Kungliga Tekniska Högskolan
Valhallavägen 79
100 44 Stockholm

Parallel Computations for Large-Scale Problems

« Homework 2 »

February 16^{th} - March 2^{nd}

Authors
Rémi Domingues 920604-T239
Johan Wärnegård 920113-4914

Teacher
Michael Hanke

Scholar year 2014-2015

1 The broadcast operation is a one-to-all collective communication operation where one of the processes sends the same message to all other processes.

1.1 Design an algorithm for the broadcast operation using only point-to-point communications which requires only $O(logP)$ communication steps

The following figure describes the steps of our algorithm. The showcase uses $P = 2^D$, but the algorithm is functional for any P.

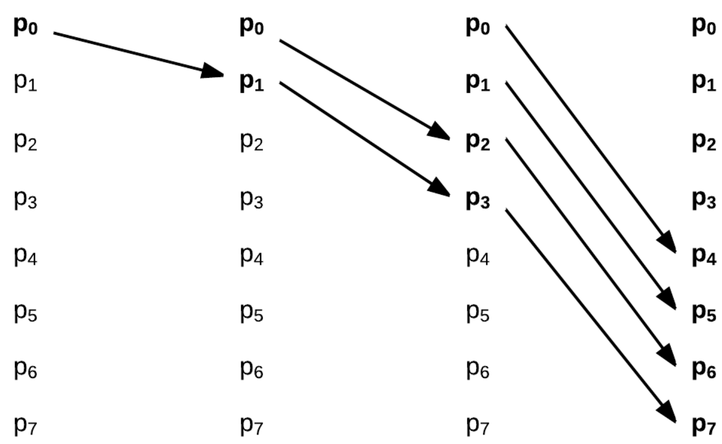

FIGURE 1 – Communication broadcast algorithm in $O(logP)$

This figure illustrates the following algorithm. For any p higher than 0, the process will receive its message from $p_{sender} = p_{receiver} - 2^D$ with D such that $2^D \leq p_{receiver} \leq 2^{D+1}$.

if p == 0 **then**
    D = 0
    msg = "message"
else
    D = trunc($log_2(p)$)
    msg = receive($p - 2^D$)
    ++D
end if

while $p + 2^D \leq p$ **do**
    send(msg, $p + 2^D$)
    ++D
end while

1.2 Do a (time-)performance analysis for your algorithm

The process involves no computation on any processor. We denote the communication time required to send the data set, consisting of n elements, between two processors t_{comm}. As usual $t_{comm} = t_{startup} + nt_{data}$. Each step in the algorithm takes time t_{comm}, in total $D = \log_2 P$ steps are required. Altogether this yields the total time as a function of P and n as :

$$T = (t_{startup} + nt_{data}) \cdot \log_2 P$$

1.3 How can the scatter operation be implemented using $O(logP)$ communication steps ?

Suppose we have a data set consisting of P elements known by a process p_0. We want to scatter the data so that a process i holds one element of the data set, also named element i. We would do this using the broadcast algorithm previously defined, replacing some calls by the following :

if p == 0 **then**
  D = 0 size
  data = loadData()
else
  D = trunc($log_2(p)$) ▷ cast from double to int
  data = receive($p - 2^D$)
  ++D
end if

while $p + 2^D \leq p$ **do**
  array = split(data, 2) ▷ Split the dataset in two parts of equal size
  data = array[0]
  send(array[1], $p + 2^D$)
  ++D
end while

After the first iteration, two processes will each hold one half of the data. After 2 iterations 4 processors hold one fourth each, and so forth until all P processors hold exactly one element. The number of iterations is :

$$D = \frac{\log(P)}{\log(2)} \Rightarrow T \propto \log_2(P)$$

2 Consider a matrix A distributed on a $P * P$ process mesh. An algorithm has been given in the lecture for evaluating the matrix-vector product $y = Ax$. While x is column distributed, y is row distributed. In order to carry out a further multiplication Ay, the vector y must be transposed.

2.1 Design an algorithm for this transposition. You may use the results from problem 1.

In order to perform the transposition of the vector y, two steps are necessary (if we consider the current state as the one immediately after the internal multiplication of x_c by $A_{r,c}$ by $p_{r,c}$). We consider here that A is a square matrix (since we juste performed the Ax operation and want to perform Ay) which is distributed on the $P * P$ mesh grid. We also assume that $P = M$, with A a matrix of $M * M$.

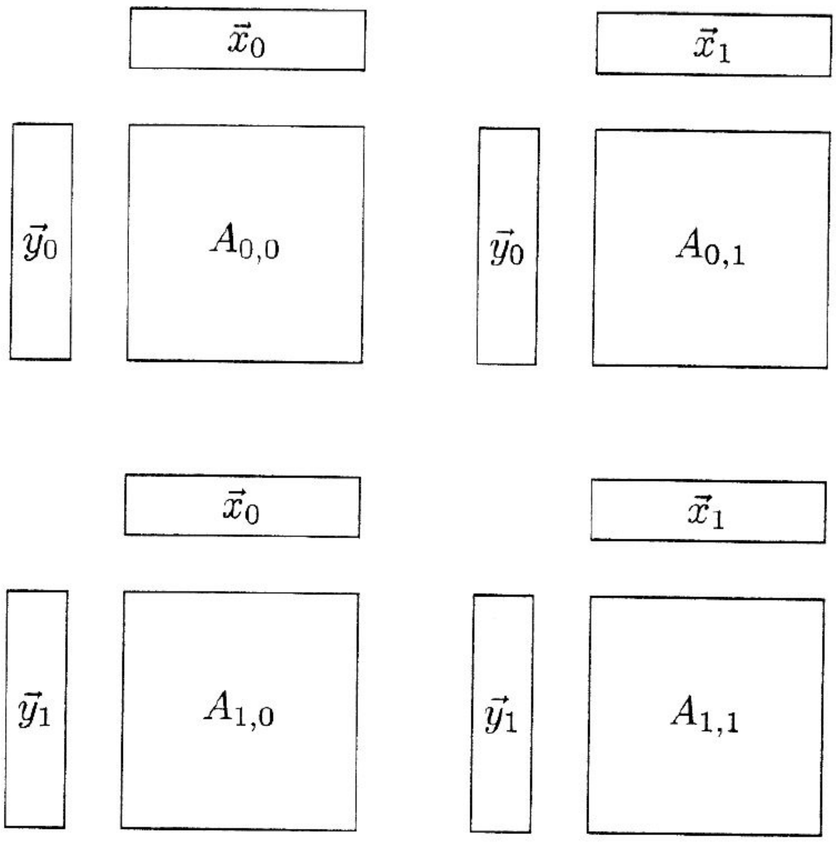

FIGURE 2 – Data distribution for each processor, with A a square matrix

Therefore, our transition algorithm would first require an efficient information exchange between the processes of the same row to compute y_r (this should be achived using recursive doubling with a complexity of $O(logP)$ for the whole matrix), then an axial symmetry communication to achieve the final transposition (we assume that each process knows its row r and column c).

Let $2^D < P < 2^{D+1}$

s = y_r ▷ y_r is still incomplete and is only a part of the real y_r
if $p \geq 2^D$ **then**
    send(s, bitflip(p, $p_{r,D}$))
end if
if $p \leq P - 2^D$ **then**
    receive(h, bitflip(p, $p_{r,D}$))
    $s = s + h$
end if
if $p \leq 2^D$ **then**
    for d = 0 :D-1 **do**
        send(s, bitflip(p, $p_{r,D}$))
        receive(h, bitflip(p, $p_{r,D}$))
        s = s+h
    end for
end if
if $p \leq P - 2^D$ **then**
    send(s, bitflip(p, $p_{r,D}$))
end if
if $p \geq 2^D$ **then**
    receive(s, bitflip(p, $p_{r,D}$))
end if
y_r = s ▷ y_r has now been summed and is known by every process of the same row
if r != c **then** ▷ No permutation required if the process is on the diagonal
    MPI_Sendrecv(y_r, $p_{c,r}$, y_c, $p_{c,r}$) ▷ y_r is sent and we retrieve y_r
end if

On the other hand, if we do not have a vector to sum and transpose but a square matrix of size PxP distributed on a PxP mesh grid, the algorithm is the following :

if r != c **then** ▷ No permutation required if the process is on the diagonal
    MPI_Sendrecv(y_r, $p_{c,r}$, y_c, $p_{c,r}$) ▷ y_r is sent and we retrieve y_r
end if

2.2 Make a performance analysis

The complexity of the first algorithm (summing the parts ov the vector $\vec{y}$ then applying a transposition to prepare the system for the multiplication yA) is $O(log(P) + 2)$.

The complexity of the second algorithm for a square matrix is O(1).

3 Parallel implementation of the Jacobi iteration

See the following pages.