Elements of correction for the exercises of CS2101A Lab 3

Instructor: Marc Moreno Maza, TA: Li Zhang

Wednesday, October 1, 2014

1 Exercise 1

```
@everywhere function fib(n)
          if (n < 2) then
              return n
          else return fib(n-1) + fib(n-2)
          end
       end

function runningtime(f,n)
       tic()
       y = f(n)
       t = toc()
       t
       end

N = 12

## We store the timings for the serial runs in T_1
T_1 = [runningtime(fib,35+i) for i=1:N]


## In fib_approximate, n is the input index and t is the threshold
@everywhere function fib_parallel(n, t)
          if (n < t) then
              return fib(n)
          else
              x = @spawn fib_parallel(n-1, t)
              y = fib_parallel(n-2, t)
              return fetch(x) + y
          end
       end
```

```
function runningtime(f,n,t)
       tic()
       y = f(n,t)
       t = toc()
       t
       end

H = 16

[ (35+i,30+t) for i=1:N, t=1:2:H]

## We store the timings for the parallel runs in T_4
## (my laptop has four cores): we use 8 different thresholds
T_4 = [runningtime(fib_parallel,35+i,30+t) for i=1:N, t=1:2:H]

## The array S stores the speedup ratios
S = [T_1[i] / T_4[i, iceil(t/2)]  for i=1:N, t=1:2:H]

using Winston

Sizes = [35+i for i=1:N]
C1 = Winston.Curve(Sizes, [S[i,1] for i=1:N])
C2 = Winston.Curve(Sizes, [S[i,2] for i=1:N])
C3 = Winston.Curve(Sizes, [S[i,3] for i=1:N])
C4 = Winston.Curve(Sizes, [S[i,4] for i=1:N])
## We plot for speedup curves in one frame
p=FramedPlot()
add(p, C1, C2)
add(p, C3, C4)

C5 = Winston.Curve(Sizes, [S[i,5] for i=1:N])
C6 = Winston.Curve(Sizes, [S[i,6] for i=1:N])
C7 = Winston.Curve(Sizes, [S[i,7] for i=1:N])
C8 = Winston.Curve(Sizes, [S[i,8] for i=1:N])
## We plot for speedup curves in a second frame
p2=FramedPlot()
add(p2, C5, C6)
add(p2, C7, C8)

## The first four curves seem to give better results
```

2 exercise 2

```
function mmult(A,B)
```

```
        (M,N) = size(A);
        C = zeros(M,M);
        for i=1:M
          for j=1:M
            for k=1:M
              C[i,j] += A[i,k]*B[k,j];
            end
          end
        end
        C;
      end

for d in [500,1000,1500,2000]
      a=rand(d,d)
      b=rand(d,d)
      @time mmult(a,b)
      end
elapsed time: 0.3470189571380615 seconds
elapsed time: 3.170802116394043 seconds
elapsed time: 23.14421010017395 seconds
elapsed time: 62.64995193481445 seconds
```

Theoretically, the number of arithmetic operations required to multiply two square matrices of order n is proportional to n^3. Hence, one could expect that the running time for $n = 1000$ should be $8 = (1000/500)^3$ times that for $n = 500$. But the above experimental results show a ratio close to 10. This is due to the high rate of cache misses in the naive algorithm for matrix multiplication. We saw in class better algorithms for this operation, in particular one based on a blocking strategy.

3 exercise 3

```
function qsort!(a,lo,hi)
          i, j = lo, hi
          while i < hi
              pivot = a[(lo+hi)>>>1]
              while i <= j
                  while a[i] < pivot; i = i+1; end
                  while a[j] > pivot; j = j-1; end
                  if i <= j
                      a[i], a[j] = a[j], a[i]
                      i, j = i+1, j-1
                  end
              end
              if lo < j; qsort!(a,lo,j); end
```

```
                lo, j = i, hi
            end
            return a
        end

function sortperf(n)
            qsort!(rand(n), 1, n)
        end

issorted(sortperf(5000)) ## to test whether your alog is correct
## should be true

[@time sortperf(2^e*1000000) for e=[0 1 2 3 4 5 6 7]]


elapsed time: 0.2307720184326172 seconds
elapsed time: 0.21168804168701172 seconds
elapsed time: 0.4441850185394287 seconds
elapsed time: 0.912261962890625 seconds
elapsed time: 1.9140989780426025 seconds
elapsed time: 3.977203130722046 seconds
elapsed time: 8.138957023620605 seconds
elapsed time: 16.795485973358154 seconds
```

Theoretically, the number of integer comparisons required to sort a list of n integers (using quick-sort) is proportional to $O(n \log(n))$. Hence, one could expect that the running time for $n = 2^7\,1000000$ should be 7/3 times that for $n = 2^6\,1000000$. (To verify this claim approximate 1000 with 2^{10}.) And, indeed, the above experimental results show a ratio close to 2. This is due to the relatively low rate of cache misses in quick sort algorithms.