COMP3010 Assignment 1 Analysis Portfolio

Harris C. McRae, 47083573

Approach

My initial plan to solve the problem was with what I referred to as a 'heatmap' approach. Given a matrix of random, positive integers, each address on the matrix can be transformed to be a sum of the neighbouring addresses, which can then be interpreted as areas of 'high load'.
Next, the P highest load addresses (where P is the number of processors available) can be chosen as origin seeds of the appropriate sub-matrices, and each sub-matrix can then be grown outwards from each other to produce P sub-matrixes *hopefully* possessing a minimal sum (maximum load).

Upon checking this, I found that this did not work. It might have been possible to utilize an approach like this, but it involved too many possibilities and an algorithm for it may have been needlessly inefficient.

My next solution based upon the concept of finding an optimal case for P=2. Given a matrix $M * N$, to split it into 2 such that each resulting sub-matrix has a minimal sum, there are $(M-1) + (N-1)$ ways to divide the matrix – splitting it between two rows, or between two columns.

This alludes to a recursive solution; if there is only one processor, then allocate the entire bounds of the sub-matrix to that one processor. If there are two, then split the sub-matrix into two split sub-matrices and process both sub-matrices separately for each processor.

In terms of my algorithm, it will split it based on the ratio of one sub-matrix to the other sub-matrix. When deciding how to split, it will compare the 'upper half' sub-matrix's sum to the 'lower half' sum and create a real value of at least 1. Another real value will be created of the ratio of processors: this ratio can be 1 if there are an even number of processors, and greater than 1 for an odd number, capped at 2 (3 processors => 2/1). The sub-matrix ratio closest to the processor ratio will be picked to allocate into.
There exists a secondary check to catch greatly disproportionate sub-matrix ratios; in this case, processors are allocated as per the sub-matrix ratio instead of an ideal processor ratio.

To determine the sum of a sub-matrix, the matrix is converted from its initial state (each address containing only its own value) to a cumulative state such that each address contains the sum of each address in a sub-matrix from $(0,0)$ to itself.

(2)	(3)	(1)
(4)	(8)	(5)
(3)	(2)	=28

Each address is equal to the sum of the address in the -Y direction, the -X direction, and subtracted the address in the $-XY$ direction. By performing this operation on each address from top->bottom, left->right so that each address has its neighbours satisfy this relation, the entire matrix can be set to satisfy this relation. Note that all values below $(0,0)$ are treated to be 0.

This allows the sum of a sub-matrix to be determined in $O(1)$ time after an initial $O(MN)$ operation – a sub-matrix can be a single row, a single column, or any rectangular region of the matrix. The value of a sub-matrix from (A,B) to (X,Y) is determined by $(X,Y) - (A-1,Y) - (X,B-1) + (A-1,B-1)$.

Analysis

The program runs as follows:

1) First, the entire matrix is operated upon so that each address in the matrix represents the sum of all preceding cells. This runs in $O(MN)$ time without fail.
2) Next: the recursive operation.
   Each step of the process requires solving of 2 subproblems, and the complexity of a subproblem is $(m+n)-2$; solutions do not need to be combined and can be said to be 1. For each subproblem, the values of m and n grow smaller by 1 quarter *on average*, while P halves until P=1. As such, the recurrence equation of this phase is:

$$T(m,n,1) = 1$$
$$T(m,n,P) = 2T\left(\frac{3m}{4},\frac{3n}{4},\frac{P}{2}\right) + (m+n) - 2 + 1$$
$$T(K,P) = 2T\left(\frac{3K}{4},\frac{P}{2}\right) + (K-2) + 1$$

   This equation covers average inputs (well-randomized, without bias) – not worst case scenarios.
   In a worst case scenario, for each step of the recursive process the program will divide into two sub-matrices, one with 1 processor and the other with P-1 processors. In this case, the performance is:

$$T(m,n,1) = 1$$
$$T(m,n,P) = T\left(\frac{3m}{4},\frac{3n}{4},P-1\right) + (m+n) - 2 + 1$$
$$T(K,P) = T\left(\frac{3K}{4},P-1\right) + K - 1$$

To prove the average-case performance we will use a substitution; if we are to treat K as static and let P be the only changing variable, $T(P) = O(P \log P)$; we can check for this;
for all $P \geq p_0$ or $P \geq 2, c = 8$, we will find $T(10,P) \leq 8\ (P \log P)$ for the average-case performance of the program.

For every value of $m+n$ or K, there exists a c so that the above statement holds true. Generally, c is close to K. Ultimately, the values of m,n do not factor significantly into the O time of the algorithm.

For a worst-case performance, it still correlates to a $O(P \log P)$ time complexity, as per the same setup as before; for $P \geq 2$ there exists a c corresponding to K ($c \geq K$) so that $T(K,P) \leq c\ P \log P$.

Ultimately, the program runs at $O(nm + (n+m) * P \log P)$ - $O(nm)$ for the initial matrix setup process, and $O((n+m) * P \log P)$ for the recursive process.