TDT4200: Problem set 2, Part 1, Theory

Parallel Computing

Neshat Naderi

September 19, 2016

Problem 1

a) Total elements ditributed and gathered, assumming one of nodes is master node and does not need communication with itself.

$$\underline{\underline{Elements = \frac{n^2}{p}(p-1)}}$$

b) Grid elements sent in periodic boarder exchange

$$\underline{\underline{Elements = 2(\frac{n}{r} + \frac{n}{q})}}$$

c) If n is total number of elements to be sent(calculated in b), $T_s = s$ is latency(time taking to start transmition and recieving data) and β is size of each element in Bytes then communication time T_{comm} is defined as

$$T_{comm} = T_s + \beta n$$

$$\underline{\underline{T(q,r) = s + 2(\frac{n}{r} + \frac{n}{q}) \cdot \frac{d}{b}}}$$

d) $q = r = 8$

Problem 2

If odd-even sort is applied when $n = p$ the communication overhead would be much higher than a compare-swap operation, so parallizing would be nonsense. According to *Pachebo* it is assumed that $n/p > 1$. Here n is the number of elements in a list that is evenly divisable by p, and p is the number of processes.

a) Speedup and efficiency.

$$S(n,p) = \frac{T_{serial}}{T_{parallel}},$$

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

In this case the serial program runs in T_s and parallel runtime is T_p

$$T_s = n^2$$

$$T_p = p \cdot \frac{n}{p} + (\frac{n}{p})^2 = n + (\frac{n}{p})^2$$

By this we get speedup and efficiency as below

$$\underline{\underline{s(n,p) = \frac{n^2}{n + (\frac{n}{p})^2} = \frac{n}{1 + \frac{n}{p^2}}}}$$

$$\underline{\underline{E(n,p) = \frac{n^2}{np + \frac{n^2}{p}} = \frac{n}{p + \frac{n}{p}}}}$$

Setting up a table for a different values for n and p

Table 1: Efficiency calculated with some values for n and p

n	p	$E(n,p)$
64	4	3.2
128	4	3.6
128	8	5.3
128	16	5.3
256	4	3.8
256	8	6.4
512	16	10.7

b) Linear speedup? To achieve linear speedup, efficiency must be equal 1.0. Since speedup expression above shows $S(n,p) \neq p$ and then the program does not obtain linear efficiency.

To see if the problem is scalable, we increase the p by factor of k and problem size n by x factor. So number of processes are kp and problem size is xn. In our case

$$E = \frac{xn}{kp + \frac{xn}{kp}}$$

c) Weakly scalable? No, because the efficiency does not stay constant as the problem size increases at the same rate as proceses. Also when $x = k$. If we increase the problem size to $2n$ and processes to $2p$ according to table 1, we don't obtain constant efficiency. Therefor not weakly scalable.

d) Strongly Scalable? Yes, because efficiency stays almost constant if only problem size increases(Table 1). For example if we increase p with factor of 2 and keep problem size 128, the efficiency stays constant.

Problem 3

Process is heavy-weight. Memory blocks are private and memory blocks are allocated for stack and heap. Descriptor of resources, process state information and security informations are stored in a process. **Thread** is lighter-weight than a process. It shares memory resources and mostly all process specific things, except stacks and program counters.

Problem 4

By using private variables, performance of the serial critical section is improved. The partial sum is added to global sum only once. Moving busy-wait flag outside the loop avoids threads to alternate between waiting and executing. Incrementing and waiting increase the runtime when busy-wait flag is inside the loop.

Problem 5

a) **Race condition** happens when threads or processes try to access resources at the same time. A resource, for example memory area can be assigned only to one thread/process at the time. Therefore, there is a competition for accessing the resource before other threads/processes.

b) **Critical section** is a block of code that could be executed by only one thread at the time. Often when a thread enters such a critical section, it blocks until it finishes executing that section.

c) Schemes for protection access to critical sections:

1. **Mutual exclusion lock**(mutex) is basiclly a lock that protects a critical section. When a thread enters a critical section, it calls the mutex function to lock. When it finishes the execution it calls the unlock function. So that other threads could start execution. In this case the critical section is serial because a mutex does not allow more than one thread to execute the code at a time.
   pros:
   (a) Critical section is owned/locked by a thread until the execution is done.
   (b) Guarantees that one thread excludes all other threads in a critical section. So the chance of collisions is rare.
   (c) Better performance than busy-waiting when the number of threads increas.

   cons:
   (a) The order of what thread enters critical section is random. It creates problems in uncommutative operations like matrix multiplication.
   (b) All other threads must wait until the owner has unlocked. For example, if the thread which owns the lock enters an infinite loop, other threads never execute the critical section and they wait forever.

2. **Busy-waiting** is another alternative to protect a critical section. It is often very unefficient because in busy-waiting a thread enters a loop and wait for a condition. So the thread could stuck in the loop and waste the system resources by testing a condition continously.
   pros:
   (a) Only one thread can enter the critical section. Thus resulting correctly.
   (b) It is simple and threads execute a critical section in order.

   cons:
   (a) Computer optimizing can affect correctness of busy-waiting reseults. Because compiler do not know the program is multithreaded. Turning off computer optimizer degrades performance.
   (b) It wasts CPU resources.

3. **Semaphores** are similar to mutexes but they are more powerfull. In semaphores threads do not own the critical section. Semaphores are a type of thread synchronization.
   pros:
   (a) Threads are synchronized and they execute by order.(Produser-consumer synchronization)
   (b) No waste of CPU ressurser due to busy-waiting.

   cons:
   (a) Threads must moniter other threads' updates.
   (b) Correctness of program cannot be verified easily.