h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Darmstadt University of Applied Sciences
– Faculty of Computer Science –

Lab report for the second high-performance computing exercise

by
Zebreus

INTRO

In the second lab we will write and benchmark a program that multiplies two matrices using multiple threads and shared memory. Analyze your program and predict the relationship between the run-time and the width and height of the matrices. At last we will measure the performance to verify our prediction.

IMPLEMENTATION

Listing 1 shows the sequential implementation of the matrix multiplication. The algorithm is pretty straight forward. We iterate over the rows of the first matrix and the columns of the second matrix. For each element we calculate the dot product of the row and column. The result is stored in the `result` matrix.

Listing 1. Sequential implementation

```cpp
CMatrix multiply(CMatrix const& m1, CMatrix const& m2) {
    CMatrix result(m2.width, m1.height);  // allocate memory
    for (unsigned int row = 0; row < m1.height; row++) {
        for (unsigned int col = 0; col < m2.width; col++) {
            double sum = 0.0;
            for (unsigned int dot = 0; dot < m2.height; dot++) {
                sum += m1[row][dot] * m2[dot][col];
            }
            result[row][col] = sum;
        }
    }
    return result;
}
```

We then parallelized the version using openMP, which can be seen in Listing 2. We instruct OpenMP to collapse the two outer for loops into one and to schedule all iterations of it's body among the threads. This way every thread repeatedly does iterations of the innermost loop with different values for `row` and `col`. `m1` and `m2` are shared among all threads, as they will only be read. We also share `result` among all threads, as every iteration of the inner loop writes to a different position in it, so there should be no race conditions. The sum variable is private to every iteration, as it is defined inside of the loop.

By setting the schedule to `static` we ensure that the workload is evenly distributed among the threads. In this case this should lead to the best performance, as the workload is the same for every iteration of the inner loop.

Listing 2. Parallel openMP implementation

```cpp
CMatrix
multiply(CMatrix const& m1, CMatrix const& m2, unsigned int numberOfThreads) {
    CMatrix result(m2.width, m1.height);  // allocate memory
#pragma omp parallel for collapse(2) schedule(static) \
    num_threads(numberOfThreads)
    for (unsigned int row = 0; row < m1.height; row++) {
        for (unsigned int col = 0; col < m2.width; col++) {
            double sum = 0.0;
            for (unsigned int dot = 0; dot < m2.height; dot++) {
                sum += m1[row][dot] * m2[dot][col];
            }
            result[row][col] = sum;
        }
    }
    return result;
}
```

We also implemented the algorithm in Rust as shown in Listing 3. It is common in Rust to use fewer explicit `for` loops then in C++ and instead use iterators and filter/map/reduce functions to process data. This is also the case here. We use the `chunks_exact(columns)` function to create an iterator that iterates over the rows. For each row we do the same to get an iterator over all columns of the other matrix. Notice that we transposed the other matrix before, so all of its columns are continous memory regions. This allows us to also use `chunks_exact()` here.

To calculate the do product of the row and column, we create iterators for both and zip them together to get an iterator over the pairs of elements at the same position. We then map the pairs to their product and sum the results up to get the dot product.

We mapped every row to an iterator so we now have an iterator over iterators. We use flatten to merge those iterators into one iterator of the results. We then collect the results into a vector, which is then converted into a matrix.

Listing 3. Sequential Rust implementation

```rust
    pub fn multiply_sequential(&self, other: &Matrix) -> Matrix {
        let transposed_other = other.transpose();

        let columns = self.cols;

        let result = self
            .data
            .chunks_exact(columns)
            .map(|row| {
                transposed_other
                    .data
                    .chunks_exact(columns)
```

```
                    .map(|column| row.iter().zip(column.iter()).map(|
  (a, b)| a * b).sum())
            })
            .flatten()
            .collect::<Vec<_>>();

        Matrix {
            rows: self.rows,
            cols: other.cols,
            data: result,
        }
    }
```

We can now parallelize this using the Rayon Data-parallelism library. Rayon provides parallelizing iterators as a drop in replacement for the normal Rust iterators. As you can see in Listing 4 we only needed to replace the `chunks_exact()` function with `par_chunks_exact()`. This will create a parallel iterator instead of a sequential one that will split the work among the threads.

In the inner part we still use `iter()` instead of Rayons `par_iter()` because we want to do not want to parallelize the inner part. This is because the inner part is very small and the overhead of parallelizing is probably bigger than the benefits.

Listing 4. Rayon implementation

RUST

```
    pub fn multiply(&self, other: &Matrix) -> Matrix {
        let transposed_other = other.transpose();

        let columns = self.cols;

        let result = self
            .data
            .par_chunks_exact(columns)
            .map(|row| {
                transposed_other
                    .data
                    .par_chunks_exact(columns)
                    .map(|column| row.iter().zip(column.iter()).map(|
  (a, b)| a * b).sum())
            })
            .flatten()
            .collect::<Vec<_>>();

        Matrix {
            rows: self.rows,
            cols: other.cols,
            data: result,
        }
    }
```

In the two examples above we did transpose the matrix beforehand, which can bring a significant performance boost over the C++ implementation. To be able to compare the performance of the two implementations we also benchmarked a C++ implementation that transposes the matrix before multiplying it. This can be seen in Listing 5. The algorithm is similar to Listing 2 except that we transpose the second matrix before multiplying it.

Listing 5. Transposed openMP implementation

```cpp
CMatrix multiply(CMatrix const& m1, CMatrix& m2, unsigned int numberOfThreads) {
    CMatrix result(m2.width, m1.height);  // allocate memory

    m2.transpose();

#pragma omp parallel for collapse(2) schedule(static) \
    num_threads(numberOfThreads)
    for (unsigned int row = 0; row < result.size; row += m1.width) {
        for (unsigned int col = 0; col < m2.size; col += m2.width) {
            double sum = 0.0;
            auto rowBase = m1.container + row;
            auto colBase = m2.container + col;
            for (unsigned int dot = 0; dot < m2.width; dot++) {
                sum += rowBase[dot] * colBase[dot];
            }
            result[row / m1.width][col / m2.width] = sum;
        }
    }
    return result;
}
```

We also implemented the algorithm in Rust without transposing the matrix first, so we can compare it to the original C++ implementations. We met two main problems when implementing it. First, openMP parallelizes loops, while Rayon parallelizes iterators. This makes all the comparable implementations a bit awkward, as we create iterators over number ranges to replicate the C++ loops. In all follwing examples we create iterators over `(row, column)` tuples to replicate the C++ behaviour. The second problem is that the Rust compiler guarantees that safe Rust does not contain any data races. As a result it does not allow multiple threads to have mutable access to the same data. So we cannot share the result matrix among the threads, like we did in C++.

A workaround is to use unsafe Rust, which allows us to take a pointer to the result Matrix and share it among multiple threads. Listing 6 shows our implementation using unsafe Rust.

Listing 6. Unsafe Rust implementation

```rust
    pub fn multiply_faithful_unsafe(&self, other: &Matrix) -> Matrix {
        unsafe {
            let mut result = vec![0.0f64; self.rows * other.cols];
            let result_matrix_pointer = result.as_mut_ptr() as usize;

            (0..self.rows)
                .into_par_iter()
                .flat_map(|row| {
                    return (0..other.cols).into_par_iter().map(move |col| (row, col))
;
                })
                .for_each(|(row, col)| {
                    let mut sum = 0.0;
                    for i in 0..other.rows {
                        sum += self.read_at(&row, &i) * other.read_at(&i, &col);
                    }
                    let result_pointer =
                        (result_matrix_pointer as *mut f64).add(row * other.cols + co
l);
                    *result_pointer = sum;
                });

            return Matrix {
                rows: self.rows,
                cols: other.cols,
                data: result,
            };
        }
    }
```

The more correct way would be to write our code in a way that guarantees that no thread can write into the memory of another thread. Listing 7 shows our implementation using this approach. For this we do not only create an iterator over the `(row, column)` coordinate pairs, but we also add a mutable reference to the associated location in the result matrix to the tuples. This way we iterate over something like `(row, column, resulting_value)` instead. This way every thread can only mutate the one value in the result matrix it is supposed to write.

Listing 7. Rust implementation with pairs

```rust
    pub fn multiply_faithful_pairs(&self, other: &Matrix) -> Matrix {
        let mut result = vec![0.0f64; self.rows * other.cols];

        let parallel_iterator_with_coordinates =
            result.par_iter_mut().enumerate().map(|(i, slice)| {
                let row = i / other.cols;
                let col = i % other.cols;
                (row, col, slice)
            });
```

```
10
11      parallel_iterator_with_coordinates.for_each(|(row, col, resulting_value)| {
12          let mut sum = 0.0;
13          for i in 0..other.rows {
14              sum += self.read_at(&row, &i) * other.read_at(&i, &col);
15          }
16          *resulting_value = sum;
17      });
18
19      return Matrix {
20          rows: self.rows,
21          cols: other.cols,
22          data: result,
23      };
24  }
```

In the previous examples we allocated the result matrix before creating the iterators to be as close to the C++ implementation as possible. It should result in the exect same behaviour, if we map the coordinate pairs to their resulting value and collect them into a matrix. Listing 8 shows that approach.

Listing 8. Rust implementation without transposing

```
RUST
1   pub fn multiply_faithful_iterators(&self, other: &Matrix) -> Matrix {
2       let data = (0..self.rows)
3           .into_par_iter()
4           .flat_map(|row| {
5               return (0..other.cols).into_par_iter().map(move |col| (row, col));
6           })
7           .map(|(row, col)| {
8               let mut sum = 0.0;
9               for i in 0..other.rows {
10                  sum += self.read_at(&row, &i) * other.read_at(&i, &col);
11              }
12              sum
13          })
14          .collect::<Vec<_>>();
15
16      return Matrix {
17          rows: self.rows,
18          cols: other.cols,
19          data,
20      };
21  }
```

Finally we could also just protect the result matrix with a Mutex. This way we can share it among all threads, but only one thread can access it at a time. This can have a small performance overhead, as we assert memory safety at runtime. Listing 9 shows this approach.

Listing 9. Rust implementation with a Mutex

```rust
    pub fn multiply_faithful_mutex(&self, other: &Matrix) -> Matrix {
        let result_mutex: Mutex<Vec<f64>> = Mutex::new(vec![0.0f64; self.rows * other.cols]);

        (0..self.rows)
            .into_par_iter()
            .flat_map(|row| {
                return (0..other.cols).into_par_iter().map(move |col| (row, col));
            })
            .for_each(|(row, col)| {
                let mut sum = 0.0;
                for i in 0..other.rows {
                    sum += self.read_at(&row, &i) * other.read_at(&i, &col);
                }
                let mut result = result_mutex.lock().unwrap();
                result[row * other.cols + col] = sum;
            });

        return Matrix {
            rows: self.rows,
            cols: other.cols,
            data: result_mutex.into_inner().unwrap(),
        };
    }
```

COMPARE THE TRANSPOSED RUST AND OPENMP IMPLEMENTATION

Figure 1 and Figure 2 show the performance of different implementations of the same algorithm.

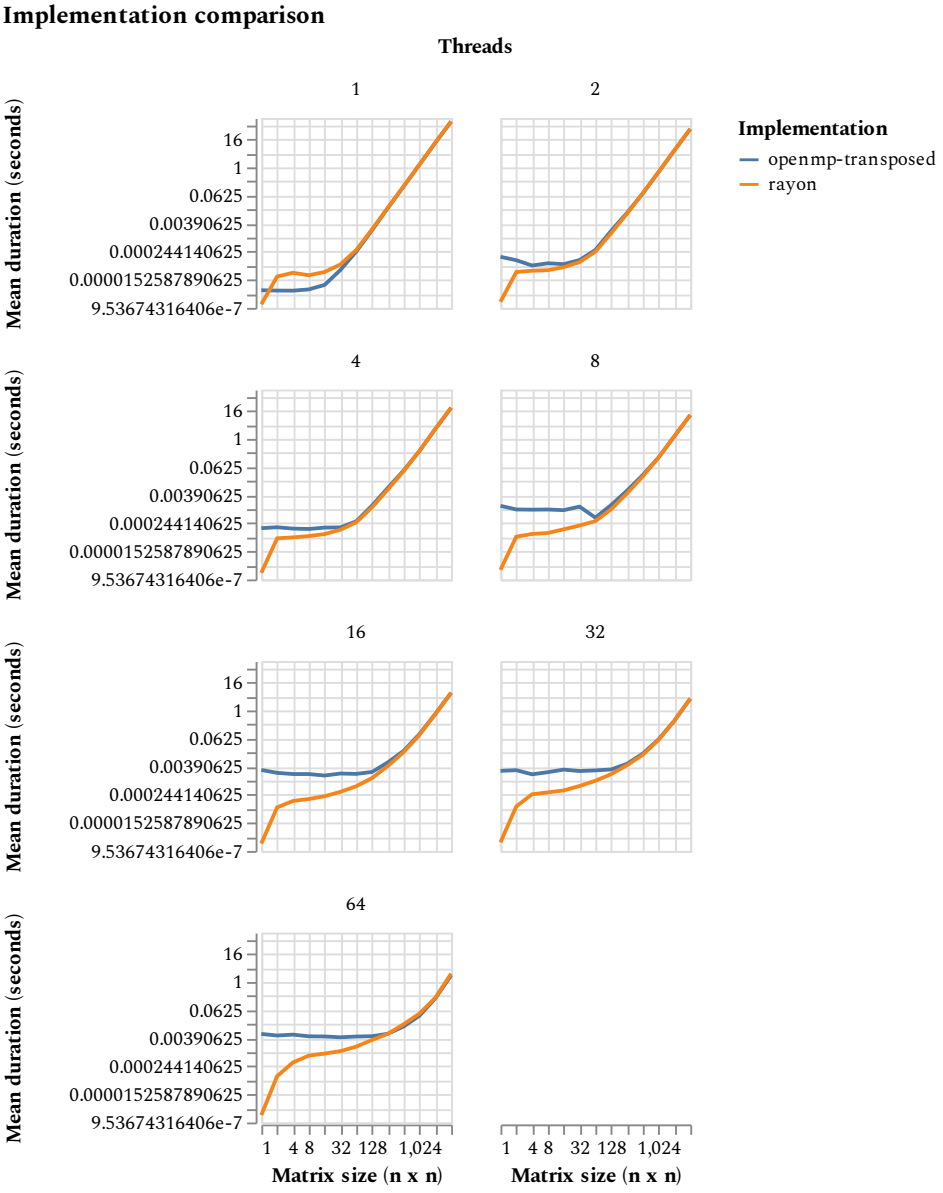

Figure 1. Transposed Rust and openMP implementations

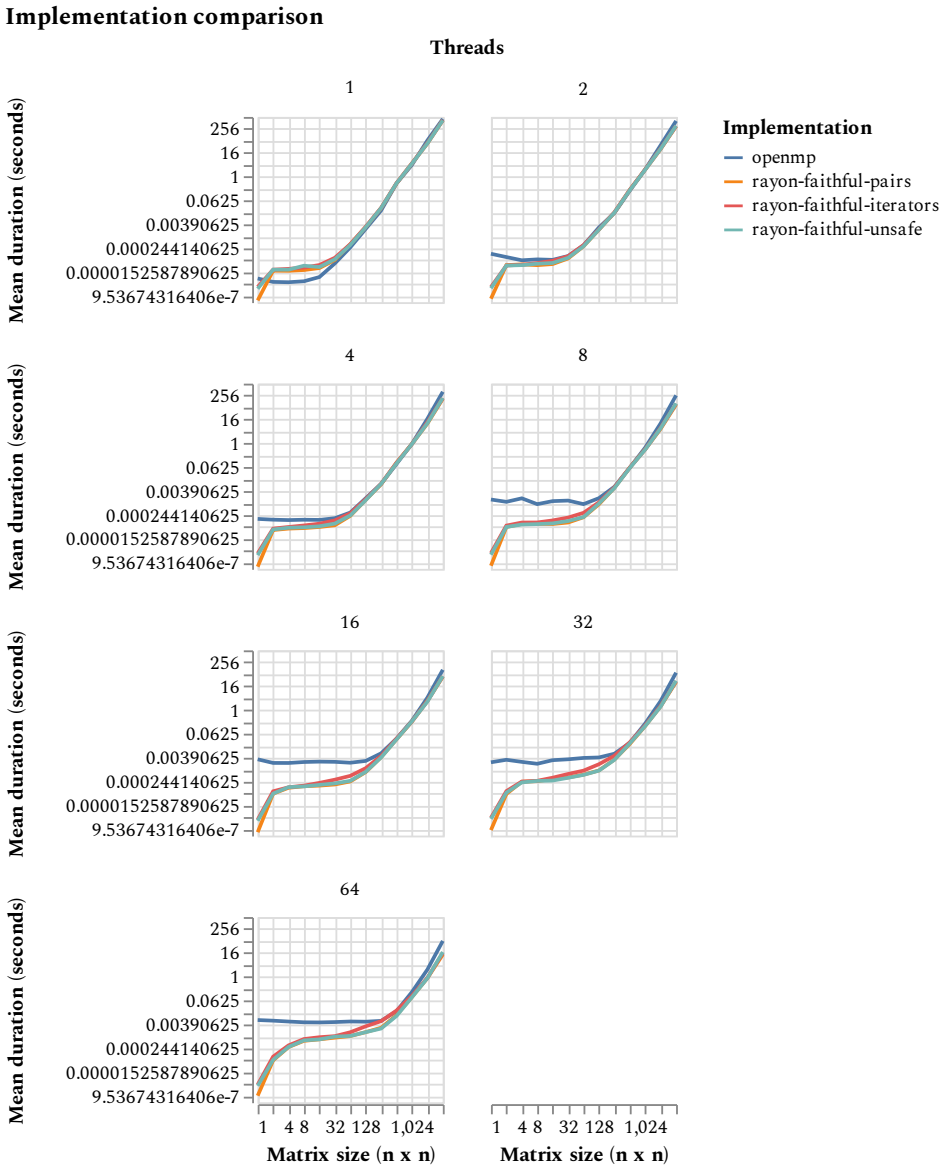

Figure 2. Original Rayon and openMP implementations

Rayon and openMP deliver nearly identical performance in every case, with openMP being slightly faster for single-threaded small matrices and rayon being slightly faster everywhere else. We can see that Rayon allocates its threadpool at application startup and openMP creates it only right before the parallel section. There is probably a setting in openMP and Rayon to change this behaviour, but we already ran our benchmarks. Figure 3 and Figure 4 shows the performance for 64 threads again.

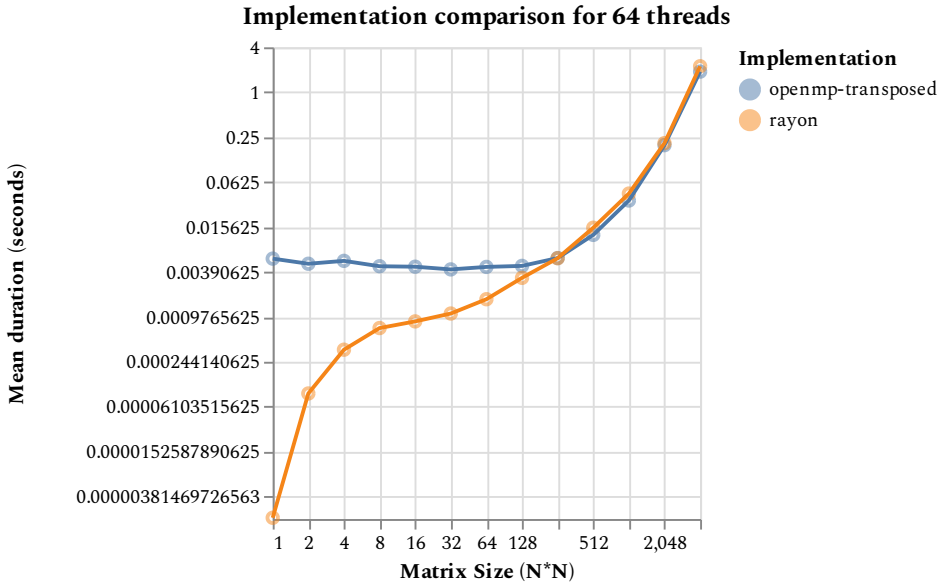

Figure 3. Detailed comparison of the transposing implementations

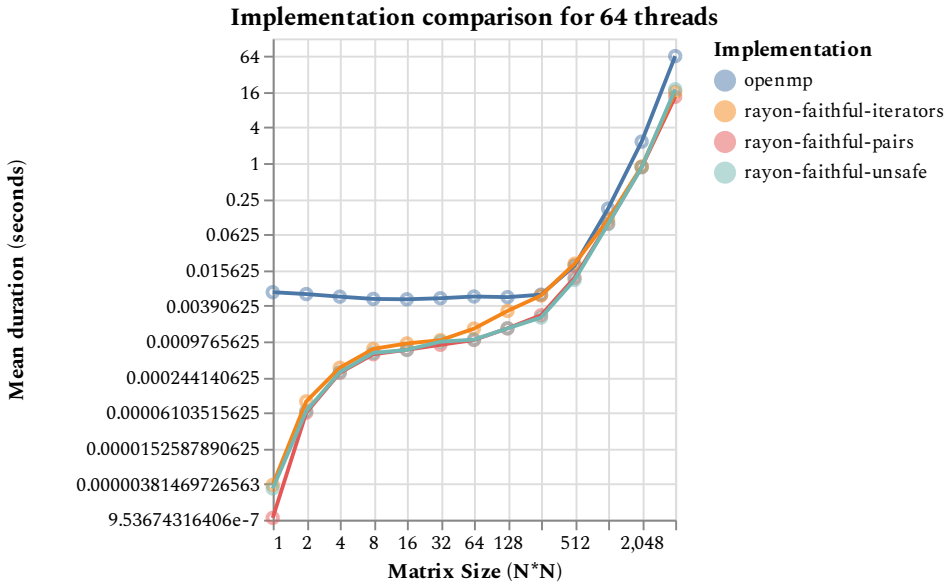

Figure 4. Detailed comparison of the normal implementations

COMPARING THE TRANSPOSING AND NON-TRANSPOSING IMPLEMENTATIONS

Figure 5 shows the performance of the different algorithms. The transposing algorithms seem to be slower for small matrices but from a given size onward they are faster. Mutexes are slower, surprisingly they approach the performance of the other non-transposing implementation for large matrices. Probably because the time spend calculating the dot product becomes much larger than the time spend locking and unlocking the mutex.

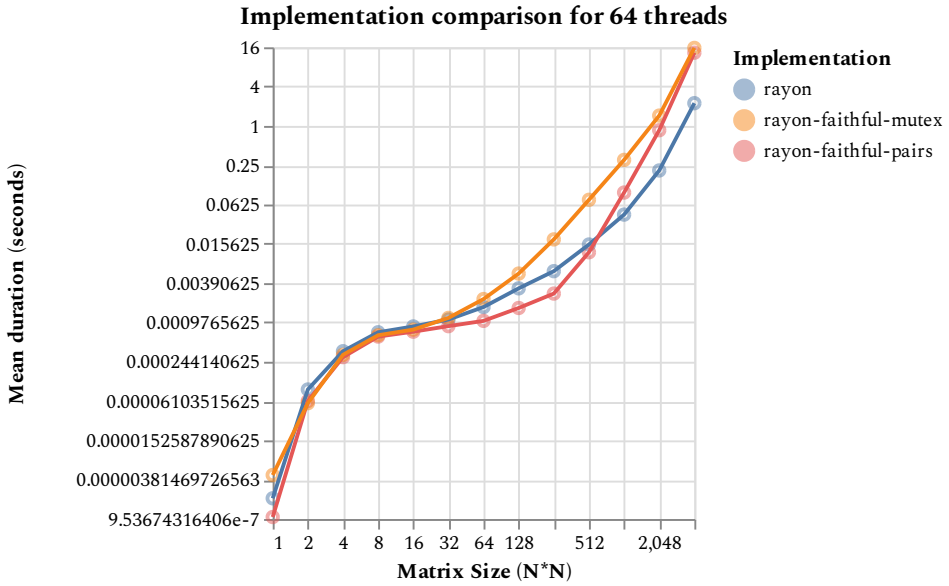

Figure 5. Detailed comparison of the transposing vs normal vs mutex implementations

CONCLUSION

Transposing the matrix is an easy way to gain a bit of performance here. Performance of CPP and rayon is comparable.

FUTURE WORK

We did not look at SIMD instructions, we could probably gain a bit of performance by using them.

Our benchmarks included startup overhead for openMP, but did not for rayon.

We did not have enough time to calculate efficiency and speedup.