**A Communicative Deadlock Detection Algorithm**

**Tak Wong**

**9/27/2010**

**University of Massachuesetts Lowell**

*Conventional Deadlock detection algorithms rely on the discovery of loops and knots among processes accessing shared resources. These algorithms often provides information only on “potential” deadlocks in a snapshot of the running processes. There is no certainty from running the detection algorithms that deadlocks will occur and when. Moreover, separate procedures are required for avoiding deadlocks. Therefore, the management of deadlocks is often complicated and expensive, with not very effective results. We present a communicative algorithm which does not depend on construction of wait-for graphs. The algorithm requires each process that would require shared resources to communicate repeatedly with a process scheduler which deterministically detects deadlocks and avoids them by ordering the process schedule optimally.*

**Part I: Optimal Dependency Ordering**

We introduce an algorithm to order the parallel tasks according to processing dependencies, so that tasks that do not have dependencies will be run and completed first, and their dependent tasks can be run after that. By a dependent task, we mean its order of execution - that the task cannot be run until some other task which it depends on has completed first. The motivation for the ODO algorithm is that many parallel tasks often have timing and state dependencies on other related tasks. If the system has fewer processors than the number of tasks, it pays to queue and schedule dependent tasks after the independent tasks have completed and freed up resources. In addition, if a group of tasks has circular dependencies – ie every task in the group depends on at least one other task in the group – then deadlock occurs, and these mutually dependent tasks cannot be activated. The ODO algorithm not only generates a dependency ordering for the tasks, but also detects circular dependencies in a poorly designed parallel algorithm.

First, we need to state some conditions up front:

- We assume the separate tasks together implement a parallel algorithm. As such, there is a priori information on the dependencies between tasks. In other words, given a set of tasks for a parallel algorithm, we know for each task in the set if it depends on another task in the same set.

- The parallel algorithm will work only if there are no deadlocks in the dependencies. For example, if task A depends on task B, and B depends on C, and C depends on A, then it is not possible to run the algorithm to completion, since the tasks A, B, C will be in a deadlock which prevents any of the three to execute first. However, if a deadlock does exist, the ODO algorithm will be able to identify it and aborts the algorithm.

- The ODO algorithm orders the tasks entirely based on the condition of dependencies. The size of the individual tasks does not contribute to the ordering.

- We assume there are a finite number of processors available to run the separate tasks, each processor executing a single task at time. If the number of tasks exceeds the number of processors, we execute the tasks in this manner:

Given ordered sets of tasks S0, S1, S2, …, Sn, where S0 contains independent tasks, and Sy contains tasks dependent only on S(y-1), where y = 1, 2,…n, initially we execute as many tasks in S0, as the number of processors allow, and then the tasks in S1, S2, …, Sn, in this strict order. This is to guarantee that no dependent task is executed until the task(s) it depends on have completed.

These are the steps of the algorithm:

 Input: A finite set T of N individual tasks : T= { t1, t2, t3, …, tN }

 Output: A group of ordered sets of tasks, S0, S1, …, Sn, based on processing dependencies

 Steps:

1) From T, take each task tx that is independent of any other task in T and independent of any task already in S0, move it into set S0. Let y = 1.

2) From T, take each task tx that is independent of any other tasks remaining in T and independent of any tasks already in Sy, move the task into Sy.

3) If at any stage in 2), T is non-null, but there are no independent tasks in T, then a deadlock is found. This algorithm cannot run to completion, because at least 2 tasks are inter-dependent, so no task can be executed first. Stop and output error.

4) When T is null, then stop and output the sets S0, S1, S2, … in this order. If T is non-null, increment y by 1, go to step 2).

Claim: The sets S0, S1, S2, … are a group of ordered sets of tasks, where the tasks in S0 are executed before the tasks in S1, and S1, before S2, and so forth.

Proof:

For any implementation of a parallel algorithm, there exists at least 1 task that is independent of all other tasks. Any of this type of tasks can be executed before other tasks. Step 1) makes sure all independent tasks belong to set S0. Since they are also independent of each other, all the tasks in S0 can be activated to run in parallel. Tasks in any other set must have dependencies on some other tasks. Take set Sy. Step 2) in the algorithm guarantees that each task tx in Sy is independent of other tasks in Sy, but it is dependent on some task in S(y–1), because, if tx is not dependent on S(y – 1), then tx would be included in S(y –1) by step 2). So, by our definition of dependency, it is clear that S0 must be processed before any other set, and S(y–1) must be processed before Sy, for y = 1, 2, 3…

Corollary:

The ordering of S0, S1, S2, …. represents the optimal ordering in the sense that the least number of sets are generated, which implies the largest number of tasks that can possibly be executed in parallel, This can be shown as follows:

Give a number of tasks that implement a parallel algorithm, suppose the ODO algorithm generates sets S0, S1, … Sn. Then there must be 1 task in Sn that is dependent on at least 1 task in S(n–1). By the same logic, this task in S(n–1) must depend on a task in S(n-2), and so forth. So in total, there is at least 1 task in Sn that depends on n-1 tasks. If there is another algorithm that can generate a group of ordered sets S’0, S’1, … S’m, where m < n, then there is no task in any of S’0, S’1, S’2,… that can be dependent on n – 1 tasks, since m < n, and any task in S’0, S’1, S’2,… can be dependent at most on m – 1 < n – 1 tasks. So this task cannot be found in any of S’0, S’1, S’2,…,S’m

Performance Analysis:

Given T a set of N separate tasks, each task t must be compared with every other task in order to determine if t is an independent task in step 1). The number of compares is N x (N – 1). Subsequently, in step 2), each task remaining in T needs to be compared with all other task to determine if it is independent of the remaining tasks in T. Step 2) is performance repeatedly until all tasks are tested or if a deadlock is found. In the worst case scenario, each round is in the order of O(N2), and there can be as many as N rounds. Therefore, the ODO algorithm overall is in the order of O(N3).

As a demonstration, we want to apply the ODO algorithm on the MBRP (Macro Block Region Partition) processing (Sun, Wang, and Chen, 2007). The idea of MBRP processing is to sub-divide an MPEG4 image into equal-size smaller blocks (Macro Blocks):

|  |  |  |
| --- | --- | --- |
| 1 | 3 | 6 |
| 2 | 5 | 8 |
| 4 | 7 | 9 |

Each of the Macro Blocks can be processed by a separate task with dependencies on neighboring blocks directly to the left, directly above, and diagonally to the above right. For example, Macro Block 5 in the above diagram has dependencies on Macro Blocks 2, 3, and 4. Essentially, the same task is run on different processors to work on the different Macro Blocks. If we apply the ODO algorithm, we will get the following ordered sets as the result:

 S0 = { t1 }

 S1 = { t2, t3 }

 S2 = { t4, t6 }

 S3 = { t5 }

 S4 = ( t7, t8 }

 S5 = { t9 }

Where tx is the task that processes Macro Block x.

We note that t5 stands alone in S3, that is because Macro Block 5 depends on Macro Block 4 diagonally. The MBRP algorithm processes the Macro Blocks from a left to right in a horizontal sweep. Therefore, t5 cannot execute until t4 is completed.

However, the Diagonal –MBRP algorithm transforms the horizontal bits in a Macro Block to a diagonal order, so that Macro Blocks 4 and 5 can be processed simultaneously, because there is always enough diagonal information from Macro Block 4 for processing Macro Block 5, thus freeing t5’s dependence on t4. In this case, the ordered sets from the ODO output are:

 S0 = { t1 }

 S1 = { t2, t3 }

 S2 = { t4, t5, t6 }

 S3 = ( t7, t8 }

 S4 = { t9 }

The above execution ordering corresponds exactly to the Diagonal-MBRP execution order. So the ODO algorithm can be seen as another way of showing the Diagonal-MBRP algorithm is the optimal procedure for processing n x n Macro Blocks.

**Part II: Communicative Deadlock Detection**

The procedure of the ODO algorithm demonstrates a method for scheduling related tasks optimally. With some additional steps, this algorithm can be adopted for deadlock detection and prevention for a set of coordinated tasks running in parallel.

As stated in Part I, the interdependencies between tasks have to be known a priori for the scheduler to order the tasks properly. For the purpose of deadlock detection, an instrumentation has to be in place for each task to communicate with a monitor when shared resources are to be accessed. Formally, we define the following components:

 T = { set of related tasks running asynchronously and concurrently }

 Task Monitor - communicates with tx, member of T through messages.

 Messages: Request-to-Access(S)

 Grant-to-Access(S)

 Notify-to-Release(S)

 When a task tx of T desires to access some shared resources S, tx sends a

 Request-to-Access to the Task Monitor. The Task Monitor checks the current

 state of dependencies among the running processes. It will send tx back a

 Grant-to-Access message only if no deadlocks is found, if tx is allowed to run. Tx

 will access the shared resource only if the Grant-to-Access for that resource is

 received. After tx is finished with the shared resource, it sends the Task Monitor

 a Notify-to-Release message.

Dependency in this procedure is defined strictly as order of access to shared resources. As an example:

If tx has requested to access resource R1 and is granted to do so, and then ty requests to access resource R1 while tx still holding the resource, then ty is said to be dependent on tx. In this scenario, if ty is holding resource R2, and tx requests to access R2 before release R1, the Task Monitor will detect a deadlock, if the request is granted. The procedure of the algorithm is as follows:

 Task Monitor listens forever for requests from the tasks

 A Request-to-Access(S) is received from tx. The Task Monitor records the request and checks all the resource holding state of all the running tasks. Iif some other task ty already holds resource S, marks tx as dependent on ty.

 The Task Monitor runs the ODO algorithm on the set T. If all the members can be ordered, then

there is no deadlock. The Task Manager sends a Grant-to-Access(s) to tx.

 If deadlock is detected, the Task Manager waits until a Notify-to-Release(S) message is received

from ty. The Task Monitor updates the resource holding state for all tasks. Goto step 3).

Basically, the above procedure repeatedly tests snapshots of the running tasks, whenever any of which wants to access shared resources. Strictly speaking, the Task Monitor avoids deadlocks by preventing cycles among the tasks to occur. The Task Monitor discovers the dependencies from communications with the tasks. The overall cost for each round (or snapshot) of processing is O(n^3), with additional exchanges of O(n) messages, and updating of O(n) tasks for dependencies. Note that the ODO algorithm in step 3) can be replaced by any efficient algorithm that can show a deadlock will occur if tx is granted the access to shared resource.