Neural network training

Gal Lindič, Ožbej Golob, Žan Jonke

Neural network training

- Forward pass algorithm
- Loss calculation
- Backward pass algorithm
- The network has 1 hidden of variable size layer
- We experimented with varying batch size and hidden layer size

Dataset

- US Adults dataset
- 32561 entries (24421 train and 8140 test)
- 14 features and 1 class variable (sex)

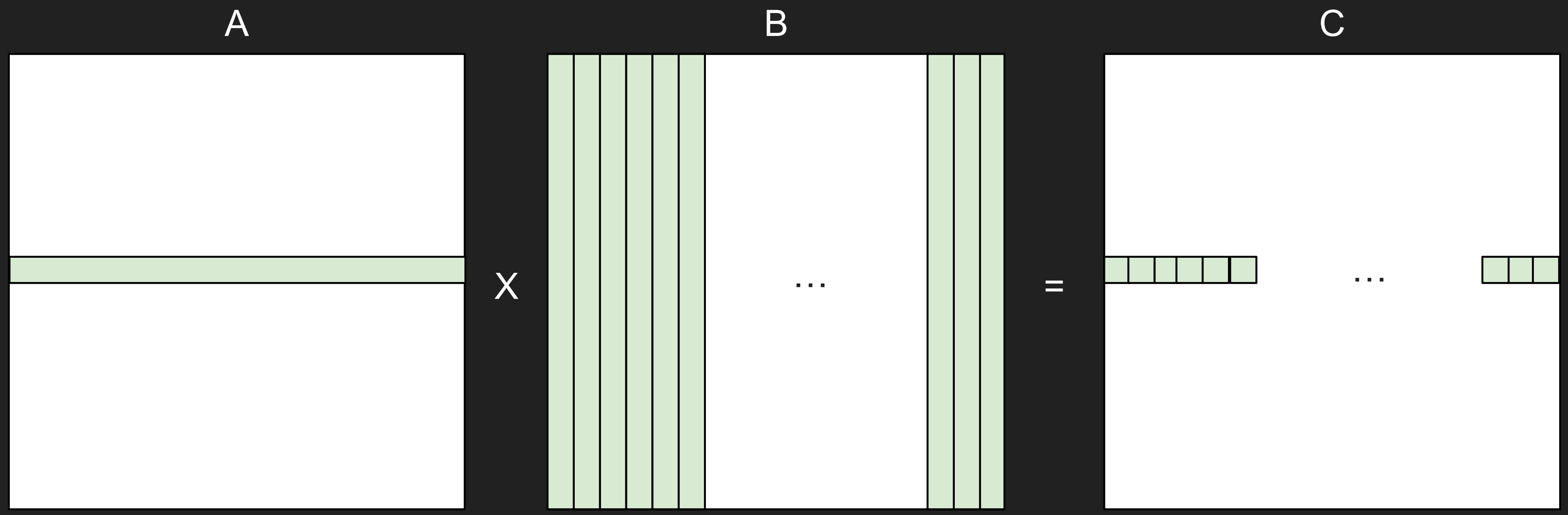
	age	workclass	fnlwgt	education	education-num
0	39	State-gov	77516	Bachelors	13
1	50	Self-emp-not-inc	83311	Bachelors	13
2	38	Private	215646	HS-grad	9
3	53	Private	234721	11th	7
4	28	Private	338409	Bachelors	13

	marital-status	occupation	relationship	race	sex
0	Never-married	Adm-clerical	Not-in-family	White	Male
1	Married-civ-spouse	Exec-managerial	Husband	White	Male
2	Divorced	Handlers-cleaners	Not-in-family	White	Male
3	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male
4	Married-civ-spouse	Prof-specialty	Wife	Black	Female

	capital-gain	capital-loss	hours-per-week	country	salary
0	2174	0	40	United-States	<=50K
1	0	0	13	United-States	<=50K
2	0	0	40	United-States	<=50K
3	0	0	40	United-States	<=50K
4	0	0	40	Cuba	<=50K

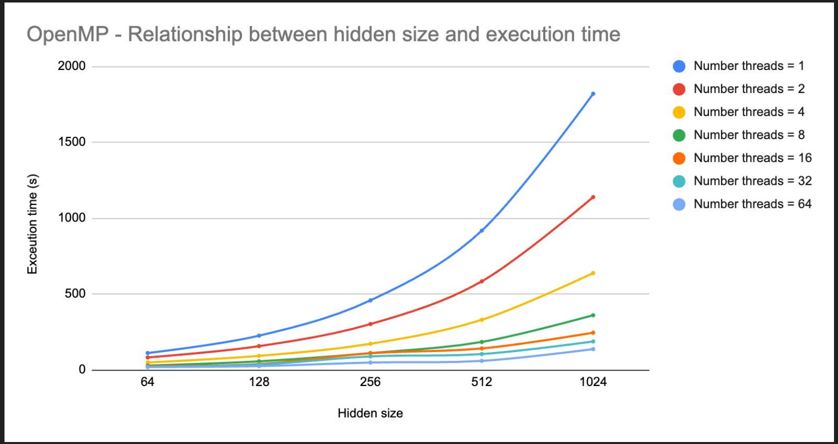
OpenMP implementation

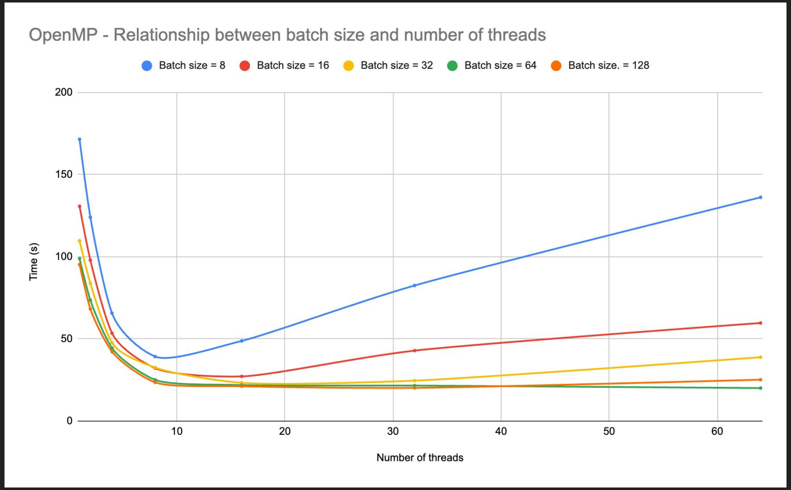
- Parallelization done row-wise ie. one thread per row
    - In matrix multiplication one thread per dot product
    - In layer activation one thread per row
    - In loss calculation one thread per sample
    - In hadamard product one thread per row

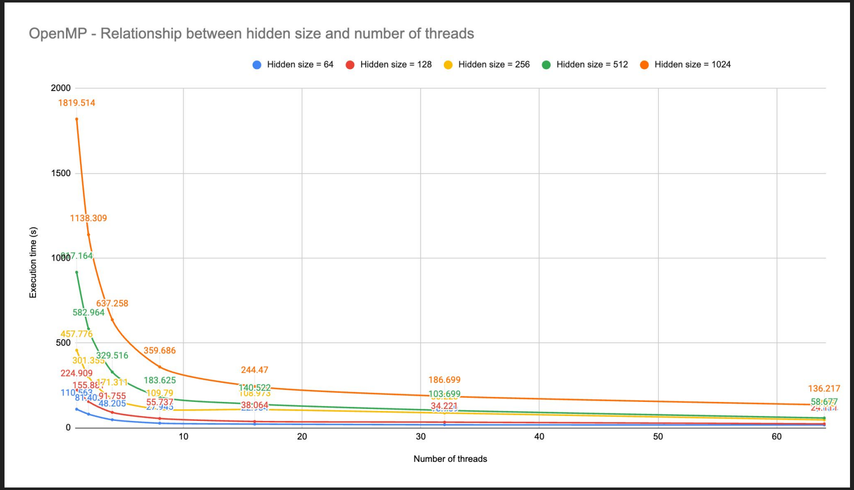


OpenMP benchmarking

- Increasing the number of threads pays off for larger hidden size
- For small batch sizes computing on large
- number of cores does not pay of

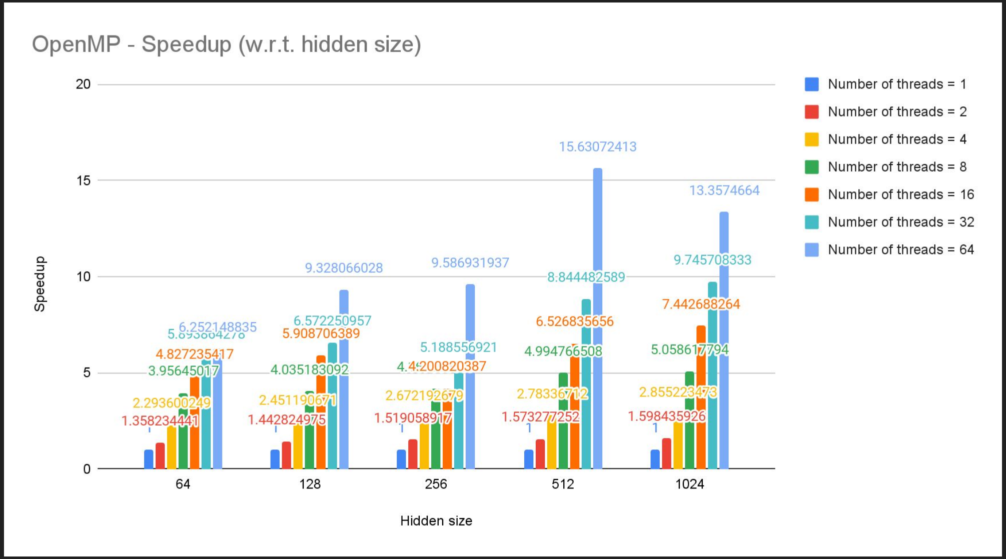


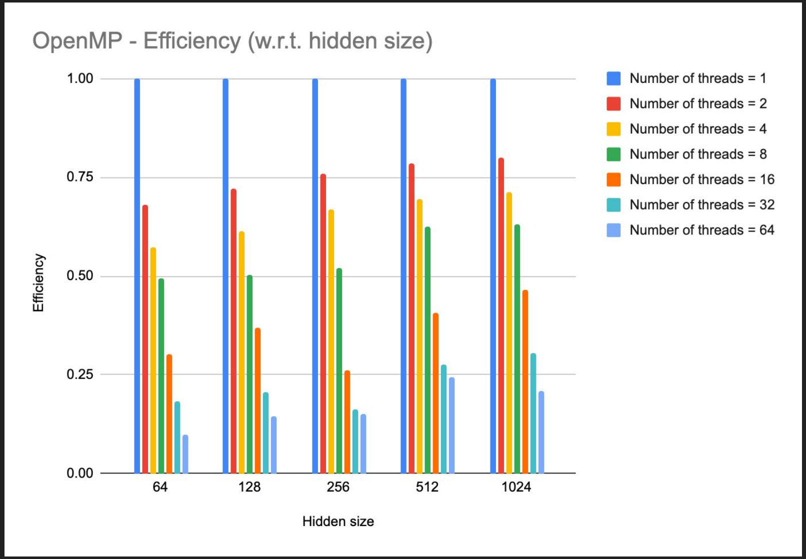




OpenMP benchmarking

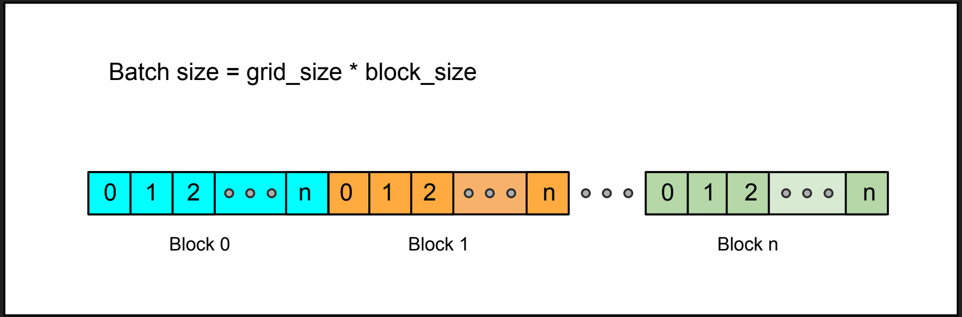
- Noticeable increase in speedup for larger hidden size
- Efficiency deteriorates





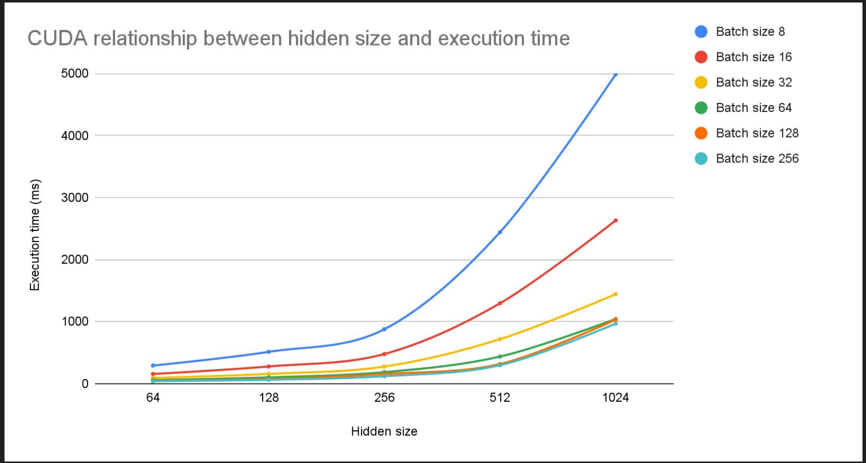
CUDA implementation

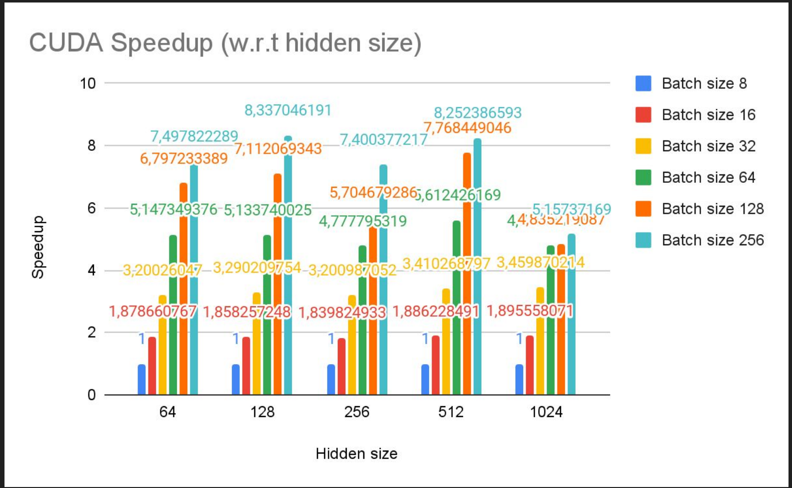
- Multiple kernels that handle different calculation tasks
- Dataset and MLP matrices stored in global GPU memory
- Number of threads in a kernel equals batch size
- Parallelization done row-wise
  - Each thread handles one row from batch data
  - Cases when there are matrices with more rows than threads, we divide the rows equally among threads

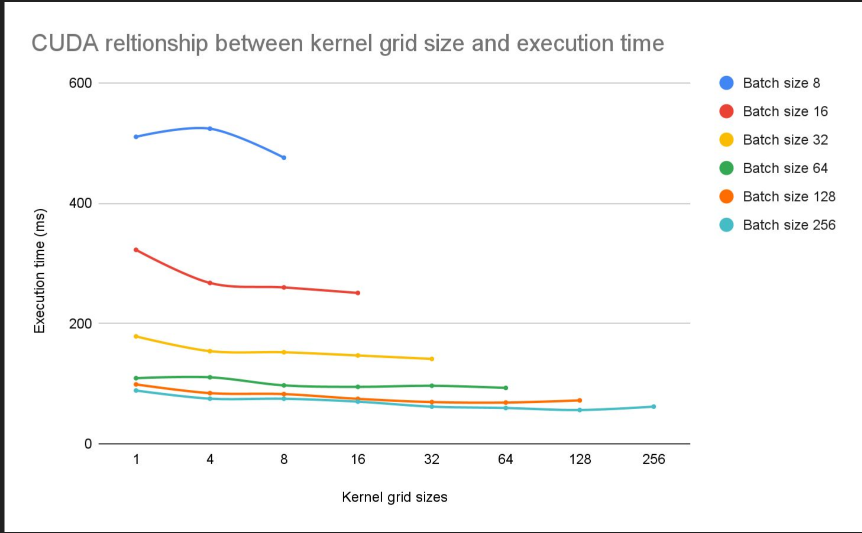


CUDA benchmarking

- Increase in batch size greatly improves execution time
- Having higher grid sizes (more blocks in kernel) slightly improves execution time

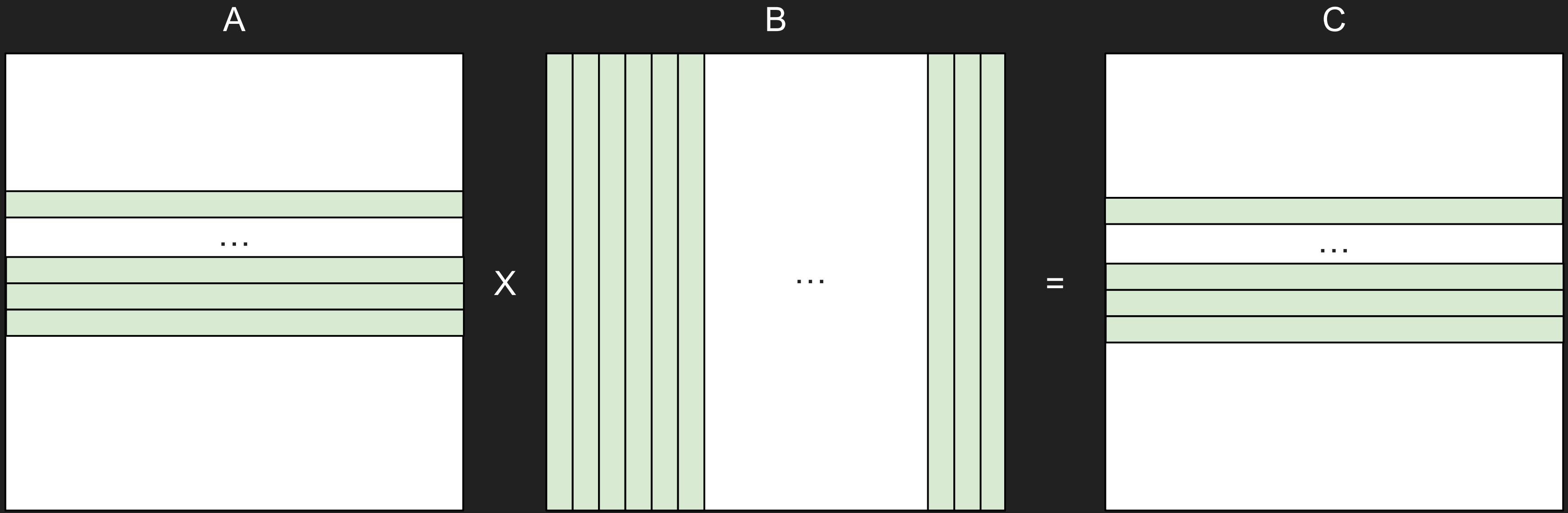






MPI implementation

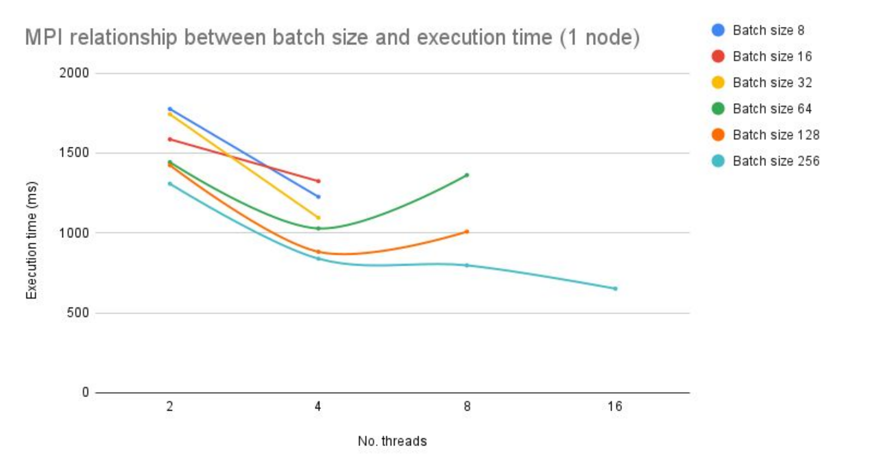
- Parallelization done by splitting into smaller problems
  - MPI_Scatterv to distribute the uneven load (matrix A)
  - MPI_Bcast to broadcast the matrix B
  - MPI_Gather to consolidate the results into matrix C
  - Other computations follow the same pattern (adding bias, hadamard product …)

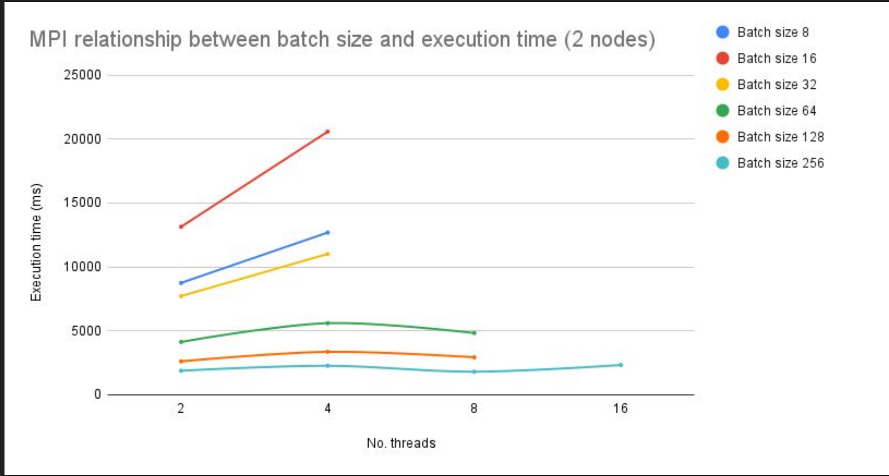


```c
void distribute_matrix(double ***M, int *rows, int cols, int * offset_rows, int myid, int procs){
    MPI_Status  status;

    int rows_per_process = rows[0] / procs; // No. of rows per process
    int remaining_rows = rows[0] % procs; // Rows not evenly distributed
    int start_row, end_row; // Start and end rows of each process
    int srows; // No. of rows to be sent to each process
    int * sendcounts; // Array of srows
    int * displs; // Array of displacements

    sendcounts = (int *)malloc(procs*sizeof(int));
    displs = (int *)malloc(procs*sizeof(int));

    for(int i = 0; i < procs; i++){
        start_row = i*(rows_per_process*cols);
        end_row = (i+1)*(rows_per_process*cols);
        if(remaining_rows > 0){
            end_row += cols;
            remaining_rows--;
        }

        srows = end_row - start_row;
        sendcounts[i] = srows;
        if(i == 0){
            displs[i] = 0;
        } else {
            displs[i] = displs[i-1] + sendcounts[i-1];
        }
    }

    int recvbuf_length = sendcounts[myid];
    *offset_rows = displs[myid] / rows[0];
    *rows = recvbuf_length / cols;

    double ** m = alloc_2d_double(recvbuf_length, cols);

    if(myid == MASTER){
        MPI_Scatterv(&M[0][0][0], sendcounts, displs, MPI_DOUBLE, &m[0][0], recvbuf_length, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
    } else {
        MPI_Scatterv(NULL, sendcounts, displs, MPI_DOUBLE, &m[0][0], recvbuf_length, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
    }

    *M = m;
}
```

```c
double ** _matmul_(double **M, double **N, int rowsM, int colsM, int colsN){
    double sum;
    double **R = alloc_2d_double(rowsM, colsN);
    for(int i = 0; i < rowsM; i++){
        for(int j = 0; j < colsN; j++){
            sum = 0;
            for(int k = 0; k < colsM; k++){
                sum += M[i][k] * N[k][j];
            }
            R[i][j] = sum;
        }
    }
    return R;
}

double ** matmul_mpi(double** M, double ** N, int rowsM, int colsM, int colsN, int myid, int procs){
    double ** finalR;

    if(myid == MASTER){
        finalR = alloc_2d_double(rowsM, colsN);
    } else {
        finalR = NULL;
    }

    MPI_Bcast(&N[0][0], colsM*colsN, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    int offset_rows;
    distribute_matrix(&M, &rowsM,colsM,&offset_rows, myid,procs);

    double ** R = _matmul_(M,N, rowsM, colsM, colsN);

    MPI_Barrier(MPI_COMM_WORLD);

    if(myid == MASTER){
        MPI_Gather(&R[0][0], rowsM*colsN, MPI_DOUBLE, &finalR[0][0], rowsM*colsN, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
    } else {
        MPI_Gather(&R[0][0], rowsM*colsN, MPI_DOUBLE, NULL, rowsM*colsN, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
    }

    return finalR;
}
```

MPI benchmarking





- 1 node works faster than 2 nodes
- Increased batch size yields lower execution times

QA