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)

diagonal matrix $E = 1$
tridiagonal matrix

calculation: $3\frac{n}{p}\gamma$ *(for each row 3 multiply-add operations, where in the first operation zero is added, and there are n/p rows per processor)*

communication: $2(\alpha + \beta)$ *(each processor sends its first and last (third) entry of the input vector to its left and right neighbour, respectively, independently of the number of rows per processor. To avoid congestion in a ring topology, all processors first send clockwise and then counterclockwise).*

$$E = \frac{S}{p} = \frac{T_s}{pT_p} = \frac{3n\gamma}{p(3\frac{n}{p}\gamma + 2(\alpha+\beta))} = \frac{1}{1 + \frac{2}{3}\frac{\alpha+\beta}{\gamma}\frac{p}{n}}$$

band matrix

calculation: $(2b+1)\frac{n}{p}\gamma$ *(bandwidth defined as maximum of lower and upper bandwidth, see wikipedia, for tridiagonal matrix* $b = 1$*)*

communication: For simplicity consider first $b \leq n/p$*, then* $2(\alpha + b\beta)$ *(each processor send the lower and upper* b *entries of its input vector to its left and right neighbour, respectively. For* $b > n/p$ *use all-to-all broadcast scheme for ring topology to avoid congestion, which is truncated after* $\lceil \frac{b}{n/p} \rceil$ *phases in both directions, and* b *entries have to be sent). Both cases are covered by* $2(\lceil \frac{b}{n/p} \rceil \alpha + b\beta)$*.*

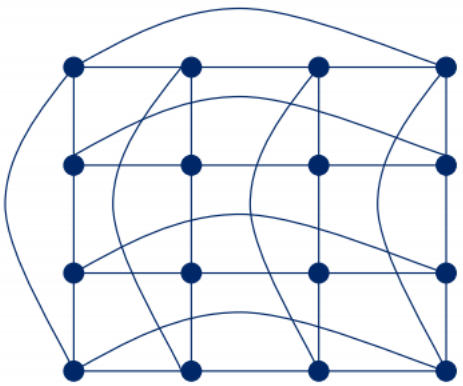
$$E = \frac{(2b+1)n\gamma}{p\left((2b+1)\frac{n}{p}\gamma + 2(\lceil \frac{b}{n/p} \rceil \alpha + b\beta)\right)} \underset{b \bmod n/p=0}{=} \frac{1}{1 + \frac{2b\alpha}{(2b+1)\gamma}\frac{p^2}{n^2} + \frac{2b\beta}{(2b+1)\gamma}\frac{p}{n}}$$

2. For which cases is the problem scalable?

In case of strong scaling (i.e. increasing the number of processors p*, while keeping the problem size constant* n*), only the diagonal case is truly scalable (it is embarrassingly parallel). Both the tridiagonal and band matrix case are weakly scalable, since the efficiency stays constant for constant* n/p *(i.e. both are increased with the same rate). Note that in terms of strong scaling the band matrix preforms worse than the tridiagonal case due to the* p^2 *(i.e. the efficient is decreasing faster).*

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.

```
while not converged
     send(data,right)
     send(data,left)
     send(data,up)
     send(data,down)
     receive(right_data,right)
     receive(left_data,left)
     receive(up_data,up)
     receive(down_data,down)
     calc(data,data,left_data,...)
end while
```

In principle the order of the send and receive statement can be chosen arbitrary, since after the data is in the buffer the processor can go on receiving, however for example the receive(right) must come after send(left) to avoid deadlocks.

2. Assume that the communication is done with non-buffered blocking communication. How do you have to adapt the code?

```
while not converged
    for i = 0 to 1 do
        if(evencol(own+1))
           send(data,right)
           receive(right_data,right)
        else
           receive(left_data,left)
           send(data,left)
        end if
        if(evenrow(own+i))
           send(data,up)
           receive(up_data,up)
        else
           receive(down_data,down)
           send(data,down)
        end if
   end for
   calc(...)
end while
```

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?

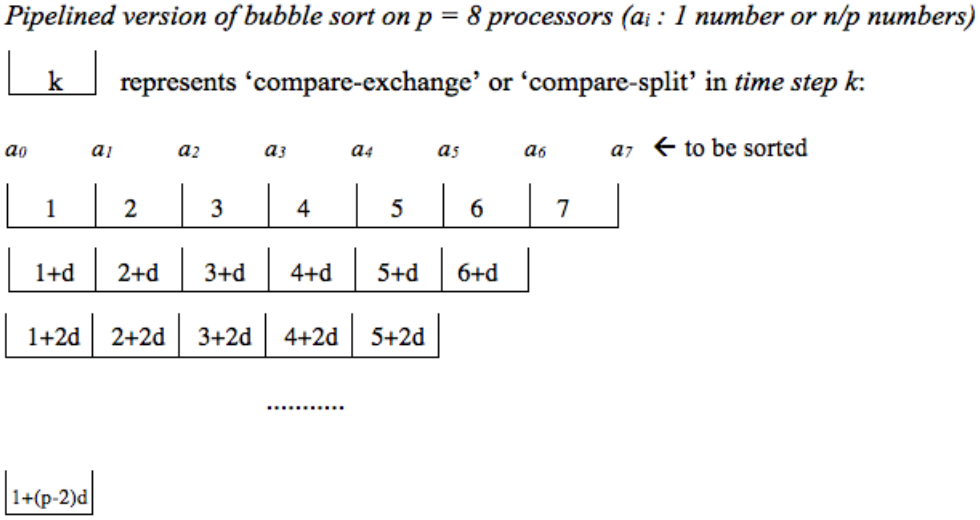
```
req1=ok, req2=,...
while not converged
    wait(req1),....
    send(data,right,req1)
    send(data,left,req2)
    send(data,up,req3)
    send(data,down,req4)
    receive(right_data,right)
    receive(left_data,left)
    receive(up_data,up)
    receive(down_data,down)
calc(...)
end while
```

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.

> *After communicating left and right, the up and down neighbours have already the information form the diagonally located processors. Therefore no additional communication is required, except that the additional data has to be transmitted. Labels can be used to identity the data.*

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?

> $d = 2$ *(after the second processor has made a compare-exchange with the third processor)*

2. Write a pseudo-code for the pipelined parallel bubble sort in the message passing programming model.

Using blocking communication

```
myrank = own()
for i=0 to (p-1) exclusive do
      if myrank==0
         send(mynumbers,myrank+1)
         receive(mynumbers,myrank+1)
      elseif(myrank==(p-1)-i)
         receive(othernumbers,myrank-1)
         send(min(othernumbers,mynumbers),myrank-1)
         mynumber=max(othernumbers,mynumbers)
         break
      else
         receive(othernumbers,myrank-1)
         send(max(othernumbers,mynumbers),myrank+1)
         send(min(othernumbers,mynumbers),myrank-1)
         receive(mynumbers,myrank+1)
      end if
end for
```

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.

$$T_p = \underbrace{O\left(\frac{n}{p}\log\frac{n}{p}\right)}_{\text{local sort}} + \Big(\underbrace{O(n/p)}_{\text{comparisons}} + \underbrace{O(n/p)}_{\text{communication}} \Big) \underbrace{(2p-3)}_{\text{odd-even: } p}$$

The total number of compare-splits stays the same for both algorithms, but in the pipelined version all the processors are fully utilized only in the last phase

Question 4: Parallel quicksort

Recall the parallel quicksort algorithm from the lecture.

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

The pivot is chosen such that the processors always have n/p of data.

$$\sum_{i=0}^{\log p-1} \underbrace{O(\log p/2^i)}_{\text{broadcast of pivot}} + \left(\underbrace{O(n/p)}_{\text{comparison}} + \underbrace{O(n/2p)}_{\text{communication}} \right) \underbrace{O(\log p)}_{\text{recursion}} + \underbrace{O(n/p \log n/p)}_{\text{final sequential quicksort}}$$

2. What is the worst-case performance?

If the pivot is always the smallest element and the upper half is defined by strictly higher, then in each round half of the processors receive no data.

$$\sum_{i=0}^{\log p-1} \left(\underbrace{O(\log p)}_{\text{broadcast of pivot}} + \underbrace{O(2^i n/p)}_{\text{comparison}} + \underbrace{O(2^i n/2p)}_{\text{communication}} \right) + \underbrace{O(n^2)}_{\text{final sequential quicksort}}$$