Computer architecture and operating systems (DAT4, SW4)

Solutions Sketch

1) Consider a program that execute the code fragment below. Would there be any benefit from using multiple-threads in this program? On a unicore-machine? On a multi-core machine? Explain precisely!

```
Thread1{
  data1=read(File); //20ms
  sum1=compute(data1);//10ms
}
Thread2{
  data2=fetch(url);  //50 ms
  sum2=compute(data2);//10ms
}
main: int sum1; int sum2; start(t1),start(t2);join(t1);join(t2); printf(sum1+sum2);
```

You will benefit significantly on a single-core machine by running two threads and thereby overlapping the two i/o operations: A possible (best case) execution scenario

- time 0: the two i/o operations would be initiated concurrently
- time 20 the first file io completes, and the core is (perhaps, depending on other processes) free to compute sum1
- time 30 compute sum1 done.
- time 50 second io completes, and the core is (perhaps, depending on other processes) free to compute sum2
- time 60. Thread 2 completes as well and the process is done.

```
2.1)
    (A==6 && B==27) (A==6 && B==24)
    (A==6 && B==31) (A==-1 && B==27)
    (A==-1 &&B==24) (A==-1 &&B==31)
    (A==3 &&B==31) (A==3 &&B==27)
    (A==3 &&B==24)
```

Observe: that high-level statements like x++ is not atomic and normally using 3 machine instructions between which pre-emption is possible:

1) load (move) from mem to register

2) add to register

3 write (move) register to memory

Hence, a possible execution is :

Thread A 7 FROM TO	Thread B 4 FROM TO	FROM	TO
		10	20
tmp1=*from;//10			
tmp1-=7; //3			
	tmp2=*from//10		
	tmp2-=4 //6		
*from=tmp;1//3		3	
	*from=tmp2 //6	6	
tmp1=*to;//20			
tmp1+=7; //27			
	tmp1=*to;//20		
	tmp1+=4; //24		
*to=tmp1; //27			27
	*to=tmp2;		24
		6	24

2.2)
In this case it is sufficient to hold the lock only when incrementing or decrementing as money is transferred:
Variant A)

```
transfer(int * from, mutex *from_lock, int * to, mutex*to_lock, int amount){
 lock(from_lock)
 *from-=amount
 unlock(from_lock)
 lock(to_lock)
 *to+=amount
 unlock(to_lock)
}
```

It would be ok to wrap the entire transaction by acquiring all required locks: simpler and safe (but in this case not performance wise optimal, as the from lock is held when not strictly needed)

Variant B)

```
transfer(int * from, mutex *from_lock, int * to, mutex*to_lock, int amount){
 lock(from_lock);
 lock(to_lock);
 *from-=amount
 *to+=amount
 unlock(to_lock)
 unlock(from_lock)
}
```

2.3)
//direct translation, but multiple exit points: do we always remember to unlock in these cases?

```
transfer(int * from, mutex *from_lock, int * to, mutex*to_lock, int amount){
 lock(from_lock)
  if(amount <0 || *from<amount) {
     unlock(from_lock);
     return false
  }
 *from-=amount
 unlock(from_lock)
 lock(to_lock)
 *to+=amount
 unlock(to_lock)
}
```

```
//Simpler and generally better (but minimal longer locking periode of "to")
transfer(int * from, mutex *from_lock, int * to, mutex*to_lock, int amount){
 int transferred=false;
 lock(from_lock);
 lock(to_lock);
  if(amount >0 && *from>=amount) {
    *from-=amount;
    *to+=amount;
    transfered=true;
 }
 unlock(to_lock);
 unlock(from_lock)
return transfered;
```

```
}
```

2.4)

```
Pthreads code here.
```

2.5 Variant A in 2.2 will not work if there is possible concurrent "getBalance()" as this may execute between the middle unlock and lock, and report a balance where an amount has been withdrawn from one account, but not yet deposited on another. Variant B works as it locks both involved accounts for the entire transaction.

2.6) Variant B would not work as it has potential to deadlock: (see next lecture)

Thread 1	thread 2
lock(A)	
	lock(B)
lock(B) //cannot succeed: held by thread 2	
	lock(A) //cannot succeed held by thread 1

A fix is to ensure A and B is locked in the order: first A, then B (Why? see next lecture). A simple change that would work (**only if** you used variant B in 2) is to swap to and from locks in one of the threads

```
Thread mand(){transfer(&kontoA,&lockA, &kontoB,&lockA,7));
Thread kone(){transfer(&kontoB,&lockA, &kontoA, &lockB,4));  //WAS  Thread kone(){transfer(&kontoB,&lockB, &kontoA,
&lockA,4));
```

```
3)
```

3.1) You need something like the code below: Remark that the result will be wrong if you only hold the locks one at a time as worker threads may increment/transfer meanwhile. Also lock local first then the global one (to prevent transfer to global whlie we getAll)

```
int getAll(counter_t *c)
{
    for(int i=0;i<NUMCPUS;i++) {
            Pthread_mutex_lock(&c->llock[i]);
    }
    Pthread_mutex_lock(&c->glock);
    int val = c->global;
    for(int i=0;i<NUMCPUS;i++) {
         val+=c->local[i];
    }
    Pthread_mutex_unlock(&c->glock);
    for(int i=0;i<NUMCPUS;i++) {
       Pthread_mutex_unlock(&c->llock[i]);
    }
    return val;
}
```

3.2) Using our getAll is reasonable when used infrequently. If there are many frequent calls to getAll the entire data-structure will be essentially sequential, and then we are no better off than using the accurate single lock version

3.3) No answer

4) See book

Challenge 13.

1) In ijk each worker gets its own private slice of the matrix, so no shared variable; no need for mutex

2) there is a large overhead of creating thread, joining and scheduling threadst: in the order 500000 cycles (use a small data-size and increase the number of threads>> no cores; then you will see an increase in execution time caused by thread overhead: the inclination on a a graph showing execution time vs number of threads will indicate that overhead.

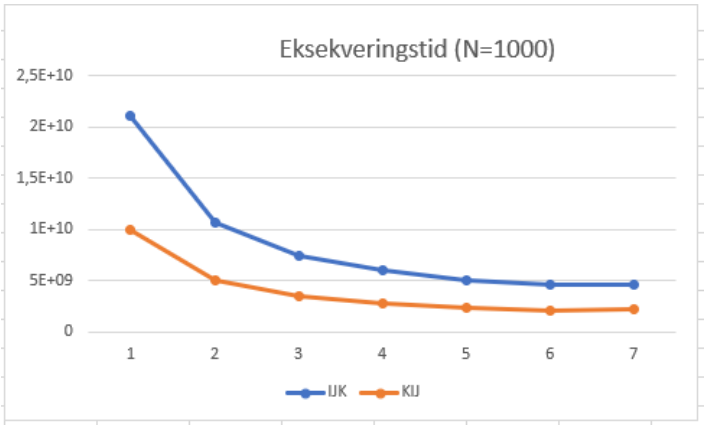
3) Your results may vary: with 6 cores I get a speedup up of 4.6. ie less than the ideal factor 6. This is despite being an easy parallelizable problem (at least our approach is), this reduction is caused by thread overhead, and load impalance (slice for the workers may not be entirely equal)

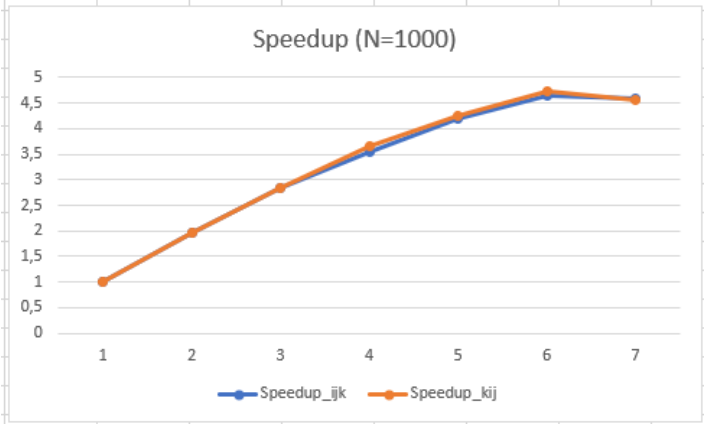
4) In my case I see no significant speedup for problem sizes less than a 100*100 matrix.

5) See code; be care that your kij version is not sliced such that the workers add contributions to the same cell! Observe that the kij method on one core actually outperforms the ijk method on 2 cores!! So locality matter!!

Code archieve;

Excel with graphs





Last modified: Wednesday, 13 April 2022, 11:27 AM

◄ Exercises

Jump to...

12-Intro-slides ►