Parallel Computing: Exercise session 2

Emil Løvbak & Sahar Chehrazad

26 October 2020

This exercise session consists of pen-and-paper exercises and covers the following topics:

- Complexity of parallel sparse matrix vector products
- Scalability
- Message passing programming model
- Parallel sorting

For questions you can contact Emil Løvbak (`emil.loevbak@kuleuven.be`) or Sahar Chehrazad (`sahar.chehrazad@kuleuven.be`).

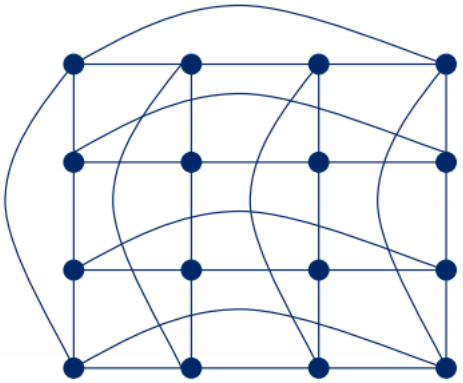
Question 1: Sparse matrix-vector product

Consider the sparse matrix-vector product over a network of processors in a ring topology. The matrix of size $n \times n$ is distirbuted in block rows over the p processors and the input vector is distributed accordingly. Assume that a single multiply-add takes time γ, each communication has a startup cost α and the time per word is β.

1. What is the parallel efficiency for a diagonal, tridiagonal matrix and a band matrix with bandwidth b? (don't treat the first and last rows specially)
2. For which cases is the problem scalable?

Question 2: Message passing

Consider a 2D mesh of processors, a matrix of data which is distributed accordingly and an iterative algorithm on that data that uses a 5-point computational molecule, i.e. it requires the local data and the ones of the neighboring processors in the horizontal and vertical direction in each step of the iteration.
Following functions are available for the message passing



- **send(in** *var*, **in** *procnum* [, **in** *label*])
- **receive(out** *var*, **in** *procnum* [, **in** *label*])
- **own()** : own processor number
- **left()**, **right()**, **up()**, **down()** : processor numbers of the neighbours

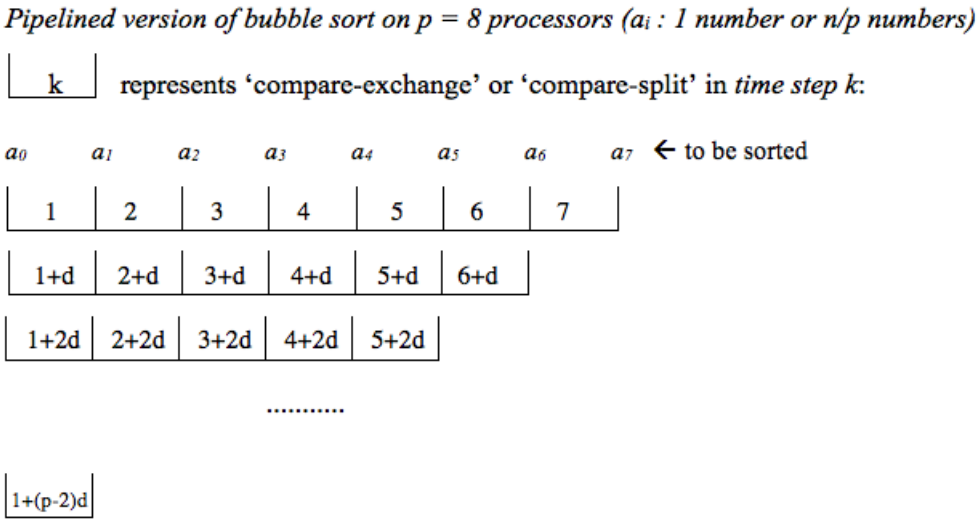
The function **calc(out** *new_local_data*, **in** *local_data* , **in** *left_data*, **in** *right_data*, **in** *down_data*, **in** *up_data*) makes one iteration step.

1. Assume that the communication is done with buffered non-blocking communication and write a pseudo-code avoiding deadlocks.
2. Assume that the communication is done with non-buffered blocking communication. How do you have to adapt the code?
3. Assume now that a non-blocking non-buffered send primitive is used:
   - **isend(in** var, **in** procnum, **in** label, **out** request **)** : *request* is an object that holds the status of the message
   - **wait(in** request **)** waits until the variable of the message for which request is the status object can be overwritten again in a safe way

   How do you have to adapt the code?
4. How do you organize the communication for a 9-point computational molecule (square), i.e. which it also involves the diagonal neighbours? Adapt the code for one of the above cases.

Question 3: Pipelined bubble sort

The odd-even transposition sort algorithm can be seen as a modification of the classical bubble sort algorithm, such that it can be implemented in parallel. Alternatively, the bubble sort algorithm can also be parallelized using the idea of a 'pipeline', see the figure below.



1. What is the 'delay' d, i.e. the number of steps that a processor must wait before it can start with the second phase of the algorithm?
2. Write a pseudo-code for the pipelined parallel bubble sort in the message passing programming model.
3. Assume that a list of length n is sorted on p processors. What is the parallel run-time? Compare this algorithm with the odd-even transposition sort.

Question 4: Parallel quicksort

Recall the parallel quicksort algorithm from the lecture.

1. What is the best-case performance (in big O notation)?
2. What is the worst-case performance?