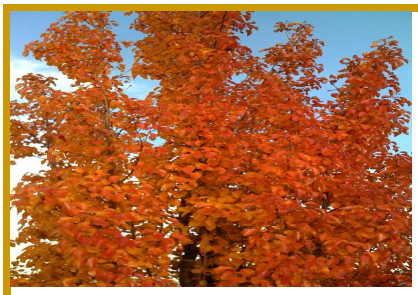

(../../index.php)

CS 475: Parallel Programming

Computer Science Department

(http://www.cs.colostate.edu)

Fall 2016

Assignment 3 - OpenMP Knapsack

Colorado State University

(http://welcome.colostate.edu/)

Objective

The objective of this assignment is for you to implement the Divide and Conquer Dynamic Programming algorithm for the 0/1 knapsack problem and parallelize it in OpenMP. The assignment has three parts. One of them is the sequential algorithm and the other two are different parallel implementations.
Work in an incremental fashion for debugging purposes. We will again be using the state capital machines. You must use these machines to measure and report performance. When studying performance, run your codes on a quiet capital machine.

- **knap1.c (ME KnapSack): knap1(name of the executable)**
  Start from the Full-table knapsack version (**knap.c**) provided in PA3.tar (PA3.tar).
  The names of input files are of the form knnxcc.txt, where nn represents N, the number of objects and cc represents the approximate total capacity. e.g. k120x4M.txt corresponds to input file for 120 objects and total capacity of ~4Million.

  The file format is:

  ```
  // first line: number of objects, knapsack capacity, // next lines: wei
  and profit of each object (1 object per line)
  ```

  For a simpler case use k10x25.txt (provided in the PA3.tar) to debug your code. It has a verbose option as the 3rd argument as seen below. knapSeq k10x25.txt should produce:

```
cs475@poudre [14:31] ~...PA3 11>./knapSeq k10x25.txt The number of objects
10, and the capacity is 25.  The optimal profit is 874 Time taken : 0.0000

cs475@poudre [14:31] ~...PA3 12>./knapSeq k10x25.txt 1 The number of objec
is 10, and the capacity is 25.  The optimal profit is 874 Time taken :
0.000003.  Solution vector is: 0 1 0 0 0 1 1 1 0 1

cs475@poudre [14:31] ~...PA3 13>./knapSeq k10x25.txt 2
The number of objects is 10, and the capacity is 25.
The optimal profit is 874
Time taken : 0.000002.
1       0 0 0 0 0 0 0 0 0 0
2       0 200 200 200 200 200 200 200 200 200
3       0 200 200 200 200 200 200 200 200 200
4       0 200 200 200 200 200 200 200 200 200
5       0 200 200 200 200 390 390 390 390 390
6       0 200 200 200 215 390 390 390 390 400
7       0 200 200 200 215 390 390 390 390 400
8       0 200 200 220 220 390 390 390 390 400
9       10 200 300 300 300 405 405 405 405 590
10      10 200 300 300 300 405 405 405 405 590
11      10 210 300 300 300 410 410 410 419 590
12      10 210 300 300 300 490 549 549 549 590
13      10 210 300 300 315 490 549 549 549 605
14      10 210 300 300 315 490 549 549 549 605
15      10 210 300 320 320 490 549 549 549 619
16      10 210 300 320 320 505 564 564 564 749
17      10 210 300 320 320 505 564 564 564 749
18      10 210 310 320 320 510 569 569 578 749
19      10 210 310 320 335 510 649 649 649 749
20      10 210 310 320 335 510 649 674 674 764
21      10 210 310 320 335 510 649 674 674 764
22      10 210 310 320 335 525 649 674 674 778
23      10 210 310 320 335 525 664 674 674 849
24      10 210 310 330 335 525 664 689 689 874
25      10 210 310 330 335 525 669 689 689 874
```

Modify the given code to implement the Memory Efficient Divide and Conquer algorithm explained in class.
We suggest the following steps:

- Create a separate function (`get_last_row`) to compute the last row of the given inputs from a start to an end point.
- Write a recursive function (`solve_kp`). This function will take as inputs, the starting and ending points and a capacity, C. It will divide the set of objects into two nearly equal parts, and call `get_last_row` for each part. This produces two rows of profits, A1 and A2. It will then calculate C*

based on these two rows. Do not forget to free the memory (if any) allocated for the calculation of rows.
    - Make two recursive calls to your `solve_kp` function (one for each half of the objects) with respective capacities based on the calculated value of C*.
    - Add the base case to the recursive function to update the solution vector.
    - Make the weights, profits and solution array as function parameter instead of a global array.

  You will have a recursive algorithm that will compute a series of values of C*. Since the solution to the knapsack problem may not be unique (multiple combinations of the objects may have the same profit) you need to be careful about how to resolve ties so that the answer you produce is the same as what we generate. Here are the rules to follow.

    - When partitioning the table into two nearly equal halves, when N is odd, divide the objects such that the first half is smaller, e.g., if N = 5, the first half should have 2 objects and second half should have 3.
    - Also, when you compute the value of C* as *argmin(MAX(A1[i]+A2[C-i]))*, make sure that all ties are resolved towards the smaller index, e.g., if A1[2]+A2[C-2] = 10 (and it happens to be the max) and A1[5]+A2[C-5] = 10, then C* should not be 5, but rather, 2 (provided no value of i smaller than 2 also gives 10).

  Keep the verbose options(default and 1) and the corresponding print statements similar to that of the given code.

- **knap2.c (Coarse-grain implementation): knap2(name of the executable)**
  Parallelize the implementation of knap1.c using OpenMP's coarse grain parallelization where large chunks of work will be done in parallel, but the chunks themselves are sequential, e.g., parallelize the complete **calls** to `solve_kp`. **Do not parallelize the for loops**. An additional option to think about is that in each call itself, there are two sub-tables to fill up, so they too can be executed as two separate parallel chunks. Therefore, we have three variations possible as follows:

    1. Parallelize only the recursive calls to `solve_kp`.
    2. Parallelize only the calls to `get_last_row`.
    3. Parallelize both recursive calls and get_last_row calls.

  Do not expect to see much improvement in performance as yet. Observe that these recursive calls build a tree. Introduce a new command line argument called `depth`. Modify your code such that at a recursion level that exceeds `depth` the code executes sequentially. A value of 0 for depth means that the entire tree is executed sequentially. In other words, allow parallelism only until certain depth in the tree.

- **knap3.c (Fine-grain implementation): knap3(name of the executable)**
  Parallelize the implementation of knap1.c using OpenMP's parallelization directives where "for loop" is executed in parallel. For this part, parallelize the

get_last_row function only. **Do not use any coarse grain parallelization as you did in knap2.c**.

- **knap4.c [Extra-Credit] (Combination of Coarse-grain and Fine-grain implementation) knap4(name of the executable)**
  Combine the parallelizations of knap2.c and knap3.c using OpenMP's parallelization directives to reflect both coarse-grain and fine-grain parallelism. For this part, parallelize both the get_last_row and solve_kp functions.
- **Report (see more details (PA3_Report.php))**
  All experiments are to be done on the capital machines in the CS department.

  You can see the Grading Rubric (PA3_grading_rubric.pdf) here.

  Your submission (PA3.1.tar and PA3.2.tar) must contain a Makefile, knap1.c, knap2.c, knap3.c and other extra-credit files if any. The Makefile should generate executables named knap1, knap2, knap3 respectively.
- **Preliminary Testing**
  5 graded tests and 1 Ungraded test.
  Ungraded : Un-Tar of your files.
  Test-1,2: Clean and Compile of your executable.
  Test-3,4,5: (note the verbose option = 1). Retain the verbose no-option and option-1 in your knap-1,2,3 version. For knap2.c, include depth as the 2nd command line argument and verbose as 3rd.

```
./knap1 k10x25.txt 1[verbose] | sed 's/Time .*$//'
./knap2 k10x25.txt 2[depth] 1[verbose] | sed 's/Time .*$//'
./knap3 k10x25.txt 1[verbose] | sed 's/Time .*$//'
```

  and the output of all the 3 should be

```
The number of objects is 10, and the capacity is 25.
The optimal profit is 874

Solution vector is: 0 1 0 0 0 1 1 1 0 1
```