Parallel Algorithms and Programming

Parallel Algorithms in Distributed Memory Systems (Part 2)

Thomas Ropars

Email: thomas.ropars@univ-grenoble-alpes.fr

Website: tropars.github.io

In this lecture

- Performance of parallel distributed algorithms
- Stencil algorithms
- Some major concepts
    - Greedy algorithm
    - Bulk communication
    - block-cyclic allocation

Introduction

About the performance of distributed parallel algorithm

Assumptions

- A distributed memory system
    - Processes communicate by sending and receiving messages
- A **virtual ring** topology

Main costs to consider

- 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)

Cost of a distributed parallel algorithm

- A computational term:

$$\text{nb of flops} \times \text{time per flop}$$

- A bandwidth term:

$$\text{amount of data moved} \times \frac{1}{bandwidth}$$

- A latency term:

$$\text{nb of messages} \times latency$$

Cost of a distributed parallel algorithm

We should remember that (in general):

$$\text{time per flop} \ll \frac{1}{bandwidth} \ll latency$$

Consequence

- Minimizing communication is important for performance

- It is also important for energy efficiency
  - Moving data from/to DRAM or over the interconnection network are the most energy consuming operations

About memory accesses

Cost of moving data between the memory and the processor

- Can often be ignored
    - Cost mostly hidden by hardware prefetchers

Prefetchers

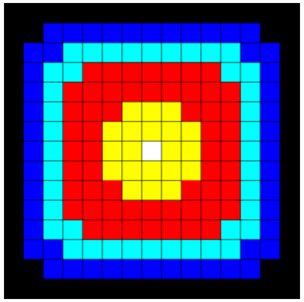
- Hardware mechanisms to load data in the cache before it is needed
- Based on the memory access pattern of the program
- Works well for regular access patterns
    - It is the case for the programs studied in this lecture

Stencil algorithms

About stencil algorithms

A common class of parallel applications

- Operate on cells that divide a discrete domain



Main properties of the cells

- Cells can hold one or multiple values
- Cells are in general organized in a N-dimensional grid (often 2D or 3D)
    - They are non-overlapping
    - They are of equal size (load balancing)
- Cells can be distributed over multiple nodes to execute a program in parallel
    - Deal with large problems

About stencil algorithms

Definition of a stencil

- Iterative algorithm
- Applies a pre-defined function to update the value of the cells based on the value of the neighboring cells

- Specifying a stencil boils down to defining:
    - The **location of a cell's neighbors**
    - The **function used to update cell values**

- The combination of both forms a *stencil*
    - Applied to all cells in the domain

2-dimensional domains

In the following, we consider stencils applying to 2-dimensional domains

- In this case, a cell can have at most 8 neighbors
  - Cardinal coordinates: N, NE, E, SE, S, SW, W, NW.

NW	N	NE
W	Cell	E
SW	S	SE

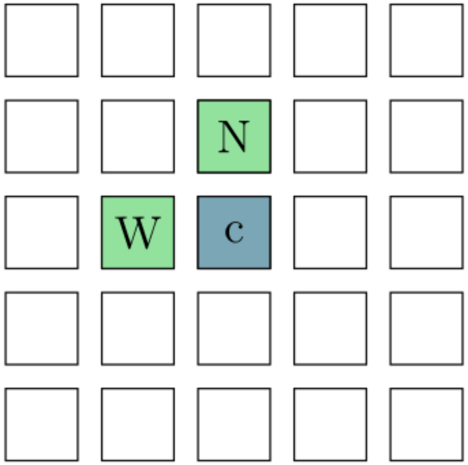
- 9-point stencil: The stencil function uses as input the value of the cell and of the 8 neighbors

A case study

A stencil algorithm

Definition of the stencil

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



- 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.

Some comments

Stencil of practical importance

- At the root of some numerical algorithms

Case of the border cells

- Might not have `West` or `North` neighbors
- A modified `Update` function is applied to these cells
    - For instance, using a constant value for the non-existent cells.

Main assumptions that we make

- $n \times n$ cells
- p processors
    - Unidirectional ring

Greedy algorithm

```
*Greedy* = the algorithm tries to put all processors to work as
early as possible
```

First version (assuming $n = p$)

- Each processor stores 1 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}.

Notation used in the following

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

Parallel execution of the greedy algorithm

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

- The number associated with each cell is the step in which the cell is updated
  - In step k, the k-th anti-diagonal can be computed.

Description of the algorithm

Some special cases (in terms of communication)

- Processor P_0 does not receive `North` values from another processor
- Processor P_{n-1} does not need to send its updated values

General algorithm

- At iteration $i + j$, processor P_i performs the following operations:
    1. Receives $A_{i-1,j}$ from P_{i-1};
    2. Computes $A_{i,j}$;
    3. Sends $A_{i,j}$ to P_{i+1}.

Detailed description of the algorithm

```
double A[n]; /* one row of A, assumed to be already initialized*/
double north = 0; /* to recv North values */

int rank = my_rank();
int P = num_procs();

if (rank == 0){
  A[0] = Update(A[0], NULL, NULL);
  Send(A[0], rank+1);
}
else{
  Recv(north, rank-1);
  A[0] = Update(A[0], NULL, north);
}

for(j=1; j<n; j++){
  if (rank == 0){
    A[j] = Update(A[j], A[j-1], NULL);
    Send(A[j], rank+1);
  }
  else if(rank == P-1){
    Recv(north, rank-1);
    A[j] = Update(A[j], A[j-1], north);
  }
  else{
    Send(A[j-1], rank+1);
    ||
    Recv(north, rank-1);
    A[j] = Update(A[j], A[j-1], north);
  }
}
```

Generalizing the algorithm

What if $n > p$?

- How to assign rows to processors ?

First solution

- Assigning a set of consecutive rows to the same processors

Generalizing the algorithm

What if $n > p$?

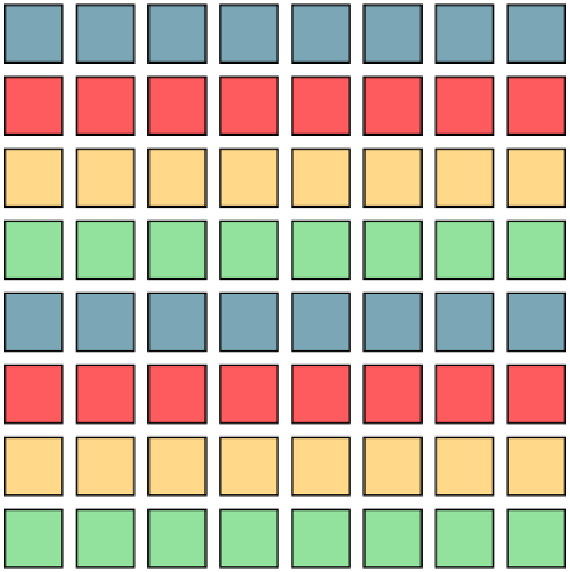
- How to assign rows to processors ?

First solution

- Assigning a set of consecutive rows to the same processors
  - Problem: The algorithm is not greedy anymore
    - P_1 can start working when P_0 has computed the first value on each row assigned to it
    - P_1 waits at least $\frac{n}{p}$ steps
    - P_2 waits at least $2 \times \frac{n}{p}$ steps
    - **Processors do not start computing as early as possible**

Cyclic row assignment

Solution to ensure that each processor starts as early as possible



- Row j of the domain is assigned to processor P_k with $k = j \mod p$

Performance of the algorithm

Extra notation

- b -- time to transfer one cell: $b = \frac{sizeof(cell)}{B}$
- w -- cost of executing the stencil `Update()` function for one cell

Time to run one step of the algorithm

- 3 operations need to be run by one process
    1. Receiving `north` value for step k
    2. Computing one cell
    3. Sending value for step $k+1$
- Comment: The send operation (at iteration n) can occur in parallel with the reception (for iteration $n+1$)

Performance of the algorithm

Extra notation

- b -- time to transfer one cell: $b = \frac{sizeof(cell)}{B}$
- w -- cost of executing the stencil `Update()` function for one cell

Time to run one step of the algorithm

- 3 operations need to be run by one process
  1. Receiving `north` value for step k
  2. Computing one cell
  3. Sending value for step $k+1$
- Comment: The send operation (at iteration n) can occur in parallel with the reception (for iteration $n+1$)

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

Performance of the algorithm

Total number of steps

When does the last processor computes its last cell?

Performance of the algorithm

Total number of steps

When does the last processor computes its last cell?

- 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.

Performance of the algorithm

Total number of steps

When does the last processor computes its last cell?

- 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.

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

Speedup achieved by the algorithm

Some comments:

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

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

- The sequential execution time is:

$$n^2 \times w$$

Speedup on p processors

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

Speedup achieved by the algorithm

Some comments:

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

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

- The sequential execution time is:

$$n^2 \times w$$

Speedup on p processors

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

Can we do better?

Speedup achieved by the algorithm

Some comments:

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

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

- The sequential execution time is:

$$n^2 \times w$$

Speedup on p processors

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

Can we do better?

- The network latency L can have a high impact on the performance of parallel algorithms
  - Let's try to reduce the latency term!

Bulk communication

- To reduce the impact of latency, the idea is to send less messages
    -

Bulk communication

- To reduce the impact of latency, the idea is to send less messages
    - We need to send larger messages
    - **A processor computes k values on one row before sending updates to the next processors**

New performance

- Execution time of one step:

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

- Number of steps:
  - 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

Execution time:

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

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

Discussion on performance

Some comments

- We obtain the expected result: the latency term is divided by k
- This solution does not perform that well when n is not large enough
  - In this case, the startup time cannot be ignored

Discussion on performance

Some comments

- We obtain the expected result: the latency term is divided by k
- This solution does not perform that well when n is not large enough
  - In this case, the startup time cannot be ignored

What values of k ensure that a processor is never idle ones it has started computing ?

- Case of processor P_0
  - P_0 has to process $\frac{n}{k}$ chunks before starting computing its second allocated row
  - It takes p steps until P_0 receives a first update from P_{p-1}

- Condition to be met:

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

Reducing the amount of communication

To further improve the performance we need to reduce the amount of data sent over the network.

-

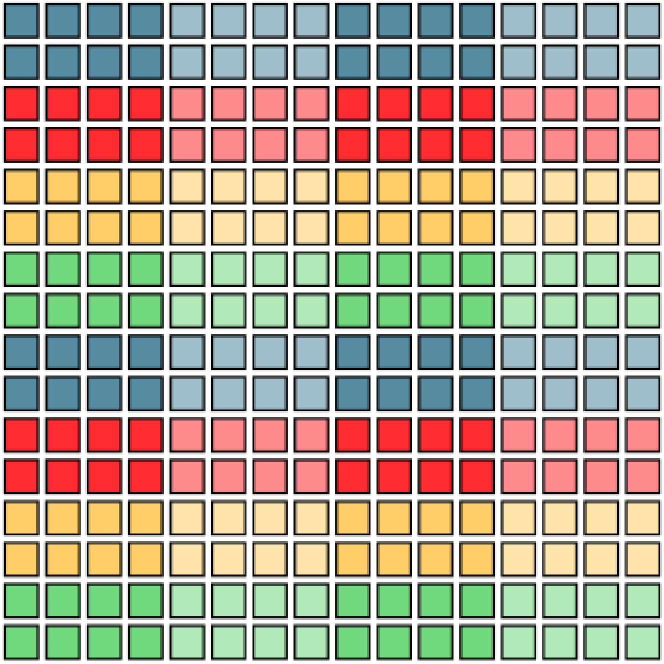
Reducing the amount of communication

To further improve the performance we need to reduce the amount of data sent over the network.

Solution

- Allocate blocks of r consecutive rows to the same processor
- **Concept of block-cyclic allocation**
  - Blocks of size $r \times k$
  - Total amount of communication is divided by r
    - Only the last row of each block is sent

Block-cyclic allocation



- 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.

Performance of the block-cyclic-allocation version

- Execution time for one step
  - = time required to process one $r \times k$ block

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

- Number of steps:
  - 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

Execution time

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

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

Performance

Comments on this new version

- The block-cyclic allocation helps reducing both the bandwidth and the latency term
- However, if r is too large, the startup time is going to become too costly

Comments on the implementation

Concept of ghost cells

Problem

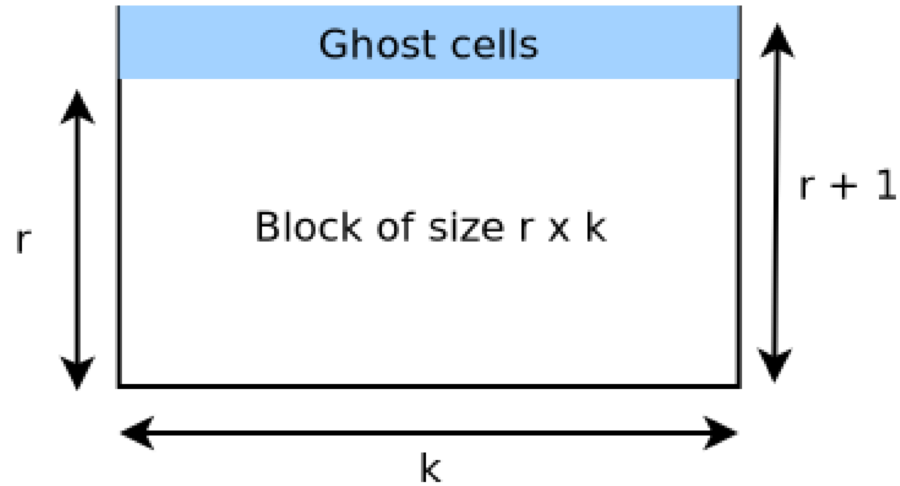
- We consider the case of a block allocation
  - A process computes over several consecutive rows
- Data are stored in different buffers (based on the figure presented in Slide 18)
  - The local domain: A (one allocation)
  - The values received from the `North` processor (*north vector*).

- **How to avoid having to describe a special case for the computation of the first row of each block?**
  - Special case: Read the north values from the *north vector* when applying `Update` to the first row of a block.

Concept of ghost cells

Solution: Allocating ghost cells

- Allocating a block of size $(r+1) \times k$ instead of $r \times k$
  - The extra row is used to receive data from north



Conclusion

- Study of stencil parallel algorithms in distributed environment
    - Impact of assignment of sub-domains to processes on performance

- Some general (and sometimes contradictory) principles 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
    - **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

- Section 4.1 and 4.3 of the book "Parallel Algorithms" (by Casanova, Robert, and Legrand).