed first. Applied rule prevents the situation, that in next prepared adaptive conflictless schedules tasks from fixed set of task classes begin execution, but other task classes queues are never emptying. In some situations, when one of active tasks finish its executions, many alternative conflictless schedule can exist. Then the arbitration rule choose one of these alternative schedules. The arbitration rule prevents task starvation, because it choose to execution a subset of oldest waiting tasks that have no resource conflict with each other. This assures that in conflictless schedule preparation the oldest waiting task will be preferred than many other task, that have conflict with the oldest task. This eliminate frequently repeated tasks set in conflictless schedule and assures that each waiting task will be executed in finite time. Taking into schedule all requested tasks causes many exceptional situations.

Next chapters present how to use adaptive conflicteless scheduling to solve classical concurrency problems like readers and writers and dining philosophers. The main advantage of this approach is that using the same data structures and algorithm different concurrency problems can be solved. The programmer using conflictless scheduling does not have to choose dedicated synchronization mechanisms or algorithm for allocation shared resources to executed task. There is no need to use any other task synchronization mechanism, conflictless schedule guarantees that task execution is realized without interruption due to resource access. In each considered further concurrency problem that were solved using conflictless scheduling will be presented:

- tasks boundaries as the shortest sequence of operations in which global resources are required,
- resource groups and assigned task classes,
- statically assigned binary resource identifiers for tasks.

For identified tasks classes the conflict array is presented with preparation of conflictless schedule. Flexibility of adaptive conflictless scheduling bases on adjusting the task execution plan to actual state of task processing environment, it is realized by preparation conflictless schedules.

II. Solution of readers and writers problem using conflictless scheduling

The classical readers and writers concurrency problem was formulated by Courtois et al. [12]. In main problem definition for one global reusable resource there are two type of tasks readers and writers. Each task belonging to readers group performs only read operation on instance of global reusable resource. Tasks from writers group perform write operation, which change state of global resource. Parallel executions of writer task with any other task is not possible due to resource conflict, but parallel executions of readers tasks is of course possible.

Tasks definition in readers and writers concurrency problem show that in adaptive conflictless schedule there will be distinguished two task classes: readers task class C_r and writers task class C_w. For determined task classes resource conflict

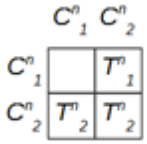

Fig. 1. Conflict array for readers and writers concurrency problem.

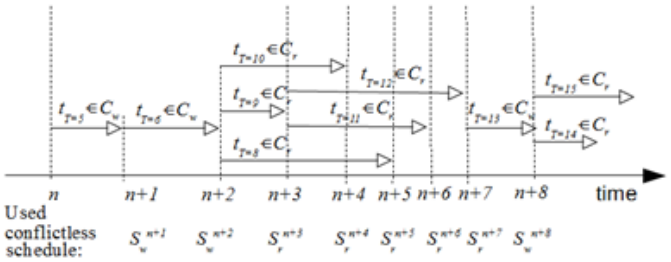

Fig. 2. The sample adaptive conflictless schedule for readers and writers tasks.

are detected and stored in conflict array. Using conflictless scheduling each new requested reader or writer task has to be verified, that its execution is possible without resource conflicts. At first according to values of binary resource identifiers of new requested task its class is determined. Then basing on conflict array shown on the Fig. 1 and collection of active task classes is determined whether new requested task can be executed without conflicts. Only one from two classes can be active and active class has always at least one executed task. If new requested task belongs to one of active class or has conflict with one of active task, its execution does not begin and task is suspended in class FIFO queue.

Control which of two task class queues can be emptying bases on the prepared conflictless schedule. For readers task class C_r binary resource identifiers $IRW_r = IR_r$, so executions of all waiting in this class queue readers tasks will be resumed at once. For writers task class C_w, where resource binary identifiers $IRW_w \neq IR_w$, only the longest waiting task can leave class queue and begins the execution. The longest waiting task can be determined in simple way, because class queue is managed by FIFO algorithm.

In classical readers and writers problem preparation of conflictless schedule for $n + 1$ point in time can be done in two situations, when active task set R^n includes many readers tasks or only one writer task. This determines the column of conflict array which will be used to prepare adaptive conflictless schedule and bases on detected resource conflicts between classes and analysis of logical time values determined for the oldest waiting tasks in each class queues that has conflict with finished active task.

Let's assume, that two class queues are not empty and readers tasks are executing, so readers class is temporarily mark as active. According to the algorithm of adaptive conflictless schedule preparation for each active class the oldest task has to be determined for any class that has conflict with that active

class or for that active class itself. If the oldest waiting task belongs to writers class C_w, then after finished execution of last reader task a execution of writer task will be started. Otherwise, if the oldest task is from readers class C_r, more readers tasks will be executed in parallel.

Now, let's consider the case, that classes C_r and C_w FIFO queues are not empty and single writer task is executing. In this situation the writers task class is active, so preparation of conflictless schedules has to compare values of logical time of oldest tasks from classes C_r and C_w, in $n-th$ point in time denoted accordingly as T_r^n and T_w^n. If $T_w^n > T_r^n$ then after finished writer task another writer task begin execution, otherwise all waiting readers tasks start execution after writer task execution will be finished.

It should be noted that all finished task executions should be notified, even this is the result of the task processing error. Adaptive conflictless schedule is constructed as a sequence of selected conflictless schedules, each of them is chosen according to finished execution one of active tasks. Every finish of active task fixes time points, in which according to prepared conflictless schedule tasks executions are started.

The Fig. 2 presents sample adaptive conflictless schedule determined for readers and writers concurrency problem. Presented tasks belongs to readers class C_r or writers class C_w and task subscript index shows its logical time value. It should be noted, that some of presented in Fig. 2 conflictless schedules are empty collections i.e. S_r^{n+4}, S_r^{n+5}, S_r^{n+6}. This is a result of logical time verification for waiting tasks. It is also shown, that many tasks from readers class C_r can be executed in parallel, but when writer task is executed, all other tasks have to wait.

Conflictless scheduling guarantees that there is no resource conflict between parallel executed tasks. Each active task can execute without any additional delays associated with obtaining access to the required resources. The suspension of conflicted task causes additional delay, but this is the result of elimination of resource conflicts between readers and writers.

III. Solution of dining philosophers problem using conflictless scheduling

The classical dining philosophers problem was originally formulated by Dijkstra as five computers competing to five tape devices. Dining philosophers problem was reformulated by Hoare at al. in present form [12], [13]. The classical philosopher problem is defined for five philosophers which sit at a round table with five forks. Each philosopher has to change its state alternately form eating to resting, which illustrates philosopher processing progress. Forks are localized on table between philosophers and before eating each philosopher requires the two nearest forks. When eating phase is finished two forks are returned on table and then philosopher changes his state to resting.

To use conflictless scheduling task has to be defined as philosopher eating phase and forks are its global resources. Each fork is shared by two neighboring philosophers. Single philosopher will request tasks sequentially, each one task

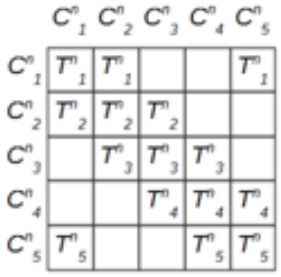

Fig. 3. The conflict array for five dining philosophers concurrency problem

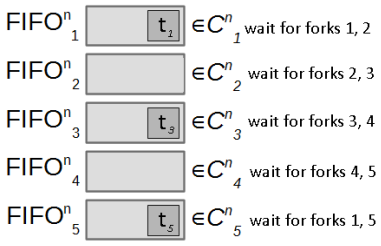

Fig. 4. FIFO queues of task classes for five dining philosopers problem

requested by the same philosopher require identical set of resources (two nearest forks for philosopher). The next task will be requested by philosopher only after finished resting phase. Number of resource groups is identical with number of philosophers. For example for five dining philosopher problem specification, there are five groups of resources, which are exclusive accessed by tasks (defined as eating state for selected philosopher). Therefore in solution of five dining philosopher using conflictless scheduling only five resource class are featured. The Fig. 4 presents task classes with its queues for five dining philosophers problem. Minimal length of binary resources identifiers is determined by number of used global reusable resources, in presented dining philosopher problem there are five forks. Each task class C_k has assigned next two forks, which are identified in IRW_k. Example of binary resource identifiers values for five task classes C_k, $k = 1..5 : IRW1 = (11000)b, IRW2 = (01100)b, IRW3 = (00110)b, IRW4 = (00011)b, IRW5 = (10001)b$. The second binary resources identifier IR_k for C_k, $k = 1..5$, therefore resource conflict detection between two task classes bases only on conjunctions of two IRW resource identifiers [7]. Using resource conflict detection the conflict array is determined for task classes $C_k, k = 1..5$, which will be used in preparation of conflictless schedules. In dining philosopher problem state of conflict array determined for five task classes is static in time. All requested tasks belong to one of these five classes.

In each class queue there will be at most one task, because in classical problem each philosopher eating phase requires finished resting and resting phase requires finished eating. In $n-th$ point in time the state of task class FIFO queue represents C_k^n. For each not empty FIFO queue of task class C_k^n the longest waiting task is determined, its time is marked as T_k^n. Those logical time values are required in preparation

of adaptive conflictless schedule to prevent task starvation.

The determined conflict array in $n-th$ point in time for five dining philosophers is shown on the Fig. 3. Empty field in conflict array represents the fact, that no resource conflict exist between two classes selected by row and column. The specification of the dining philosopher problem assures that many philosophers can eat simultaneously, but never two directly sitting at table neighbor philosophers can eat at the same time. This means that in conflictless scheduling many tasks can be executed in parallel, so many active class can exist. Active task execution time is not known, therefore for each active task class an adaptive conflictless schedule need to be prepared.

A preparation of conflictless schedule requires to determine for each active class C_k^n the class C_k^{*n} which FIFO queue includes oldest waiting conflicted task. Determination of the class C_k^{*n} bases on finding the lowest value of logic time T_k^n in column of conflict array that is assigned to the active class C_k^n. If class C_k^{*n} is determined for any active class $C_k^n, k = 1..count(R^n)$, then any adaptive conflictless schedule S_k^{n+1} does not include any task, that class has resource conflict with any one of determined classes C_k^{*n}. This rule prevents task starvation in waiting in class FIFO queue. Therefore conflictless scheduling assures that execution of each requested task will be started in finite time, this rule provides liveness for philosophers. The task deadlock in solution of dining philosopher is eliminated by execution schedule that assures elimination of resource conflict occurrences. Therefore the situation, when five parallel tasks reserve only one fork, never occurs.

The Fig. 5 presents sample conflictless schedule for dining philosophers problem. On figure each task is represented as eating phase of selected philosopher, the Fig. 5 does not presents resting phase for dining philosophers. The subscript index for task determines its logical time value and its class C_k subscript represents philosopher identifier $k = 1..5$. According to dining philosophers problem specification the prepared schedule provides parallel task processing without any resource conflict. Logical time values determine order of task execution, therefore task starvation problem never occurs. Some of conflictless schedules presented on the Fig. 4 is empty i.e. S_3^{n+3}, S_2^{n+6}. The schedule S_1^{n+4} includes only task from class C_2^{n+4} because waiting task execution from class C_4^{n+4} is blocked by running conflicted task from active class C_5^{n+4}. Therefore execution of task from class C_4^{n+4} is running just in $n+5$ point in time after finished execution of task from class C_5^{n+4}.

Another important aspect in conflictless scheduling presented on the Fig. 5 is the number of conflictless schedules that have to be prepared for each point in time. For $n+1$ point in time two schedules S_2^{n+1} and S_4^{n+1} should be prepared, but only one of them will be chosen to execution. For $n+2$ point in time also two conflictless schedules need to be prepared S_2^{n+2}, S_5^{n+1} and for $n+3$ point in time three conflictless schedules are required S_1^{n+3}, S_3^{n+3}, S_5^{n+3} e.t.c., but from prepared conflictless schedules only one is used. It also should

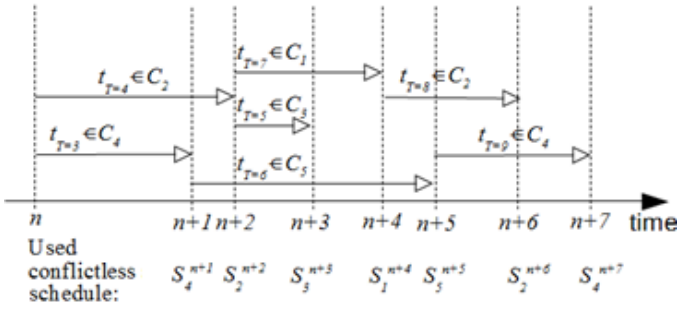

Fig. 5. The sample adaptive conflictless schedule for tasks that represents eating phase of five dining philosophers.

be noted, that running task sequence is not always consistent with order of its logical time values. In $n+1$ point in time execution of task from class C_5^{n+1} is started, that has logical time value lower than task from class S_3^{n+2} which execution is started later. This means that in conflictless scheduling longer waiting task can be executed later that task from other class.

IV. Conclusions

The conflictless schedule provides alternative coordination method of tasks in high contention environment with limited number of shared resources. Presented approach has elastic task definition, therefore can be used in various task processing environments to prevent resource conflict occurrence. Those resource conflict generates outlier in task processing environments, because conflicted tasks can not be executed in parallel. This novel approach of adaptive conflictless scheduling bases on resource group allocation to tasks, management of task classes and its representation that enables rapid resource conflict detection. Task that execution could not start are waiting in FIFO class queue, each class represents group of global resources and way how task access them with distinction of read and write operations. Prepared conflictless schedule determines group of suspended tasks to execute them in parallel and in each case is dedicated to situation, when specific active task was finished.

Universality of conflictless scheduling base on that it can be used to solve various concurrency problems. This paper presents how conflictless scheduling was used to prepare solution in two classical concurrency problems: readers and writers and dining philosophers. In both presented concurrency problems the number of conflictless schedules to prepare is determined by number of active classes, which varies in time. The value of logical time fixed for each coordinated task was used to eliminate occurrence of waiting task starvation in class FIFO queue. In some situations the task with lower value of its logical time was included in adaptive conflictless schedule than the older waiting task, this is the result of elimination of resource conflicts and task starvation. All calculations of conflictless schedule for fixed moment in time can be realized in massively parallel computing environment provided by modern GPU.

As other known solutions of readers and writers concurrency problem the adaptive conflictless scheduling executes reader tasks in parallel and each writer task was executed alone. In this concurrency problem there will be only two task classes, each one has own FIFO queue with suspended tasks respectively for readers and writers tasks. Usage of adaptive conflictless scheduling to solve dining philosophers problem will provide many taks classes, the number of task classes is determined by number of philosopohers. Also number of managed by adaptive conflictless schedule FIFO queues is determined by the number of philosophers. It is worth to mention that adaptive conflictless schedule ensures processing progress for each of five philosophers and eliminate deadlock occurrence.

Usage of conflictless scheduling simplify programmer work, because software developer only need to determine group of resources and select boundaries for each task with its binary resource identifiers. Access to each determined group of resources is controlled by adaptive conflictless schedule, which is created as a sequence of prepared conflictless schedules. Software developer that use conflictless task scheduling is not obliged to create valid solution of concurrency problem or even use of other synchronization mechanisms. Adaptive conflictless scheduling simplifies the program preparation because the developer does not have to analyze the task and resources dependency to prevent resource conflicts and task deadlocks and does not need to take care of shared resources allocation to tasks. Using presented scheduling concept provides effective task processing without resource conflicts and resulted by them deadlocks. Conflictless scheduling assures liveness and fairness for controlled tasks. The usage of adaptive conflictless scheduling eliminates task starvation, so execution of each one coordinated task always will be started in finite time.

REFERENCES

[1] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Prentice Hall Press, 2014.

[2] W. Stallings, *Operating systems, Internals and Design Principles*. Pearson Education, 2015.

[3] C. C. Aggarwal, *Outlier Analysis*. Springer Science and Business Media, 2013.

[4] K. Pun and G. G. Belford, "Perfornance study of two phase locking in single-site database systems," *IEEE transactions on software engineering*, no. 12, pp. 1311–1328, 1987.

[5] P. A. Bernstein and E. Newcomer, *Principles of transaction processing*. Morgan Kaufmann, 2009.

[6] M. Smolinski, "Coordination of parallel tasks in access to resource groups by adaptive conflictless scheduling," in *International Conference: Beyond Databases, Architectures and Structures*. Springer, 2015, pp. 272–282.

[7] M. Smoliński, "Conflictless task scheduling concept," in *Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology–ISAT 2015–Part I*. Springer, 2016, pp. 205–214.

[8] A. Duraj, "Conflictless task scheduling using association rules," in *International Conference: Beyond Databases, Architectures and Structures*. Springer, 2015, pp. 283–292.

[9] P. Harish and P. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *International Conference on High-Performance Computing*. Springer, 2007, pp. 197–208.

[10] P. Bakkum and K. Skadron, "Accelerating sql database operations on a gpu with cuda," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010, pp. 94–103.

[11] B. He and J. X. Yu, "High-throughput transaction executions on graphics processors," *Proceedings of the VLDB Endowment*, vol. 4, no. 5, pp. 314–325, 2011.

[12] P.-J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with readers and writers," *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, 1971.

[13] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *Acta informatica*, vol. 1, no. 2, pp. 115–138, 1971.