Matrix Sparsity Calculation using Multithreading: OpenMP vs PThreads

Om Dave (CO22BTECH11006)

September 4, 2024

1 Introduction

The goal of this report is to analyze the computation of matrix sparsity using different parallel methods. Sparsity is defined as the ratio of zero-valued elements in a matrix to the total number of elements. To measure the matrix sparsity, we utilize OpenMP for parallelization and compare the results with the multithreading approach (PThreads) from the previous assignment. We analyze the efficiency and performance of four methods: Chunk, Mixed, Dynamic, and Mixed-Chunk, under different experimental conditions.

2 Program Design and Implementation

2.1 System Specifications

Table 1: System Specifications

Detail	Specification
CPU(s)	12
Model name	AMD Ryzen 5 5600H
Thread(s) per core:	2
Core(s) per socket:	6

2.2 Code Structure and Workflow

The program is designed using a modular approach, with each parallelization method (Chunk, Mixed, Dynamic, and Mixed-Chunk) implemented as separate functions. The workflow for each method follows a consistent structure to ensure maintainability, flexibility, and ease of testing.

2.2.1 Main Program Flow

The main program flow is as follows:

- **Input:** The matrix is read from an input file provided by the user. The user can also specify the number of threads and sparsity to be checked.

- **Parallel Execution:** The selected method is executed using OpenMP or PThreads, distributing the workload among threads based on the specific method's strategy (static or dynamic scheduling).
- **Output:** The total number of zero-valued elements (matrix sparsity) is computed and displayed. The program also reports the execution time in milliseconds for performance analysis.
- **File Output:** Optionally, results (including timing information and computed sparsity) are written to an output file for further analysis.

2.2.2 Input and Output

Input:

- The program accepts command-line arguments for specifying matrix size, sparsity, number of threads, row increment (for dynamic methods), and method type.
- The program can also read a matrix from an input file (e.g., in CSV format), allowing for custom matrix configurations.

Output:

- After execution, the program prints the total number of zero-valued elements (i.e., the sparsity) and the execution time in milliseconds.
- Optionally, results can be saved to an output file for further use in experiments or performance comparisons.

2.2.3 Thread Management

- **OpenMP:** Thread management is handled automatically using OpenMP directives. The number of threads is set using the `num_threads()` directive, and work distribution is handled using `schedule(static)` or `schedule(dynamic)` based on the selected method.
- **PThreads:** Threads are manually created using `pthread_create()` and synchronized using mutexes to ensure that global variables (such as the total zero count) are updated in a thread-safe manner.

2.2.4 Timing and Performance Measurement

- The `clock_gettime()` function is used to measure the execution time.
- After computation, the average execution time over multiple runs is displayed to minimize the effect of transient system performance fluctuations.

2.3 Computational Methods

In this section, we describe the design and implementation of the four different parallelization techniques for computing matrix sparsity.

2.3.1 Chunk Method

In the chunk method, the matrix is divided into equally sized chunks. Each chunk is assigned to a separate thread, and each thread is responsible for calculating the number of zero-valued elements in its assigned rows. This method ensures static allocation of work to threads, thereby reducing overheads associated with dynamic allocation but may lead to uneven workload distribution if the sparsity pattern varies significantly across chunks.

Implementation in OpenMP:

```
chunk = max(n / k, 2); //using minimum chunk size as 2 other wise method
    ↪ would reduce to mixed
int total_zeros = 0;

#pragma omp parallel num_threads(k) reduction(+:total_zeros)
{
    int id = omp_get_thread_num();
    int total_zeros_local = 0;

    #pragma omp for schedule(static,chunk)
    for(int i = 0; i < n; i++) {
        // if(id==1) printf("Thread %d processing row %d\n", id, i); //
            ↪ Log the distribution
        for(int j = 0; j < n; j++) {
            if(A[i][j] == 0) {
                total_zeros_local++;
            }
        }
    }
    total_zeros += total_zeros_local;
    zeros_by_thread[id] = total_zeros_local;
}
```

In the chunk method, the matrix rows are divided into contiguous blocks (or chunks) and statically assigned to each thread using OpenMP's `schedule(static, chunk)` directive. The chunk size is calculated by dividing the total number of rows by the number of threads, ensuring that each thread processes a block of consecutive rows. Each thread then counts the zero elements within its assigned rows, storing the result in a local variable (*total_zeros_local*). After processing its chunk, each thread updates the global zero count using the OpenMP `reduction(+:total_zeros)` clause, ensuring synchronization without needing `atomic` operations.

2.3.2 Mixed Method

In the mixed method, rows are assigned to threads in an interleaved manner. Each thread processes every 'k'-th row of the matrix, where 'k' is the total number of threads. This method helps to distribute the workload more evenly compared to the chunk method, as sparsity is often not uniformly distributed across consecutive rows.

Implementation in OpenMP:

```
#pragma omp parallel num_threads(k) reduction(+:total_zeros)
{
    int id = omp_get_thread_num();
    int total_zeros_local = 0;

    // Mixed distribution: each thread processes every k-th row
    for (int i = id; i < n; i += k) {
        // if(id==1) printf("Thread %d processing row %d\n", id, i); //
            ↪ Log the distribution
        for (int j = 0; j < n; j++) {
            if (A[i][j] == 0) {
                total_zeros_local++;
            }
        }
    }
    zeros_by_thread[id] = total_zeros_local;
    total_zeros += total_zeros_local;
}
```

Each thread increments by k in the row index to process its assigned rows. Like the chunk method, the zero counts are updated locally first and then added to the global total using the `reduction(+:total_zeros)` clause. The interleaved assignment ensures better load balancing when the sparsity is non-uniform across the matrix.

2.3.3 Dynamic Method

In the dynamic method, the work is distributed to threads dynamically, i.e., each thread fetches a set of rows to process as soon as it completes its previous assignment.

Implementation in OpenMP:

```
#pragma omp parallel num_threads(k) reduction(+:total_zeros)
{
    int id = omp_get_thread_num();
    int total_zeros_local = 0;

    #pragma omp for schedule(dynamic,rowInc)
    for (int i = 0; i < n; i++) {
        // if(id==1) printf("Thread %d processing row %d\n", id, i);
        for (int j = 0; j < n; j++) {
            if (A[i][j] == 0) {
```

```
                total_zeros_local++;
            }
        }
    }

    zeros_by_thread[id] = total_zeros_local;
    total_zeros += total_zeros_local;
}
```

In the dynamic method, rows are dynamically assigned to threads using OpenMP's `schedule(dynamic, rowInc)` directive, where 'rowInc' determines how many rows each thread processes at a time. Threads dynamically pick up new rows once they finish processing their current rows. This method is beneficial for load balancing, as it allows threads that finish early to take on more work. The global zero count is safely accumulated using the `reduction(+:total_zeros)` clause.

2.3.4 Mixed-Chunk Method

In the mixed-chunk method, rows are divided into chunks of size 'rowInc' and these chunks are assigned to threads in a round-robin (cyclic) manner. Each thread processes its assigned chunk of rows, which helps achieve a balance between the static nature of chunking and the dynamic distribution of work across threads.

Implementation in OpenMP:

```
#pragma omp parallel num_threads(k) reduction(+:total_zeros)
{
    int id = omp_get_thread_num();
    int total_zeros_local = 0;

    #pragma omp for schedule(static,rowInc)
    for(int i = 0; i < n; i++) {
        // if(id==1) printf("Thread %d processing row %d\n", id, i); //
        ↪ Log the distribution
        for(int j = 0; j < n; j++) {
            if(A[i][j] == 0) {
                total_zeros_local++;
            }
        }
    }
    zeros_by_thread[id] = total_zeros_local;
    total_zeros += total_zeros_local;
}
```

In the mixed-chunk method, rows are divided into fixed-sized chunks ('rowInc'), and these chunks are assigned to threads in a cyclic (round-robin) fashion. This balances the workload by ensuring that each thread processes non-contiguous chunks, reducing the risk of uneven distribution of zero values. The global zero count is accumulated using OpenMP's `reduction(+:total_zeros)` clause, eliminating the need for atomic operations.

3 Performance Analysis

We conducted four experiments to analyze the performance of the four methods under different conditions. The results are presented below in the following format: (a) Time vs. Matrix Size, (b) Time vs. Number of Threads, (c) Time vs. Sparsity, and (d) Time vs. Row Increment for the dynamic method.

3.1 Experiment 1: Time vs. Matrix Size (N)

3.1.1 Methodology

In this experiment, we varied the size of the matrix from 1000×1000 to 5000×5000 while fixing the sparsity at 40%, the number of threads (K) at 16, and the row increment for the dynamic method at 50.

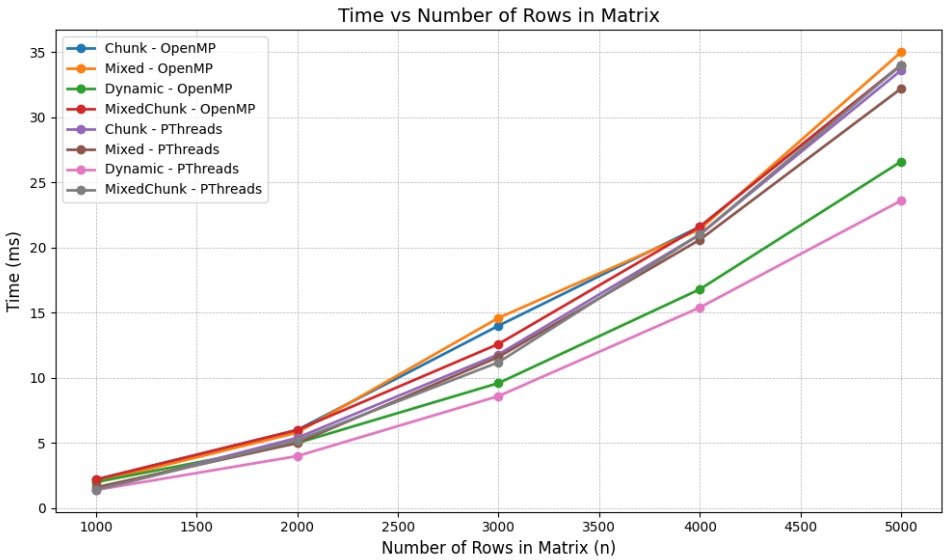

Figure 1: Time vs. Matrix Size for various methods

Table 2: Time (ms) for Different Methods with Matrix Size from 1000 to 5000

Method	**1000**	**2000**	**3000**	**4000**	**5000**
Chunk - OpenMP	2.2	6.0	14.0	21.6	34.0
Mixed - OpenMP	2.0	5.8	14.6	21.4	35.0
Dynamic - OpenMP	2.0	5.0	9.6	16.8	26.6
MixedChunk - OpenMP	2.2	6.0	12.6	21.6	34.0
Chunk - PThreads	1.4	5.4	11.8	21.0	33.6
Mixed - PThreads	1.6	5.0	11.6	20.6	32.2
Dynamic - PThreads	1.4	4.0	8.6	15.4	23.6
MixedChunk - PThreads	1.4	5.2	11.2	21.0	34.0

3.1.2 Observations and Reasoning

As the matrix size increases, the computation time increases across all methods. This is expected, as larger matrices contain more elements that need to be processed, increasing

the workload. Dynamic methods, both in OpenMP and PThreads, slightly outperform the other methods as they dynamically balance the workload better, especially for larger matrix sizes. Among all methods, the performance difference between OpenMP and PThreads is minimal, showing that both parallelization strategies are effective in handling the increased workload with a similar efficiency.

3.2 Experiment 2: Time vs. Number of Threads (K)

3.2.1 Methodology

The matrix size is fixed at 5000×5000, and the number of threads varies from 2 to 32, doubling each time (2, 4, 8, 16, 32). Sparsity is set at 40%, and row increment is 50 for dynamic.

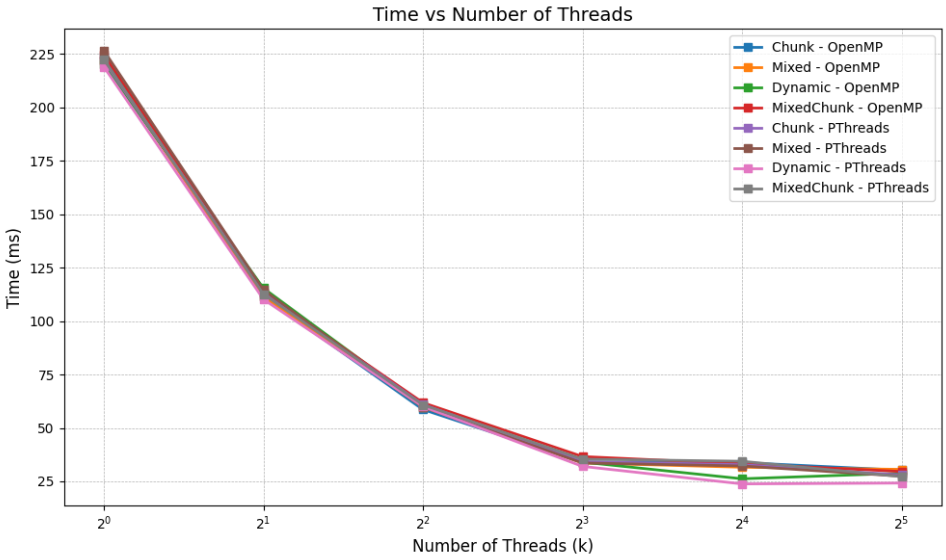

Figure 2: Time vs. Number of Threads for various methods

Table 3: Time (ms) for Different Methods with Number of Threads (K)

Method	**1**	**2**	**4**	**8**	**16**	**32**
Chunk - OpenMP	224.0	111.4	58.6	34.0	33.8	30.2
Mixed - OpenMP	221.8	111.4	59.8	33.6	31.6	30.6
Dynamic - OpenMP	220.0	115.4	60.4	34.0	26.2	28.8
MixedChunk - OpenMP	224.8	113.2	61.8	36.6	33.2	29.6
Chunk - PThreads	221.8	113.2	61.0	34.2	32.8	28.0
Mixed - PThreads	226.6	114.6	59.8	33.6	32.2	27.4
Dynamic - PThreads	219.0	110.2	60.0	32.0	23.8	24.2
MixedChunk - PThreads	222.4	112.6	60.8	35.4	34.4	27.2

3.2.2 Observations and Reasoning

As the number of threads increases, computation time decreases, which is expected due to the workload being divided among more threads. However, after a certain number of

threads (8 in this case), the time becomes nearly constant or even increases slightly due to hardware limitations (in this case, a maximum of 12 threads on the CPU). Beyond this point, the overhead of managing additional threads outweighs the benefits, leading to slight increases in computation time. Once again, dynamic methods outperform the other methods, and both OpenMP and PThreads implementations show similar performance, indicating that thread management overheads are comparable in both approaches.

3.3 Experiment 3: Time vs. Sparsity (S)

3.3.1 Methodology

The sparsity of the matrix is varied from 20% to 80%, keeping the matrix size fixed at 5000×5000, number of threads (K) at 16, and row increment at 50 for dynamic.

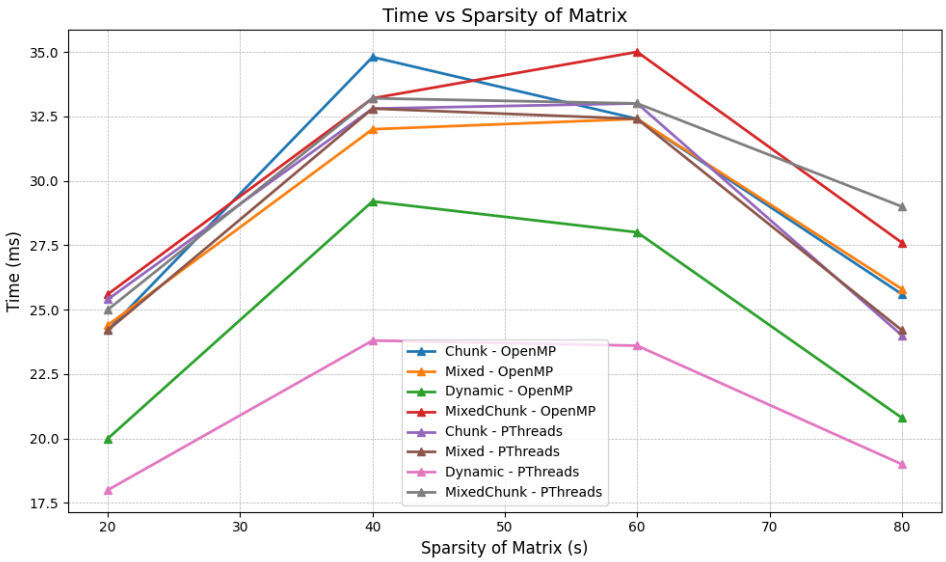

Figure 3: Time vs. Sparsity for various methods

Table 4: Time (ms) for Different Methods with Sparsity from 20% to 80%

Method	**20%**	**40%**	**60%**	**80%**
Chunk - OpenMP	24.2	34.8	32.4	25.6
Mixed - OpenMP	24.4	32.0	32.4	25.8
Dynamic - OpenMP	20.0	29.2	28.0	20.8
MixedChunk - OpenMP	25.6	33.2	35.0	27.6
Chunk - PThreads	25.4	32.8	33.0	24.0
Mixed - PThreads	24.2	32.8	32.4	24.2
Dynamic - PThreads	18.0	23.8	23.6	19.0
MixedChunk - PThreads	25.0	33.2	33.0	29.0

3.3.2 Observations and Reasoning

When increasing sparsity, the computation time first increases and then decreases for all methods. This behavior can be explained by branch prediction in modern processors.

Branch prediction works by using historical data to guess the outcome of a branch decision before it is actually calculated. Modern CPUs have sophisticated algorithms that allow them to predict the direction of a branch with high accuracy. If the prediction is correct, the instruction pipeline continues without any delay. However, if the prediction is incorrect, the pipeline has to be flushed and reloaded, which incurs a significant penalty. Thus, the varying accuracy of branch prediction in the context of sparse matrices leads to changes in computation time.

```
if(A[i][j]==0) total_zeros_local++;
```

leads to branch prediction in this case. At low and high sparsity values (near 20% and 80%), branch prediction is more effective because the patterns of zeros and non-zeros are more consistent, reducing time. In the mid-range (40-60%), branch misprediction increases, leading to higher computation times. Dynamic methods again show better performance compared to others due to their efficient workload distribution across threads.

3.4 Experiment 4: Time vs. Row Increment (rowInc)

3.4.1 Methodology

For this experiment, we vary the row increment size from 10 to 50 (10, 20, 30, 40, 50) while keeping the matrix size at 5000×5000, sparsity at 40%, and number of threads at 16.

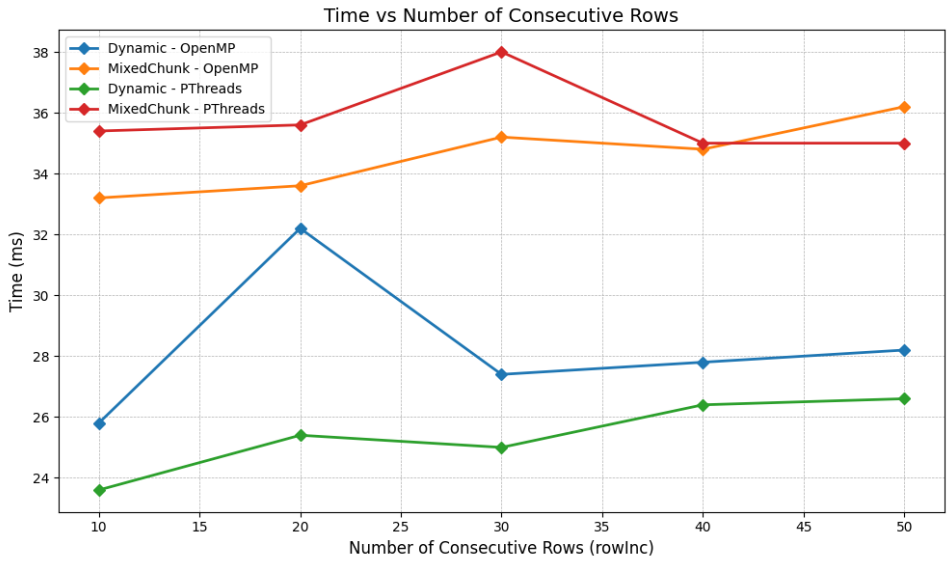

Figure 4: Time vs. Row Increment for various methods

Table 5: Time (ms) for Different Methods with Row Increment from 10 to 50

Method	**10**	**20**	**30**	**40**	**50**
Dynamic - OpenMP	25.8	32.2	27.4	27.8	28.2
MixedChunk - OpenMP	33.2	33.6	35.2	34.8	36.2
Dynamic - PThreads	23.6	25.4	25.0	26.4	26.6
MixedChunk - PThreads	35.4	35.6	38.0	35.0	35.0

3.4.2 Observations and Reasoning

As the row increment ('rowInc') increases, the computation time remains fairly constant with slight fluctuations. The dynamic methods outperform the mixed-chunk methods across all row increment values. While no significant trends are observed, the slight fluctuations suggest that there may be an optimal 'rowInc' value specific to the matrix's sparsity pattern and the location of zeros, which leads to better cache locality and reduced overhead. Further fine-tuning of 'rowInc' could reveal an optimal configuration for a specific matrix type.

4 Conclusion

From the experiments, it is evident that dynamic methods in both OpenMP and PThreads consistently perform better across different conditions, such as varying matrix sizes, number of threads, sparsity, and row increments. The use of OpenMP's 'reduction' clause helps prevent false sharing, which occurs when multiple threads modify adjacent elements in the same cache line, leading to cache coherence delays. By using 'reduction', OpenMP ensures that each thread maintains its own copy of the variable and combines the results at the end of the parallel region, thereby eliminating false sharing and achieving performance comparable to PThreads.

Both OpenMP and PThreads implementations demonstrate similar overall performance, highlighting that both approaches are effective in parallelizing matrix sparsity calculations. The choice between them can depend on the specific application's needs and the ease of implementation.