Lecture notes: Parallel Algorithms in Distributed Memory Systems

Master M1: Parallel Algorithms and Programming

2019

This lecture studies some parallel algorithms and their implementation in a distributed memory system. Three problems are considered: a matrix-vector multiplication, a matrix-matrix multiplication, and a *stencil* computation. To reason about the algorithms we assume that the processes are organized in a *virtual ring* topology.

1 About performance of parallel distributed algorithms

The execution of an algorithm in a distributed memory parallel system involves two costs:

- The cost of running computation (floating point operations in our context)
- The cost of moving data
  - from the memory to the processors
  - between nodes (over the interconnection network)

The total cost of a distributed parallel algorithm is the sum of 3 terms:

1. A computational term: `nb of flops` $\times$ `time per flop`
2. A bandwidth term: `amount of data moved` $\times$ $\frac{1}{bandwidth}$
3. A latency term: `nb of messages` $\times$ `latency`

A parallel programmer should remember that in general:

$$time\_per\_flop \ll \frac{1}{bandwidth} \ll latency$$

It follows that minimizing communication in a distributed parallel algorithm is important to save time. Note also that moving data from/to DRAM or over the interconnection network are the most energy consuming operations.

In the following, we will ignore the cost of moving data from/to the memory when computing the execution time of an algorithm. In practice, this cost is mostly hidden by hardware prefetcher in modern processors when the data access patterns are regular (as it is the case in the algorithms presented below).

2 Matrix-vector multiplication

The first problem we are studying is a matrix-vector multiplication $y = Ax$ where A is a matrix of size $n \times n$ and x,y are vectors of size n. Figure 1 describes the sequential implementation of the algorithm.

```
1     for(i=0; i<n; i++){
2       y[i] = 0;
3       for(j=0; j<n; j++){
4         y[i] = y[i] + A[i][j] × x[j]
5       }
6     }
```

Figure 1: Sequential matrix-vector multiplication

2.1 Parallel algorithm

We can observe that the computation of each value of the result vector y is the scalar product between a row of A and the vector x. Each scalar product is independent: They can be executed in any order. As such, a simple parallel version of the algorithm consists in distributing the rows of A over p processors so that each processor can compute $r = \frac{n}{p}$ scalar products in parallel[1].

2.2 Data distribution

Figure 2 presents the basic distribution of data to implement the matrix-vector product algorithm. In this version, it is assumed that all processors have a copy of the vector x in their memory.

Note that distributing the data and the computation over several nodes is not only good to speed-up the computation. It can enable to solve larger problems than with a single node. Indeed, it can be the case that the matrix A is so big that it does not fit in the memory of one node. In this case, distributing the rows of matrix A over p nodes allows storing a much larger matrix in memory.

As already mentioned, Figure 2 assumes that all processors have a copy of the vector x. However, when implementing a library function that computes a matrix-vector multiplication, in general, one prefer assuming that blocks of x are distributed over the nodes in the same way as matrix A. The motivation behind this assumption is that in practice, one might want to run several matrix-vector multiplications in a sequence. Imagine that one wants to run $z = By$ after $y = Ax$. In this case, at the beginning of the second matrix-vector multiplication, the vector y is already distributed over the nodes as shown in Figure 2. Thus, having a code that can deal with this case is good for modularity.

Considering a virtual ring network topology, the basic steps of one iteration of the new matrix-vector multiplication algorithm are:

- Each processor computes a partial result using the elements of x it has in its local memory
- Each processor sends its block of x to its successor and receives from its predecessor in the virtual ring.

[1]For the sake of simplicity, we will always assume that p divides n.

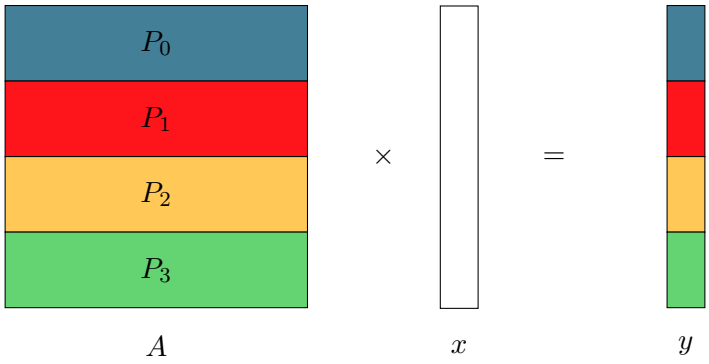


Figure 2: Distributed matrix-vector multiplication over 4 processors. All data with the same color are on the same processor

The algorithm has to run over p iteration to terminate.

Figure 3 illustrates the execution of this algorithm with 4 processes. If A is a $n \times n$ matrix, each process stores $\frac{n}{4}$ rows of A and $\frac{n}{4}$ values of x.

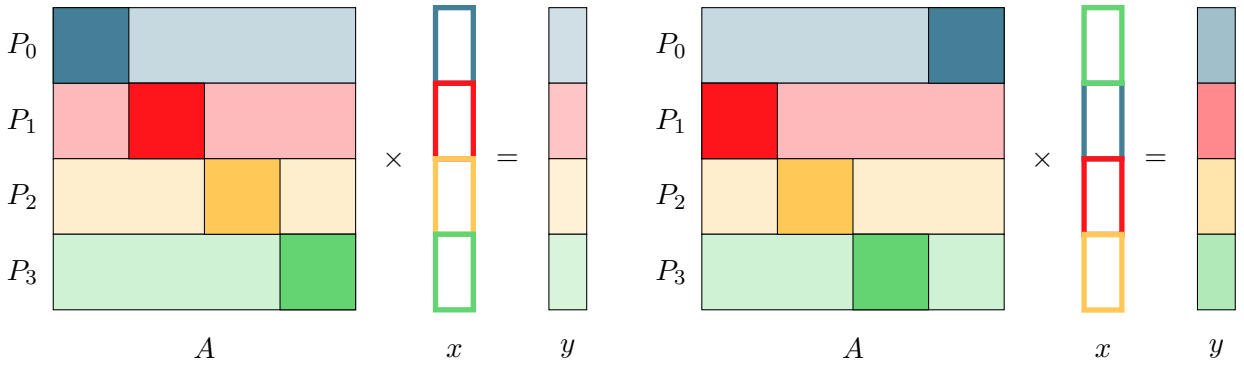
The algorithm presented in Figure 4 details the parallel version of the matrix-vector multiplication. A few points should be discussed about this algorithm:

- The algorithm is designed to allow the communication and the computation inside one step to occur in parallel (symbol $\|$).
- Running the communication and the computation in parallel requires to allocate an extra buffer ($tempR$) for the communication.
- One can notice that the communication is not mandatory in the last iteration of the algorithm. However, it allows restoring the initial distribution of the blocks of x, which can be a desirable property.
- As it can be observed in Figure 3, when process k computes based on a block q of values of x (i.e., the values at index $q \times r$ to $((q+1) \times r) - 1$ of the initial vector x), the computation applies to the block q of the sub-part of matrix A stored locally ($A_{k,q}$). Hence, the index of the block of A to use for process with id $rank$ in a given step if there are P processes in total is:

$$block = (rank - step) \mod P$$

We can compute the execution time of the algorithm described in Figure 4 assuming p processes and a matrix of size $n \times n$ with $r = \frac{n}{p}$. To compute the execution time, we additionally use:

- L: the latency of a network communication
- B: the bandwidth of a network communication
- w: the computation time for one basic unit of work (two floating point operations in line 19)



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

Figure 3: Distributed matrix-vector multiplication over 4 processors with blocks of vector x distributed over the processors. Colors borders for blocks of vector x refer to the initial position of the blocks.

We assume that the time to receive of message is the same as the time to send a message ($= L + \frac{r}{B}$). Since there are p iterations of the algorithm and since computation occur in parallel with communication, the execution time is:

$$\begin{aligned} T_{MV}(p) &= p \times \max(r^2 \times w, L + \frac{r}{B}) \\ &= \max(\frac{n^2}{p} \times w, p \times L + \frac{n}{B}) \end{aligned}$$

When n becomes large, then the execution time is asymptotically equal to:

$$T_{MV}(p) = \frac{n^2}{p} \times w$$

which implies that a speedup of p is to be observed compared to the sequential version of the algorithm when using p processors.

```
1     double A[r][n]; /* rows of A, assumed to be already initialized*/
2     double x[r]; /* values of x, assumed to be already initialized*/
3     double y[r];
4
5     int rank = my_rank();
6     int P = num_procs();
7
8     double tempS[r], tempR[r];
9
10    tempS ← x;  /* copy of the values */
11    for(step = 0; step < P-1; step++){
12      Send(tempS, (rank + 1) mod P);
13      ||
14      Recv(tempR, (rank − 1) mod P)
15      ||
16      for(i=0; i<r; i++){
17        for(j=0; j<n; j++){
18          int block= (P+rank-step) % P;
19          y[i] = y[i] + A[i, r × block mod P)] × tempS[j]
20        }
21      }
22      tempS ← tempR;
23    }
```

Figure 4: Parallel matrix-vector multiplication. The symbol ‖ implies that lines 12, 14 and 16 can be executed in parallel. The symbol ← implies a memory copy.

About the implementation in MPI For an MPI implementation of the algorithm presented in Figure 4, we recall that using the `MPI_Send()` and the `MPI_Recv()` for communication would prevent the communication and the computation to occur in parallel. To allow them to occur in parallel, one should use the non-blocking functions `MPI_Isend()` and `MPI_Irecv()` instead. Note that even in this case, one might not observe a full overlapping of the communication and computation in practice depending on the characteristics of the NIC (Network Interface Controller).

Data distribution The algorithm described in Figure 4 assumes that the data are initially already distributed among the processes. In practice, the data might already be distributed over the processes or might be hosted by a single process originally:

- The case where the data are on a single process can correspond to a scenario where one process initially read the data from a file. Note however that IO libraries implementing parallel accesses to files exists (e.g., MPI-IO).
- The case where the data are already distributed can correspond to a scenario where the data is the result of a previous distributed computation, as seen previously.

If one assumes a model where the data is located on a single node initially and should be aggregated on a single node again at the end of the execution, then *scatter* and *gather* collective operations can be used to implement these extra data movements.

3 Matrix-matrix multiplication

We consider the matrix-matrix multiplication problem $C = A \times B$ where A, B, and C are matrices of size $n \times n$. The sequential implementation of the algorithm is described in Figure 5.

```
1     for(i=0; i<n; i++){
2       for(j=0; j<n; j++){
3         C[i][j] = 0;
4         for(k=0; k<n; k++){
5           C[i][j] = C[i][j] + A[i][k] × B[k][j];
6         }
7       }
8     }
```

Figure 5: Sequential matrix-matrix multiplication

The parallel version of the matrix-matrix multiplication algorithm in a virtual ring topology with p processes is an extension of the algorithm previously studied for matrix-vector multiplication.

The distribution of the data and the main ideas of the algorithm are described in Figure 6. In a nutshell, the main principles are:

- The three matrices are distributed over the processes based on the rows. Each process stores $r = \frac{n}{p}$ rows of each matrix.
- In each iteration, a process run partial matrix-matrix multiplication computation based on the data that are available locally.
- In each iteration, a process sends r rows of B to the next process and receives r rows of B from the previous process in the virtual ring.

The algorithm requires p iterations to terminate.

As can be seen in Figure 6, in each iteration of the loop, a process runs p matrix-matrix multiplications on matrix blocks of size $r \times r$. Only the rows of B are exchanged between the processes. Each process computes a sub-part of matrix C: process i is in charge of rows $i \times r$ to $((i+1) \times r) - 1$.

The algorithm for the parallel matrix-matrix multiplication on a logical ring of processes is described in Figure 7. It is very similar to the one for matrix-vector multiplication but an additional index (named l in Figure 7) is introduced to iterate over the blocks of B and C.

As for the matrix-vector multiplication algorithm, we can compute the execution time of the matrix-matrix multiplication algorithm for matrices of size $n \times n$ and for p processes, as:

$$\begin{aligned} T_{MM}(p) &= p \times \max(n \times r^2 \times w, L + \frac{n \times r}{B}) \\ &= \max(\frac{n^3}{p} \times w, p \times L + \frac{n^2}{B}) \end{aligned}$$

When n becomes large, then the execution time is asymptotically equal to:

$$T_{MM}(p) = \frac{n^3}{p} \times w$$

which implies an optimal efficiency (speedup of p with p processors).

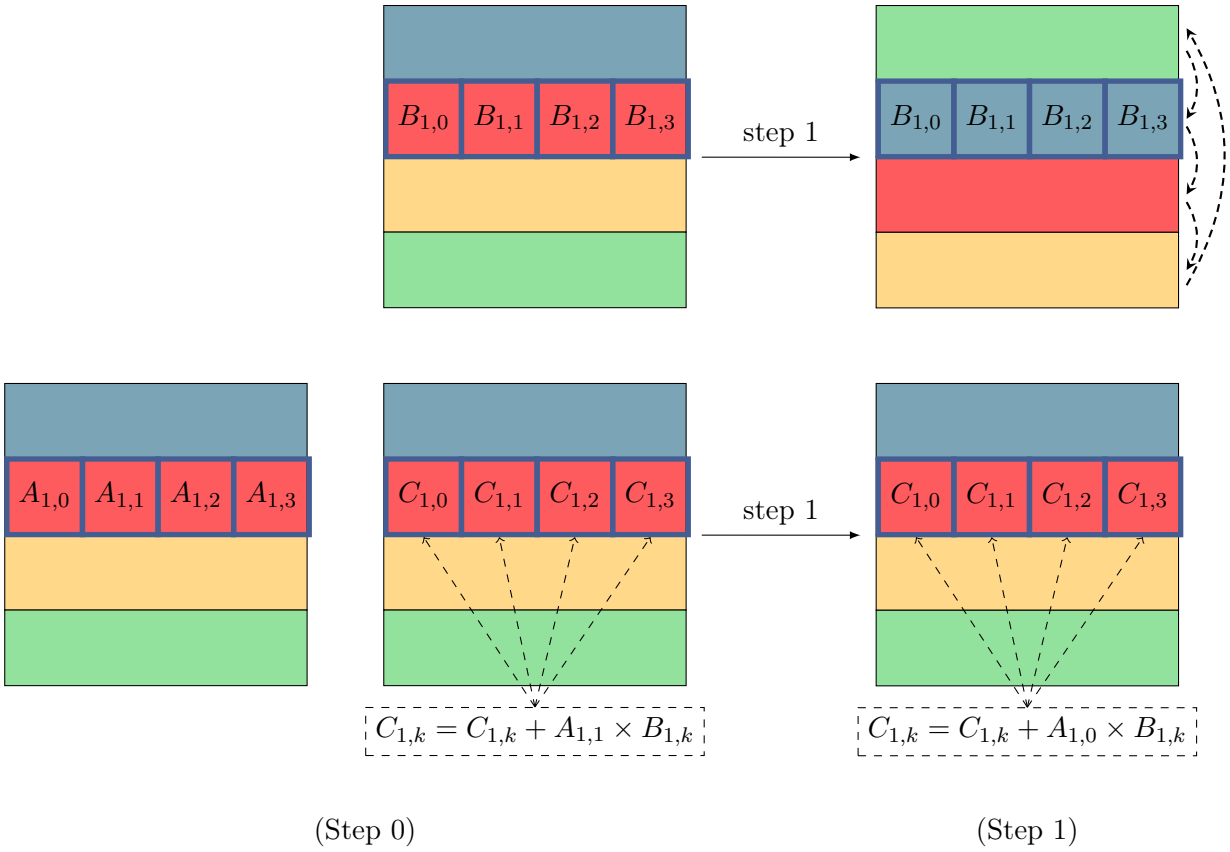


Figure 6: Parallel matrix-matrix multiplication with 4 processes. The figure presents 2 iteration of the algorithms and focuses on the computation run by process P_1. The notation $A_{k,l}$ refers to the block l in the part of matrix A stored by process k.

Bulk communication It is interesting to note that the parallel matrix-matrix multiplication could also be implemented as n matrix-vector multiplications (one for each column of B). In this case, the execution time of the algorithm would be:

$$\begin{aligned} T_{MM-alt}(p) &= n \times T_{MV}(p) \\ &= \max(\frac{n^3}{p} \times w, n \times p \times L + \frac{n^2}{B}) \end{aligned}$$

Compared to the execution time $T_{MM}(p)$, the execution is still asymptotically optimal but the latency term is multiplied by n ($n \times p \times L$). This is due to the fact that is this algorithm, the data will be exchanged column by column, whereas in the algorithm of Figure 7, a set of rows is sent as once. This might have a significant impact on the performance in practice.

Sending data in *bulk* is a general technique to reduce the cost of communicating over the network in parallel algorithms, and more specifically, to reduce the impact of latency.

```
1    double A[r][n]; /* rows of A, assumed to be already initialized*/
2    double B[r][n]; /* rows of B, assumed to be already initialized*/
3    double C[r][n];
4
5    int rank = my_rank();
6    int P = num_procs();
7
8    double tempS[r][n], tempR[r][n];
9
10   tempS ← B;  /* copy of the values */
11   for(step = 0; step < P-1; step++){
12     Send(tempS, (rank + 1) mod P);
13     ||
14     Recv(tempR, (rank − 1) mod P)
15     ||
16     for(l=0; l<p; l++{
17       for(i=0; i<r; i++){
18         for(j=0; j<r; j++){
19           for(k=0; k<r; k++){
20            block = (P + rank - step) % P;
21              C[i][l × r + j] = C[i][l × r + j] + A[i][block × r + k] × tempS[k][l × r + j];
22       }
23     }
24     tempS ← tempR;
25   }
```

Figure 7: Parallel matrix-matrix multiplication. The symbol ‖ implies that lines 12, 14 and 16 can be executed in parallel. The symbol ← implies a memory copy.

4 Stencils

TO BE DONE

References

Some references can complement the material presented in these lecture notes:

- Section 4.1 and 4.2 of the book "Parallel Algorithms" (by Casanova, Robert, and Legrand).