Problem 1. (16%)

1. if $P = NP$, then $NPC \subseteq P$.

   True, If $P = NP$, then for any language $L \in NPC$ (1) $L \in NPC$ (2) L is NP-hard.

   By the first condition, $L \in NPC \subseteq NP = P \Rightarrow NPC \subseteq P$

2. if $NPC \subseteq P$, then $P = NP$.

   True. All the NP problem can be reduced to arbitrary NPC problem in polynomial time, and NPC problems can be solved in polynomial time because $NPC \subseteq P$. $\Rightarrow NP$ problem solvable in polynomial time $\Rightarrow NP \subseteq P$ and trivially $P \subseteq NP$

   $\Rightarrow NP = P$

3. When performing an amortized analysis with the potential method, we usually want the potential after any given number of operations to be larger than or equal to the initial potential.

   True. Because we want the amortized cost to be an upper bound of actual cost. or you can write down the cost function to reveal the relationship. $\sum\limits_{i=1}^{n} \hat{c}_i = \sum\limits_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0) \geq \sum\limits_{i=1}^{n} c_i$ $\Rightarrow \phi(D_n) \geq \phi(D_0), \forall n$

4. Let $P_1 = \{L \subseteq \{0,1\}^*$: there exists an algorithm A that **decides** L in polynomial time $\}$ and $P_2 = \{L \subseteq \{0,1\}^*$: there exists an algorithm A that **rejects** L in polynomial time $\}$. Then $P_1 = P_2$.

   True. $P_1 \subseteq P_2$ trivially.

   So we only need to consider $P_2 \subseteq P_1$

   $\Rightarrow \forall L$ in P_2, there exists an algorithm A that **rejects** L in polynomial time.

   $\Rightarrow$ there exist some c,n_0, such that A rejects L takes at most cn^k steps for all $n \geq n_0$

   $\Rightarrow$ then we can construct A_0, if A rejects L in cn^k steps, then **rejects** else **accept** which runs also at most $O(n^k)$ times, so it can decide L in polynomial time.

   $\Rightarrow P_2 \subseteq P_1$

   $\Rightarrow P_2 = P_1$

Problem 2. (29%)

1. (6%)

   (3%) race condition 所造成的 bug 不一定每次執行時都會發生，可能每次的執行結果會不一樣，所以難以發現 bug，也難以 debug。(common mistake: race condition 造成的 bug 也可能很頻繁的發生，端看實際例子。所以如果以「肯定的語氣」寫「因為在大部分的情況下執行結果都會正確，只有在很少很少的機率才會發生錯誤」當理由，酌扣一分。)

   (3%) 只將平行運用在獨立的運算上，也就是說，只運用在彼此之間沒有關聯性的運算上；注意變數和記憶體的使用。

2. (4%)

   1) 將 tasks 分成少於 18 hrs 的小 task

   2) 將每個工程師曾經做過的 task 的「預估完成時間」和「實際完成時間」標記在統計表上，把「預估完成時間」和「實際完成時間」相除就能得到「velocity」，每一位工程師都會有許多 velocity 的歷史紀錄。

   接下來才是 Monte Carlo 的部分。

   3) 使用 Monte Carlo 來模擬各種可能的未來情況。

   總共跑 100 種未來情況 (scenarios)，每種情況跑出來的結果都 assign 1% 的機率。

   每一種未來情況的預估完成時間是這樣計算的：randomly 從該工程師曾有的 velocity 中挑出一個，以之除他的預估時間，這樣算出每個小 task 的預估時間後 (每個小 task 挑的 velocity 都不一樣)，再相加，得到的就會是整個 project(大 task) 的預估時間。

   最後得到的結果可以做成圖表，例如：25hrs 有 1% 的機率、35hrs 有 5% 的機率 (代表有 5 個情況跑出來都是 35hrs)、80hrs 有 30% 的機率，...。

   common mistake：

   Monte Carlo 最重要的就是跑 n 種情況，每種情況都使用 random 的方法跑出結果。所以如果沒寫到 scenario、random 之類的關鍵字，就會酌扣分數。

3. (9%)

(3%) The work $= T_1$

$= \text{sum}(\ T_A + T_B + T_C + T_{spawnForParallel} + (T_{D_1} + T_{D_2} + ... + T_{D_n})\ )$

$= \text{sum}(\ O(\log n) + O(\log n) + O(\log n) + \Theta(\log n) + \frac{(1+O(n))\cdot n}{2}\ )$

$= O(n^2)$

(3%) The span $= T_\infty$

$= \max(\ T_A\ , T_B\ , T_C\ , T_{spawnForParallel} + \max(T_{D_1}\ , T_{D_2}\ , ...\ , T_{D_n})\ )$

$= \max(\ O(\log n)\ , O(\log n)\ , O(\log n)\ , \Theta(\log n) + O(n)\ )$

$= O(n)$

(3%) The parallelism $= \frac{work}{span} = \frac{T_1}{T_\infty} = \frac{O(n^2)}{O(n)}$

But we can't use $O(n)$ as a divisor, so we need to consider the *span* and the *work* in cases.

By equations above, we can conclude that

the $work = \Theta(\log n + n \cdot T_{D_n})$, the $span = \Theta(\log n + T_{D_n})$

If $T_{D_n} = \Omega(\log n)$, then $work = \Theta(n \cdot T_{D_n})$, $span = \Theta(T_{D_n})$, parallelism $= \frac{\Theta(n \cdot T_{D_n})}{\Theta(T_{D_n})} = \Theta(n)$

Else if $T_{D_n} = O(\log n)$, then $work = O(n \log n)$, $span = \Theta(\log n)$, parallelism $= \frac{O(n \log n)}{\Theta(\log n)} = O(n)$

Therefore the answer to this question is $O(n)$.

(You don't have to do this complicated process to get full credits. You can get credits as long as you divide *work* by *span* and derive an anser like $O(n)$ or $\Theta(n)$)

4. (4%)

(2%) advantage: 方便快速地就能做出 prototype，節省時間以及金錢。在開發團隊之間要互相溝通、或者測試使用性、測試設計時，都可以用到。

(2%) disadvantage: 與實際的成品可能相差比較大、測試者較難真正的感受使用介面。

5. (6%)

(3%)

parallelism $\rho = \frac{work}{span} = \frac{T_1}{T_\infty}$

1) 平均可在 critical path 上 (computation dag 的最長路徑上) 每一個步驟平行處理的工作量

2) parallelism 是對 perfect linear speedup 可能性的限制

3) parallelism 是可能得到的最大 speedup

(3%)

number of processors $P > \rho$ 代表離 perfect linear speedup 越來越遠，沒辦法將每個 processor 充分平行利用

***Problem* 3.** (28%)

1. This problem is equivalent to check whether one graph is a bipartite graph or not, while two nodes has a undirected edge iff the corresponding APs interfere each other. The pseudo code is shown below.

```
def solve(N = APs, E = interfering relations)
    color[N] = 0 // 0 for channel 6 and 1 for channel 11
    for i in 1 to N
        if color[i] != 0 continue
        color[i] = 1
        q = queue(i)
        while q not empty
            x = q.pop()
            for y in E[x] // APs that may interfere with x
                if color[y] != 0
                    if color[y] == color[x] // interfering happens
                        return false
                    else continue
                else
                    color[y] = !color[x]
                    q.push(y)
    return true
```

   Common mistake: the graph formed by APs and interfering relations may have more than one component (APs that are far enough so that no interfering), thus it's necessary to travel all vertices to make sure that all of them is colored (assigned channel).

2. Each vertex will be colored at most one time, then each edge will be visited at most 2 times, so the time complexity is $O(|V| + |E|)$, and since that there are at most $|V| * (|V| - 1)$ edges, the time complexity can be represent in simple form $O(|V|^2)$.

3. (a) Create 3 nodes and 3 edges $\Rightarrow O(3 + 3) = O(1)$
   (b) Create 2n nodes for variables and their negations, n edges for variables with theirs negations, and 2n edges for variables and theirs negations connected with node 'Other' $\Rightarrow O(2n + n + 2n) = O(n)$
   (c) For each clause create 5 nodes, 5 edges for inner connection, 2 edges linked with node 'True', and 3 edges linked with literals' node $\Rightarrow m\ clauses * O(5 + 5 + 2 + 3) = O(m)$

   Totally $O(1 + n + m)$ = polynomial time to do the reduction.

4. Since the APs created in step 2 are all linked to the node 'Other', so they can only in state True or False, and because of the edges between x_i and $\neg x_i$, we can treat the states of these APs are equivalent to the variables in 3-CNF-SAT problem.
   Consider the all possible states combination of the three APs related to one clause, we can find that if all of them is False, the left upper AP in that clause must also be False, which implies the right upper AP must be Other, which implies that the right lower AP must be False, but there is a link between the right lower AP and one of related AP which is also False, a contradiction.
   If not all of them is False, we can simply construct a valid assignment by hands for all possible seven cases, here is the list of them shown from upper left to lower right: TOFOTTTT, TFFOOTTF, TFFOOTFT, TFFOOTFF, TFOFOFTT, TFOFOFTF, FOTOFFFT. Thus each clause is satisfiable iff there are at least one True state in the corresponding three APs x_1, x_2, and x_3, which equivalent to (x_1, x_2, x_3) must be True in 3-CNF-SAT problem.
   So each valid channel assignment will map to a valid 3-CNF-SAT solution, and vice versa.

   * x_i and $\neg x_i$ have edge and are linked to node 'Other', so exactly one of two is assigned True and the other is False, thus no contradiction.
   * If the corresponding APs problems has solutions, in each clause at least one literal is assigned True, so ϕ is satisfiable.

   Since 3-CNF-SAT is a well known NP-Complete problem, and we can solve 3-CNF-SAT by polynomial reduction to 3-channel APs problem, this 3-channel APs problem is in NP-Hard.

5. It is easy to see that we can encode the assignment into a certificate y, and make a simple algorithm A(x, y) where x is the encoded graph of APs, it simply to validate whether all vertices have different channel with their adjacencies in $O(|V| * (|V| - 1)) = O(|V|^2)$.
   So we obtain a polynomial time verify algorithm, thus this problem $\in$ NP.
   In addition, from 4. we know that this problem $\in$ NP-Hard.
   As the result, this problem $\in$ NP-Complete.

***Problem* 4.** (?%)

```
Y  = MAT-VEC(A,X)
    n = A.rows
    let Y be a new vector of length n
    parallel for i from 1 to n do   //Loop1
        Y(i) = 0
    parallel for i from 1 to n do //Loop2
        Y(i) = INNER-PROUCT(A, X, i, 1, n)
    return Y

INNER-PRODUCT(A,x, i, l, r)
    if l == r do
        reutrn A(i, l) * x(l)
    else do
        mid = floor((l+r)/2)
        spawn left = INNER-PRODUCT(A, X, i, l, mid)
        right = INNER-PRODUCT(A, X, i, mid+1, r)
        sync
        return left + right
```

Analysis:

Work:

Loop1: $\theta(n)$

Loop2: $\theta(n)$ * time of INNER-PRODUCT

INNER-PRODUCT: $T(n) = 2T(\frac{n}{2}) + \theta(1) = \theta(n)$ By Master Theorem

So the total work is Loop1 + Loop2 = $\theta(n) + \theta(n) * \theta(n) = \theta(n^2)$

Span:

Loop1 : $\theta(logn)$

Loop2 : $\theta(logn)$ * span of INNER-PRODUCT

INNER-PRODUCT; $T(n) = T(\frac{n}{2}) + \theta(1) = \theta(logn)$ By Master Theorem

So the total span is $\theta(logn)$

Parallism:

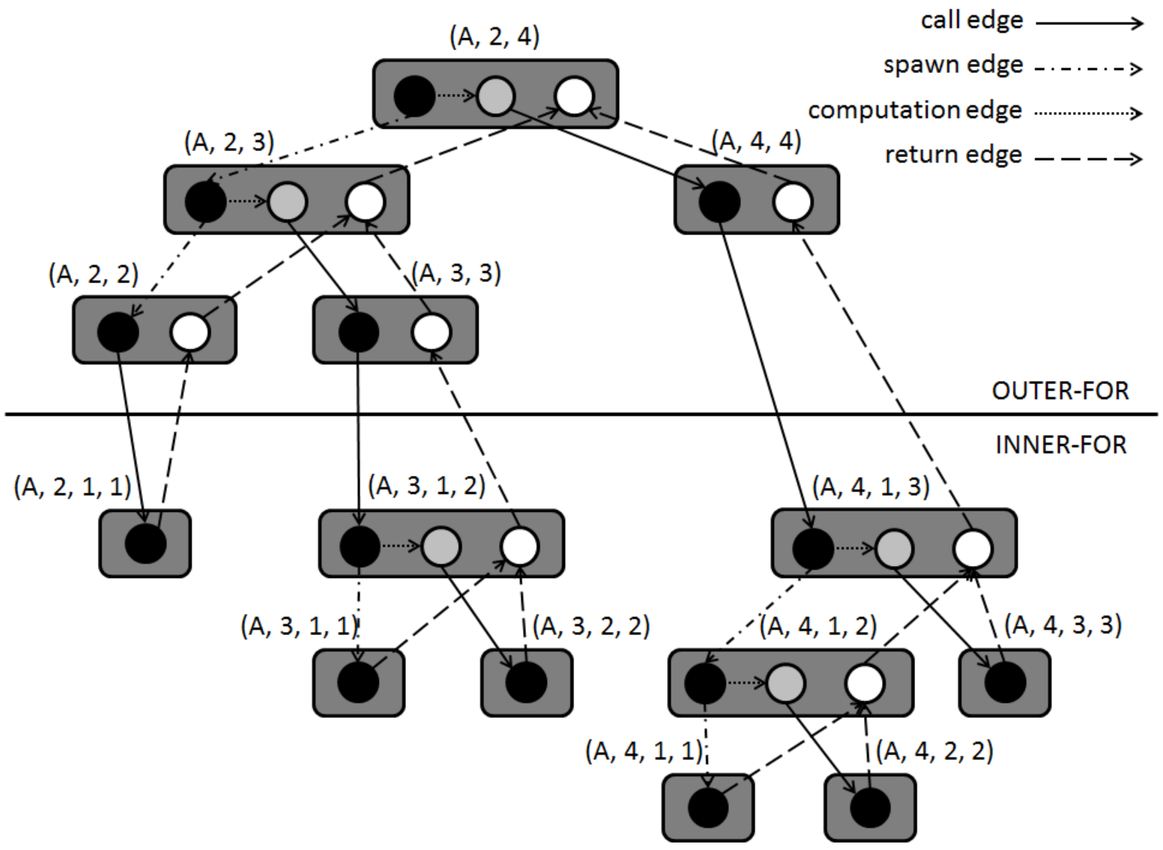
Work/Span = $\theta(\frac{n^2}{logn})$

Problem 5. (24%)

1. The compiler would produce two functions below to implement the parallel for.

```
OUTER-FOR(A, j, j')
    if j == j'
        INNER-FOR(A, j, 1, j - 1)
    else
        mid = floor((j + j') / 2)
        spawn OUTER-FOR(A, j, mid)
        OUTER-FOR(A, mid + 1, j')
        sync

INNER-FOR(A, j, i, i')
    if i == i'
        exchange A(i, j) with A(j, i)
    else
        mid = floor((i + i') / 2)
        spawn INNER-FOR(A, j, i, mid)
        INNER-FOR(A, j, mid + 1, i')
        sync
```



2. work of inner for: $T_1^I(n) = \Theta(n)$

   work: $T_1(n) = \Theta(n) + \sum_{i=1}^{n-1} T_1^I(i) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$

   span of inner for: $T_\infty^I(n) = \Theta(\log n) + \max_{1\le i\le n} iter_\infty(i) = \Theta(\log n) + \Theta(1) = \Theta(\log n)$

   span: $T_\infty(n) = \Theta(\log n) + \max_{2\le i\le n} iter_\infty(i)$

   $= \Theta(\log n) + \max_{2\le i\le n} T_\infty^I(i-1)$

   $= \Theta(\log n) + \Theta(\log n)$

   $= \Theta(\log n)$

   parallelism$= \frac{T_1}{T_\infty} = \frac{\Theta(n^2)}{\Theta(\log n)} = \Theta(\frac{n^2}{\log n})$

3. Work would remain the same. $T_1(n) = \Theta(n^2)$

   span of inner for: $T_\infty^I(n) = \Theta(n)$

   span: $T_\infty(n) = \Theta(\log n) + \max_{2\le i\le n} iter_\infty(i)$

   $= \Theta(\log n) + \max_{2\le i\le n} T_\infty^I(i-1)$

   $= \Theta(\log n) + \Theta(n)$

   $= \Theta(n)$

   parallelism$= \frac{T_1}{T_\infty} = \frac{\Theta(n^2)}{\Theta(n)} = \Theta(n)$

Problem 6. (12%)

Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(\log n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\log n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works: (Hint: one way to formulate the potential function involves the depths of the nodes in the heap. Think of a way to combine them.) (12 points, 4 points each)

1. Define your potential function Φ. Prove that it always satisfies $\Phi(D_i) \geq \Phi(D_0), \forall i$, where D_i is the data structure after performing the i-th operation.
2. Show that the amortized cost of INSERT is $O(\log n)$.
3. Show that the amortized cost of EXTRACT-MIN is $O(1)$.

Sol:

1. Denote n_i as the number of elements in D_i and $d_i(x)$ as the depth of x in D_i. Suppose that each INSERT or EXTRACT-MIN operation takes at most $k \log n$ time, where k is a contanst > 0. Define $\Phi(D_i) = k \sum_{x \in D_i} d_i(x)$. Initially the heap has no items, so $\Phi(D_0) = 0$, and thus $\Phi(D_i) \geq \Phi(D_0)$, for all i.

2. Suppose that the i-th operation is an INSERT, then

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \log n_i + k \sum_{x \in D_i} d_i(x) - k \sum_{x \in D_{i-1}} d_{i-1}(x) \\
&= k \log n_i + k \lceil \log(n_i + 1) \rceil \\
&= O(\log n).
\end{aligned}$$

3. Suppose that the i-th operation is an EXTRACT-MIN, then

$$\begin{aligned}
\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \log n_i + k \sum_{x \in D_i} d_i(x) - k \sum_{x \in D_{i-1}} d_{i-1}(x) \\
&= k \log n_i - k \lceil \log(n_i + 1) \rceil \\
&= O(1).
\end{aligned}$$