CS 416 - Operating Systems Design

Assignment 01
Parallel transitive closure of a directed graph

January 29, 2013
Due by Feb, 20 - Midnight

1 Announcements

January 29, 2013- Assignment posted
Group size: 4 people per group
Due date: Feb/20/2013 at 12:00 a.m.
Submission: Electronically, via Sakai.
Logistics: This assignment should be carried out on the iLab machines

2 Assignment Overview

This assignment will give you the opportunity to gain some experience with concurrent programming. You will have to implement four versions of a parallel algorithm to find the transitive closure of a directed graph. Two versions should rely on threads (in particular, the pthreads library), whereas the other two should be based on multiple processes. You will be dealing with shared-memory communication issues and the related synchronization problems that arise, such as ensuring mutual exclusion for accessing a shared resource. You will also have to synchronize threads/processes to make sure that the operations they perform are executed in the correct order, so that the correct transitive closure of the graph is produced at the end.

More details about the assignment are given below.

3 Background

3.1 Transitive Closure of a directed graph

Given a directed graph $G = (V, E)$ with vertex set $V = \{1, 2, \ldots, n\}$. The transitive closure of G is defined as the graph $G^* = (V, E^*)$, where $E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\}$.

Here, G^* contains the same vertex set $|V|$. However, it contains an edge $(i, j) \in E^* \iff$ there is a path from vertex i to vertex j in G.

The fundamental operation in Warshall's algorithm is to determine whether there is a path going from vertex i to vertex j for all $i, j \in V$. It is based on the following observation.

Let the vertices of G be $V = \{1, 2, \ldots, n\}$, and consider a subset $\{1, 2, \ldots, k-1\}$ of vertices for some k.

For any pair of vertices $i, j \in V$, the algorithm checks whether there exists a path from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k-1\}$. Then it considers vertex k in the intermediate vertices set.

There may be two cases:

1. There was already a path from i to j whose intermediate vertices are all drawn from $\{1, 2, \ldots, k-1\}$.
2. There was no path from i to j when intermediate vertices are all drawn from $\{1, 2, \ldots, k-1\}$. In this case, it checks whether there is a path from i to k and also a path from k to j.

For $i, j, k = 1, 2, \ldots, n$, let's define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set $\{1, 2, \ldots, k\}$, and 0 otherwise.

Transitive closure $G^* = (V, E^*)$ can be formed by putting edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$.

A recursive definition of $t_{ij}^{(k)}$ is: $t_{ij}^{(0)} = \begin{cases} 0 & \text{if } (i,j) \notin E \\ 1 & \text{if } (i,j) \in E \end{cases}$

And for $k \geq 1$
$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$

3.2 Representing the Graph

Let $n = |V|$, so that the graph can be represented by an $n \times n$ adjacency matrix $A = a_{ij}$
$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$

The output will be an $n \times n$ matrix $T^{(n)} = t_{ij}^{(n)}$, where entry $t_{ij}^{(n)}$ will be 1 if there exists a path from i to j considering all n vertices, otherwise it will be 0.

3.3 The sequential Warshall's Transitive Closure algorithm

Algorithm 1 TRANSITIVE-CLOSURE(A)

```
n ← rows[A]
// initialization
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $t_{ij}^{(0)} = a_{ij}$
    end for
end for
// computation of Warshall's Transitive Closure
for $k = 1$ to $n$ do
    for $i = 1$ to $n$ do
        for $j = 1$ to $n$ do
            $t_{ij}^{(k)} = t_{ij}^{(k-1)}$ or $(t_{ik}^{(k-1)}$ and $t_{kj}^{(k-1)})$
        end for
    end for
end for
return $T^{(n)}$
```

3.4 Intuitive Concurrent Implementation

The performance of the code shown in Section 3.3 above may be improved if the computation is performed in parallel. One way to achieve such parallelism is to have one process (or thread) work on a set of rows. Note that the maximum number of processes (or threads) the algorithm can use is n, one for each row. Each process (or thread) has one or more adjacent rows and is responsible for performing computation on those rows.

Algorithm 2 TRANSITIVE-CLOSURE(A)

```
n ← rows[A]
// initialization
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $t_{ij}^{(0)} = a_{ij}$
    end for
end for
// computation of Warshall's Transitive Closure
for $k = 1$ to $n$ do
    for $i = 1$ to $n$ in parallel do
        for $j = 1$ to $n$ do
            $t_{ij}^{(k)} = t_{ij}^{(k-1)}$ or $(t_{ik}^{(k-1)}$ and $t_{kj}^{(k-1)})$
        end for
    end for
end for
return $T^{(n)}$
```

3.5 Bag of Tasks

The code shown in Section 3.4 should statically distribute the load among the processes/threads, each process/thread should work in a pre-defined set of rows (e.g., process i should work on all rows such that n mod #processes = i).

Another work distribution approach is to have each process (or thread) work on any row. The parent process (or main thread) creates a queue that has all the rows that need to be processed, as described in Algorithm 3. Whenever a worker (working thread/process) is idle, it can access the queue to get another row to work on, as in Algorithm 4.

Algorithm 3 TRANSITIVE-CLOSURE(A)

```
1: $n \leftarrow$ rows[$A$]
2: // initialization
3: **for** $i = 1$ to $n$ **do**
4:     **for** $j = 1$ to $n$ **do**
5:         $t_{ij}^{(0)} = a_{ij}$
6:     **end for**
7: **end for**
8: // computation of Warshall's Transitive Closure
9: **for** $k = 1$ to $n$ **do**
10:     **for** $i = 1$ to $n$ **do**
11:         enqueue( $i$, $k$ )
12:     **end for**
13:     **wait** until the queue is empty
14: **end for**
15: **return** $T^{(n)}$
```

Algorithm 4 WORKER( )

```
1: **while** queue is not empty **or** $k < n$ **do**
2:     $i, k \leftarrow$ get one row from the queue
3:     // computation of Warshall's Transitive Closure
4:     **for** $j = 1$ to $n$ **do**
5:         $t_{ij}^{(k)} = t_{ij}^{(k-1)}$ **or** $(t_{ik}^{(k-1)}$ **and** $t_{kj}^{(k-1)})$
6:     **end for**
7: **end while**
```

In this version, it is important to ensure the consistency of the queue with locks.

4 The Assignment

You are required to write a program, source file **wtc.c**, that creates p processes or threads to find the transitive closure of a directed graph. Based on the command line arguments to **wtc**, it creates either processes (1) or threads (2) to work in parallel using the intuitive Algorithm 2, or it creates processes (3) or threads (4) to find the transitive closure using the bag of tasks method. Another argument is the input graph G, the file format of which is detailed below in Section 4.5. The output of **wtc** is the screen (stdout) and it is detailed in Section 4.6. As an example, calling **wtc** to find the transitive closure of the graph specified in the file *input.in* in parallel with threads, a user should use the following command line:

```
% wtc 2 input.in
```

4.1 Process-level concurrency

You are required to write this module in the source file **wtc_proc.c**. It creates p processes to find the transitive closure of a directed graph. Your program should read the input graph from a text file, the name of which is given in the command line, and print out the solution on the screen (stdout).

The program should perform the followings steps:

- Create a shared-memory area for holding data and synchronization variables. You will be required to use semaphores for both synchronization and achieving mutual exclusion.
- Initialize the semaphores to the appropriate values.
- Create p child processes. Each process will perform its share of work. The processes will exit after its rows computation is done and the results have been written to the shared memory.
- The parent process waits until each child process is done. Then it will print out the results and the time spent in computation.

4.2 Thread-level concurrency

You are required to write this module in the source file **wtc_thr.c**, which creates p threads to find the transitive closure of a directed graph. Your program should read the input graph from a text file, the name of which is given in the command line and print out the solution on the screen (stdout).

The program should perform the following steps:

- Since all threads share the same address space, all global memory can be shared among them. No special shared memory should be created.
- Initialize the mutexes (locks) and condition variables (if you wish) to the appropriate values.
- Create p execution threads, performing the same functions of the processes above.
- The main thread waits until the other threads are done. Then it will print out the results and the time spent in computation. Even though no explicit shared memory is created, each access to a shared variable should be protected against race conditions. In this case, you should use pthread locks.

4.3 Process-level concurrency - Bag of tasks

You are required to write this module in the source file **wtc_btproc.c**. It creates p processes to find the transitive closure of a directed graph. Your program should read the input graph from a text file, the name of which is given in the command line and print out the solution on the screen (stdout). This time, each process has to get the row from a queue and work on it. The queue is maintained by the main process (parent).

4.4 Thread-level concurrency - Bag of tasks

You are required to write this module in the source file **wtc_btthr.c**, which creates p threads to find the transitive closure of a directed graph. Your program should read the input graph from a text file, the name of which is given on the command line and print out the solution on the screen (stdout). This time, each thread has to get the row from a queue and work on it.

4.5 Format of the input file

The input text file describing the graph should conform to the following specification:

```
p
n
<from> <to>
.
.
.
```

Listing 1: Input file format

Where, p is the number of processes/threads your program should launch, n is the number of vertices on the graph G and each row, an edge in the graph from vertex 'from' to vertex 'to'. One example of such a file would be:

```
2
4
2 1
2 3
2 4
3 4
4 1
```

Listing 2: Input file (sample)

4.6 Format of the output file

The output describing the transitive closure should print out the matrix $T^{(n)}$ on the screen. Use space character to separate elements of the same row, and the newline character (
n) to distinguish rows o $T^{(n)}$. After the matrix, print the time spent to find the transitive closure.

One example of such a file would be:

```
    0    0    0    0
    1    0    1    1
    1    0    0    1
    1    0    0    0
Time:  12  us
```

Listing 3: Output file (sample)

5 Hints

The following system calls might be necessary during implementation

- Process creation and synchronization
  - fork(), wait(), exit()
- Threads
  - pthread_create(), pthread_exit(), pthread_join()

- Shared memory
  - shm_open(), shm_unlink(), mmap(), munmap(), ftruncate()
- Critical section, semaphores
  - sem_init(), sem_open(), sem_close(), sem_destroy(), sem_getvalue(), sem_wait(), sem_post(), sem_unlink()
  - pthread_mutex_lock(), pthread_mutex_unlock(), pthread_cond_wait(), pthread_cond_signal()
- Time measurement
  - gettimeofday()

The most complete documentation regarding the above functions is to be obtained using man pages. You can type man <function_name> in the Linux shell to obtain a detailed description of a given function.

6 What to hand in

You should hand in the source code and Makefile in a single tar-gz file. The file names and the input format should perfectly match the ones presented above. An additional text file should be provided, containing the time measurements experienced for all process and thread implementations, for 2-512 processes (threads) and 32 samples of various sizes of graphs. **An analysis and result interpretation should be provided as well, based on time measurement**.

7 Grading

This assignment is worth 100 points that are distributed in the following way. 60 points for functionality, 20 for performance and 20 for the report.

7.1 Functionality

Your solution will be evaluated by several test cases. It is a pass/fail evaluation that gives 15 points for **wtc_proc**, 15 points for **wtc_thr**, 15 points for **wtc_btproc**, and 15 points for **wtc_btthr**. Warnings during compilation, memory leaks during execution will result in losing of points.

7.2 Performance

The performance of your solution can result in up to 20 points. It will be made a ranking of the 10 fastest solutions (average of all implementations). The fastest group gets 10 points, the second fastest 9 points and so on.

The same will happen with memory usage. The solution which uses less dynamic memory will get 10 points, second gets 9, and so on.

8 Test cases

Assume numbers are separated by one space character. You are supposed to test with one thread/process up to the number of vertices of thread/processes.

```
3
6
1 2
1 6
2 5
3 2
3 4
4 5
5 3
5 1
6 4
```

Listing 4: Test case 1 - input1.in

```
3
9
1 2
2 4
4 5
5 3
3 6
6 5
6 1
3 9
9 1
9 8
9 7
7 2
8 7
8 4
3 8
```

Listing 5: Test case 2 - input2.in

9 Notes

- Use Linux and C language.
- Your solution has to work on iLab machines.
- Do not assume any static limit in memory allocations, we have test cases with tens of thousands of vertices to evaluate your solution.
- Do not copy the solution from other students. You are encouraged to discuss and share ideas, not the implementation.
- Submit one solution per group.