MA398 Matrix Analysis and Algorithms: Exercise Sheet 5

1. Consider an algorithm that performs a sorting operation on an array of size n. This algorithm has a cost of $C(n) = n \log n$ operations. As $n \to \infty$, what can be inferred about the time complexity of this algorithm? Furthermore, if this sorting algorithm is to be run on a parallel computer architecture, what considerations might come into play in terms of the computational cost?

   Answer: An algorithm with a cost of $C(n) = n \log n$ operations is considered to have a time complexity of $O(n \log n)$. This indicates that the time it takes for the algorithm to run increases logarithmically with the size of the input. As $n \to \infty$, the time it takes for the algorithm to run will increase, but not as dramatically as in an algorithm with a time complexity of, for example, $O(n^2)$ or $O(n^3)$.

   In terms of a parallel computer architecture, there are additional considerations that might come into play in terms of the computational cost. One of the main benefits of parallel computing is that it allows for the simultaneous execution of multiple tasks, which can potentially lead to significant reductions in computational time. However, not all algorithms are equally suited to parallel execution.

   In the case of the sorting algorithm, one would have to consider whether the algorithm can be effectively parallelized. Some sorting algorithms, like mergesort, are inherently parallelizable, because the division of the input data and the merging of the sorted data are both operations that can be performed in parallel.

   However, even with parallelizable algorithms, there are additional considerations that might affect the computational cost:

   - Overhead of parallelization: Starting, synchronizing, and terminating parallel tasks all incur overhead that can affect the overall computational cost. If the overhead is high, it might outweigh the time saved from parallel execution.
   - Load balancing: In parallel computing, it is important that all processors have an approximately equal amount of work. If some processors finish their tasks earlier and remain idle, this could affect the overall computational efficiency.
   - Data dependencies and communication: If the algorithm involves data dependencies, where the computation of one part depends on the results of another, this could limit the degree of parallelism. Also, the need for processors to communicate and share data can also add to the computational cost.

   So, while the computational cost of the sorting algorithm on a single processor might be $n \log n$, the actual computational cost on a parallel computer architecture could be different and would depend on a variety of factors including those mentioned above.

2. Consider a simple Python function that performs a multiplication operation on two matrices A and B of size $n \times n$. Write a Python function using numpy's built-in np.dot() function to perform this operation and a function to compute the cost of this algorithm. Assume each multiplication and addition operation counts as a single operation.

   In your computation, consider:

   - The cost of multiplying two elements (one operation)
   - The cost of adding two elements (one operation)
   - The nested loops required to perform the multiplication (consider the number of times the operation is repeated).

Does your cost calculation match with the standard cost of matrix multiplication, which is $O(n^3)$? Can you explain why?

For simplicity, you can assume that all matrices are square and of the same dimension, $n \times n$.

3. Implement Strassen's algorithm for matrix multiplication in Python. Your Python function should take two square matrices as input and return the product matrix as output.

   For the input, you may assume that the matrices are 2D numpy arrays of shape (n, n), where n is a power of 2. Also, assume that each element of these matrices is a complex number.

   After implementing the function, test it with two example matrices and verify the output by comparing it to the output from numpy's built-in matrix multiplication function (np.dot()).