1. (a) Add an implementation of Strassen’s Algorithm to your matrix class from the last assignment. You can control which method is called by checking whether the symbol \_OPENMP is defined. If so, you should do the parallel algorithm. If not, you should do the sequential algorithm.

(b) Do a parallel scalability analysis of the parallel, using the sequential code as a baseline. You can use one of the polyps nodes to do your analysis. You should test your algorithm on up to 16 cores. Produce graphs to illustrate your findings.

(c) Do an in-depth analysis of the overhead of your parallel algorithm. How much overhead is there in general? What are the sources of it? How much of the overhead is predictable theoretically and how much comes from sources that are difficult to include in our model of computation? You can use profiling and time measurement routines to obtain your data.

(d) In the sequential case, when optimizing, cache effects are a big part of the overall picture. What additional considerations arise when trying to optimize your parallel implementation with OpenMP? Do you see any obvious ways to go about improving performance?

2. This problem considers the classic sorting algorithms “insertion sort” and “merge sort” that we have discussed in class. The insertion sort has a worst case running time of $O(n^2)$, in contrast with merge sort, which has a running time of $O(n \log n)$ (provably optimal).

(a) On what inputs is insertion sort effective? Try to formalize your analysis of this question by expressing the running time as a function of some property of the list besides just its size.

(b) How would you optimize insertion sort by taking into account cache effects? Would your answer be different for different kinds of inputs?

(c) The standard insertion sort procedure uses a linear search to scan through the already-sorted part of the list to find the insertion point. Since the subarray is already sorted, can we improve the worst-case running time by using binary search to find the insertion point? If so, what would be the improved worst-case running time? Try to express the running time using the same measure of complexity you used in the first part.

(d) Is it possible to parallelize insertion sort in any reasonable way? How about merge sort?

3. Consider an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries in each column are in sorted order from top to bottom.
We will call such a matrix a *sorted matrix*. Note that some of the entries in the matrix may be marked as empty, in which case we consider the value to be infinite.

(a) Argue that all entries of a sorted matrix must be empty if the entry (1, 1) is empty and that the matrix must be completely full if the entry (m, n) is filled.

(b) Give an algorithm to extract the minimum element of the matrix in \(O(m + n)\) time (the main challenge is in restoring the state of the matrix after deleting the minimum element). Your algorithm should use a recursive subroutine that solves an \(m \times n\) instance by solving either an \(m \times n - 1\) instance or a \(m - 1 \times n\) instance. Give and solve a recurrence for the running time in terms of \(p = m + n\) that yields the desired running time.

(c) Give an \(O(m + n)\) algorithm for inserting a new element into a non-full sorted matrix.

(d) Explain how to sort \(n^2\) numbers in \(O(n^3)\) time using the above methods without calling any other sorting algorithm as a subroutine.