Algorithms: Assignment #7

Due on April 30, 2016

Zhaoyang Li (2014013432)

Contents

Problem 1

CLRS Exercises 35.3-2, Show that the decision version of the set-covering problem is NP-complete by reducing it from the vertex-cover problem.

Part I SET-COVERING $\in$ NP.
Trivially, for an given instance $(X, \mathcal{F}, k)$ and its solution S, we can check if S is a subset of $\mathcal{F}$ consisting of less than k subsets of X, and whether S covers X, in polynomial time.
Part II Conversion
Given an instance of VERTEX-COVERING, say (V, E, j). Let $X = E, k = j$. For all vertexes $v_i \in V$, construct $E \supseteq E_i = \{e \in E : \exists v_j \in V s.t.(v_j, v_i) = e\}$. Let $\mathcal{F} = \{E_i\}$. Since $\mathcal{F}$ consists of subsets of E, the converted instance of SET-COVERING is valid. Such construction can be done in polynomial time.
PART III Proof of correctness
(a) An instance of VERTEX-COVERING is accepted **if** it's converted instance of SET-COVERING is accepted.
Suppose we have a solution S to $(X, (F), k)$, which covers X. $\forall S_i \in S$, there is a corresponding $v_i \in V$. Let the corresponding v_i's form a set V'. We know that $|S| = |V'| = k = j$. Since S covers X, it's obvious that V' covers (V, E).
(b) An instance of VERTEX-COVERING is accepted **only if** it's converted instance of SET-COVERING is accepted.
Suppose we have a solution S to (V, E, j). There exists a corresponding $C \subseteq 2^E$ for each $v \in S$. Since $j = k$, $|C| \leq k$. $\forall x \in X$, it corresponds to an $e \in E$. Since X is covered, E is covered.
In conclusion, SET-COVERING $\leq_P$ VERTEX-COVERING, and SET-COVERING $\in$ NP. Thus, SET-COVERING $\in$ NP-Complete.

Problem 2

CLRS Exercises 35.3-3,
Show how to implement GREEDY-SET-COVER in such a way that it runs in time $O\left(\sum_{S \in \mathcal{F}} |S|\right)$

As far as I can see, to get such a low time complexity, we have to perform some space-time tradeoff, meaning that we need some spacial data structure.
Let's maintain an table T_1 of all the elements $S_i \in \mathcal{F}$, keeping them sorted by a $s_i = |\{x \in S_i : \text{not covered yet}\}|$.
Each time, we choose $S_j \in \mathcal{F}$ from T_1 with the maximum s_j, update all the s_i's in T_1 (where "update" is always "decrease"), and repeat until $\forall j, s_j = 0$.
T_1 is supposed to support DECREASE-KEY and FIND-MAX efficiently. I believe we can find some data structure to meet the requirements. Heaps, for instance.
Now, in order to update our s_i's efficiently, we have to keep track of all $x_k \in X$: we need to know all the S_i's x_k is in, all the s_i's in which x_k counted. That requires tables mapping from k to lists of S_i. In order to make locating quicker, we may want to add more internal links between all the tables.
Now let's describe our algorithm informally:

- Create T_1 mapping from i to decreasingly-sorted integers, initially $|S_i|$. Create T_2 mapping from i to $\{j : x_i \in S_j\}$. Let $C = \emptyset$.
- Repeat until $\forall j, T_2(j) = 0$: choose the maximum item in T_1 as t. Let $C = C \cup \{t\}$. For all $x \in t$, for all $y \in T_2(x)$, $T_2(x) = T_2(x) \setminus \{y\}$, decrease the corresponding $T_1(y)$.
- Return with C.

Problem 3

CLRS Problems 27-2,
Saving temporary space in matrix multiplication

a

Modifications on the original P-MATRIX-MULTIPLY-RECURSIVE are listed as follows.

- add an initialization $\forall i, j, c_{ij} = 0$ outside, in parallel, in $\Theta(\lg n)$
- add a **sync** statement after the 4th **spawn**
- modify line 3 into $c_{11} = c_{11} + a_{11} \cdot b_{11}$

$T_\infty(n) = 2T_\infty(n/2) + \Theta(\lg n) = \Theta(n)$

b

Work: $T_1 = \Theta(n^3)$. Span: $T_\infty = \Theta(n)$.

c

Parallelism: $T_1/T_\infty = \Theta(n^2) = 1000^2 = 10^6$.
Though it's 10 times less than the original, since most parallel computers still have far fewer than 1 million processors, it's reasonable to conclude that there is no significant difference.

Problem 4

Implement multithreading versions of merge sort and quick sort with OpenMP.

Implementation and environment

Language The C++ programming language

IDE Visual Studio 2015

Compiling MSVC2015 64bit Debug[1] with OpenMP support

OS Windows 10 Education, Build 10586

Hardware Intel Core i7-6600U @ 2.60GHz (2 Cores), RAM 16GB LPDDR3, Model ThinkPad X1 Carbon 20FB

Verification of correctness

Correctness is verified by comparing results given respectively by the algorithms, on randomly generated arrays. 100% correctness.

[1] Release with optimization disabled gives similar results.

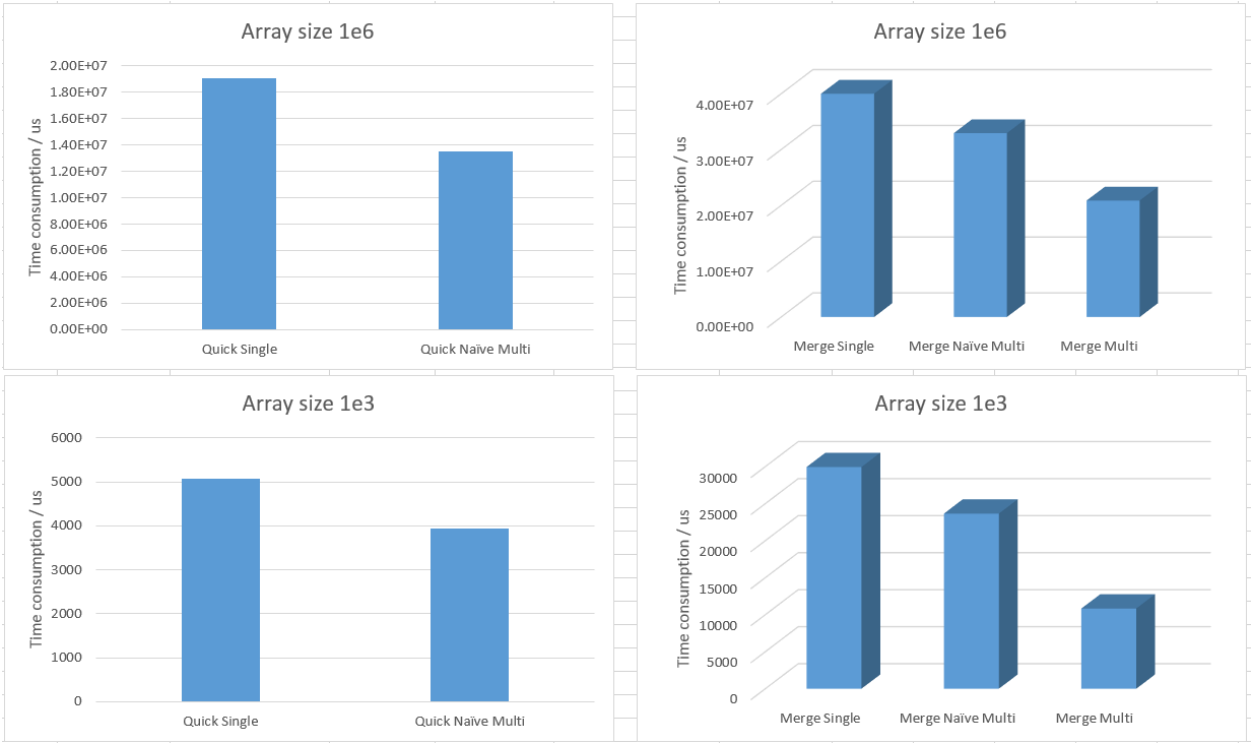
Problem 4 continued on next page...

Timing

Each number in the table below is an average over several different randomly generated arrays.

milliseconds	Quick Single	Quick Naive Multi	Merge Single	Merge Naive Multi	Merge Multi
Array size 1e6	1.90E+07	1.35E+07	4.16E+07	3.29E+07	2.09E+07
Array size 1e3	5081.22	3929.41	29941.2	23651.2	10828

Illustrated as follows.



It can be concluded that multi-threading merge sort saves about 50% of the running time, as is expected on a 2-core CPU. Naive multi-threading algorithms save 20% to 30%.

Problem 5

Implement GPU version of matrix multiplication with OpenCL or CUDA.

Implementation and environment

Language The C++ programming language and the OpenCL C Language

IDE Visual Studio 2015

Compiling MSVC2015 64bit Release, Intel SDK for OpenCL Applications 6.0

OS Windows 10 Education, Build 10586

Hardware Intel Core i7-6600U @ 2.60GHz, GPU Intel HD Graphics 530, RAM 16GB LPDDR3, Model ThinkPad X1 Carbon 20FB

Problem 5 continued on next page...

Verification of correctness

Correctness is verified by comparing results given respectively by the two implementations, on randomly generated matrices. 100% correctness.

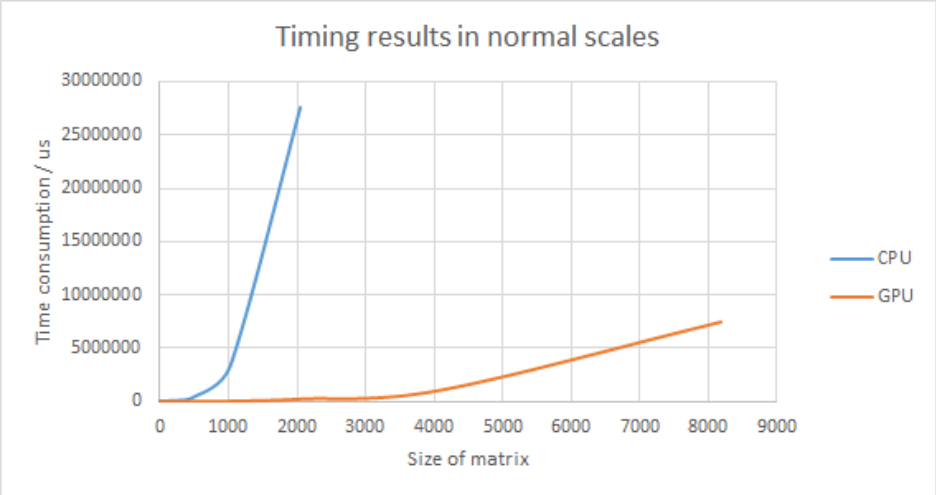
Timing

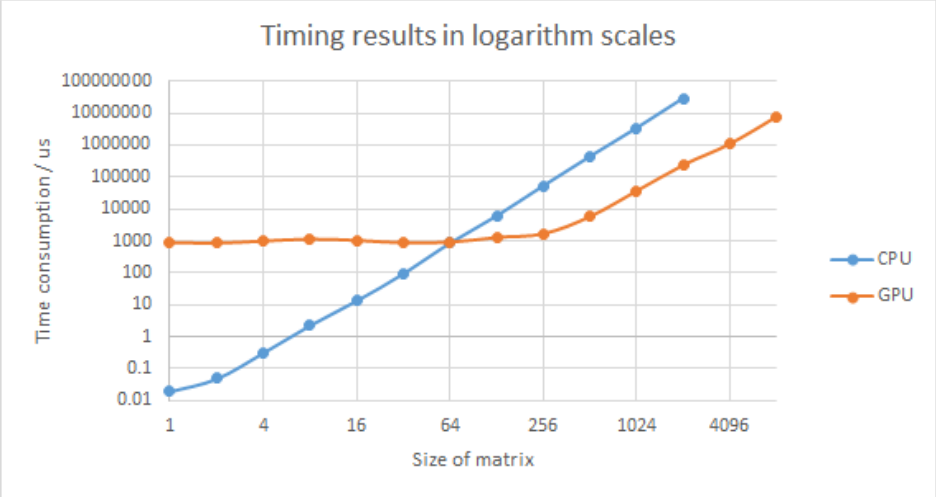
Each number in the table below is an average over several different randomly generated matrices.

size	CPU/us	GPU/us
1	0.02	915.825378
2	0.05	867.252197
4	0.32	1014.651611
8	2.29	1159.98291
16	13.44	1057.659912
32	94	891.381287
64	850	938.103577
128	6000	1313.00354
256	52000	1707.822388
512	427000	5865.757813
1024	3.28E+06	36941.33203
2048	2.76E+07	236610.8594
4096		1075285.25
8192		7530680.5

Illustrated as follows.

Problem 5 continued on next page...





It can be concluded that for large matrices, GPU calculation is about 100 times faster. As the two lines on the log-log plot looks parallel with each other, we know that growth rate of running time on CPU and GPU are basically the same, which is $\Theta(n^3)$.

It's not expected that for small matrices, GPU has a fixed running time, regardless of the exact size of the matrix. I didn't look into details; a reasonable guess is that time spent on scheduling outweighs.

(End of all.)