CS5610 Assignment 1

Gautam Singh
CS21BTECH11018

Contents

***Abstract*—This report analyzes the various techniques implemented for a multi-threaded solution to computing the sparsity of a matrix. The sparsity of a matrix is defined as the number of zero entries in the matrix. The report is organized as follows. In section 1, we introduce a new technique to compute sparsity. Then, we provide a low-level overview of the C++ implementation. In section 2, we describe the low-level program design on which our experiments are run. In ??, we present the results of various experiments and analyze them. Finally, we conclude the report in section 4.**

1 Dynamic Block Allocation

In this section, we describe a new technique to compute the sparsity of a square matrix. In this method, we divide the matrix into square blocks of size M, for a total of $\left(\left\lceil \frac{N}{M} \right\rceil\right)^2$ blocks. Each thread gets a block and computes the sparsity of a matrix by iterating through the elements in that block.

To make this *block allocation* technique dynamic in nature, we allocate ids to each block in row-major order. Formally, if $P = \left\lceil \frac{N}{M} \right\rceil$, then the j-th block in the i-th row of blocks will be given the id $Pi+j$, where i and j are zero-indexed. Therefore, to obtain blocks, threads will increment a shared counter atomically until the limit on the number of blocks (which is P^2) is reached.

Allocation of blocks attempts to exploit spatial as well as temporal locality of the rows of the matrix to prevent cache misses.

2 Program Design

We describe the low-level program design of our C++ implementation used to compute the sparsity of the input matrix.

2.1 Command Line Options

The program takes various arguments that can be found in the `help` function of the code. In particular, the technique to be used is specified using the command line and parsed during execution. Runner functions for each technique have been implemented. Based on the chosen technique, all threads run one of the implemented functions to compute the sparsity of the matrix.

2.2 Threads and Thread Information

All of the runner functions receive a `ThreadInfo` struct which is defined as follows.

Listing 1 The ThreadInfo struct.

```
struct ThreadInfo {
    /// Thread id
    uint64_t id;
    /// Result of thread computation
    int res;
};
```

On executing the runner function, the sparsity computed by this thread is stored in the field `res`. For static methods, the `id` field is used to compute the allocation of rows to this thread. Since thread ids assigned by the operating system may not be in the range $[0, N)$, we assign our own thread ids for use.

Threads are created using the `thread` class in the C++ standard library. Based on the technique to be used, they are instantiated with the runner function and their corresponding `ThreadInfo` struct that contains all the necessary information.

2.3 Atomic Shared Counters

For dynamic methods, such as the dynamic row and block allocation techniques, we create a `Counter` class, whose memebers and methods are defined as follows. Full implementation details can be found in the source code, and are omitted for brevity.

Listing 2 The `Counter` class.

```
template<class T>
class Counter {
    /// 64-bit atomic unsigned counter.
    std::atomic<T> ctr;
public:
    Counter(T n = 0);

    T get();

    T getAndIncrement(T inc = 1);
};
```

The `Counter` class is a template class, meaning one can instantiate a counter using integral data-types of various sizes and signedness. For our purposes, we use a 64-bit unsigned integer as the integral data type. The atomic operations are provided by the `atomic` class in the C++ standard library. The following methods are implemented:

1) `T get()`: Read the current value stored in the counter atomically. Threads use this to check how many rows or blocks have their sparsity computed.
2) `T getAndIncrement(T inc)`: Read the current value and then increment the counter by `inc`. Threads use this to get the id of the row or block on which sparsity is to be computed.

2.4 Timing

The `chrono` class provided by the C++ library was used to measure execution time. Specifically, the time taken from creation of threads to their joining was measured in milliseconds.

3 Results and Analysis

The following experiments have been performed on an Intel i9-11900H 8-core 16-thread CPU. We analyze the variation of execution time on varying different parameters one-by-one. Note that the variable `rowInc` is taken as block increment for the dynamic block allocation method.

3.1 Time vs. Size

Figure 1 shows the execution time for various methods as matrix size N changes. We set the number of threads $K = 16$, row increment $rowInc = 50$ and sparsity $S = 40\%$. The total number of zero elements is shown in Table 1. We observe the following.

1) Execution time for all methods increases with size quadratically.
2) Dynamic row allocation performs the best but dynamic block allocation performs the worst. However, the time difference is within 1.5 ms for all methods for a matrix of size 5000.

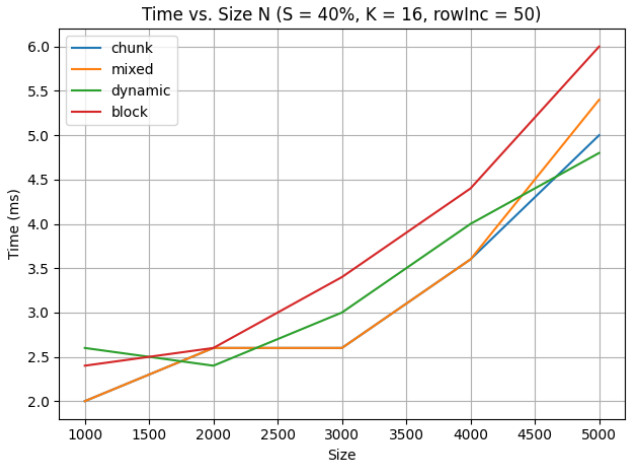

Figure 1. Execution time vs. matrix size for various techniques.

Size N	1000	2000	3000	4000	5000
No. of Zeros ($\times 10^6$)	0.4	1.6	3.6	6.4	10

TABLE 1. NUMBER OF ZEROS FOR 40% SPARSITY AS A FUNCTION OF MATRIX SIZE.

3.2 Time vs. Number of Threads

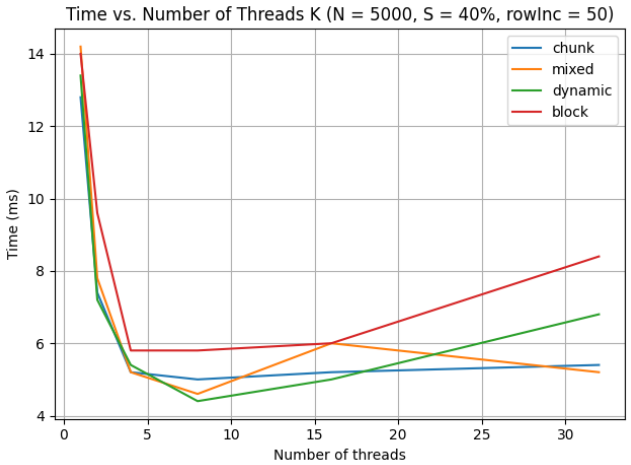

Figure 2. Execution time vs. number of threads for various techniques.

Figure 2 shows the execution time for various methods as the number of threads increases. We set $N = 5000$, $S = 40\%$ and $rowInc = 50$. The following observations can be made.

1) The best performance occurs when using 8 threads. As the number of threads increases, initially the execution time reduces drastically. However, on still

increasing the number of threads, the delay during context switches dominates, thus the total execution time increases.

2) The static allocation methods outperform the dynamic allocation methods, since less work is given to each thread. In fact, the execution times level out for both static methods.
3) The larger execution times for the dynamic methods may also be due to competition for incrementing the shared atomic counter.

3.3 Time vs. Sparsity

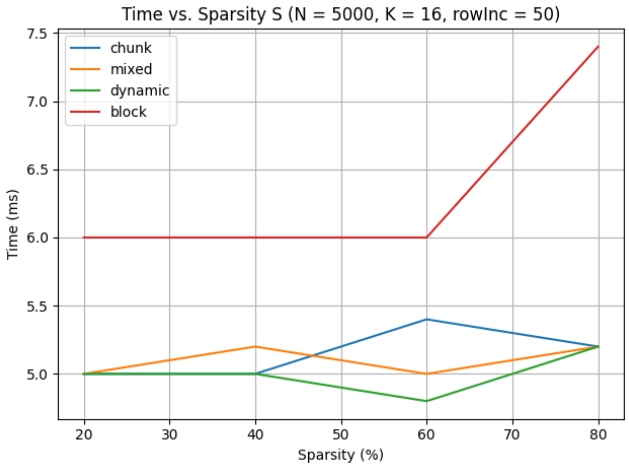

Figure 3. Execution time vs. sparsity for various techniques.

Figure 3 shows the execution time for various methods as the sparsity of the matrix increases. We set $N = 5000$, $K = 16$ and $rowInc = 50$. The number of zeros computed in each case are depicted in Table 2. The following observations can be made.

Sparsity	20%	40%	60%	80%
No. of Zeros ($\times 10^6$)	5	10	15	20

TABLE 2. NUMBER OF ZEROS FOR VARIOUS SPARSITY LEVELS WHERE $N = 5000$.

1) Except for the block method, other techniques are more or less at the same performance despite an increase in sparsity.
2) For the block method, the larger execution times are a result of threads having to compete for incrementing the shared counter, as well as doing the extra work of decoding the location of the block from the block id provided by the shared counter. This results in a slower execution by about 2 milliseconds.

3.4 Time vs. Row Increment

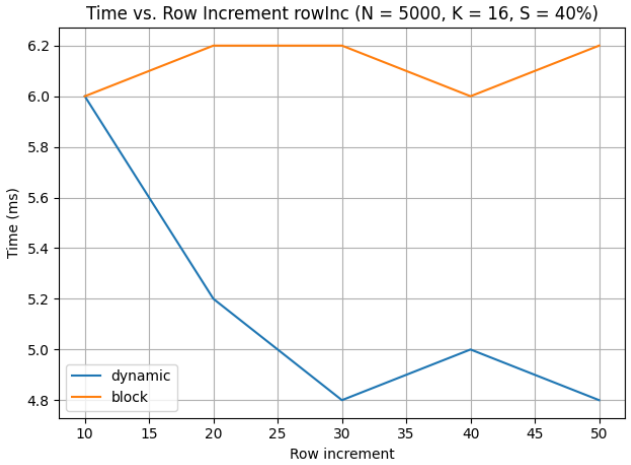

Figure 4. Execution time vs. Row increments for dynamic allocation methods.

Figure 4 shows the execution time for various dynamic methods as the row increments change. We set $N = 5000$, $S = 40\%$ and $K = 16$. The following observations can be made.

1) The execution time for dynamic block allocation remains the same on increasing the block increment, whereas it decreases for dynamic row allocation.
2) In a dynamic setting, row allocation beats block allocation, by 1.4 milliseconds on average. Threads have less decoding to do from the shared counter during dynamic row allocation, which explains the speedup.

4 Conclusion

We conclude that both chunk and mixed methods give very good performance as the number of threads increases. However, dynamic row allocation is better with a lesser number of threads. Because of the excessive decoding work in this implementation, dynamic block allocation did not perform as well as dynamic row allocation on increasing the size of the matrix and also on increasing sparsity. Therefore, one should consider allocation methods and techniques that have very little overhead.