Multiprocessor Systems - Assignment I (MPI)

Adrian Holfter, Lucie Labadie

April 23, 2017

1 Implementation

1.1 MPI-based Blocked Matrix-Matrix Multiplication

Based on the provided one-dimensional row-based implementation, we created an implementation using a two-dimensional block-wise checkerboard data distribution. This means that for two given matrices A, B with $N \times N$ elements each, every worker will compute $m \times n$ values of the result matrix C with $m, n \leq N$.

The work patch size is determined by starting with the size $N \times N$ of the whole matrix and the number of available worker nodes n_w. While $n_w \geq 2$, the work patch size is split in half, using alternating split dimensions, and n_w is divided by two. This assumes that N is a multiple of n_w.

To compute the result value $C_{x,y}$, the x^{th} row of matrix A and the y^{th} column of matrix B are needed. In order to simplify the transfer of the columns of matrix B to the worker nodes, matrix B is transposed in the beginning so it is in column-major order in memory. The master node then proceeds to send the required information to the workers. Each worker will receive:

offset_row : The row index to start from

offset_col : The column index to start from

num_rows : The number of rows in the work patch

num_cols : The number of columns in the work patch

A : num_rows elements of the A matrix, starting from $offset\_row$. Since this matrix is in row-major order, consecutive rows are adjacent in the memory and can be copied / sent as one block.

B : num_cols elements of the B matrix, starting from $offset\_cols$. Since this matrix is transposed and now in column-major order, consecutive columns are adjacent in memory and can be copied / sent as one block.

The master will have the patch with offset $(0, 0)$. Both the master and the workers then do the actual calculation for their assigned part of the result matrix. Since the B matrix was transposed, the matrix multiplication routine had to be changed slightly.

Once they are finished, they send back the data to the master. Since every worker has a 2D patch of the whole C matrix which cannot be sent as a single block (since the rows not adjacent in memory in most cases), every worker sends back the entire C matrix. The master receives this data into an additional buffer matrix D and copies over the relevant parts of the result into its own C matrix.

For more details on the algorithm, see Algorithm 1 in Appendix A.

1.2 MPI-based Laplace Approximation

First we chose to write the row-wise MPI-implementation of the Laplace Approximation. So we split the matrix on row dimension only given the number of available nodes.

After splitting the matrix we have to send each part to the workers. Each worker will receive :

offset : the chunk position in the whole matrix;

rows : the number of rows the worker have to compute;

matrix : the matrix part the worker should compute with the extra rows and columns needed for the computation (one after and one before).

Since the original implementation allocates always a memory block of `MAX_SIZE` × `MAX_SIZE`, the matrix rows are not adjacent in memory. To circumvent this problem, our implementation dynamically allocates exactly the needed amount of memory, leading to a memory layout in which matrix rows are adjacent. This allows to send multiple rows at once and helps to avoid excessive amounts of `MPI_Send()` calls.

The master will then do its part of the computation and wait to receive the results of the workers.

The workers will receive their data and start the computation, after computation they will send back the results to the master : offset and result matrix. We do not need to send the number of rows again since it is already known by the master.

The work executed by each worker and the master starts by exchanging the extra rows on top and bottom. During the first turn this operation is not really necessary but it will serve for the next since it is needed for the further computations. The first pass of computation is executed respecting the turn we are in (ODD or EVEN). The stopping condition is checked by taking the maximum of each row and keeping only the biggest sum. If this sum is converging (difference between two runs of the same type approaches 0), the computation is stopped. To do so, each worker is computing its own maximum value and sends it to the master which will check whether to stop or not and send the result back to the workers. Then the turn type is switched (ODD or EVEN) and another iteration is started.

For more details on the algorithm, see Algorithm 2 and 3 in Appendix B and C.

2 Measurements

The measurements were taken on the *kraken.tek.bth.se* Server. The executables were compiled with the `-O2` option, to enable compiler optimizations. Every measurement was taken 10 times and the smallest value was used to account for background load and operating system caches.

2.1 MPI-based Blocked Matrix-Matrix Multiplication

Table 1 shows the runtimes of the different versions of the blocked matrix-matrix multiplication.

Version	**shortest runtime [s]**	**speedup**
Sequential single-threaded	30.40	1.0
MPI 1 thread	2.54	11.97
MPI 2 threads	0.95	32.00
MPI 4 threads	0.57	53.33
MPI 8 threads	0.47	64.68

Figure 1: Runtime comparison for matrix multiplication

There is a substantial speedup noticeable for the MPI version in comparison to the original sequential values, even when running single-threaded. The explanation for this is most probably that the matrix B is now stored in column-major order in memory, which means that the memory layout now more closely matches the access pattern, allowing for a far better cache-hit ratio.

Figure 2 shows the runtime breakdown for the MPI version per number of workers. It becomes clear that the percentage of time spent copying back the result matrix increases rapidly, which is because every worker sends back the *whole* `C` matrix, even though only a small part of it actually contains work results.

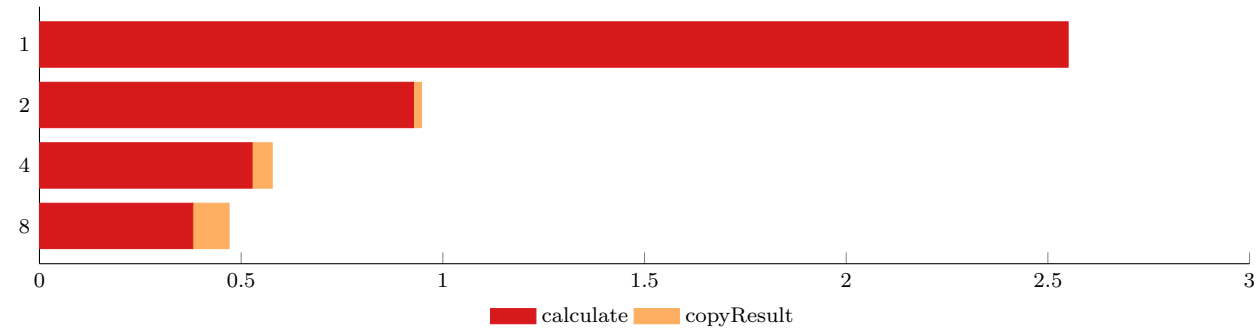

Figure 2: Runtime breakdown for matrix multiplication per number of workers

2.2 MPI-based LaPlace Approximation

Table 3 shows the runtimes of the different versions of the row-based LaPlace approximation using successive over-relaxation. Figure 4 also visualizes this.

Version	**shortest runtime [s]**	**speedup**
Sequential single-threaded	41.03	1.00
MPI 1 thread	41.42	0.99
MPI 2 threads	23.85	1.72
MPI 4 threads	16.45	2.49
MPI 8 threads	13.56	3.05

Figure 3: Runtime comparison for LaPlace approximation

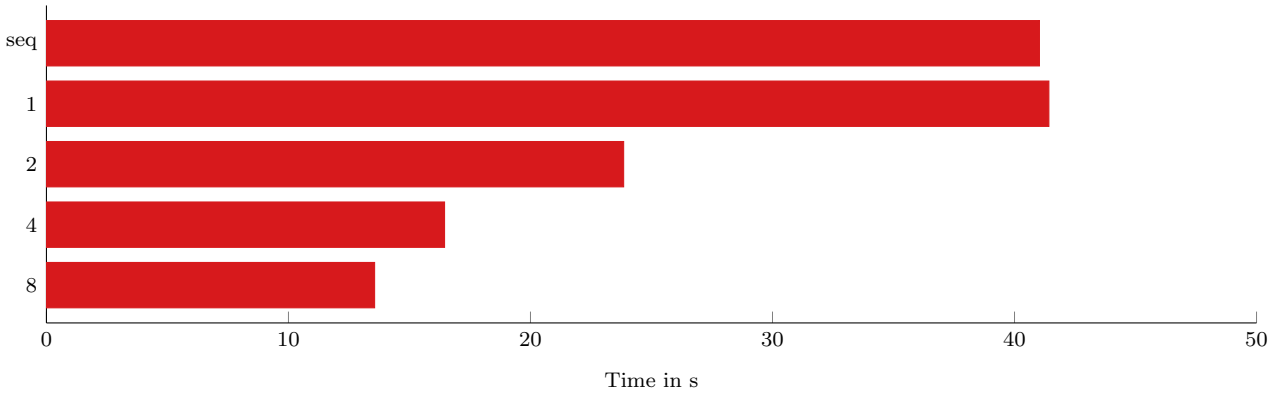

Figure 4: Runtime comparison chart for LaPlace approximation

Appendix A: Matrix multiplication algorithm

Algorithm 1 Matrix multiplication block-wise parallel implementation algorithm

```
MPI initialize
if master then
  matrix initialize
  transpose matrix B
  split the matrix
  for p from 1 to number of cores do
    MPI Send(offset row to worker p)
    MPI Send(offset column to worker p)
    MPI Send(number of rows to worker p)
    MPI Send(number of columns to worker p)
    MPI Send(matrix A rows to worker p)
    MPI Send(matrix B columns to worker p)
  end for
  Matrix multiplication computation on the block
  for p from 1 to number of cores do
    MPI Recv(offset row from worker p)
    MPI Recv(offset column from worker p)
    MPI Recv(number of rows from worker p)
    MPI Recv(number of columns from worker p)
    MPI Recv(matrix C from worker p)
  end for
else
  MPI Recv(offset row from master)
  MPI Recv(offset column from master)
  MPI Recv(number of rows from master)
  MPI Recv(number of columns from master)
  MPI Recv(matrix A rows from master)
  MPI Recv(matrix B columns from master)

  Matrix multiplication computation on the block

  MPI Send(offset row to master)
  MPI Send(offset column to master)
  MPI Send(number of rows to master)
  MPI Send(number of columns to master)
  MPI Send(matrix C to master)
end if
MPI finilize
```

Appendix B: Laplace Approximation global algorithm

Algorithm 2 Laplace Approximation row-wise parallel implementation : global algorithm

```
MPI initialize
matrix initialize
split the matrix
if master then
  for p from 1 to number of cores do
    MPI Send(offset to worker p)
    MPI Send(number of rows to worker p)
    MPI Send(matrix to worker p)
  end for
  Call work(number of rows, offset)
  for p from 1 to number of cores do
    MPI Recv(offset from worker p)
    MPI Recv(matrix from worker p)
  end for
else
  MPI Recv(offset from master)
  MPI Recv(number of rows from master)
  MPI Recv(matrix from master)

  Call work(number of rows, offset)

  MPI Send(offset to master)
  MPI Send(matrix to master)
end if
MPI finilize
```

Appendix C: Laplace Approximation work algorithm

Algorithm 3 Laplace Approximation row-wise parallel implementation : work function

```
turn = EVEN
while not finished do
  iteration++
  if not the first row then
    MPI Recv(top row from above worker)
    MPI Send(top row to above worker)
  end if
  if not the last row then
    MPI Send(bottom row to below worker)
    MPI Recv(bottom row from below worker)
  end if
  Computations Laplace Approximation
  Max row sum calculation
  if not the master then
    MPI Send(max row sum to master)
  else
    Check stopping conditions
  end if
  MPI Broadcast(finished status to every worker)
  turn changing
end while
return iteration
```