1) A) The algorithm is work-efficient (work-optimal) and so the algorithm satisfies $W(n) = \theta(T^*(n)) = O(T^*(n))$. We have $W(n) = O(n \log n)$ and $T(n) = O(\log^2 n)$ and so $T^*(n) = O(n \log n)$. A work-optimal PRAM algorithm on a p-processor system would take $T_p(n) = O((T^*(n)/p)+T(n))$. We have many choices here. We can choose $p = n$ and that way we would obtain $T_p(n) = O(\log n + \log^2 n) = O(\log^2 n)$. However, we can choose the smallest value of p to get optimal time and that value is $n/\log(n)$. When $p = n/\log(n)$, we obtain $T_p(n) = O(\log^2 n + \log^2 n) = O(\log^2 n)$. The minimum value of p for optimal speedup is $n/\log(n)$. As long as $p > (n/\log(n))$, we have optimal speedup.

   B) We assume that the critical and atomic regions contain regular (non-fp) additions

   1. OpenMP atomic pragma: This is the fastest of all operations. The purpose of atomic pragma is similar to critical region. It only allows one thread to perform operations in its region at a time. However, it has less overhead compared to critical region but less functionality.

   2. OpenMP critical region: In this region, only one thread is allowed to run at a time. Considering that our operations being performed in the critical region and atomic pragma are non-floating point additions, we can rank them as less expensive compared to the other operations. We have to keep in mind however, that threads perform context switches adding to the overhead.

   3. A floating point add: Although this is a FP operation, it is definitely less costly compared to a multiplication (division). A multiplication may require many adders but depending on the architecture, it might be less costly than a FP add (if core has multipliers commonly as seen in GPUs). But in general, a floating point add is faster than a floating point divide.

   4. A floating point divide: A floating point divide is very costly. It is similar to a fp multiply but even more costly as it is the least supported operation by hardware. It is definitely more expensive compared to an fp add but should be slower on average compared to an exp() function. This depends on the value of x, but the performance with exp() should degrade exponentially.

   5. exp(x): Exponential is basically multiple multiplies and each multiply is multiple additions. This is the costliest of operations depending on x.

2)

Binary summation:

N-> P|	8	128	2048	32768	131072	1048526
1	0.00028	0.00038	0.000196	0.003556	0.015845	0.187947
2	0.009483	0.00094	0.000268	0.003448	0.014667	0.159981
4	0.001073	0.0001488	0.000297	0.003310	0.015617	0.142618
8	0.083227	0.000808	0.000461	0.008338	0.017302	0.129882
16	0.000608	0.000907	0.001315	0.004865	0.031346	0.140117
32	0.01167	0.001693	0.002451	0.006390	0.102656	0.138619
64	0.002915	0.003699	0.004722	0.008404	0.22365	0.189091
128	0.004228	0.010467	0.008339	0.014276	0.065036	0.194300

Partition summation:

N-> P|	8	128	2048	32768	131072	1048526
1	0.00028	0.00013	0.00016	0.000169	0.000607	0.004620
2	0.009483	0.00055	0.004856	0.000198	0.000687	0.004184
4	0.001073	0.003996	0.000105	0.000281	0.000591	0.003964
8	0.083227	0.000393	0.002992	0.12617	0.015578	0.011531
16	0.000608	0.000408	0.000561	0.000613	0.000981	0.004800
32	0.01167	0.000735	0.000796	0.056218	0.022839	0.004187
64	0.002915	0.001582	0.001506	0.001590	0.001937	0.073312
128	0.004228	0.007319	0.003210	0.003029	0.003618	0.008473

I did not include graphs as they were not good indicators for patterns. (they were jumbled up.

From the data we notice that performance increases when the number of threads increase. However, we then reach a maximum of performance from which performance starts to decrease. This is fairly evident as described by Amdahl's law and Brent’s theorem. Amdahl’s law indicates that no matter how much optimization you make for a certain program, overall performance is affected by only the fraction of time the part being optimized takes from the overall program. And so, a performance increase in one aspect of the program is limited. Brent’s theorem explains how a program needs at least a certain amount of time to complete no matter how good optimizations are. This is shown in our results when having around 16 to 32 threads. Performance reaches a maximum and then starts increasing. It start increasing because the number of context switches between threads is so high that performance benefits are overshadowed by overhead and the program actually becomes slower.