Recomender System

Pedro Mendes (97144), Filipe Lucas (78775), Ricardo Pereira (86506)

April 3, 2020

1 Data Structure

There were multiple approaches to store matrix A, the most basic one was storing the values in a full matrix however as matrix A is sparse we opted to store it as compressed sparse row (CSR) matrix. A CSR only stores non-zero values in a single array making this a good choice in terms of caching since all the values are stored in adjacent memory positions.

The other matrices were left as full matrices since they needed to be indexed in $O(1)$ time and a CSR does not permit this.

2 Matrix B multiplication

A new matrix B needs to be calculated in every iteration, however, since the only positions of matrix B that are needed in each iteration are the ones that coincide with the non zero elements of matrix A, we optimized the code such that only these positions are calculated.

Since the full matrix B is needed for the final output, after the iterations are done the full matrix B is calculated.

3 Solution

The approach we took for parallelization was quite methodical. First we opted to optimize the sequential version as far as we could. We experimented with using *openMP* in the inner loops but it always made the program very slow, even slower than the sequential version.

3.1 Matrix B

As mentioned before there are two versions of the calculation. The simplest one is the `full` one. Where each thread gets a distinct row and therefore there are no race conditions possible.

```c
#pragma omp parallel for
for (size_t i = 0; i < l->rows; i++)
    for (size_t j = 0; j < r->columns; ++j)
        for (size_t k = 0; k < l->columns; ++k)
            // only write to shared memory: two threads never have the same i
            *MATRIX_AT(b, i, j) += *MATRIX_AT(l, i, k) * *MATRIX_AT(r, k, j);
```

For the second version of we also have guarantees of no data races because each thread gets a chunk of the indexes of A that do not overlap.

```c
Item const* const end = a->items + a->current_items;
#pragma omp parallel for
for (Item const* iter = a->items; iter < end; ++iter)
    double bij = 0.0;
    for (size_t k = 0; k < l->columns; k++)
        bij += *MATRIX_AT(l, iter->row, k) * *MATRIX_AT(r, k, iter->column);
    // only write to shared memory: two threads never have the same iter->row
    *MATRIX_AT(matrix, iter->row, iter->column) = bij;
```

3.2 Matrix R

The calculation of matrix R was the easiest to parallelize, since, from the sequential version, the top level for loop was over the rows of R we could distribute theses rows over the threads and since the only write to shared memory happens to R in distinct rows we have the guarantee that there are no race conditions.

```
#pragma omp parallel for
for (size_t k = 0; k < r->rows; k++)
    Item const* iter = // ...
    Item const* const end = // ...
    for (size_t column = 0; column < r->columns; column++)
        double aux = // sum of deltas of this column of A
        // only write to shared memory: two threads never have the same k
        *MATRIX_AT(aux_r, k, column) =
            *MATRIX_AT(r, k, column) - alpha * aux;
```

3.3 Matrix L

Matrix L was also simple to parallelize once we figured out how to split the rows of A between the threads. The outer loop is over the non empty rows of A so we had to change this to be over all the possible rows of A, empty or not. This would mean a loss of performance for the sequential version but it was needed in order to split sets of rows between the threads. What this translates to is a while loop that makes sure the A iterator is synchronized with the outer loop.

In terms of race conditions the same reasoning applies. The only write to shared memory depends on a variable that is private to each thread.

```
Item const* iter = a->items;
Item const* const end = iter + a->current_items;
#pragma omp parallel for firstprivate(iter) schedule(guided)
for (size_t row = 0; row < a->n_rows; ++row)
    size_t row_len = a->row_lengths[row];
    while (iter != end && iter->row < row) iter += a->row_lengths[iter->row];
    for (size_t k = 0; k < l->columns; k++)
        double aux = // sum of deltas for this row of a
        // only write to shared memory: two threads never have the same row
        *MATRIX_AT(aux_l, row, k) = *MATRIX_AT(l, row, k) - alpha * aux;
    iter += row_len;
```

4 Speed up

Disregarding the small instances, the speed up obtained, using 4 physical cores, was close to 3x the serial version. Sometimes even reaching 3.5 times speed up on some instances.

As for the scheduling, using guided brought a better speed up in some cases, but also made some instances slower. The difference, however, was negligible.

The instances where guided was slightly better were the ones where the number of non zero elements in matrix A were not evenly distributed between rows, so we attribute the minimal difference to the fact that this unbalance of was also minimal in all instances.

For these reasons we decided it was best to keep the guided version as it is more resilient to possible unbalanced instances without incurring a significant cost to the balanced ones.

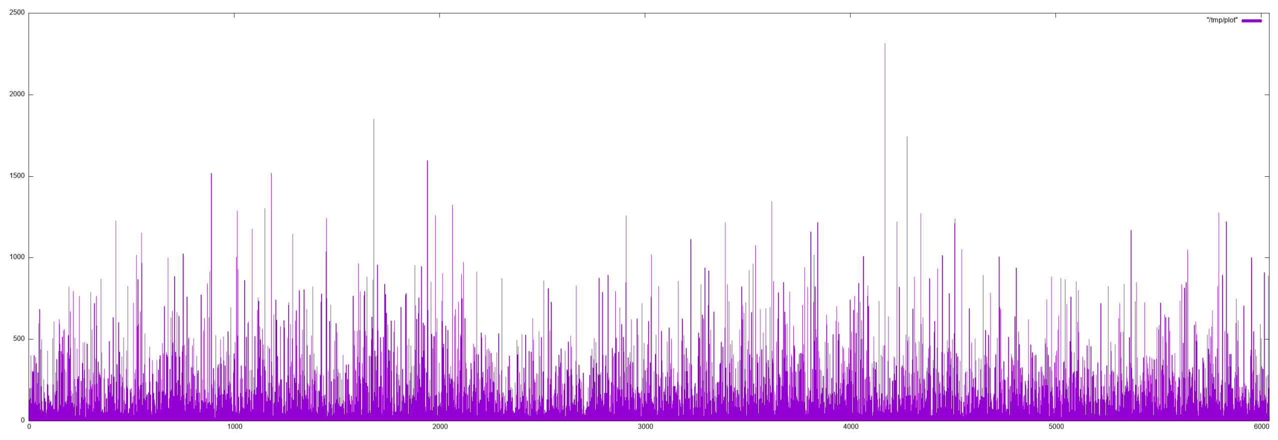


Figure 1: Number of non zero elements in each row of Matrix A. Example where some imbalance can be seen.

It is also important to note that, in bigger instances, the parsing creates a significant overhead, making the speed up less noticeable.

5 A note on hyper threading

As can be seen in Appendix A enabling hyper threading on 4 physical cores has an interesting effect. When compared with the version with 4 threads it brings little to no benefit, but when compared to forcing 8 threads on 4 physical cores it does improve the performance.

Sorted from best to worse

- 8 threads on 4 hyper threaded cores;
- 4 threads on 4 cores;
- 8 threads on 4 cores;

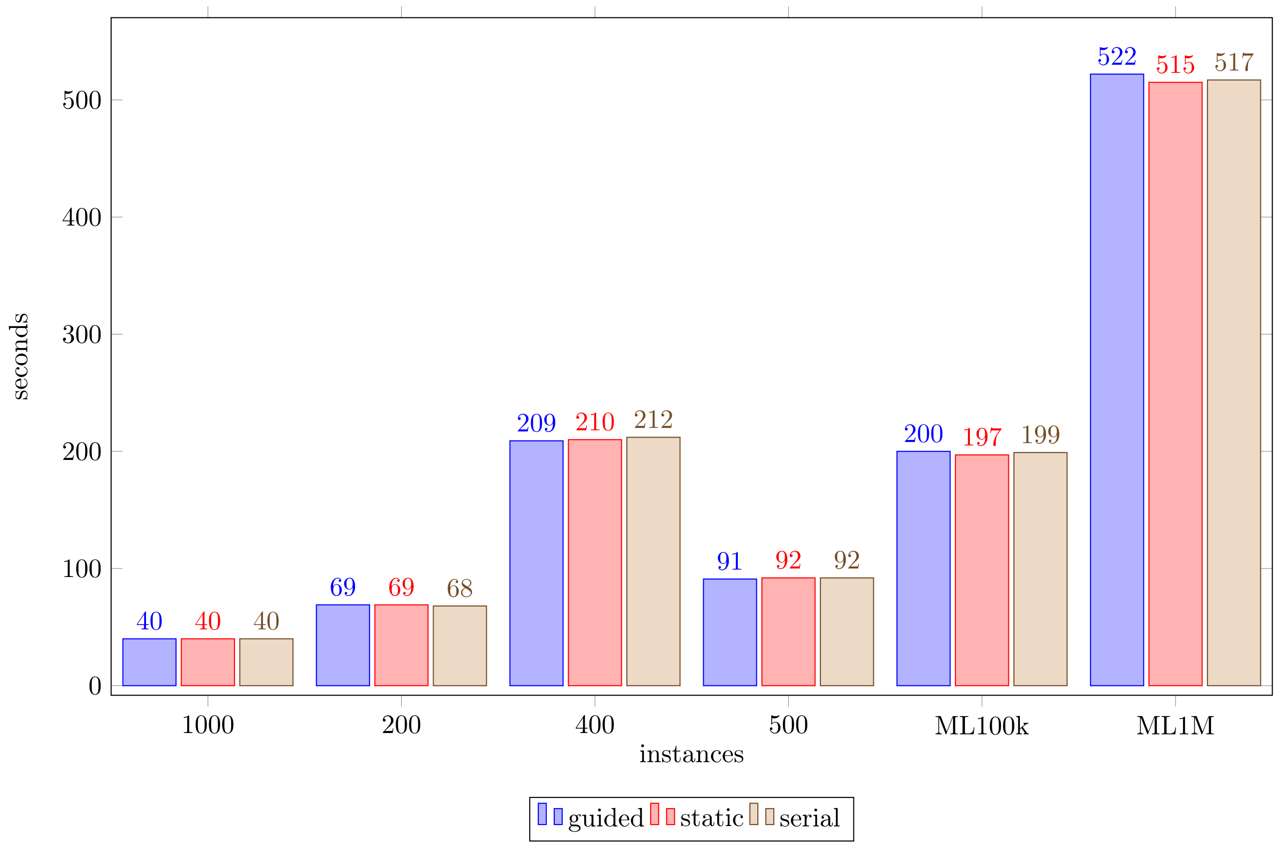
6 Future Work

In a future implementation the parser could be made parallel to see if could bring further improvements. This seems unlikely as the chunks of the parsed input need to be aggregated at the end of the parsing and this could prove to be too much to compensate the overhead it might bring to smaller instances.

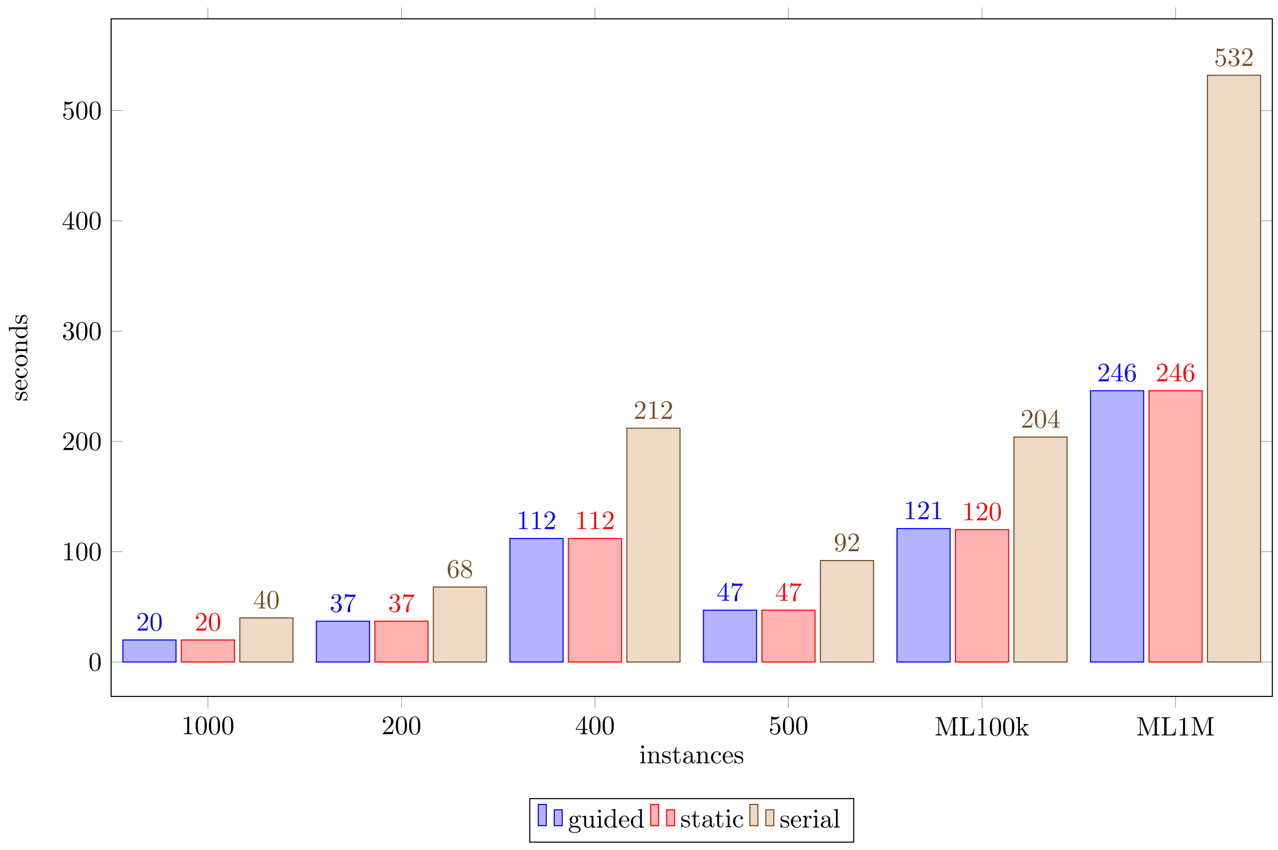
A Benchmarks

These benchmarks were run on an Intel i5-8250U @ 3.400GHz with 4 physical cores with hyper threading disabled as to not interfere with the measurements. The last graph shows the results with hyper threading enabled for comparison.

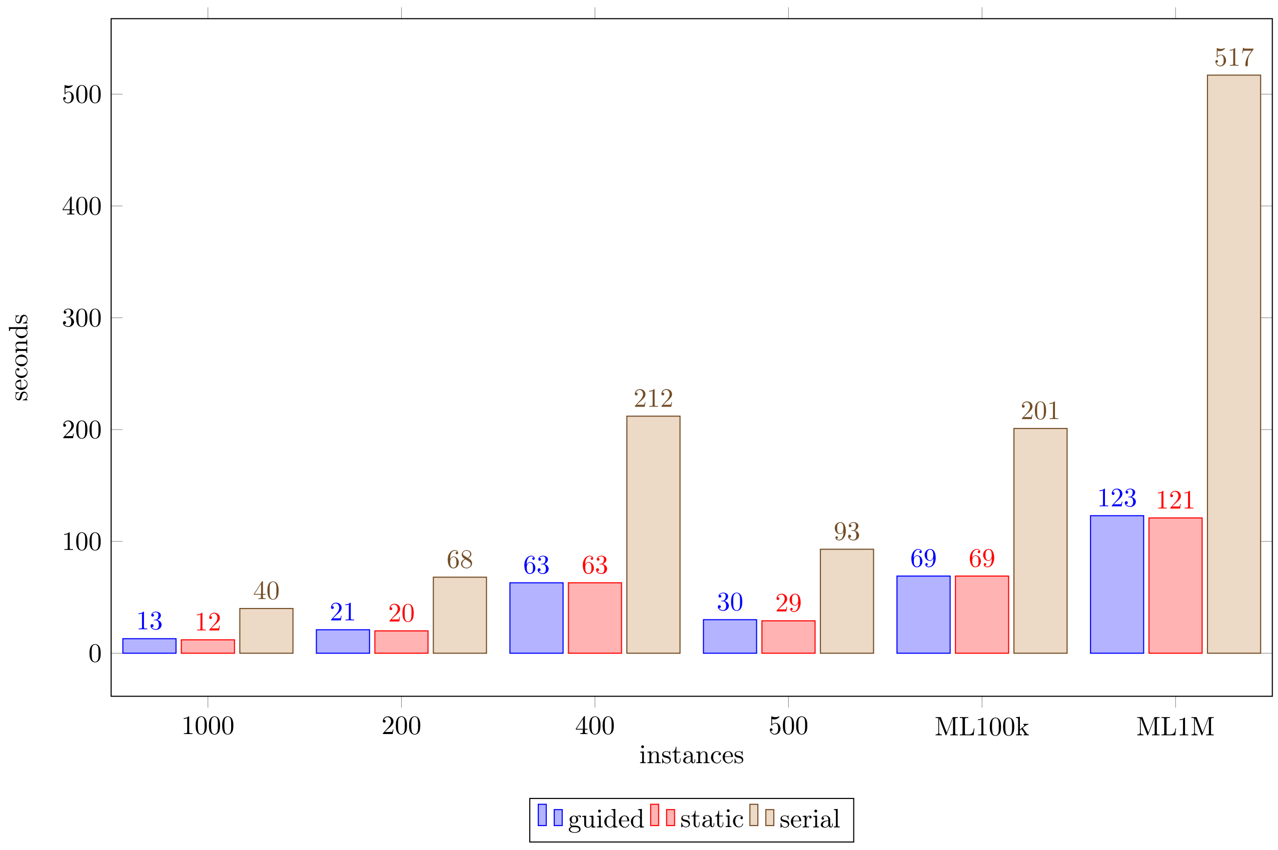
A.1 Number of threads equal to 1



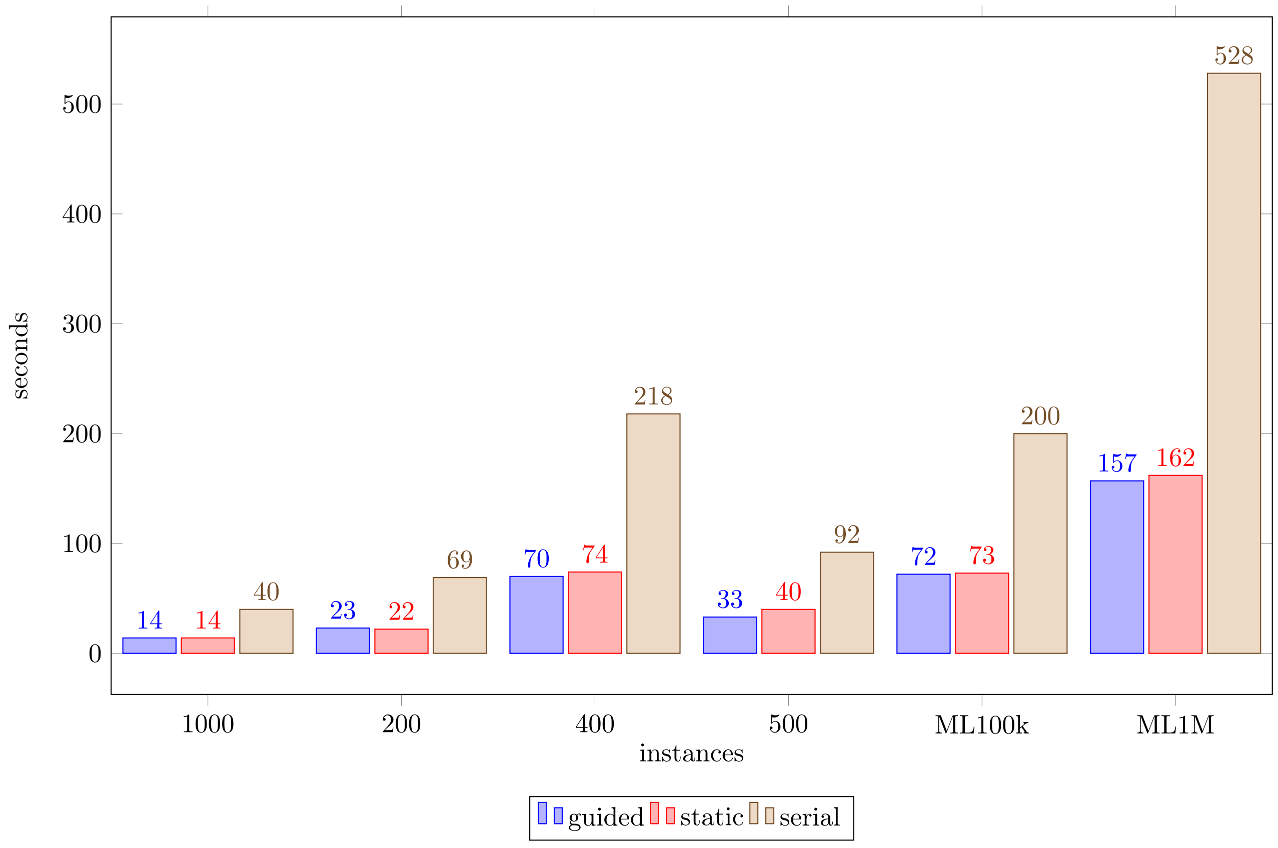
A.2 Number of threads equal to 2



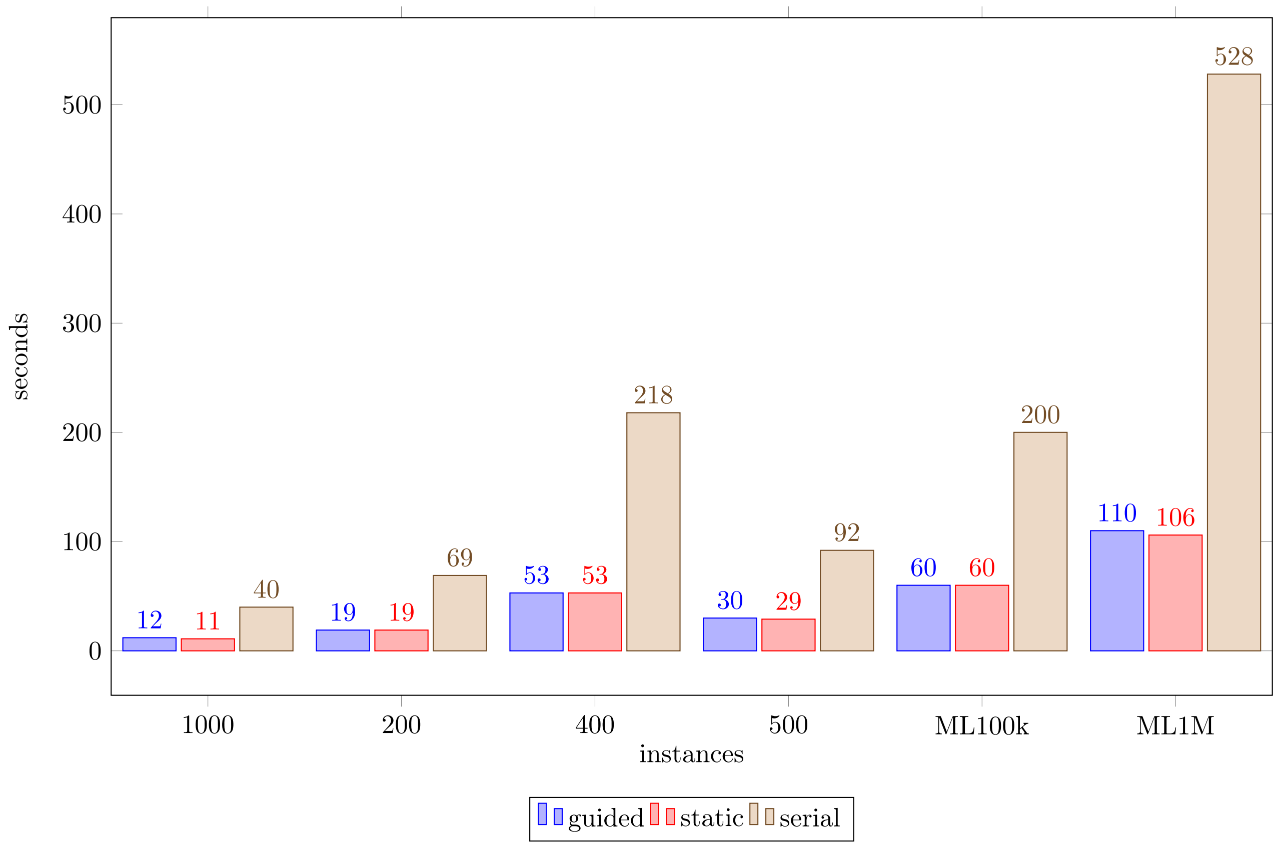
A.3 Number of threads equal to 4



A.4 Number of threads equal to 8



A.5 Number of threads equal to 8 with hyper threading


seconds
ML100k
ML1M
instances
guided
static
serial