TDT4200 Parallel Computing

Problem Set 4

Stian Hvatum (hvatum)
MTDT

October 18, 2012

Contents

Task 1 Theory

1. Differences between processes and threads

1. Processes have their own private virtual memory space provided by the operating system, while threads within one process share this memory space among each other.
2. Processes do not usually share resources with each other, while threads within one process usually share the allocated resources.
3. Processes are usually scheduled by the operating system, while thread scheduling is much more controlled by the programmer[1].

2. OpenMP vs. POSIX Threads

b) 1. Benefits of OpenMP

OpenMP code is generally more readable than POSIX Thread programs. OpenMP is also much easier to use when you parallelize an already existing serial program, since you add almost only pragmas instead of having to rewrite the entire program.

b) 2. Benefits of POSIX Threads

POSIX Threads and the pthreads library gives the programmer much more fine grained control of the threads than OpenMP, while you still can get equal speed as OpenMP. pthreads is also in my opinion easier to use when you have a real world application that do different things at the same time, eg. listening for incoming network traffic while prompting the user for input.

3. Mutex vs. Semaphore

Mutexes and semaphores are both utilities for synchronization in parallel programs. A mutex is a kind of lock, such that one thread can only acquire the lock if no one else has acquired it. Once a lock is required, it must be released by it's owner before someone else can lock it. This provides us with a method to allow only one thread through a critical section at a time. The first thread will acquire lock on the mutex, while later threads will wait at the lock(mutex)-call until the owner unlocks it. Then, one of the waiters will acquire lock, and passes by, while the remaining threads still wait for their turn. Similar to mutexes, semaphores also provides this kind of mechanism, but semaphores uses a kind of resource counter to allow a certain number of threads/calls to pass through.

A common example for using semaphores are the producer/consumer-pattern, where two semaphores represents number of produces elements and number of empty spaces to put produced elements in. At each production, the produced counter is

[1] Still, threads are usually preemptive scheduled by the OS, but the programmer can sleep, wake, create and join threads as he wants

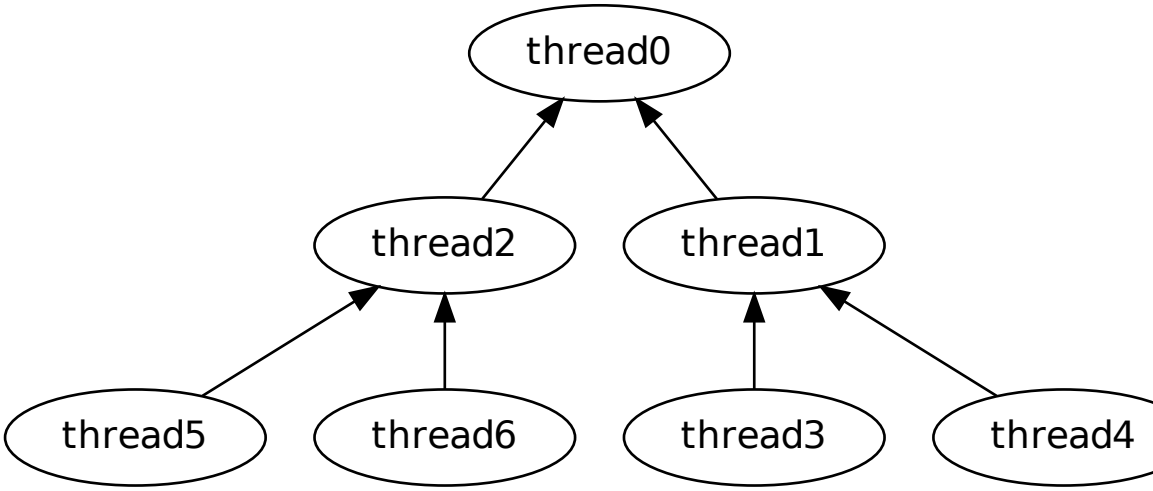

Figure 1: Passing results

incremented, and the number of spaces are decremented. When there are no more spaces left, the producer will go to sleep until the semaphore in incremented by the consumer. The consumer increments the number of empty spaces and decrements the number of produces elements at each "consume", and goes to sleep if number of produces elements is zero.

A good example of using mutexes is programs that contains critical sections, eg. global variable updating, and needs to ensure that only one thread at a time updates the variable. If this global variable is updated multiple places in the same program, we can use the same mutex on the different locations to ensure that only one thread are running in only one of the locations.

4. Reduction

When we are updating a number in parallel, we may run into a race condition. If we are using the same arithmetic operation in each thread, we can divide the problem such that each thread calculates their part, and then we group threads, eg. into a binary tree, and pass the resulting parts against the root. In this way, we get $log_2(N)$ arithmetic operations that has to be synchronized instead of N operations. One possible implementation is illustrated in Figure 1.

```c
#include <stdio.h>

int main(void)
{
    int sum = 0;
    #pragma omp parallel for
    for (int i = 0; i < 10; i++)
        sum += i;
    printf("Sum is %d\n", sum);
}
```

Figure 2: Race condition

5. Race condition

The code in Figure 2 is a simple program that adds all numbers from 0 to 9 (excluding 10). The code is really simple, but the inclusion of OpenMP generates a severe problem. When we parallelize the loop, multiple threads will read `sum`, add `i` to it, and store the value back into `sum`. Since threads often are interleaved, one thread may write back `sum` right after another thread has read it, but before it has added to it. If this happens, the last thread will overwrite `sum` with a sum that skipped the other thread's addition. This phenomenon is called a race condition since the threads "races" to gain access to a resource.

6. Thread safety

If a function is *thread-safe*, it can safely be used in a multithreaded program. When a function is *thread-safe*, it basically means that multiple threads will not get in each others way if they call the function simultaneously. If all modified variables are one the current threads stack, or they are locked in a proper way, a function will in most cases be thread safe[2].

The `fibonacci`-function can not be considered *thread-safe*. The reason is that it makes use of `static` variables, which are not completely local. A static variable is a "local" variable that is stored somewhere else than on the stack[3]. Since it is not stored on the stack, it's value is kept between calls, and subsequent calls can continue where the last stopped, like in the `fibonacci`-function.

These static variables creates a problem for multithreaded programs, since if an other thread calls the function while a first thread is in the middle of it, the new thread will alter the value of the static variable, and suddenly both threads call the function within the same "state". This is the same as for global variables, and can not be considered thread safe.

[2] Given they don't call any thread-unsafe functions

[3] There is no standard place for static variables, but GCC tends to store it at .bss (Block Started by Symbol)

The Serial, pthreads and OpenMP-values are wall clock time for running the program with 100 time steps, # threads and 1002 planets. The speedup-columns are the running time of the serial program devided by the running time of the parallelized program in the left-of column.

# threads	Serial	pthreads	Speedup	OpenMP	Speedup
1	1.034	1.107	0.934	1.036	0.998
2	1.034	0.843	1.227	0.535	1.933
3	1.035	0.632	1.638	0.363	2.851
4	1.035	0.504	2.054	0.278	3.723
5	1.035	0.427	2.424	0.225	4.600
6	1.035	0.365	2.836	0.202	5.124
7	1.034	0.328	3.152	0.170	6.082
8	1.034	0.293	3.529	0.154	6.714

Table 1: Speedup

Task 2 Programming

1. OpenMP

See nbody_openmp.c

2. POSIX threads

See nbody_pthreads.c

3. Speedup

I have tested both programs against the serial program on clustis3.idi.ntnu.no, and the speedup was as shown in Table 1. I receive almost linear speedup in the OpenMP-version, this is probably because the most computational-heavy parts of the program has been parallelized. In the pthreads-version, everything is block-scheduled among the threads, which may bad for cache if threads are evenly interleaved. On my personal computer[4], the difference where much less, though the wall clock time for 9 threads[5] for both where about 0.360 seconds, OpenMP being slightly faster. The number of planets in the test where 1002, as in the provided planets.txt-file. If I use some more than 8 threads, the speedup for both are minimal, since this is the number of cores (including HT) on both my private computer and clustis. If I use fewer time steps, pthreads seems to be faster, while more time steps seems to boost OpenMP. I believe this is because pthreads can schedule the work with less overhead, and thus use less work on getting started, while OpenMP have better scheduling of threads, and have better throughput once started.

[4] Intel i7 930 @ 2.8GHz

[5] On 8 threads, OpenMP used 2.3 seconds!