Problem Set 2, Part 1

21. Sept, 2016

Problem 1

a)

$n * n$

b)

For one processor, the number of grid elements to be sent is: $2 * \frac{n}{q} + 2 * \frac{n}{r}$.

So my answer is $q * r * 2n(\frac{1}{q} + \frac{1}{r}) = 2n(q + r)$.

c)

Vertical: We can do all vertical communication in two batches, where 2 and 2 processes transfer $2 * \frac{n}{q}$ bytes. So in seconds,this is $(2 * \frac{dn}{q} * \frac{1}{b}$ sec $) * 2 + 2s$.
For **horizontal** connections, it's the same but with r instead q.
My answer is thus $4s + \frac{4dn}{b}(1/q + 1/r)$.

d)

There are 3 cases: 2x32, 4x16 and 8x8.

2x32

$4s + \frac{4dn}{b}(1/2 + 1/32)$

4x16

$4s + \frac{4dn}{b}(1/4 + 1/16)$

8x8

$4s + \frac{4dn}{b}(1/8 + 1/8)$

We can see that the expression is minimal when q=8 and r=8.

Problem 2

a)

The run-times are given in Pacheco on page 135. Given these, the speedups are (given by time_parallel/time_serial):

Processes	200k	800k	3200k
2	2.0465	2.0526	2.093
4	4	4.0625	4.1860
8	7.3333	7.6470	8.1818
16	11.7333	13.4482	13.8431

And the efficiencies are (given by speedup/num_processes):

Processes	200k	800k	3200k
2	1.02325	1.0263	1.0465
4	1	1.0156	1.0465
8	0.9166	0.9558	1.0227
16	0.7333	0.8405	0.8651

b)

The program doesn't obtain linear speedups - this can be seen in the case of 16 processors: the efficiency is well below 1.

c)

Strongly scalable: constant efficiency when we only increase number of processes with constant problem size.
No.

d)

Weakly scalable: we can keep efficiency constant when we increase the problem size at the same rate as we increase the number of processes. Not quite so. The efficiency with 2 processes and 200k elements is 1.02325, and the efficiency with 8 processes and 800k elements is 0.9558.

Problem 3

1. Processes don't share memory. Threads do share memory.
2. Threads are much more light weight to create - much less overhead.

Problem 4

For every iteration, the processes have to wait for each other to execute the critical section. The critical section is in effect a serial part of the code. Once we move the critical section out of the loop, the loop will execute much faster and in true parallel - independent of other threads. The serial part is minimized to be proportional to the number of threads rather than to the problem size.

Problem 5

a)

A race condition happens when some output is dependent on the order in which events happen, when the programmer cannot control this ordering. For example, if two threads repeatedly add some value to a global sum, when they add a value at more or less the same time, it becomes a matter of fate what happens next. This is because in machine code, the threads first make a local (to the CPU) copy of the global sum, add their value, then write it back. It's not atomic.

b)

A critical section is a section of code that can only (for correct results) be executed by one thread at a time. For example due to race conditions.

c)

Busy-waiting

Round-robin style. Threads take turns entering the critical section, strictly in order.

Pros

- Works, easy to implement.
- Enforces ordering, which can be critical in some applications.

Cons

- Continually uses CPU - wasting power and computing time.
- If there are more threads than cores, a thread whose turn it is, may be blocked.
- If we don't need ordering, it's wasteful that threads wait for their turn - in case some are done earlier than others but are later in the queue.

Mutex locks

Before the critical section is a funtion call to lock the mutex. But once a thread has locked it, all other threads calling this funtion will be blocked until the mutex is unlocked.

Pros

- Doesn't enforce ordering, in the case that we don't need it.
- With more threads than cores, the performance doesn't degrade in the same way as when using busy-waiting - it doesn't depend on blocked threads.

Cons

- Doesn't enforce ordering, in the case that we need that.

Semaphores

A semaphore is a generalization of mutexes. It can be thought of as an unsigned int. When it's 0, a call to sem_wait will block. Else, calls to sem_wait will decrement the integer. sem_post increments the integer, effectively allowing someone to "consume" (sem_wait).

Pros

- More flexibility: we can do more - e.g. the number of processes in a critical section can be limited by
- And especially works well for "producer-consumer synchronization", in which there is no critical section, but the "consumer" thread just has to wait for the "producer" thread.

Cons

- Don't know.