Recomender System

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

May 20, 2020

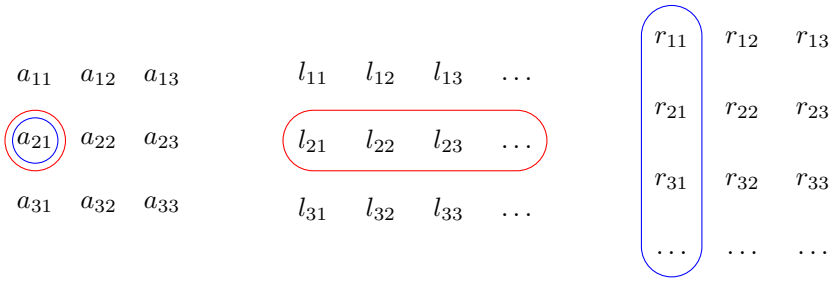
1 Serial

There was no change in the serial version, as it is working as intended. One important thing to note is that the MPI version uses the serial version when the number of lines or columns in matrix A is less than the number of processors, as it wouldn't make sense to use MPI for these cases.

2 Decomposition

When discussing how to perform the decomposition, the first implementation was to transpose L so that it would be indexable as (k, i), which then aligned with accessing R as (k, j). This had the advantage that matrices L and R could be split among nodes and the calculations of $L^{(t+1)}$ and $R^{(t+1)}$ could be made 100% independently. But this came with a huge disadvantage: Matrix B could not be computed in parallel, all nodes had to send their calculations of their slices of L and R to node 0 and this one then computed matrix B alone and sent it back. This had so much overhead that the sequential version could many times be faster. Instead of trying to make it work we decided to rewrite the program and split matrix A.

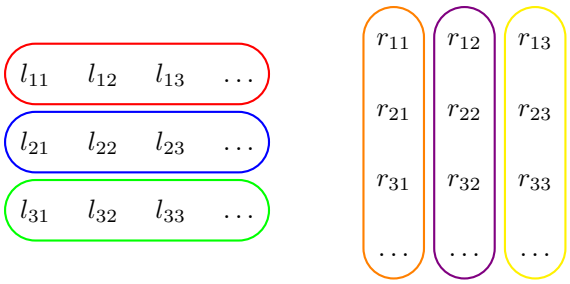
Matrix A was then split in a checker board fashion, each node getting a zone of A and the corresponding slices of L and R. The design was oriented to facilitate the calculations of matrix B. This means that for some range of lines of a, $[i, i'[$, and a range of columns, $[j, j'[$, assigned to node X, this node will also have lines of L from i to i' and all it's columns, and columns of R from j to j' and all of it's lines.



With this decomposition the calculations of matrix B can be done independently by the nodes, on the other hand this means that communication had to be shifted somewhere else, this place being in the calculations of $L^{(t+1)}$ and $R^{(t+1)}$. The way it works is since these nodes don't have enough of matrix A to calculate their parts of L or R, they do as much as possible and in the end we make use of `MPI_Allreduce` to sum all their values.

2.1 MPI Reduction

Taking advantage of the checker board layout each node only has to reduce with nodes that share their slice of L/R.



If we give each line and row of L and R a colour, we can than define a node as a combination of 2 colours, and then it only has to communicate with nodes that share a colour with it. For example, given the following table

R \ L	red	blue	green
orange	0	1	2
violet	3	4	5
yellow	6	7	8

We can see that node 0 only needs to communicate with nodes 1 and 2 to calculate matrix L and nodes 3 and 6 to calculate matrix R.

2.2 Memory

Another huge advantage of this decomposition is that at no point (excluding the very end of the algorithm) does any node have in memory an entire matrix.

In total the amount of spent memory is relative to the amount of processes, each process will at most have $\sqrt{\#processors}$ lines of matrix L and $\sqrt{\#processors}$ rows of matrix R. Of matrix `A` and `B` only the the areas defined previously (Section 2) will be stored.

At the end of the computation, all matrices except B are freed and node 0 gathers the parts of matrix B the other nodes calculated, so it can finally print the output.

3 Load Balancing

In the cases that the lines and columns are not dividable by the number of nodes the the first X nodes take one more line/column where X is the remainder of the division. This has the convenient property of helping during parsing, since node 0 has always the largest area of matrix A it can always accumulate the values that belong to other nodes without having to overcommit memory.

4 Performance

Benchmarks were made for the three biggest instances: `inst1000-80000-20-10-1000`, `inst20000-10000-40-2-50` and, `inst60000-2000-200-10-20`. Since the program can only make use of numbers of processors whose square root is an integer, the performance gains from using 32 nodes over using 16 are not great, on the other hand when bumping the number to 64 nodes, since 64 has an integer root, the performance gains are much better. The times for these benchmarks can be seen in Appendix B.

To show a more realistic performance graph we also benchmarked with the number of nodes with integer square roots. These can be seen in Appendix B. One interesting note is that the performance of 36 and 49 is worse then the performance of 16. We suspect this might be because the machines used have 4 cores each, so powers of 2 create more even distributions, improving comunication performance.

5 Future Work

There are 2 major flaws in the implementation that, given enough time, could be fixed.

The first is that the number of allowed nodes is very strict. Only numbers whose square root is an integer can be used, excess nodes are discarded.

The other improvement that could be made is node 0 not aggregating the elements of B from the other nodes as this might be too much for some instances.

A Benchmarks: Powers of two

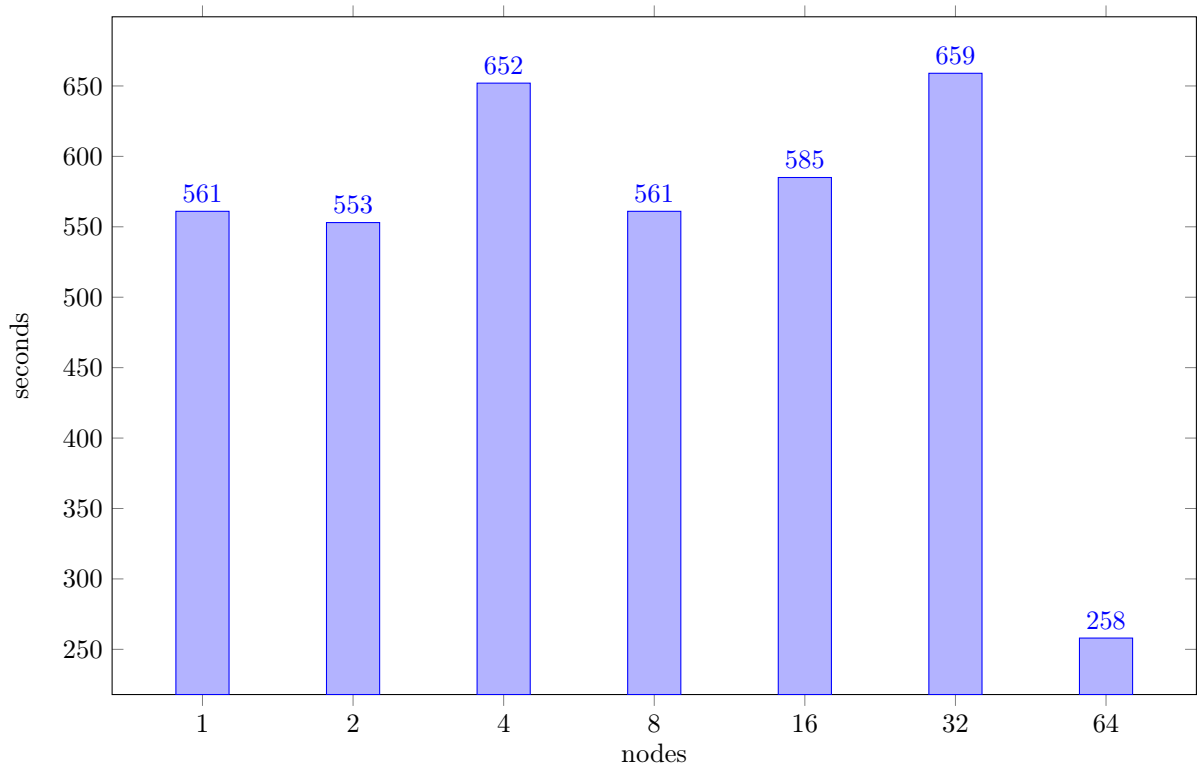


Figure 1: inst1000-80000-20-10-1000

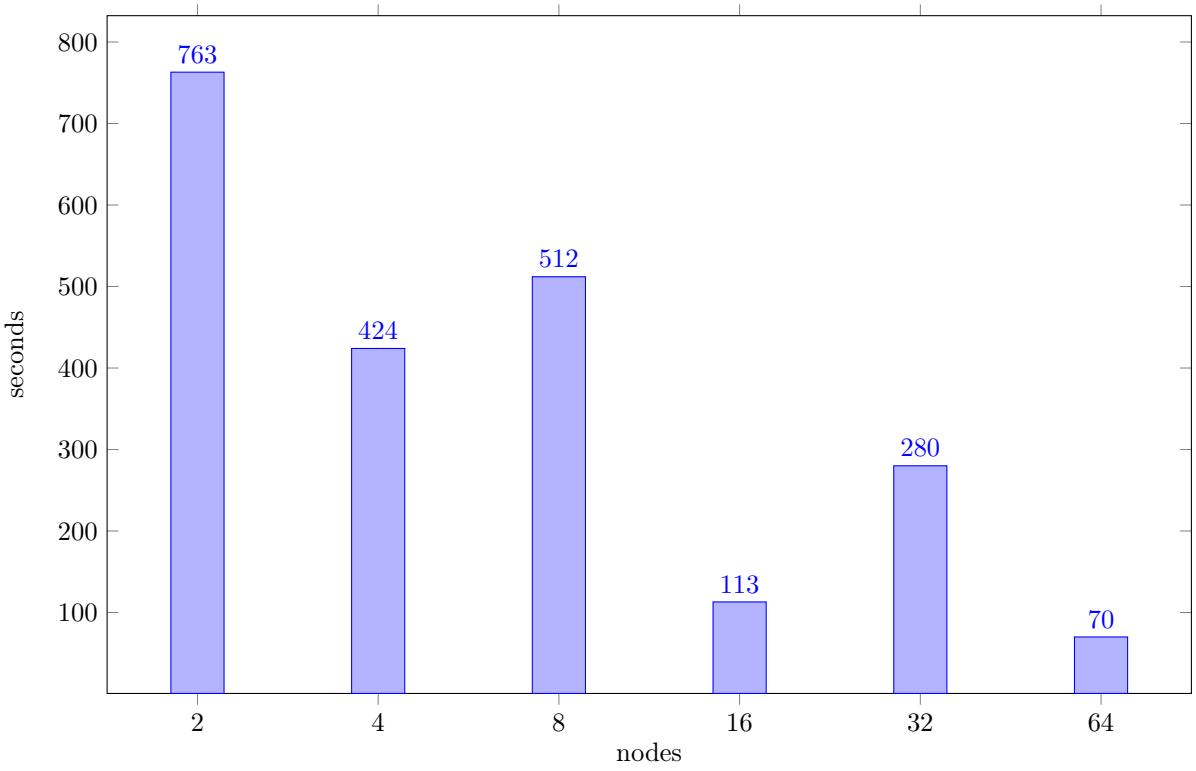


Figure 2: inst20000-10000-40-2-50

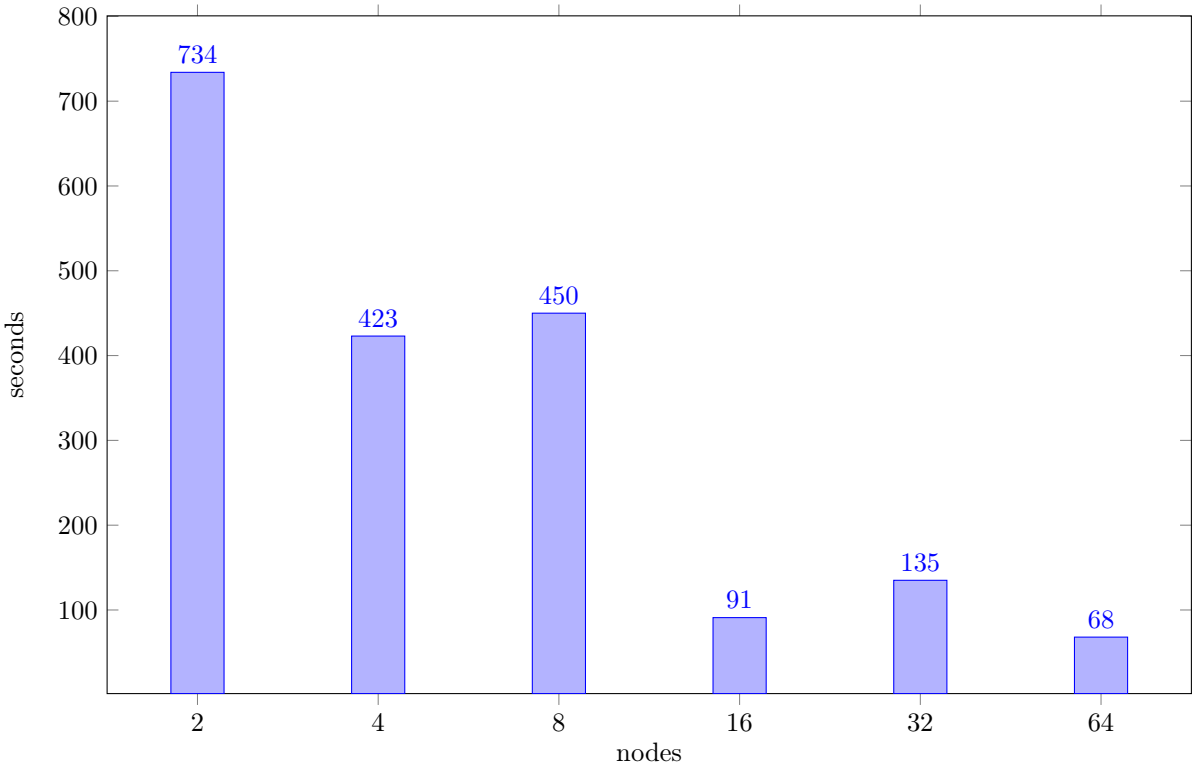


Figure 3: inst60000-2000-200-10-20

B Benchmarks: Integer Square Roots

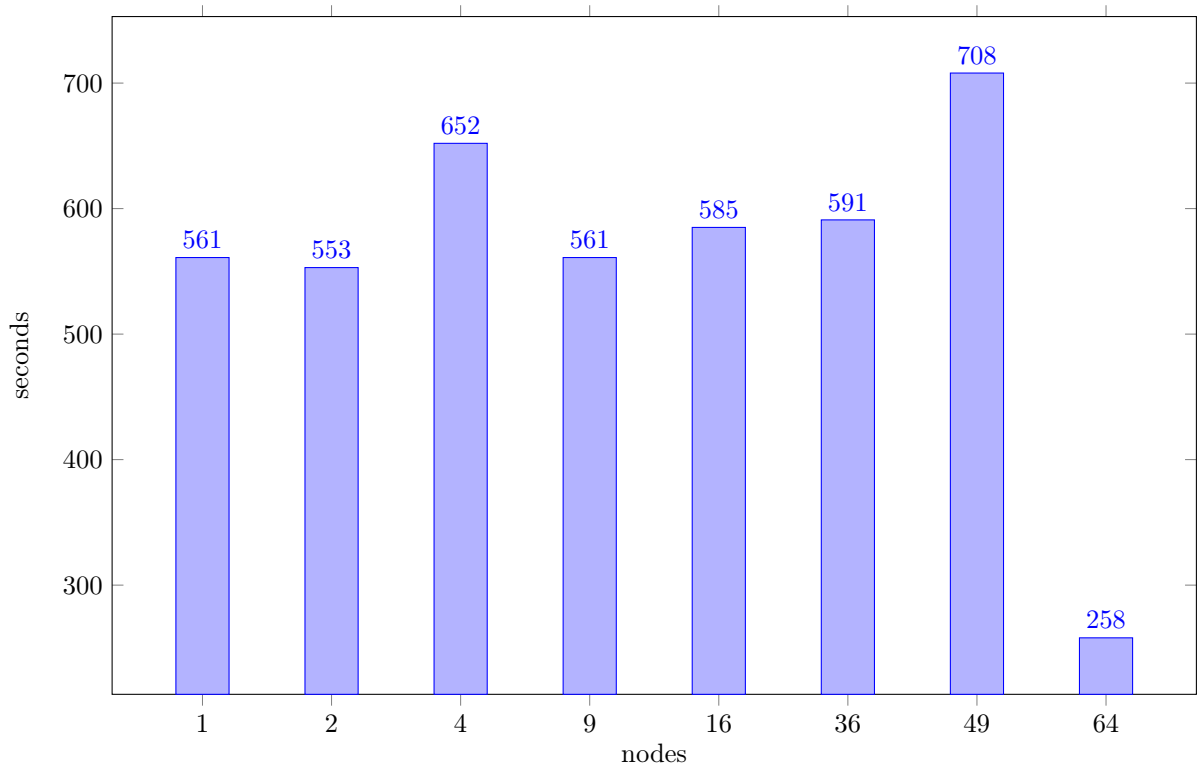


Figure 4: inst1000-80000-20-10-1000

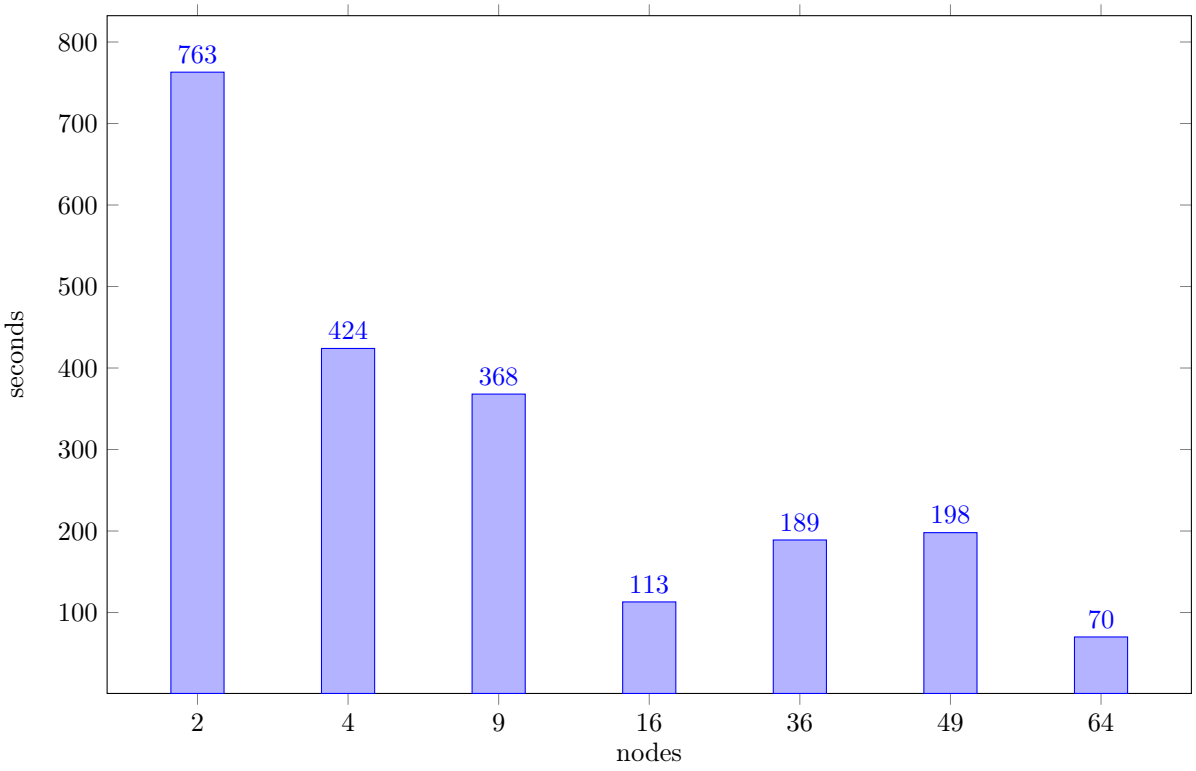


Figure 5: inst20000-10000-40-2-50

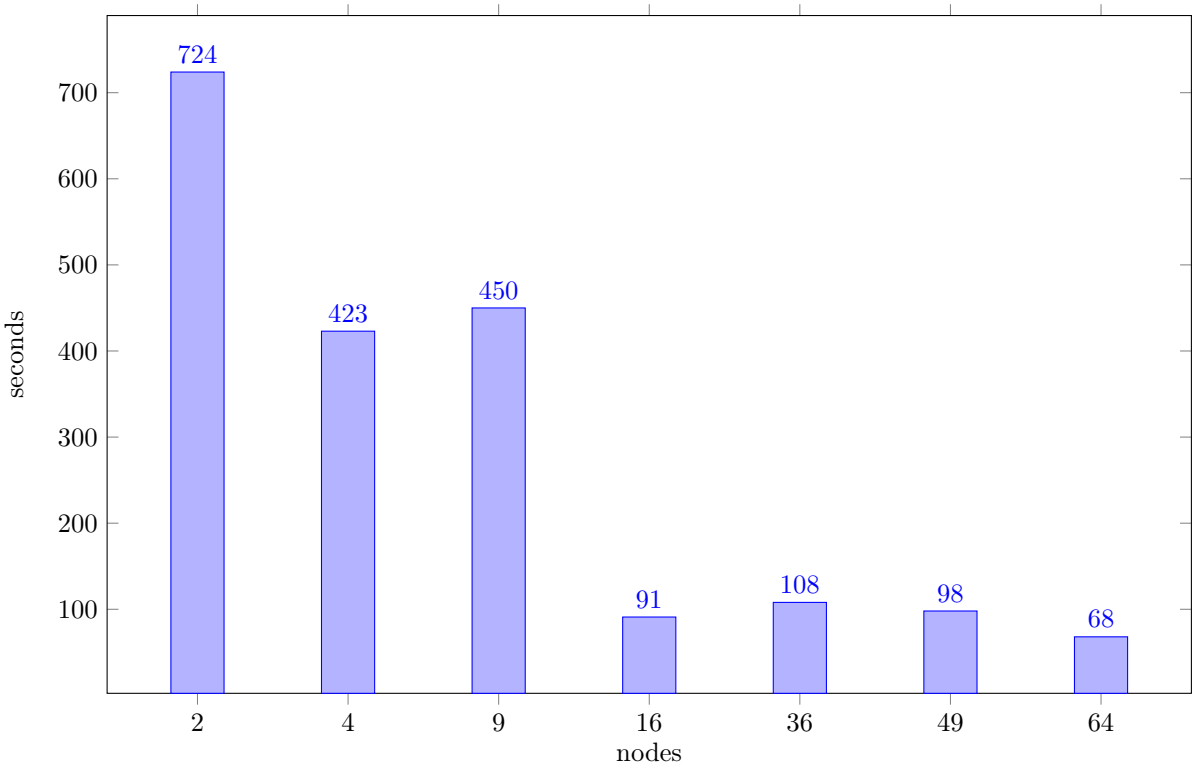


Figure 6: inst60000-2000-200-10-20