Question 1

(a) [5 marks]

EREW stands for exclusive read exclusive write, which means each memory address can be read or written to by only one processor at a time. CREW stands for concurrent read exclusive write which means that a memory address can be read by several processes simultaneously, but can be written to by only one processor at a time.

For example if the problem is to fill an array of n elements with given value, it can be solved in constant time on n processor CREW PRAM, since all of processors can read the given value simultaneously. On EREW PRAM this problem would take long time to solve.

(b)

i. [5 marks]

<TODO: write explanation>

```
for all objects in parallel {
this.jump = this.next; // copy the list

while (this.jump != NULL) {
this.temp = this.d; // this.temp is necessary since this is EREW PRAM
this.d = MAX(this.d, (this.jump).temp);
this.jump = (this.jump).jump; // move on
}
}
```

For the given algorithm we assume PRAM synchronization between statements of a loop. Asymptotic run time is Θ(log n).

ii. [5 marks]

Since each element of the array is directly accessible, we don't need to use pointer jumping, but we can borrow the main idea behind it. We can calculate the index of the next element of the array we will compare our value with and then do the comparison. Asymptotic run time will stay the same - Θ(log n). <TODO: write WHY?>

My 2 cents: I think the connection between pointers and arrays needs to be made clearer. You can jump to any index p in an int array A by A[0] + (p*sizeof(int)) (or words to that effect).

(c) [10 marks]

As the first step of the algorithm we will create an integer array of the same size *n* as the given array. Characters 'x', '(', ')' will be presented as 0, 1, -1 respectively. This conversion can be done in constant time given the fact that the number of processors is equal to number of array elements. In the next step we'll need to calculate the prefix sum for the given array and store it in a different array *b*, which can be done in logn time.
If the input character string is correctly parenthesized, then the last element of the array *b* will be equal to 0 and all other elements will be more than or equal to 0. If the array *b* will contain negative value, this would mean that the input string is not correctly parenthesized. The described check can be performed in constant time, by *i*-th processor checking the value of the array element b[i]. Can use MIN-reduction to get smallest value from all processors; then check (on single processor) that it is not less than 0.
The total asymptotic run time of the algorithm will be Θ(log n), caused by a prefix calculation time bottleneck. The cost will be Θ(nlog n).
The best sequential algorithm can solve this problem in Θ(n) time. Description of the algorithm: After converting the input n-character string into an integer array as described earlier, the algorithm will loop through the elements of the array, summing them and storing the sum in a variable *x*. If while performing the summation of the elements *x* will get negative or if *x* will not be equal to 0 at the end, this will mean that the input character string is not correctly parenthesized. Since the cost of the parallel algorithm is not equal to the run time of the sequential algorithm, our parallel algorithm is not cost-optimal.

Question 2

(a) [6 marks]

i. Processors adjacent to processor 9(1001) in hypercube are processors 1(0001), 8(1000), 11(1011) and 13(1101). Binary representations of adjacent processor indices in a binary hypercube differ only in one bit.

ii. Processors 5, 8, 10 and 13 are adjacent to processor 9 in the mesh with row-major indexing.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

iii. Processors 6, 8, 10 and 14 are adjacent to processor 9 in the mesh with snakelike row-major indexing.

0	1	2	3
7	6	5	4
8	9	10	11
15	14	13	12

(b)
i. [7 marks]

ii. [7 marks]

Question 3

a) compare-exchange constitutes of two adjacent processors P1 and P2 exchanging their values. P1 stores min(P1, P2), P2 stores max(P1,P2).
With compare-split each processors has a block of n/p elements. Adjacent processors P1 and P2 exchange their blocks and merge them. P1 keeps the lower
half, P2 - the higher. The optimal solution for n/p elements per processor is to sequentially sort each block n/p and use a series of compare-splits to
merge them together.

b) We partition 4 items per processor.

processor step	P1	P2	P3	P4
initial data	7 3 14 4	1 8 3 5	4 7 2 8	13 1 10 2
1	3 4 7 14	1 3 5 8	2 4 7 8	1 2 10 13
2	1 3 3 4	5 7 8 14	1 2 2 4	7 8 10 13
3	1 3 3 4	1 2 2 4	5 7 8 14	7 8 10 13
4	1 1 2 2	3 3 4 4	5 7 7 8	8 10 13 14
5	1 1 2 2	3 3 4 4	5 7 7 8	8 10 13 14

Step 1 we sequentially sort each 4 items in each processor.
Step 2 - odd step, we do odd-even compare splits (P1 <-> P2 and P3 <-> P4)
Step 3 - even step, even-odd compare split (P2 <-> P3)
Step 4 - odd step, we do odd-even compare splits (P1 <-> P2 and P3 <-> P4)
Step 5 - even step, even-odd compare split (P2 <-> P3)

c) Variant of PRAM doesn't matter since no two processes are ever reading/writing the same values.

d)

input 1	⊕	⊕	⊕
input 2	⊕	⊕	⊕
input 3	⊖	⊕	⊕
input 4	⊖	⊕	⊕

First column - create a bitonic sequence from the input array
Second columnt - input1 ⊕ input3 and input2 ⊕ input4
Third columnt - input1 ⊕ input2 and input3 ⊕ input4

e) Sorting networks consist of a preset collection of comparators. For CREW Quicksort we chose random pivots at which we separate the array into a tree. Because of this it is not possible to fit these random selected pivots into a sorting network.
CREW Merge sort anyone?