MPI assignment – report

Krzysztof Pszeniczny

8th June 2016

1 Implementation

Each process stores two buffers for sparse matrix entries (which are swapped after each round due to asynchronous sends/receives) and two buffers for dense matrix parts (one for the result matrix and one for the B matrix).

My implementation basically follows the algorithm description. The only difference is regarding collecting the result matrix in the Inner algorithm. Because we were told that we should start as with $c = 1$, I decided to perform the appropriate deduplication of results to the state as if $c = 1$, so e.g. respecive counts (when using `-g`) are sent by all processes, not just by their representatives in layer 0.

Moreover, because we perform repeated multiplications, I collect the parts among all the processes in a replication group instead of collecting to layer 0 and broadcasting from there.

2 Optimisations

- I impose a Cartesian virtual topology (possibly reordering the processes in order to accomodate for the possible grid-like underlying topology).
- I use custom data types for initial parameters and for sparse matrix entries (two ints – coordinates – and a double – the value itself).
- I use custom communicators (actually, sub-communicators of the Cartesian one) for communication inside the replication groups.
- Parts of the sparse matrix are sent concurrently with the computation on them (i.e. asynchronous sends and receives).
- I use the following collective communication functions:

  - `MPI_Bcast` to broadcast the initial parameters
  - `MPI_Scatter` to distribute the counts of sparse matrix entries each process should expect
  - `MPI_Scatterv` to scatter the sparse matrix itself
  - `MPI_Allgatherv` to replicate the matrices inside replication groups
  - `MPI_Gather` (if the `-v` option is used) to gather the resulting matrix
  - `MPI_Reduce` (if the `-g` option is used) to find the total number of entries greater than the given threshold
  - `MPI_Sendrecv` to perform the initial shift in the Inner algorithm
  - `MPI_Allreduce` as an ugly hack in my implementation of the Inner algorithm – to collect the parts of the result matrix inside the replication group. If a matrix cell does not belong to a process, it stores 0 in it, so reduction using `MPI_Sum` results in each process having the whole part of the matrix assigned to its replication group.

- Matrices are traversed in a way which optimises cache usage. Because the dense matrices are stored in a column-major order in my implementation, this means traversing all columns in the outer loop.