Lecture notes: Parallel Algorithms in Distributed Memory Systems

Master M1: Parallel Algorithms and Programming

2020

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 prefetchers 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 prefers 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.

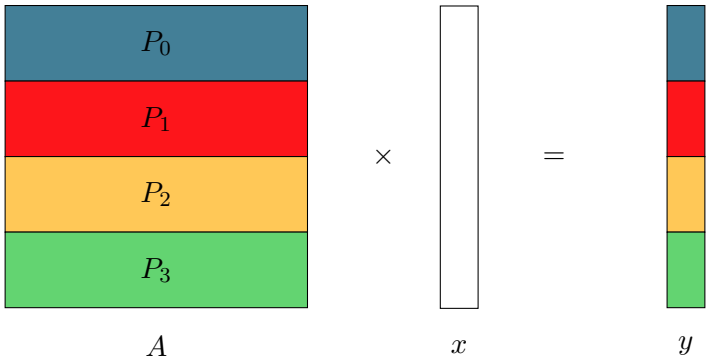


Figure 2: Distributed matrix-vector multiplication over 4 processors. All data with the same color are on the same processor

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.

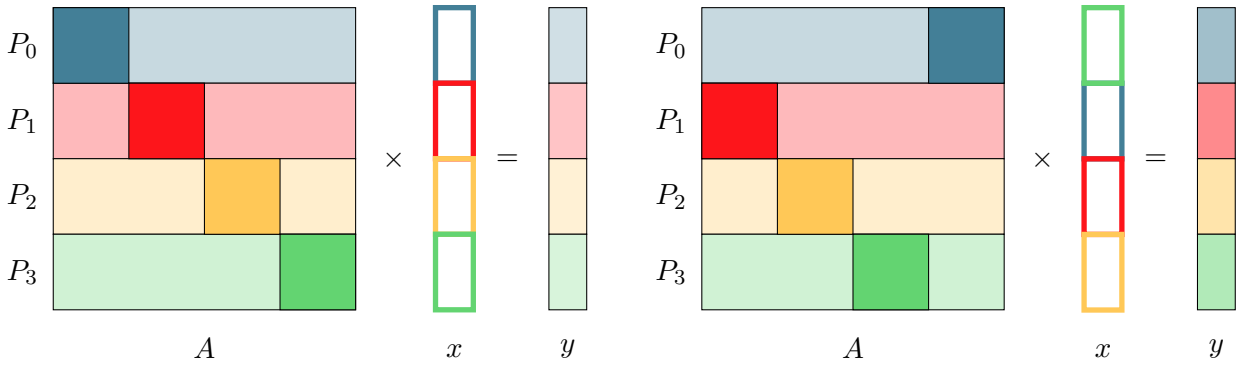
The algorithm presented in Figure 4 details the parallel version of the matrix-vector multiplication. A few points should be discussed about this algorithm:

- 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; step++){
12     Send(tempS, (rank + 1) mod P);
13     ||
14     Recv(tempR, (rank − 1) mod P)
15     ||
16     int block= (rank − step) mod P;
17     for(i=0; i<r; i++){
18       for(j=0; j<r; j++){
19         y[i] = y[i] + A[i, r × block + j)] × tempS[j]
20       }
21     }
22     tempS ← tempR;
23   }
```

Figure 4: Parallel matrix-vector multiplication. The symbol ‖ implies that lines 12, 14 and 17 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).

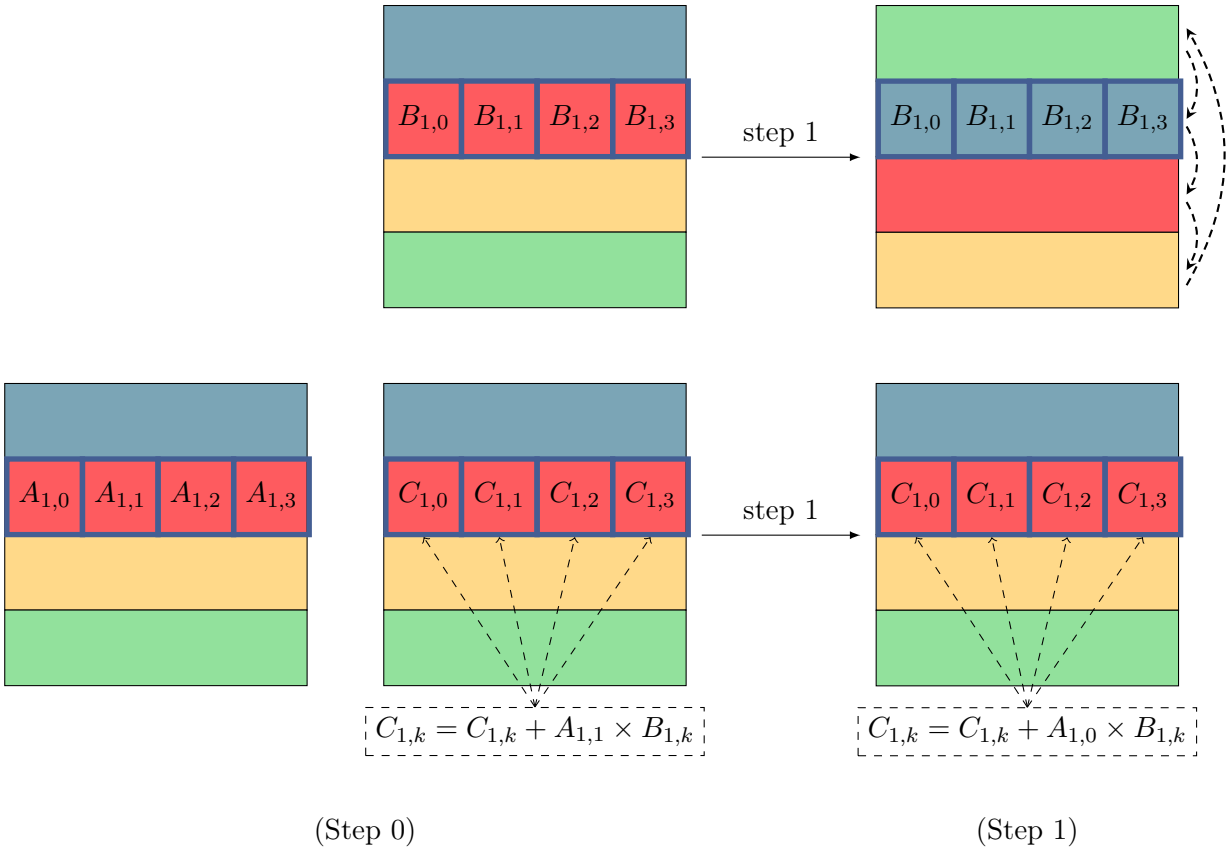


Figure 6: Parallel matrix-matrix multiplication with 4 processes. The figure presents 2 iteration of the algorithms and focuses on the computation run by process P_1. The notation $A_{l,k}$ refers to the block k in the part of matrix A stored by process l.

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 (*i.e.*, it is $n \times p \times L$). 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; step++){
12    Send(tempS, (rank + 1) mod P);
13    ||
14    Recv(tempR, (rank − 1) mod P)
15    ||
16    int block = (rank − step) mod P;
17    for(l=0; l<P; l++){
18      for(i=0; i<r; i++){
19        for(j=0; j<r; j++){
20          for(k=0; k<r; k++){
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 17 can be executed in parallel. The symbol ← implies a memory copy.

4 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 8. When the studied domain becomes very large, one may want to run the stencil in parallel on multiple nodes.

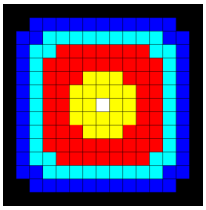


Figure 8: An example of 2D stencil presented during the first lecture (heat diffusion).

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.

5 A first stencil algorithm

The first 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 9.

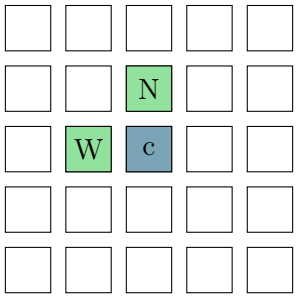


Figure 9: Illustration of the stencil

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).

5.1 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.

As in the previous algorithms, 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 10 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.

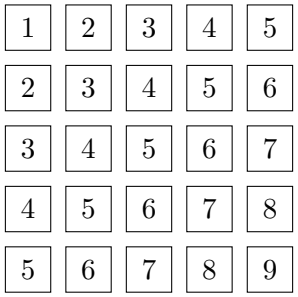


Figure 10: Parallel execution of the greedy algorithm

Figure 11 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}.

5.2 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 use the same strategy as what we used for matrix-vector and matrix-matrix multiplication. 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 12. With this solution, row j of the domain is assigned to processor P_k with $k = j \mod p$.

The corresponding version of the algorithm is presented in Figure 13. It is very similar to the algorithm presented in Figure 11. The main difference is that each processor stores $\frac{n}{p}$ rows and needs

```
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 11: Greedy algorithm for the first stencil

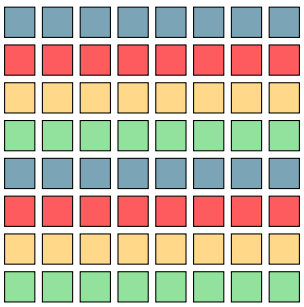


Figure 12: Cyclic row assignment with 4 processes. Each processor is associated with one color

to iterate over these rows. Note that on each processor, the allocation is made as a continuous set of rows (line 1) on each processor although it corresponds to non-contiguous rows in the domain.

```
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 13: Greedy algorithm for $n > p$

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.

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.

5.3 Bulk communication

When studying matrix-matrix multiplication, we have seen that sending large messages can help reducing the latency term in the execution time of parallel algorithms.

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[2] 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.

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:

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

5.4 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 14. 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.

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

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

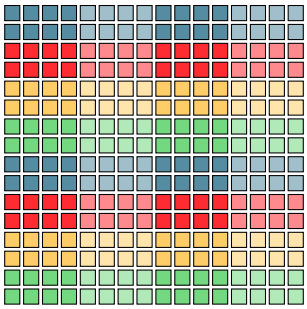


Figure 14: Bloc-cyclic allocation with $p = 4$, $n = 16$, $k = 4$, and $r = 2$. Each processor is associated with one color. Light and dark colors are used to illustrate blocks.

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.

6 Summary of some algorithmic principles

While studying these parallel algorithms, we have introduced some general (and sometimes contradictory) principles that can be applied to develop efficient algorithms in distribute shared memory:

- *Sending data in bulk* to limit the impact of network latency
- *Sending data early* to avoid having idle processors
- *Overlapping communication and computation*
- *Assigning blocks of data to processors* to limit the amount of communication and have regular access patterns for the computation
- *Applying cyclic data distribution* to increase load balancing between processors and reduce idle time.

References

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

- Section 4.1, 4.2, 4.3, 4.5, 4.6, 4.7, and 4.8 of the book "Parallel Algorithms" (by Casanova, Robert, and Legrand).

The material presented in this document is strongly inspired by these sections of the book.

7 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 15. Such a stencil is also employed in some numerical methods (e.g., jacobi numerical method).

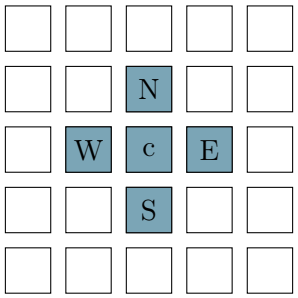


Figure 15: Illustration of the stencil

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 16: Phase 1 corresponds to lines 9-16; Phase 2 corresponds to lines 18-25; Phase 3 corresponds to lines 26-34.

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.

```
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 16: 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.