Measuring Matrix Sparsity Using OpenMP

Tushita Sharva

September 4, 2024

1 Problem Statement

To find the ***percentage sparsity of a given (square) matrix***, which is given by number of zero-valued elements divided by the total number of elements. We leverage multi threading constructs in solving this.

Parameters

- **N:** Number of rows (== Number of columns)
- **S:** Sparsity in the matrix (to be computed)
- **K:** Number of threads
- **rowInc:** Row Increment, specific parameter for one of the allocation methods

1.1 Chunk Based Allocation

- **Number of rows per thread**: $p \leq N/K$. All threads get N/K rows whereas the last thread is allotted $N\%K$ rows.
- **(Static) Allocation**: For one-based indexing, Each thread will be assigned `tid * p + 1` to `(tid + 1) * p`.

  For zero-based indexing:

  - Threads will run from 0 to K - 1, Matrix Rows will run from 0 to N - 1
  - Assignment is a design choice, which is mentioned in upcoming section.

1.2 Mixed Allocation

- **Number of rows per thread**: Number of rows whose reminder is `tid`
- **(Static) Allocation**: For one-based indexing, each thread will be assigned the set of rows whose row number is of format `n * (K) + tid` where n runs from 0 to (N / K). **For zero-based indexing:**

  - Threads will run from 0 to K - 1, Matrix Rows will run from 0 to N - 1
  - While being less than total number of rows, assign `tid`th row, `K + tid`th .. to `tid`th thread and so on...

1.3 Dynamic Allocation

- **Number of rows per thread**: Dynamic, which will be rowInc $\times$ number of times dynamically incremented
- **(Dynamic) Allocation**: The threads have a shared counter which is incremented by the *rowInc* value, and will find sparsity for those `rowInc` number of rows.

2 Design

2.1 Observations

- The function to be parallelised is the function which outputs the number of zeroes in a row given the elements of the row. So we can have a function specific to that across all the implementations.
- For **static allocation**, the threads should have the information of what rows they should look at before hand. When they know what rows to look at, they will invoke the above function for each row.
- For **Dynamic allocation**, we need to decide if the thread has got some work to do, and also simultaneously increase the counter by rowInc. This should be done in **mutually exclusive** fashion. Else, two threads might simultaneously work on same set of rows, which leads to wrong results.

2.2 Design Questions

1) How should the matrix elements be passed to the function?

1. If we put the entire matrix in the global memory, the function would just need the row number to give the zeroes.
2. We can keep in local memory, and to each thread we have to send the row as parameter to that function.

2) How to map threads to rows which works well whether the remainder is 0 or non zero in static allocation

1. **Chunk based:** This can be handled by openMP if programmed properly
2. **Mixed Allocation:** We can program that threads look at this rows whose remainder is x where $x \in [0, tid]$

3) Storing the experimental results (zeroes calculated by each thread)

1. Use an array of class/struct, that stores results of this specific thread. Later on when the threads return, we can obtain the values from the modified array.
2. Use a global counter, which can be atomically incremented by each thread. Here threads don't need to return anything to the invoking function. **Note:** Atomic is not required for first approach (using a class) because each thread `tid` is working on it's own element `arr[tid]`.
3. Use a array, and if the thread id is `tid`, it will modify `arr[tid]`.

2.3 Design choices

Using Global Memory Approach for keeping the matrix (1st Question)

- Because by passing individual subsets, we are making each thread to have a copy of local memory of it's assigned rows is a disadvantage.

Using a Class instead of Global Memory for the elements involved for results (3rd Question)

- We could use a global counter, but in that case we need to use atomic integers. Since each thread is concerned with different rows, it would be an overkill that they have to wait for each thread to modify their value. It is better to maintain individual memory for each thread and sum up them at last.
- The options of vector and class are similar in terms of memory consumption, but decided to use class so that w can leverage OOPs to make the elements private.

3 Low Level Design details

All the three implementations have the following structure:

- The parameters `N, S, K, rowInc`, and the matrix `Matrix` are in the global memory. These values are received from the input file `inp.txt` and this is handled by the `init()` function.
- `Class Logger`: A class that provides methods ***DEBUG, OUTPUT*** whose role is to log the statements in the respective debug files or output files.
- `Class ThreadData`: A class that stores the threadId and the number of zeroes calculated by the thread. The members are private and there are getter and setter methods provided to access and modify them.
- `Function findNumberofZeroesInRow(int x)`: A function which when given a row gives the number of zeroes in that row.
- `Function threadFunc(ThreadData *)` which is the function that is run in parallel by various threads. The logic for assignment of rows to threads is present here. This is the snippet that varies in all three programs.
- `Function main()`: The primary function that initialises the threads, the thread data structures. Also, when all threads join, some other calculations like the total zeroes of the matrix, time of execution, etc., are performed in this `main` function.

Specific to the Dynamic Implementation, we have

- `Class Counter`: A data structure that is designed for the implementation of Dynamic Method. This has a private counter, `rowsDone`, for maintaining the number of rows done. The class provides the method to request for modifying. If we know from the `rowsDone` that all rows are done, the method would **reject that request** (by sending negative acknowledgement), and hence the invoking function threadFunc would know that it can return the control to the `main` function.

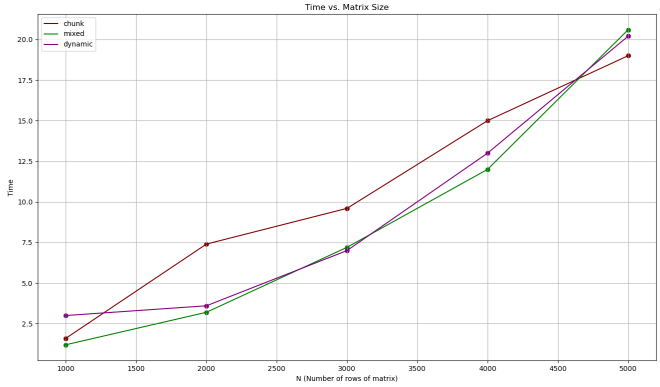
The three programs work as follows:

1. Chunk Based Implementation
   - The main thread reads input
   - The total number of rows to be read are N. So first we write the loop that reads N rows and finds sparsity
   - That is done by invoking the `Function threadFunc(ThreadData *, int i)` for each row in $i \in [0, N]$.
   - `threadFunc` in turn calls the `Function findNumberofZeroesInRow(int i)`, which returns the actual number of zeroes
   - Now we parallelise this loop by writing `pragma op parallel for num_threads(K)`. OpenMP handles them into dividing into chunks with equal distribution.
   - At last when all threads are done, we sum up the zeroes counted by each thread
2. Mixed Implementation
   - The main thread reads the input
   - K threads are spawn
   - Each thread will start and invoke the function `threadFunc`.
   - Each thread then counts the zeroes of those rows which leave a remainder of tid (threadId) when divided with K (number of threads).
   - All threads return and we then sum up zeroes calculated by each thread.
3. Dynamic Implementation
   - The main thread initializes the input matrix and sets up a shared `Counter` object to manage row assignments.

- K threads are spawned, each invoking the function `threadFunc`.
- Each thread uses a `shouldContinue` boolean to determine whether to keep processing rows.
- The thread requests a chunk of rows from the `Counter` object and processes them, counting the zero-valued elements in those rows.
- The `shouldContinue` boolean is updated based on whether there are more rows left to process.
- Once all threads have finished, the total count of zeroes in the matrix is obtained by summing the results from all threads.

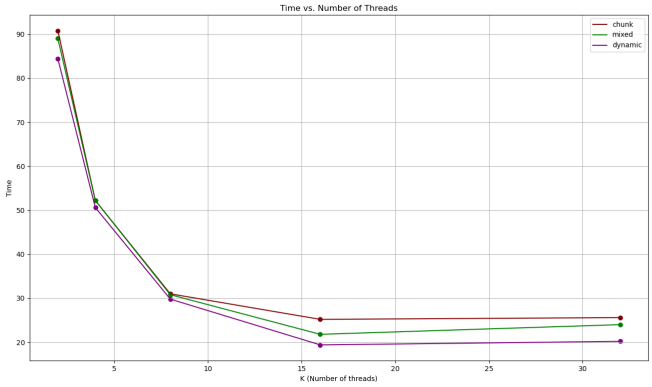
4 Performance Analysis

4.1 Matrix Size vs Time Taken



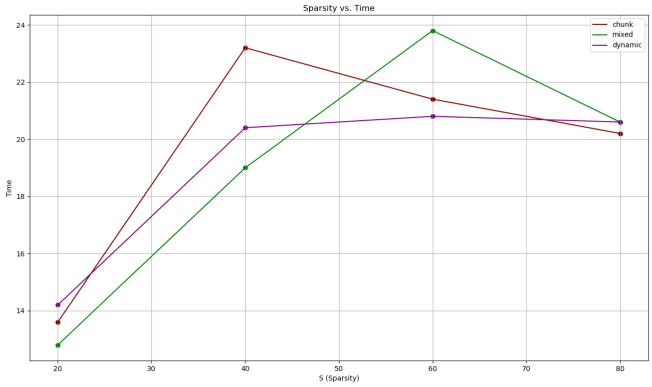
Explanation: Unlike in the previous assignment, we can see that the chunk based implementation is performing the least, and the mixed and dynamic methods are performing similarly. A special mention, the performance of openMP was better cpp threads used in the previous assignment. The performance of Dynamic was better even when it had lock over heads in it's implementation. This can be attributed to underlying OS scheduling and the optimizations of OpenMP.

4.2 Number of threads vs Time Taken



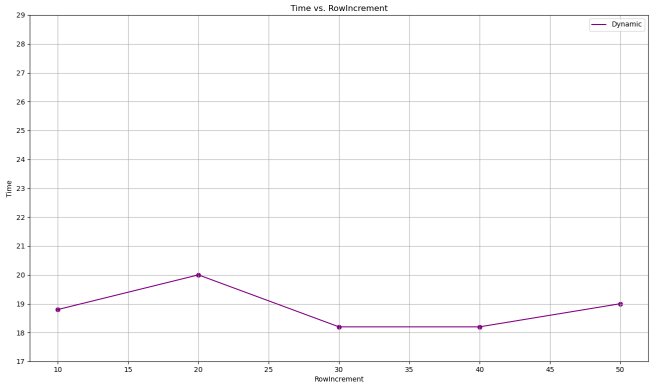
Explanation: We can see that the time taken by all the implementations significantly reduces as the number of threads increases, and then it attains an almost constant execution time. This is expected, which is the direct consequence of Amdahl's law. According to Amdahl's law, the speedup can never become infinitely large and is limited by the sequential code present. That is what we observe here.

4.3 Sparsity vs Time Taken



Explaination: The graph pattern doesnot imply anything significantly, but from theory perspective, we know that all checking whether an element is zero or not takes constant O(1) time. Adding all of them at the end also should happen in constant time, given number of threads are equal. The reason why time is fluctuating within same implementation can be beyond implementation details.

4.4 RowIncrement vs Time Taken



Explaination: We can see that by increasing rowInc, the time taken decreases (ignoring the outlier). This is because we are avoiding many **mutually exclusive** checks by increasing the number of rows checked at one.