CS5300 Assignment 2

Gautam Singh
CS21BTECH11018

Contents

1 Program Design 1
- 1.1 Command Line Options 1
- 1.2 Thread Information 1
- 1.3 Runner Functions 1
- 1.4 Pthreads Runners 1
- 1.5 OpenMP Runners 2
- 1.6 Atomic Shared Counters 2
- 1.7 Timing 2

2 Results and Analysis 2
- 2.1 Time vs. Size 2
- 2.2 Time vs. Number of Threads 3
- 2.3 Time vs. Sparsity 3
- 2.4 Time vs. Row Increment 3

3 Conclusion 4

***Abstract*—This report analyzes various techniques implemented for a multithreaded solution to computing the sparsity of a square matrix using various multithreading libraries, namely pthreads (POSIX Threads) and OpenMP (Open Multi-Processing). 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 describe the low-level program design on which our experiments are run. In section 2, we present the results of various experiments and analyze them. Finally, we conclude the report in section 3.**

1 Program Design

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

1.1 Command Line Options

The program takes various arguments that can be found in the `help` function of the code. In particular, the technique and library 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.

Code 1 The ThreadInfo struct.

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

1.2 Thread Information

Each thread has its own `ThreadInfo` struct which is defined in Code 1.

On executing the runner function, the sparsity computed by this thread is stored in the field `res`. For static methods, the `id` field can be used to compute the allocation of rows to this thread. These structs are stored in a global array accessible to any runner function.

1.3 Runner Functions

Runner functions are created for each combination of method (chunk, mixed and dynamic) and library (pthreads and OpenMP). The naming convention followed for these functions is `<LIB>_<TECH>Runner`, where `LIB` is the library to be used and `TECH` is the technique to be used.

Runner functions have the signature `void *()(void *)`. That is, runners receive a generic pointer and return a generic pointer. We create a hash table (using the C++ `unordered_map` container) mapping the string `<LIB>_<TECH>` obtained from the command line arguments to its corresponding runner function pointer to be used, or to output an error message in case of invalid input.

1.4 Pthreads Runners

For runner functions implemented using the pthreads API, threads are created and joined in the main function using `pthread_create` and `pthread_join`. For compatability with the API, runner functions have the signature described in section 1.3. Threads are passed a generic pointer to the respective `ThreadInfo` struct, since thread IDs can be arbitrary. To implement the dynamic methods, the dynamic technique runner makes use of the atomic shared counter described in section 1.6.

1.5 OpenMP Runners

In contrast to the runner functions implemented for pthreads, runner functions implemented using OpenMP are much smaller in code size, due to the following reasons.

1) Thread IDs alloted to OpenMP threads are 0-indexed, thus we can use them directly as an index to the global `ThreadInfo` array. It is for this reason that we pass a null pointer to the OpenMP runners.
2) Various OpenMP `pragma` directives can be used to implement the required computation technique. In particular, the `schedule` directive determines the method in which work should be allocated to threads. Among many other schedules, we use either `static` or `dynamic` scheduling, with chunk size being specified where needed. By default, `static` scheduling allocates roughly equal work to all threads.
   Since these directives take care of work allocation, we do not need to use any shared counters or thread IDs to determine the work of each thread.

1.6 Atomic Shared Counters

For the dynamic row allocation technique in pthreads, we create a `Counter` class, whose memebers and methods are defined in Code 2. Full implementation details can be found in the source code, and are omitted for brevity.

Code 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.

1.7 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.

2 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.

2.1 Time vs. Size

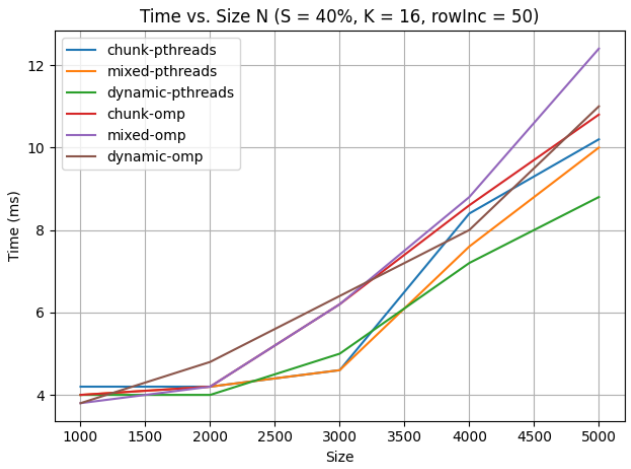

Figure 1. Execution time vs. matrix size for various techniques.

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.

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.

1) Execution time for all methods increases with size roughly quadratically.
2) Within a library, dynamic methods perform better than static methods on average. This is because the zero entries in the matrix are randomly dispersed, so a dynamic method ensures a fairer work division among the threads.
3) Pthreads outperforms OpenMP on all techniques.

2.2 Time vs. Number of Threads

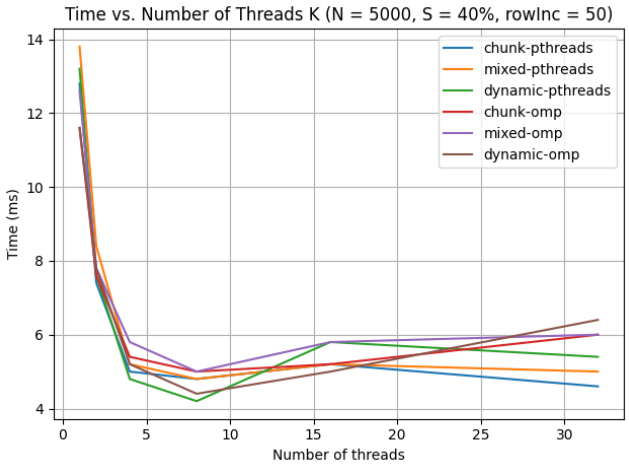

Figure 2. Execution time vs. number of threads for various techniques.

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) For larger number of threads, static methods dominate on pthreads, since there may be more competition among threads in incrementing the atomic shared counter. However, for OpenMP, the dynamic method is faster even for a larger number of threads, suggesting a different method of dynamic row allocation.
3) The optimal runtimes for dynamic methods are the best (at 8 threads).
4) Pthreads outperforms OpenMP in this setting as well.

2.3 Time vs. Sparsity

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.

1) Static methods perform comparatively more poor when the sparsity increases. This is because the sparsity is distributed at random, resulting in improper load balancing on using static methods. In such cases, dynamic methods perform better and should be preferred. Thus, for both libraries, dynamic methods perform the best.
2) Pthreads outperforms OpenMP on all methods here. In particular, pthreads outperforms OpenMP by about 1.5 milliseconds at 80% sparsity on dynamic allocation.

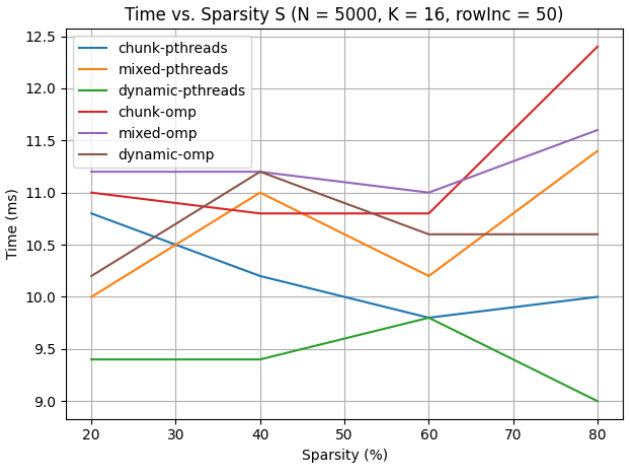

Figure 3. Execution time vs. sparsity for various techniques.

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 $N = 5000$.

2.4 Time vs. Row Increment

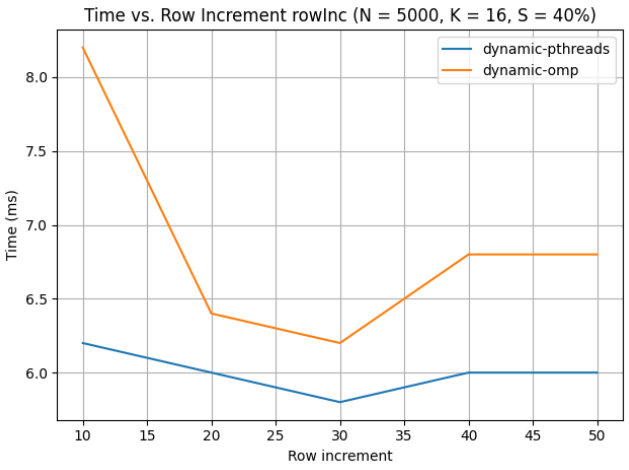

Figure 4. Execution time vs. Row increments for dynamic allocation methods.

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 optimal row increment for this setting turns out to be $rowInc = 30$ for both libraries.
2) The initial decrease in runtime as row increment increases is more drastic in OpenMP than in pthreads. In both cases, it is due to cache locality being exploited.
3) However, runtime then increases with increasing row increment because it is too large, leading to possibly improper load balancing as the chunks are too large.
4) Pthreads again outperforms OpenMP over all row increments.

3 Conclusion

We conclude that for many cases, dynamic methods of work allocation to threads is better, especially since the sparsity of the matrix is randomly distributed. Further, Pthread performed better than OpenMP on all experiments, even with using our constructs such as the atomic shared counter. This might be because the Pthreads API is lower level than OpenMP, and thus more performant for this task. OpenMP, on the other hand, is suited for portable code and faster development.