Marc Beitchman
CSE P 506
Assignment #2

Question 1:

a) pseudo code for Dekker' algorithm extended for 3 processors:

Initial shared values:

```
bool flag[3] = {0}
bool waiting[3] = {0}
uint turn = 0    // ProcessID
```

P0:

```
flag[0] = true
while(flag[1] == true or flag[2] == true)
{
    if(turn != 0)
    {
        flag[0] = false;
        waiting[0] = true;

        while(turn != 0){}

        flag[0] = true;
        waiting[0] = false;
    }
}

ENTER_CS
{
    ....
    if(waiting[1] == true and waiting[2] == false)
    {
        turn = 1
    }
    else if(waiting[1] == false and waiting[2] == true)
    {
        turn = 2
    }
    else
    {
        // no one is waiting or both are waiting
        turn = randomly choose 1 or 2
    }
```

```
        flag[0] = false
}

P1:
flag[1] = true
while(flag[0] == true or flag[2] == true)
{
        if(turn != 1)
        {
                flag[1] = false;
                waiting[1] = true;

                while(turn != 1){}

                flag[1] = true;
                waiting[1] = false;
        }
}

ENTER_CS
{
        ....
        if(waiting[0] == true and waiting[2] == false)
        {
                turn = 0
        }
        else if(waiting[0] == false and waiting[2] == true)
        {
                turn = 2
        }
        else
        {
                // no one is waiting or both are waiting
                turn = randomly choose 0 or 2
        }

        flag[1] = false
}

P1:
flag[2] = true
while(flag[0] == true or flag[1] == true)
{
        if(turn != 2)
        {
                flag[2] = false;
                waiting[2] = true;

                while(turn != 2){}

                flag[2] = true;
                waiting[2] = false;
        }
}

ENTER_CS
{
        ....
        if(waiting[0] == true and waiting[1] == false)
        {
                turn = 0
        }
        else if(waiting[0] == false and waiting[1] == true)
        {
                turn = 1
        }
        else
        {
                // no one is waiting or both are waiting
                turn = randomly choose 0 or 1
        }

        flag[2] = false
}
```

b) This algorithm guarantees mutual exclusion because only one of three processors will be allowed to enter the critical section at any one time. This holds because if more than one processor is trying to enter the critical section they will enter the waiting-loop that waits on the turn variable to be set to the processor's id. All processors have to wait until the turn variable is set to their turn before they can enter the critical section. The inner while loop that waits on the turn variable guarantees mutual exclusion because the turn variable guarantees only one processor can enter the critical section at a time.

c) Freedom from deadlock is guaranteed in the given algorithm because no process can block any other process. If there are contending processes trying to execute, all but one of them will wait until it is given a turn to run. This prevents the possibility of a circular-wait from occurring and guarantees freedom from deadlock.

d) This algorithm is free from starvation because it guarantees that if a processor is waiting in the inner loop for a turn that it will eventually get selected to run. The shared *waiting* variable indicates which processors are waiting for a turn to run. At the end of the critical section, the *turn* variable is ensured to be set on a processor that is waiting if one of the processors is waiting.

Consider if the turn variable was instead incremented by 1 and modulo the number of processors to set the turn to the adjacent processor. This method would allow for starvation in the case where the only other processor waiting was not given the turn after the busy processor exited. The logic to choose which processor's turn that was added prevents this case and ensures that the next processor selected is indeed the waiting processor and not an idle processor when there is a processor waiting to be scheduled. Therefore, no processor will be prevented from running when it is waiting for a turn.

Question 2:

a) The 0-1 knapsack problem is solved using dynamic programming by building a table of all of the sub-solutions of the problem. All combinations of items for all possible weights are solved in the table. The benefit of storing these results in a table is that each solution is solved only once but potentially used more than once.

The algorithm to solve the sub-problems is informally described as:

1) If the weight of the knapsack is 0, no items can be taken and the value in the knapsack is 0.

[code: write 0 in the table for solution at this element for this case]

2) If no items are available, there can be nothing in the knapsack and the value in the knapsack is 0.

[code: write 0 in the table for solution at this element for this case]

3) If the weight of current item is greater than the total capacity of the knapsack, then the item cannot be included in the knapsack and the value in the knapsack is the same as the combination of items for the set of elements already solved not including the current item.

[code: copy value directly above current element in the table to current element for this case]

4) If the weight of the current is less than the capacity of the knapsack, decided between taking the item if the new value is greater than the previous solution without the item.

[code: copy the greater of the value directly above the current element in the table or the value of the current element plus the value of the element that is at the position on the row above the current element that is equal to the total weight of the knapsack minus the weight of the current element]

Therefore at each element in the table, the algorithm either writes 0, copies one element from the row above or sums two elements from the row above the current element. Subsequently, each operation in each row of the table is independent of each other and only dependent on the previous row in the table.

The parallel algorithm I implemented attempts to calculate each row on each value independently. The correctness of this algorithm is guaranteed due to the fact that each row is only dependent on the previous row (as described above) and the calculations within the same row are independent. The calculations on each row can therefore be performed in any order and simultaneously. There are no locks needed in this algorithm since there is no artifact from performing simultaneous reads and there are no simultaneous writes that occur when solving the solution in the parallel.

The performance of the parallel algorithm is the weight of the knapsack divided by the number of available processors (w/p) times the number of items available to be placed in the knapsack (n). This results in performance of $o(n * \frac{w}{p})$.

The performance of the sequential algorithm is the weight of the knapsack (w) times the number of available items (n). This results in performance of $o(n * w)$.

The speed-up of this algorithm is equal to:

S = $o\ (n * w)/\ o(n * w/p)$ = $o(p)$

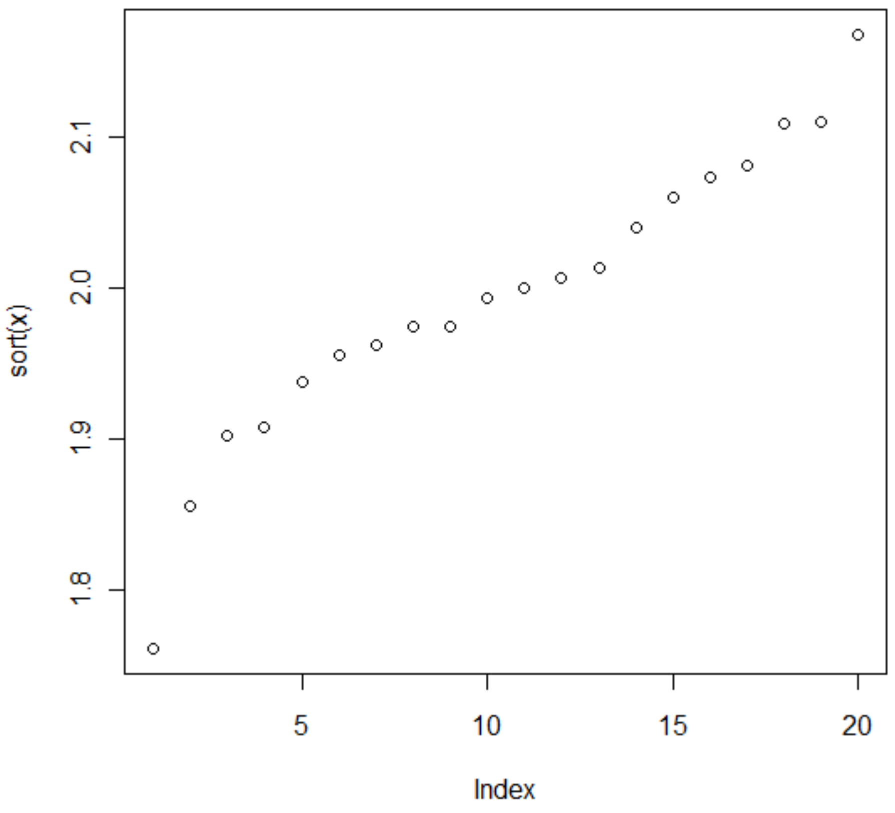
Note: I experimented with a different parallel approach that provided marginal speed-up (29%). The algorithm split the table in four quadrants. The quadrants were scheduled in the following order: upper-left, lower-left and upper-right in parallel, and lower-right. The individual tables were calculated using the above approach. It is possible that this approach could provide more speed-up than the current approach if a smaller block size is used.

b) I implemented the code for the sequential algorithm. I did use the Wikipedia articles mentioned in the assignment and the following YouTube videos to confirm the correctness of my implementation of the sequential solution.

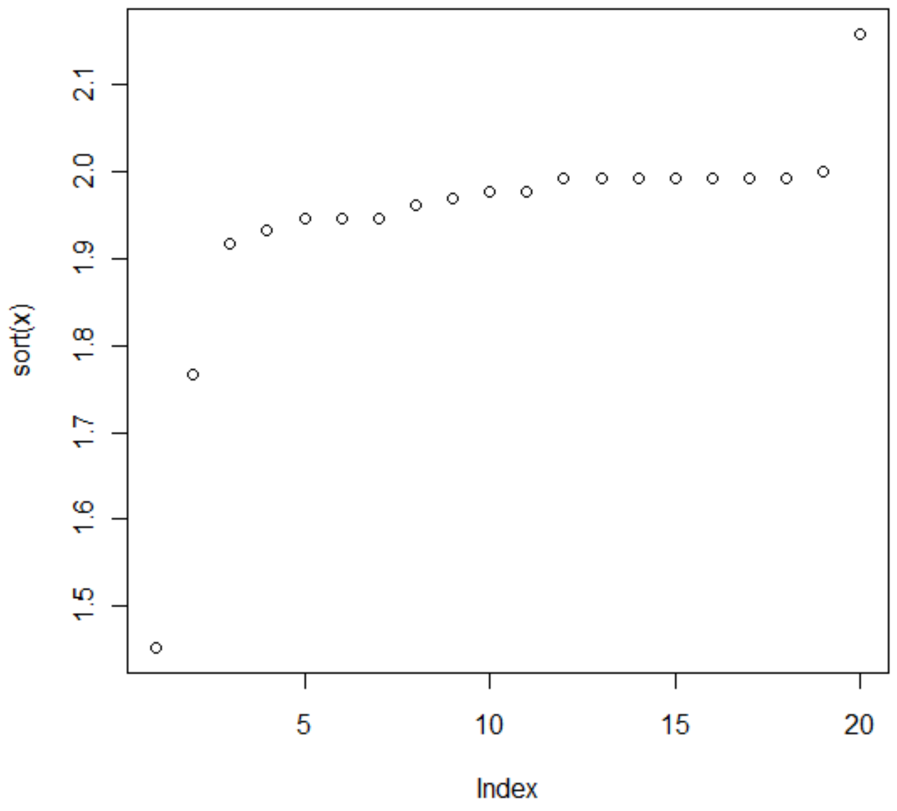
http://www.youtube.com/watch?v=hugQNiYoqUA
http://www.youtube.com/watch?v=WL6NETCcvkg
http://www.youtube.com/watch?v=Z9sMTFh5AXQ

c) The following speed-up's were found for the parallel algorithm using 1000 items with a knapsack capacity of 10,000. The Parallel.For method from the task parallel library was used to parallelize the calculation of the values for each row. Each run is the mean of 20 iterations and was found to be within 95% confidence as verified using the R software packaged. Raw iteration data can be found in the 'raw data' folder.

- **1.994256** speed-up
  - machine: eight core Intel Xenon 2.67GHz processors with 16GB of RAM
  - p-value < 2.2e-16 with 95 percent confidence interval: 1.949511 to 2.039001
  - plot of samples (x-axis: iteration, y-axis: speed-up)



- **1.944999** speed-up
  - machine: quad core IntelQ9400 2.67 GHz processor with 8 GB of RAM,
  - p-value < 2.2e-16 with 95 percent confidence interval: 1.882116 2.007883
  - plot of samples (x-axis: iteration, y-axis: speed-up)



The results between the two machines are comparable. This is due to the fact that the 8-core machine is a shared server and I do not have the ability to make sure my process is the only user code running. The quad core is not a server and I guaranteed that my code was the only process running on the machine.