COMP3010 Assignment 1

Name: Eric Huang SID: 45980071

Electronic PDF Headings available

Design Concept

These are the core design concepts of the algorithm:

- **Matrix Partitioning Method**
  Since the number of processors is always smaller than the matrix's rows and columns, we can divide the matrix into rows or columns and then assign them to each processor. This way, all regions are rectangles (rows or columns), also ensuring that all parts of the matrix are covered.
- **Optimization 1**
  Compare whether summing up the elements within a row or column results in a smaller load across all processors.
- **Optimization 2**
  When summing up partial sums, compare whether finishing adding the row or returning to the start of the row is closer to the recorded average load (`avgLoad`).
- **Optimization 3**
  For better load balancing, update the total load (`sumLoad`) and average load (`avgLoad`) each time a region is assigned to a processor.

Implementation

This is an overview of the outer-shell control program:

1. Add up all elements inside the matrix, divide the sum by the amount of proc, and we get `avgLoad`.
2. Call the base algorithm twice. First time with normal input, summing up col-by-col, row-by-row. Second time with transposed actions, realised by calling the algorithm function, but with the matrix and its metadata transposed. (**Optimization 1**)
3. Compare the two results and see which one has the smallest load across all proc. Choose the better one and save it as the output.

This is an overview of the base algorithm:

1. Record the current coordinate as the starting coordinate. Sum up elements col-by-col, row-by-row until the partial sum exceeds `avgLoad`.
2. Determine whether adding until completing the row or retreating to the row's start is closer to the `avgLoad`, and apply the better method. (**Optimization 2**)
3. Record the current coordinate as the ending coordinate. Assign the selected region to the profile of the processor.
4. Update `sumLoad` by subtracting the current partial sum. Calculate new `avgLoad` for the remaining processors. (**Optimization 3**)
5. Do some checks (ending checks and remaining processor checks) to handle ending conditions. Repeat steps 1-5 until the ending conditions are met.

Efficiency Analysis

Individual complexity is commented with labels in the source code (e.g. @complexity1)

Parameters & Conversions

```
n, proc, rows, cols
n = rows * cols             // number of elements = matrix rows * matrix cols
rows^2 = Θ(n), cols^2 = Θ(n) // theoretically, on average, the input matrix is square
```

Time Complexity Analysis

```
@complexity1 | n
@complexity2 | inverse of @complexity4 (rows & cols swapped)
@complexity4 | @complexity5 + @complexity16 + @complexity17 {
    @complexity5 | proc * (@complexity6 + @complexity8) {
        @complexity6 | O(n/proc)
        @complexity8 | [cols * cols] + O(n/proc) + [O(cols) + O(n)]
    }
    @complexity16 | proc
    @complexity17 | 4*proc
}
```

The final equation for @complexity0

```
@complexity0 | @complexity1 + @complexity2 + @complexity4
n + {proc * (O(n/proc) + [cols * cols] + O(n/proc) + [O(cols) + O(n)]) + proc + 4*proc}
+ {proc * (O(n/proc) + [rows * rows] + O(n/proc) + [O(rows) + O(n)]) + proc + 4*proc}
  // O(@complexity2) = O(@complexity4)
= O(n + 2*(proc * (O(n/proc) + [cols * cols] + O(n/proc)
       + [O(cols) + O(n)]) + proc + 4*proc))
= O(7n + (2*cols^2+10)*proc + 2*cols) // simple algebra
  // O(2*cols^2+10) = O(2*cols^2)
= O(7n + 2*cols^2*proc + 2*cols)
= O(7n + 2*cols(cols*proc+1)) // simple algebra
  // O(cols*proc+1) = O(cols*proc)
= O(7n + 2*cols^2*proc)
  // O(cols^2) = Θ(n)
= O(7n + 2n*proc)
= O(n*proc)
```

Hence, the time complexity for this program is highly dependent on the number of input elements and processors.

Conclusion

In conclusion, this project has explored algorithm design and implementation, covering matrix partitioning and relevant load distribution optimization challenges. The efficiency analysis yielded a time complexity of O(n * proc), reflecting the algorithm's dependency on the number of input elements and processors.

Looking forward, it is evident that real-world scenarios often involve more processor cores than only matrix rows or columns, especially in contexts like GPUs or supercomputers with a substantial number of cores. Strategies like strip partitioning may offer potential efficiency improvements but require addressing challenges such as handling remaining row elements. As technology advances, these optimizations are surely needed for enhancing parallel processing in the future.

References

[1] E. Huang, “erichuang1/COMP3010-Assignment-1: This project aims to design and implement a load balancing algorithm,” GitHub, 2023. [Online]. Available: https://github.com/erichuang1/COMP3010-Assignment-1.