Lecture notes: Parallel Algorithms in Distributed Memory Systems

Master M1: Parallel Algorithms and Programming

2022

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.

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 prefetchers in modern processors when the data access patterns are regular (as it is the case in the algorithms presented below).

Hence, when computing the execution time of an algorithm, we will use the following values:

- L: the latency of a network communication

- B: the bandwidth of a network communication
- w: the computation time for one basic unit of work (depends on the algorithm)

2 Stencils

We study stencil algorithms which is a class of algorithms commonly found in parallel applications.

Stencil algorithms operate on cells that divide a discrete domain. Cells can hold one or multiple values, and they have neighboring cells. Cells are in general organized in a N-dimensional grid (often 2D or 3D). They are non-overlapping and of equal size to improve load balancing. An example of 2D domain divided in cells is given in Figure 1. When the studied domain becomes very large, one may want to run the stencil in parallel on multiple nodes.

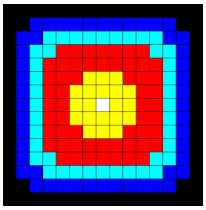

Figure 1: An example of 2D stencil presented during the first lecture (heat diffusion).

The stencil algorithm is an iterative algorithm that applies a pre-defined function to update the value of the cells based on the value of the neighboring cells. *The location of a cell's neighbors and the function used to update cell values form a "stencil" that is applied to all cells in the domain.*

In the following, we are studying stencils applying to 2-dimensional domains. A 2D stencil implies that a cell can have at most 8 neighbors that can be identified using cardinal coordinates: N, NE, E, SE, S, SW, W, NW.

3 A stencil algorithm

The stencil algorithm we consider applies the following function to update cells:

$$c_{new} = Update(c_{old}, W_{new}, N_{new})$$

It means that the new value of a cell (c_{new}) depends on the previous value of the cell (c_{old}) and the already updated value of the *west* and *north* neighbors (W_{new} and N_{new}). The stencil is represented in Figure 2.

Note that the stencil cannot be applied as it to cells at the border of the domain that do not have *north* or *west* neighbors. This is handled by applying a modified `Update` function to these cells (for instance, assuming that the value of the non-existing cells is a constant).

We point out that although simple, this stencil is of practical importance as it is at the root of some numerical algorithms (e.g., the *Gauss-Seidel* iterative method to solve a linear system of equations).

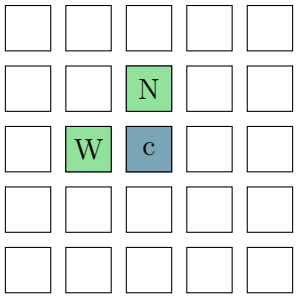

Figure 2: Illustration of the stencil

3.1 Greedy algorithm

We study a first parallel algorithm on an unidirectional ring of processors for the stencil application described above. We assume p processors. We assume that the problem includes $n \times n$ cells.

The algorithm we introduce is *greedy*. By *greedy*, we mean that the algorithm tries to put all processors to work as early as possible.

To distribute the work, we allocate rows of data to processors. To start reasoning about the algorithm, we make the simplifying assumption that there are as many processors as the number of rows, i.e., $p = n$. In this case, the main principles of this algorithm are:

- Each processor stores one row. Row i is stored on processor P_i.
- As soon as a processor P_i has computed a value, it sends it to processor P_{i+1}.

If A is the domain on which the stencil is applied, we note $A_{x,y}$ (or $A[x][y]$) the y-th element on row x.

Figure 3 shows the execution of the algorithm. The number associated with each cell is the step in which the cell is updated. As it can be observed in the figure, in the first step only $A_{0,0}$ is computed. In the second step, $A_{1,0}$ and $A_{0,1}$ can be computed. More generally, in step k, the k-th anti-diagonal can be computed.

1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8
5	6	7	8	9

Figure 3: Parallel execution of the greedy algorithm

Figure 4 details the greedy algorithm. Note that the algorithm deals with the two special cases of processor P_0 that does not receive *north* values from another processor, and P_{n-1} that does not

need to send its updated values. Ignoring these two special cases, we can say that at iteration $i+j$ processor P_i performs the following operations:

- Receives $A_{i-1,j}$ from P_{i-1};
- Computes $A_{i,j}$;
- Sends $A_{i,j}$ to P_{i+1}.

```
1    double A[n]; /* one row of A, assumed to be already initialized*/
2    double north = 0; /* to recv North values */
3
4    int rank = my_rank();
5    int P = num_procs();
6
7    if (rank == 0){
8      A[0] = Update(A[0], NULL, NULL);
9      Send(A[0], rank+1);
10   }
11   else{
12     Recv(north, rank-1);
13     A[0] = Update(A[0], NULL, north);
14   }
15
16   for(j=1; j<n; j++){
17     if (rank == 0){
18       A[j] = Update(A[j], A[j-1], NULL);
19       Send(A[j], rank+1);
20     }
21     else if(rank == P-1){
22       Recv(north, rank-1);
23       A[j] = Update(A[j], A[j-1], north);
24     }
25     else{
26       Send(A[j-1], rank+1);
27       ||
28       Recv(north, rank-1);
29       A[j] = Update(A[j], A[j-1], north);
30     }
31   }
```

Figure 4: Greedy algorithm for the first stencil

3.2 Dealing with larger domains

Until now we have assumed that $n = p$. However, in general we will have $n > p$. The question that arises is: What scheme should be used to assign rows to processors? For the sake of simplicity, we will assume in the following that p divides n.

A first answer is to assign a set of consecutive rows to each process. However, such a solution has a major drawback: It does not follow the main idea of our greedy algorithm. Indeed, before P_1 can start working, it has to wait until P_0 has computed the first value on each row assigned to it, i.e., P_1 has to wait at least $\frac{n}{p}$ steps. The same delay will be observed between the time when P_1 starts working and P_2 starts working, etc. We can conclude that such a solution will not allow to take full advantage of all the processors available, as it does not make them start computing as early as possible.

To make the processors start working as early as possible, we can assign rows to processors in a cyclic manner, as illustrated in Figure 5. With this solution, row j of the domain is assigned to processor P_k with $k = j \mod p$.

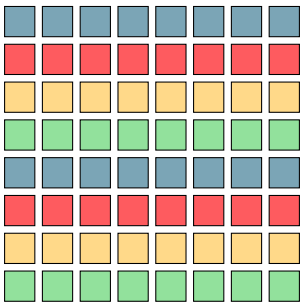

Figure 5: Cyclic row assignment with 4 processes. Each processor is associated with one color

The corresponding version of the algorithm is presented in Figure 6. It is very similar to the algorithm presented in Figure 4. The main difference is that each processor stores $\frac{n}{p}$ rows and needs to iterate over these rows. Note that on each processor, the allocation is made as a continuous set of rows (line 1) although it corresponds to non-contiguous rows in the domain.

We can compute the execution time of our greedy algorithm using cyclic row assignment as a function of p and n. To simplify the notations, we note b as the time to transfer one cell ($b = \frac{sizeof(cell)}{B}$).

First, we compute the time to run one step of the algorithm. In the general case, in a step k, 3 operations need to be run by one process: receiving *north* value for step k; computing one cell; sending value for step $k+1$. One can notice that the send operation (at iteration n) can occur in parallel with the reception (for iteration $n+1$). As such the execution time for one step is:

$$T_{step}(p, n) = w + L + b$$

Here w corresponds to the cost of executing the stencil *Update()* function for one cell.

To compute the total execution time, we need to compute the total number of such steps until the last cell has been updated. This happens when processor P_{p-1} computes the rightmost cell of its last row. We make the following observations:

- It takes $p-1$ steps before processor P_{p-1} starts computing its first cell
- From this point, processor P_{p-1} updates one cell per step
- Processor P_{p-1} has $\frac{n}{p} \times n$ cells to compute in total.

```
1     double A[n/p][n]; /* n/p rows of A */
2     double north = 0; /* to recv North values */
3
4     int rank = my_rank();
5     int P = num_procs();
6
7     for(i=0; i<n/p; i++){
8       if (rank == 0 && i == 0){
9         A[0][0] = Update(A[0][0], NULL, NULL);
10        Send(A[0][0], rank+1);
11      }
12      else{
13        Recv(north, rank-1);
14        A[i][0] = Update(A[i][0], NULL, north);
15      }
16
17      for(j=1; j<n; j++){
18        if (rank == 0 && i == 0){
19          A[i][j] = Update(A[i][j], A[i][j-1], NULL);
20          Send(A[i][j], rank+1);
21        }
22        else if(rank == P-1 && i == n/p-1){ /* case of the last row */
23          Recv(north, rank-1);
24          A[i][j] = Update(A[i][j], A[i][j-1], north);
25        }
26        else{
27          Send(A[i][j-1], (rank+1) % P);
28          ||
29          Recv(north, (rank-1) % P);
30          A[i][j] = Update(A[i][j], A[i][j-1], north);
31        }
32      }
33    }
```

Figure 6: Greedy algorithm for $n > p$

Hence the total execution time of the algorithm is:

$$T(n,p) = (p - 1 + \frac{n^2}{p}) \times (w + L + b)$$

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

$$T(n,p) = \frac{n^2}{p} \times (w + L + b)$$

The execution time of the sequential algorithm is $n^2 \times w$. We can compute the speedup obtained with p processors:

$$
\begin{aligned}
Speedup &= \frac{T_{seq}}{T_{par}} \\
&= \frac{n^2 \times w}{\frac{n^2}{p} \times (w + L + b)} \\
&= p \times \frac{w}{w + L + b} < p
\end{aligned}
$$

Since the speedup is less than p, we can conclude that the algorithm is not optimal. As discussed earlier, the network latency L can have a high impact on the performance of parallel algorithms. In the case of our algorithm, the latency term ($\frac{n^2}{p} \times L$) can have a severe impact on the parallel execution time.

3.3 Bulk communication

To reduce the latency term, the general idea is to try sending less messages. In general, sending less messages is done by sending larger messages.

To send larger messages, a solution is to have each processor computing k values on one row before sending the updates to the next processor. In this case $\frac{n^2}{k}$ messages will be sent in total instead of n^2. On the other hand, processor i will have to wait $i \times k$ steps before starting executing.

We can compute the execution time of this new algorithm. We start by computing the execution time for one step, i.e., the time one processor needs to process k values:

$$T_{step} = k \times (w + b) + L$$

To compute the total execution time, we observe that:

- It takes $p - 1$ steps until processor P_{p-1} starts working.
- Processor P_{p-1} should run $\frac{n^2}{p \times k}$ such steps

Hence, the total execution time of the algorithm is:

$$T_{bulk}(p, n, k) = (p - 1 + \frac{n^2}{p \times k}) \times (k \times (w + b) + L)$$

When n becomes large, the execution time can be simplified as:

$$T_{bulk}(p, n, k) = \frac{n^2}{p} \times (w + b + \frac{L}{k})$$

Hence, we obtain the expected result: the latency term is divided by k. However this solution does not perform that well when n is not large enough for the startup time[1] to be ignored. We can notice that this startup time is partially multiplied by k. Hence, if k is large, the startup time significantly increases.

[1] We call startup time, the time until the last processor starts computing.

A question that arises with this new solution is: what values of k ensure that a processor is never idle ones it has started computing. To answer this question, we study the case of processor P_0. Processor P_0 has to process $\frac{n}{k}$ chunks before starting computing its second allocated row. We also know that it takes p steps until P_0 receives a first update from P_{p-1}. As such the condition that ensures that P_0 is never idle is:

$$p \leq \frac{n}{k} \Rightarrow k \leq \frac{n}{p}$$

3.4 Reducing the amount of data communicated over the network

The solution presented above requires the same amount of data to be communicated over the network as the initial greedy algorithm. To further improve the performance of the algorithm, we should decrease this amount.

The solution to decrease the amount of data communicated over the network is to allocated blocks of r consecutive rows to the same processor as presented in Figure 7. Combined with the previous modification, we obtain a block-cyclic allocation with blocks of size $r \times k$. With this solution, the total amount of communication is divided by r, since only the updates of the last row of a block need to be sent to the next processor.

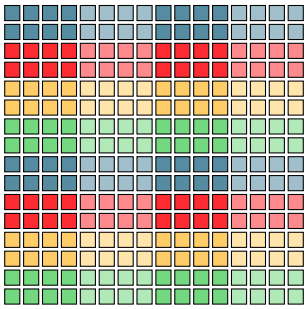

Figure 7: Bloc-cyclic allocation with $p = 4$, $n = 16$, $k = 4$, and $r = 2$. Each processor is associated with one color. Light and dark colors are used to illustrate blocks.

We can compute the execution time of the new algorithm. To do so, we start by computing the execution time for one step, i.e., the time required to process one $r \times k$ block, including the communication time:

$$T_{step} = r \times k \times w + k \times b + L$$

To compute the total execution of the algorithm, we observe that:

- It takes $p - 1$ steps until processor P_{p-1} starts working.
- Processor P_{p-1} should run $\frac{n^2}{p \times r \times k}$ such steps

Hence the total execution time is:

$$T_{block-cyclic}(n, p, r, k) = (p - 1 + \frac{n^2}{p \times r \times k}) \times (r \times k \times w + k \times b + L)$$

When n becomes large, the execution time can be approximated as:

$$T_{block-cyclic}(n, p, r, k) = \frac{n^2}{p} \times (w + \frac{b}{r} + \frac{L}{r \times k})$$

In this new version, both the latency and the bandwidth term are reduced compared to the execution time of the initial greedy algorithm. Increasing r helps improving the speedup observed with the parallel version of the algorithm when n is large. However, one should be careful when selecting r. If r is too large, the cost of the algorithm startup time is going to become prohibitive. Computing the optimal values of r and k goes beyond the scope of this lecture.

4 A second stencil (Jacobi) – TO STUDY OPTIONALLY

To have a more complete study of stencil algorithms, we consider another stencil. The function used to update cells is:

$$c_{new} = Update(c_{old}, W_{old}, E_{old}, N_{old}, S_{old})$$

The new value of the cell depends on the old value of the cell, and on the old values of the 4 neighbors (West, East, North, South), as illustrated in Figure 8. Such a stencil is also employed in some numerical methods (e.g., jacobi numerical method).

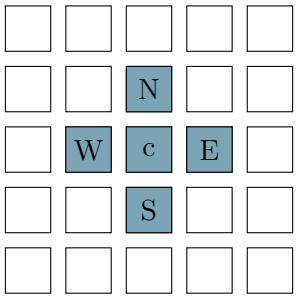

Figure 8: Illustration of the stencil

This new stencil implies 3 major changes compared to the study conducted previously:

- All computations can potentially be run in parallel since the update of a cell only depends on old values.
- It is not possible to update the cells *in place* without overwriting values that would be later needed. Therefore a second copy of the domain needs to be stored in memory with the updated values of the cells.
- Since both *north* and *south* values are needed by the update function, an unidirectional ring is not very appropriate for such a computation. We choose to use a *bidirectional* ring instead.

For this algorithm, we allocate $r = \frac{n}{p}$ consecutive rows to each process. With this allocation, processor P_i:

- Needs to receives data from processor P_{i-1} to be able to run the computation for the first row of its sub-domain;
- Needs to receives data from processor P_{i+1} to be able to run the computation for the last row of its sub-domain;
- Can immediately run the computation for all the other rows of its sub-domain.

For the sake of simplicity in the description of the algorithm, we will assume that the domain can be seen as a torus, i.e., P_0 and P_{p-1} exchange rows. Based on the previous observation, we can conclude that there are 3 main phases in the new stencil algorithm:

1. Exchanging data at the border of the sub-domains

2. Computing the updated values for the cells inside each sub-domains, excluding cells at the border.

3. Computing the updated values for the cells at the border, using the data received from the neighboring processes

Note that phases 1 and 2 can be run in parallel. The new algorithm is described in Figure 9: Phase 1 corresponds to lines 9-16; Phase 2 corresponds to lines 18-25; Phase 3 corresponds to lines 26-34.

```
1     double A[r][n]; /* r rows of A, holding the old values*/
2     double B[r][n]; /* B will hold the updated value */
3
4     double fromN[n], fromS[n];
5
6     int rank = my_rank();
7     int P = num_procs();
8
9     {  /* Phase 1: communication */
10      Send(A[0][*], (rank − 1) mod P);
11      ||
12      Recv(fromN, (rank − 1) mod P);
13      Send(A[r-1][*], (rank + 1) mod P);
14      ||
15      Recv(fromS, (rank + 1) mod P);
16    }
17    ||
18    { /* Phase 2: computation */
19      for(i=1; i<r-1; i++){
20        for(j=0; j<n; j++){
21          B[i][j] = Update(A[i][j], A[i][j-1], A[i][j+1], A[i-1][j],
22              A[i+1][j]);
23        }
24      }
25    }
26    { /* Phase 3: computation */
27      for(j=0; j<n; j++){
28        B[0][j] = Update(A[0][j], A[0][j-1], A[0][j+1], fromN[j],
29            A[1][j]);
30      }
31      for(j=0; j<n; j++){
32        B[r-1][j] = Update(A[r-1][j], A[r-1][j-1], A[r-1][j+1], A[r-2][j],
              fromS[j]);
33      }
34    }
```

Figure 9: Algorithm for the new stencil. The notation A[i][j-1] is not strictly correct. The notation $A[i][(j-1) \mod n]$ would be more precise. The notation $A[0][*]$ is used to make it explicit that a full row of A is sent.

We compute the execution time of the new algorithm. For that, we start by computing the execution time of each phase:

$$T_{phase1} = 2 \times (L + n \times b)$$
$$T_{phase2} = n \times (r - 2) \times w = (\frac{n^2}{p} - 2 \times n) \times w$$
$$T_{phase3} = 2 \times n \times w$$

Hence the total execution time of the algorithm is:

$$T = \max\left(2 \times (L + n \times b), (\frac{n^2}{p} - 2 \times n) \times w\right) + 2 \times n \times w$$

We can observe that when n becomes large, the execution time can be approximated as:

$$T = \frac{n^2}{p} \times w$$

which implies that the algorithm is asymptotically optimal. We can conclude that a virtual ring topology is very appropriate to implement parallel versions of this stencil.