Introduction to Parallel Computing
Homework 2: Parallelizing matrix operations using OpenMP.
Results report

Lorenzo Fasol
lorenzo.fasol@studenti.unitn.it
227561
https://github.com/Lory1403/PC-Homework-2

a.y. 2023/2024

This report addresses the implementation and performance analysis of parallel matrix multiplication and transposition using OpenMP. Matrix operations are fundamental in various scientific and computational domains, and efficient parallelization is crucial for handling large datasets.

1 Parallel matrix multiplication

This first task of the assignment entails the implementation of matrix multiplication, a foundational operation in linear algebra. The primary objective is to efficiently compute the product of two matrices. The exploration of diverse OpenMP directives is undertaken with the explicit purpose of enhancing computational performance. In my implementation, I opted to design two distinct algorithms, matMul and a parallel version matMulPar, capable of handling both squared and non-squared matrices.

Serial implementation

`Source code 1` presents a fundamental serial implementation of matrix multiplication. The algorithm employs a triple loop structure to iterate over the rows of the first matrix, the columns of the second matrix, and the columns of the first matrix. Despite its simplicity and intuitiveness, this implementation exhibits inefficiencies. Its time complexity is characterized by $O(2 * n^3)$ due to the multiplications and additions, while the space complexity is $O(n^2)$ owing to the size of matrix C.
A more sophisticated implementation is presented in `Source code 2`. This optimized version strategically interchanges the order of the j and k loops, resulting in a reduction in cache misses and an improvement in data locality. These optimizations contribute to enhanced overall performance.

Source code 1: Serial nested loops matrix multiplication

```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) collapse(2)
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

Source code 2: Optimized serial matrix multiplication

```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) coll
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

In an attempt to optimize matrix multiplication performance, the blocking or loop tiling technique was tested. This strategy involves partitioning large matrices into smaller blocks or tiles to enhance cache utilization and reduce cache misses as in the prior implementation. However, despite the theoretical advantages of this technique, the empirical results did not yield to an improvement.

Parallel implementation

`Source code ??` presents a parallel implementation of matrix multiplication. The algorithm is similar to the serial version, but it employs OpenMP directives to parallelize the outermost loop.

The provided code employs OpenMP directives to parallelize the computation of the matrix product C resulting from the multiplication of matrices A and B. These directives include the following:

- `#pragma omp parallel for`: this pragma directive initiates a parallel loop, distributing the computational workload across multiple threads
- `shared(A, B, C)`: the shared clause indicates that matrices A, B, and C are shared among the threads
- `reduction(+:sum)`: the reduction clause designates the variable `sum` as a reduction variable, ensuring that each thread maintains its local copy of the variable. These local copies are later aggregated using addition to compute the final result
- `collapse(2)`: the collapse clause collapses the nested loops into a single loop
- `#pragma omp simd`: this pragma directive in the innermost loop employs Single Instruction, Multiple Data (SIMD) parallelism to further enhance vectorization and exploit parallelism at the instruction level

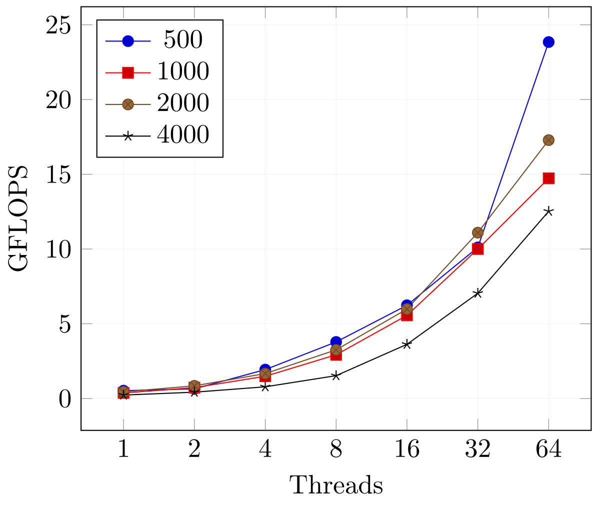
Within the loop, each element `C[i][j]` is computed as the sum of products of corresponding elements from row i of matrix A and column j of matrix B.

Source code 3: Parallel nested loops matrix multiplication

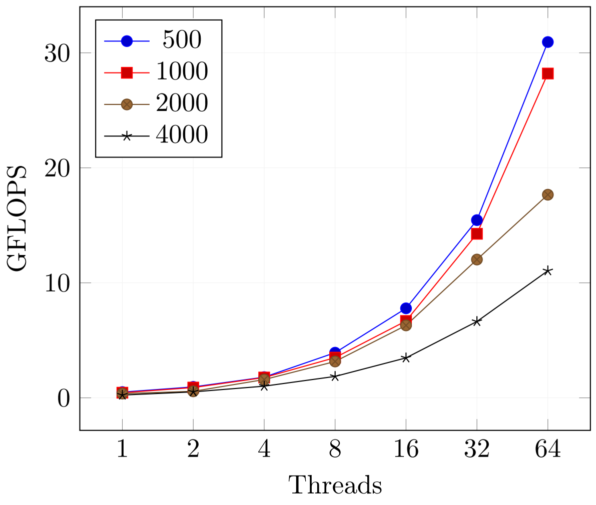
```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) collapse(2)
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

Results analysis

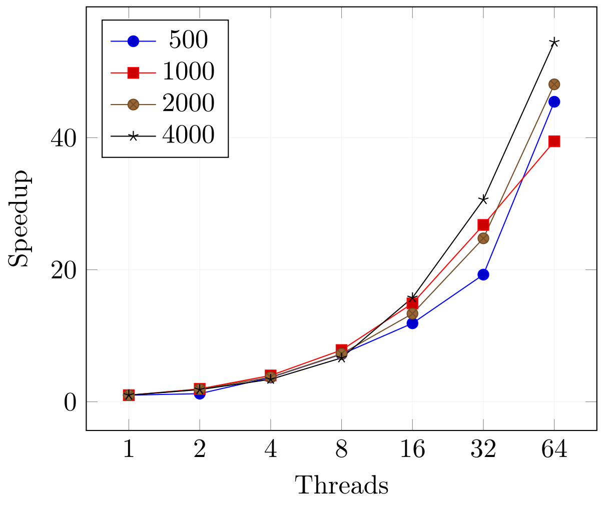
The design of the serial code did not facilitate a direct comparison with its parallel counterpart due to the optimization that I adopted. In order to facilitate a thorough analysis of results, it is essential to establish a basis for comparison. To ensure a fair comparison, the parallel code was intentionally executed with a single thread, thereby obtaining serial results for meaningful analysis.



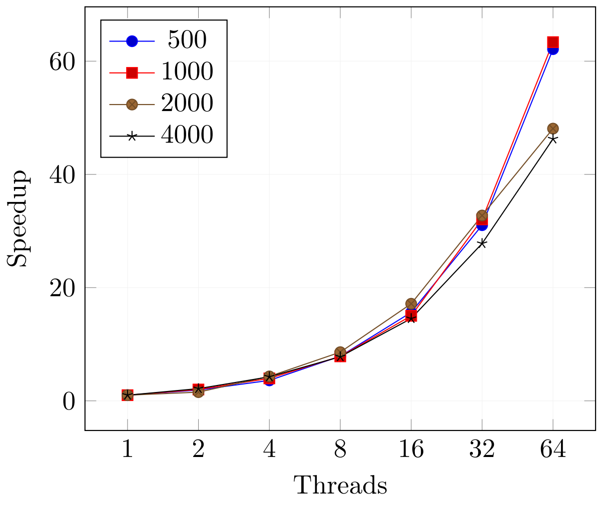
GFLOPS for Sparse Matrices



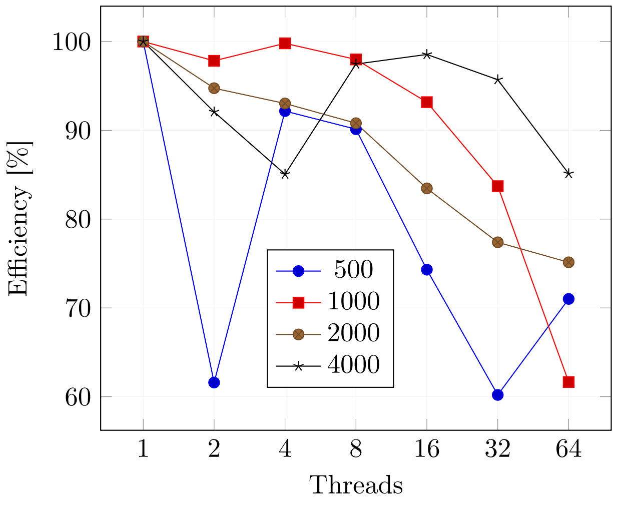
GFLOPS for Dense Matrices



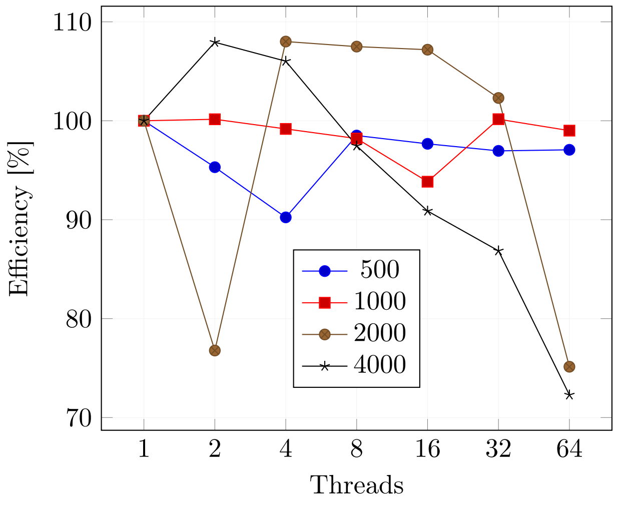
Speedup for Sparse Matrices



Speedup for Dense Matrices



Efficiency for Sparse Matrices



Efficiency for Dense Matrices

Potential Bottlenecks and Optimizations:

- **Cache Usage:** Cache misses can be a significant bottleneck. Strategies such as loop tiling or blocking, as previously discussed, can enhance cache locality.
- **Memory Access Patterns:** Irregular memory access patterns can lead to poor performance. Optimizations like loop reordering to improve data locality and diminish cache misses, for instance, through loop interchange or unrolling, are effective measures.
- **Floating-Point Arithmetic:** Floating-point operations can be a bottleneck on some systems. Hardware-specific optimizations can be taken into account, such as SIMD (Single Instruction, Multiple Data) instructions.
- **Algorithmic Optimizations:** Incorporating advanced algorithms, such as Strassen's algorithm (reducing the number of multiplications required) or the Coppersmith-Winograd algorithm (optimized for large matrices), can yield notable improvements in performance.

Differences between Parallel Algorithms for Dense and Sparse Matrices:
Setting aside the obvious differences in data structure, parallel algorithms for dense and sparse matrices differ in terms of parallelism, complexity, optimizations, and sparsity considerations.
In dense matrices, achieving parallelism is often uncomplicated, employing techniques like loop-level parallelism (e.g., OpenMP). Contrarily, sparse matrices pose a challenge due to their irregular data structures, necessitating the use of specialized algorithms.
Parallel algorithms for sparse matrices tend to be more complex, involving dynamic load balancing, data partitioning, and addressing communication overhead. In the context of dense matrices, the focus lies on optimizing cache usage and exploiting SIMD instructions. In contrast, sparse matrix parallelism centers on efficient manipulation of data structures.
Parallel algorithms for sparse matrices must also consider the specific sparsity pattern. The sparsity pattern affects parallelism and load balancing. Some parallel algorithms might perform well with specific sparsity patterns while struggling with others.

In summary, while parallelization is valuable for both dense and sparse matrices, the approaches and challenges differ significantly due to the inherent differences in data structure and complexity.

2 Parallel matrix transposition

The code produced for this task performs transposition operations on matrices, both in serial and parallel fashion using OpenMP directives. It includes functionality for transposing matrices and measuring the elapsed time for various scenarios, such as different matrix dimensions, thread counts, and block sizes for block transposition. The matrices can be also initialized in dense or sparse fashion.

Serial and parallel matrix transpose implementation

Source code 4

```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) collapse(2)
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

The serial code for the transposition is a simple nested loop that iterates over rows and columns of the matrix and swaps the elements. The iteration over the columns was prioritized due to the observed enhanced performance in maintaining cache-friendliness with the transposed matrix (TA) compared to the original matrix (A).

Serial and parallel block matrix transpose implementation

Three principal solutions have been explored for an optimal block transposition implementation.
`Source code` ?? represents the least efficient approach. It incorporates unnecessary index calculations, resulting in decreased efficiency due to redundant computations.
`Source code` ?? and `Source code` ?? exhibit similar structures. In an attempt to enhance data locality, I experimented with the order of loops. Despite replicating the logic applied in previous tasks, these variations did not yield any optimizations.

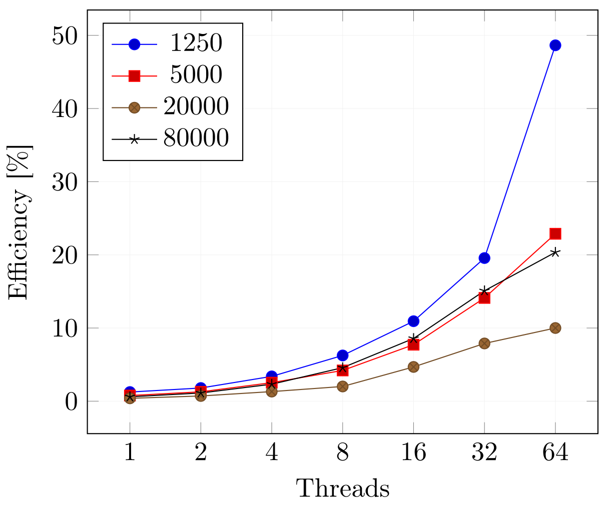
Source code 5: Matrix transposition with blocks - First implementation

```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) collapse(2)
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

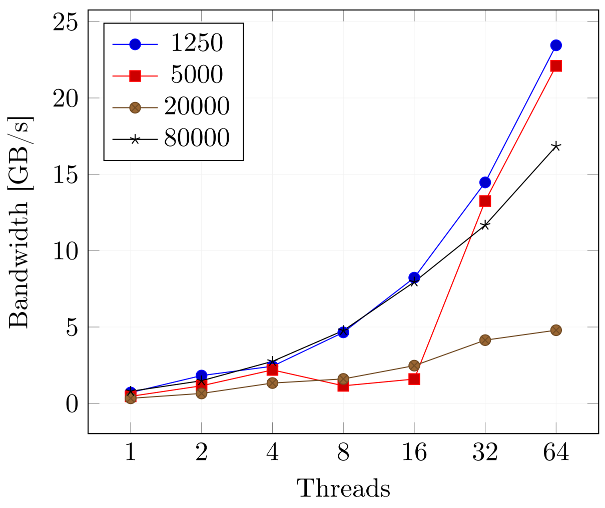
```
}
```

Source code 6: Matrix transposition with blocks - Second implementation

```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) collapse(2)
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
```



Bandwidth for Sparse Matrices without blocks



Bandwidth for Dense Matrices without blocks

Source code 7: Matrix transposition with blocks - Third implementation

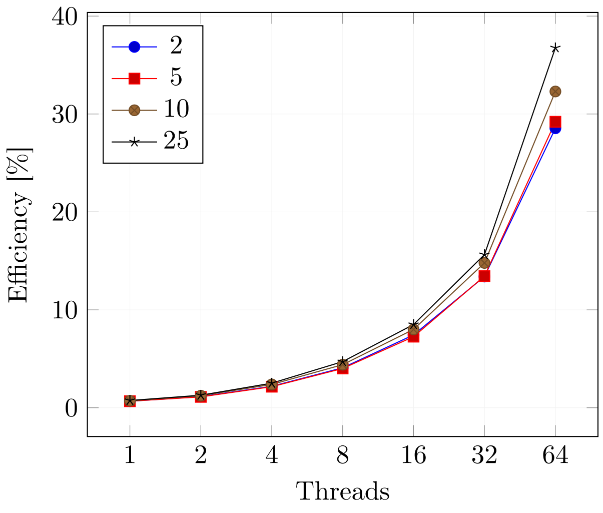
```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) collapse(2)
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

As discussed earlier, maintaining uniformity in the algorithm across both serial and parallel executions serves the purpose of establishing a robust basis for comparing results.
This approach ensures that any variations in performance can be more precisely attributed to the parallelization strategy rather than disparities in algorithmic intricacies.
After numerous tests conducted, the best pragma to perform the parallel task is the one presented in `Source code ??`. The directive is instructing the compiler to parallelize a nested loop with SIMD parallelism, collapsing four levels of loops into a single parallel loop. The inclusion of the SIMD clause is particularly significant for its substantial optimization impact, considering the increased number of variables taken into account.

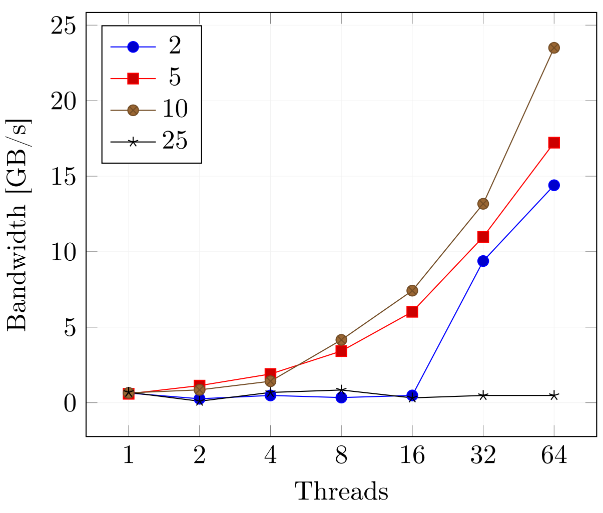
Source code 8: Matrix transposition with blocks - Third implementation

```
float sum;
#pragma omp parallel for shared(A, B, C) reduction(+:sum) collapse(2)
for (int i = 0; i < rowsA; i++) {
    for (int j = 0; j < colsB; j++) {
        sum = 0.0;
        #pragma omp simd
        for (int k = 0; k < colsA; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

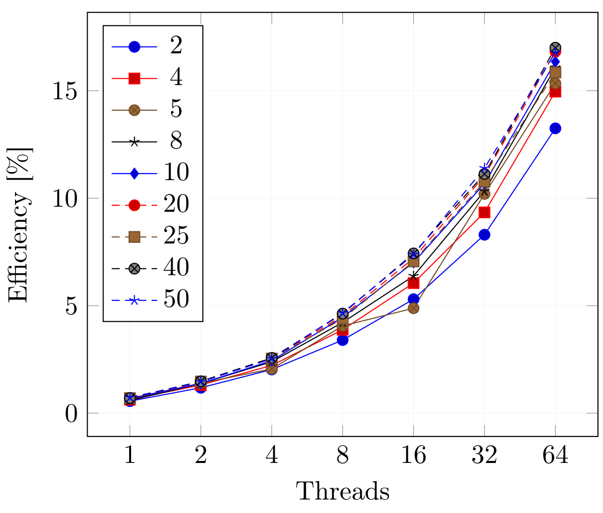
Results analysis



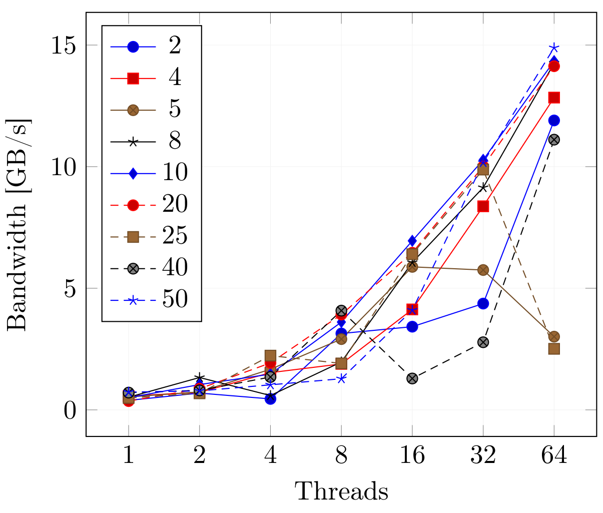
Bandwidth for Sparse Matrix size 1250 with blocks



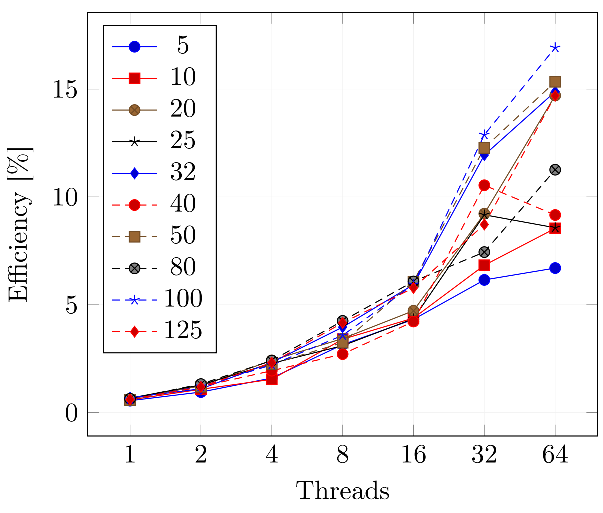
Bandwidth for Dense Matrix size 1250 with blocks



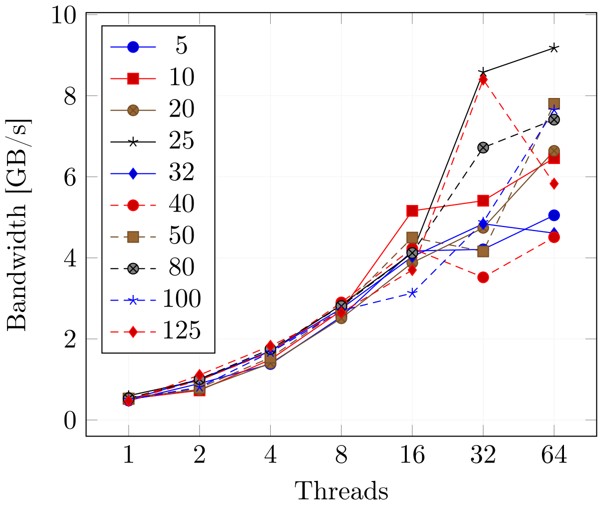
Bandwidth for Sparse Matrix size 5000 with blocks



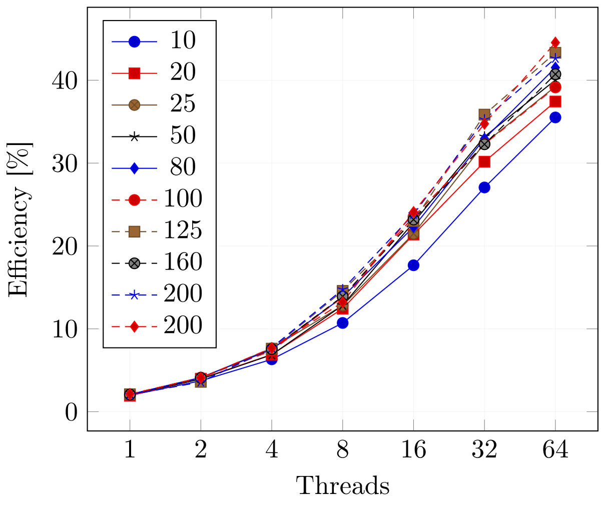
Bandwidth for Dense Matrix size 5000 with blocks



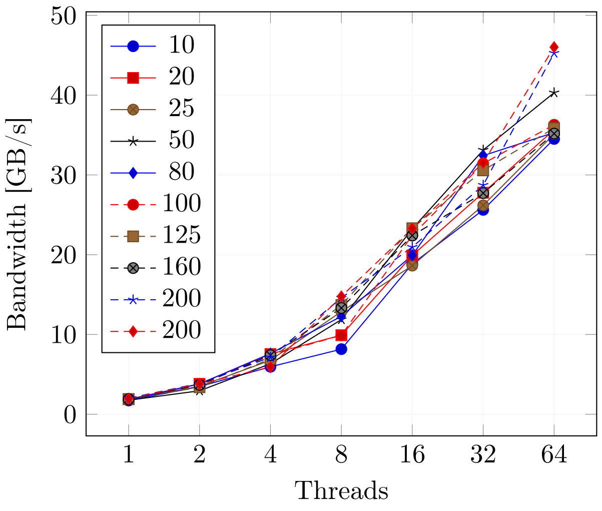
Bandwidth for Sparse Matrix size 20000 with blocks



Bandwidth for Dense Matrix size 20000 with blocks



Bandwidth for Sparse Matrix size 80000 with blocks



Bandwidth for Dense Matrix size 80000 with blocks