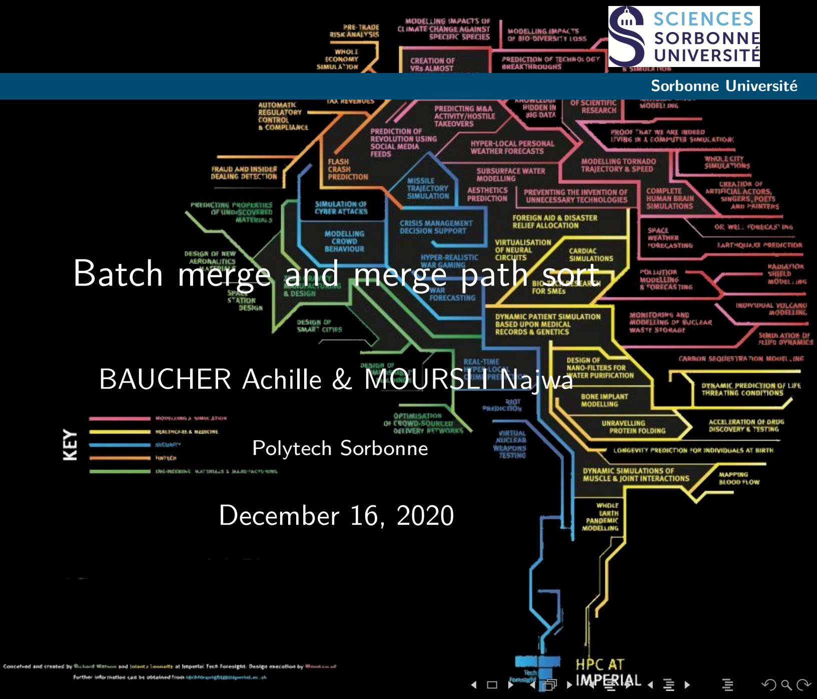

PRE-TRADE RISK ANALYSIS
MODELLING IMPACTS OF CLIMATE CHANGE AGAINST SPECIFIC SPECIES
MODELLING IMPACTS OF BIO-DIVERSITY LOSS
WHOLE ECONOMY SIMULATION
CREATION OF VRs ALMOST
PREDICTION OF TECHNOLOGY BREAKTHROUGHS
AUTOMATIC REGULATORY CONTROL & COMPLIANCE
TAX REVENUES
PREDICTING M&A ACTIVITY/HOSTILE TAKEOVERS
KNOWLEDGE HIDDEN IN BIG DATA
OF SCIENTIFIC RESEARCH
PREDICTION OF REVOLUTION USING SOCIAL MEDIA FEEDS
HYPER-LOCAL PERSONAL WEATHER FORECASTS
PROOF THAT WE ARE INDEED LIVING IN A COMPUTER SIMULATION
FRAUD AND INSIDER DEALING DETECTION
FLASH CRASH PREDICTION
MISSILE TRAJECTORY SIMULATION
SUBSURFACE WATER MODELLING
MODELLING TORNADO TRAJECTORY & SPEED
WHOLE CITY SIMULATIONS
AESTHETICS PREDICTION
PREVENTING THE INVENTION OF UNNECESSARY TECHNOLOGIES
COMPLETE HUMAN BRAIN SIMULATIONS
CREATION OF ARTIFICIAL ACTORS, SINGERS, POETS AND PAINTERS
PREDICTING PROPERTIES OF UNDISCOVERED MATERIALS
SIMULATION OF CYBER ATTACKS
CRISIS MANAGEMENT DECISION SUPPORT
FOREIGN AID & DISASTER RELIEF ALLOCATION
SPACE WEATHER FORECASTING
OIL WELL FORECASTING
EARTHQUAKE PREDICTION
MODELLING CROWD BEHAVIOUR
VIRTUALISATION OF NEURAL CIRCUITS
CARDIAC SIMULATIONS
DESIGN OF NEW AERONAUTICS
HYPER-REALISTIC WAR GAMING
RADIATION SHIELD MODELLING
POLLUTION MODELLING & FORECASTING
SPACE STATION DESIGN
& DESIGN
WAR FORECASTING
FOR SMEs
INDIVIDUAL VOLCANO MODELLING
DESIGN OF SMART CITIES
DYNAMIC PATIENT SIMULATION BASED UPON MEDICAL RECORDS & GENETICS
MONITORING AND MODELLING OF NUCLEAR WASTE STORAGE
SIMULATION OF FLUID DYNAMICS
CARBON SEQUESTRATION MODELLING
REAL-TIME
DESIGN OF NANO-FILTERS FOR WATER PURIFICATION
DYNAMIC PREDICTION OF LIFE THREATENING CONDITIONS
RIOT PREDICTION
BONE IMPLANT MODELLING
OPTIMISATION OF CROWD-SOURCED DELIVERY NETWORKS
UNRAVELLING PROTEIN FOLDING
ACCELERATION OF DRUG DISCOVERY & TESTING
VIRTUAL NUCLEAR WEAPONS TESTING
LONGEVITY PREDICTION FOR INDIVIDUALS AT BIRTH
DYNAMIC SIMULATIONS OF MUSCLE & JOINT INTERACTIONS
MAPPING BLOOD FLOW
WHOLE EARTH PANDEMIC MODELLING
KEY
MODELLING & SIMULATION
HEALTHCARE & MEDICINE
SECURITY
FINTECH
ENGINEERING, MATERIALS & MANUFACTURING
HPC AT IMPERIAL
Tech Foresight
Conceived and created by Richard Watson and Iolanta Leonardi at Imperial Tech Foresight. Design execution by
Further information can be obtained from

# Batch merge and merge path sort

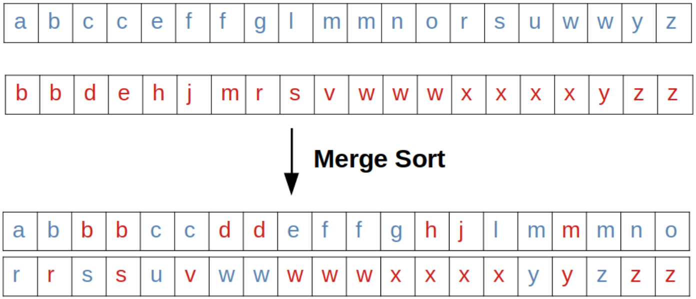
BAUCHER Achille & MOURSLI Najwa

Polytech Sorbonne

December 16, 2020

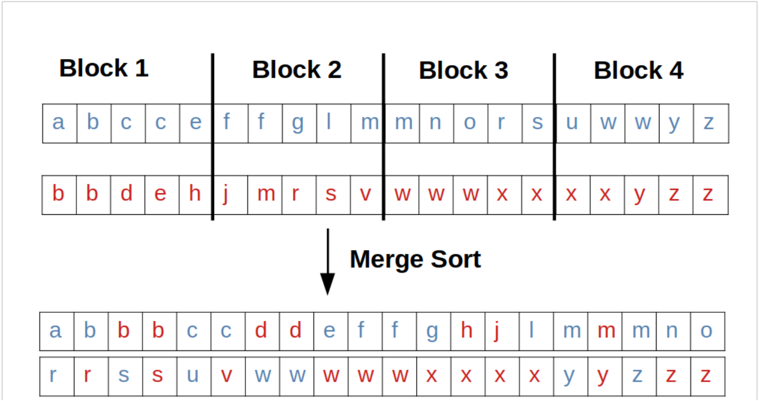
# Summary

## Main Goal



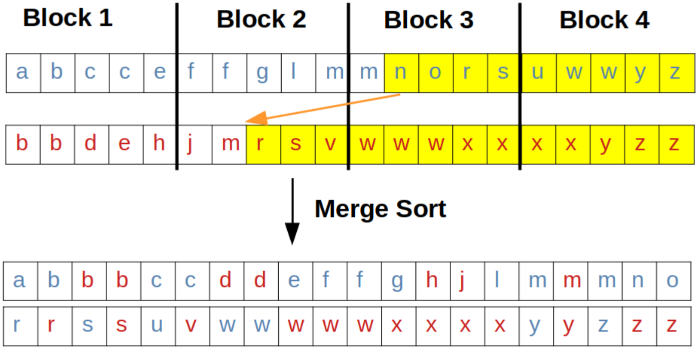
a b c c e f f g l m m n o r s u w w y z
b b d e h j m r s v w w w x x x x y z z
Merge Sort
a b b b c c d d e f f g h j l m m m n o
r r s s u v w w w w w x x x x y y z z z

## Can be distributed



Block 1
Block 2
Block 3
Block 4
a b c c e f f g l m m n o r s u w w y z
b b d e h j m r s v w w w x x x x y z z
Merge Sort
a b b b c c d d e f f g h j l m m m n o
r r s s u v w w w w w x x x x y y z z z

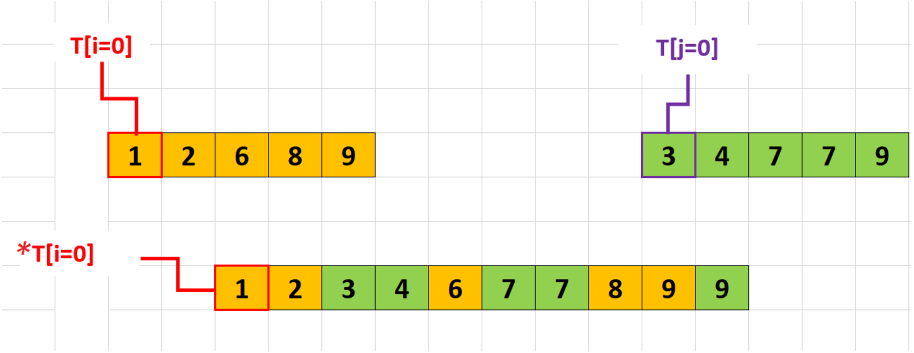
# Task distribution


Block 1
Block 2
Block 3
Block 4
a b c c e f f g l m m n o r s u w w y z
b b d e h j m r s v w w w x x x x y z z
Merge Sort
a b b b c c d d e f f g h j l m m m n o
r r s s u v w w w w w x x x x y y z z z

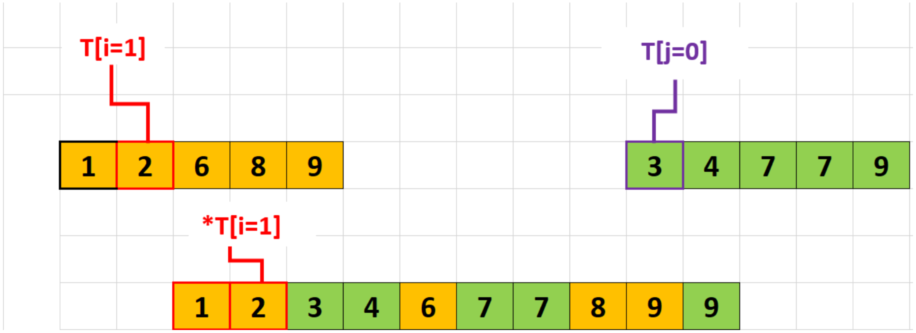
# Summary

## Serial Merge Path

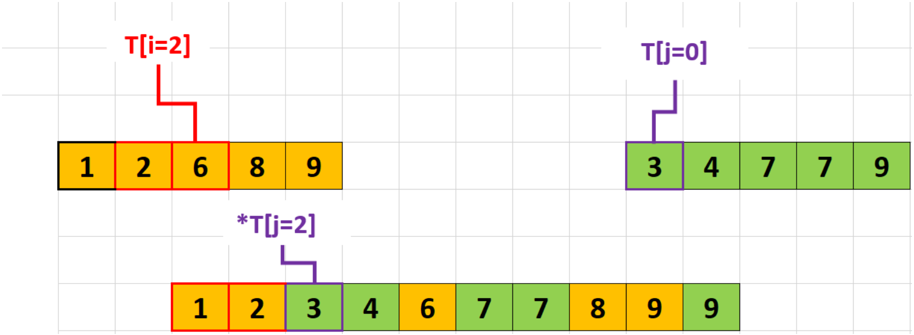
The most common idea is to use two pointers and increment one of them whenever it has the lowest value among both, while copying its value to the output array:



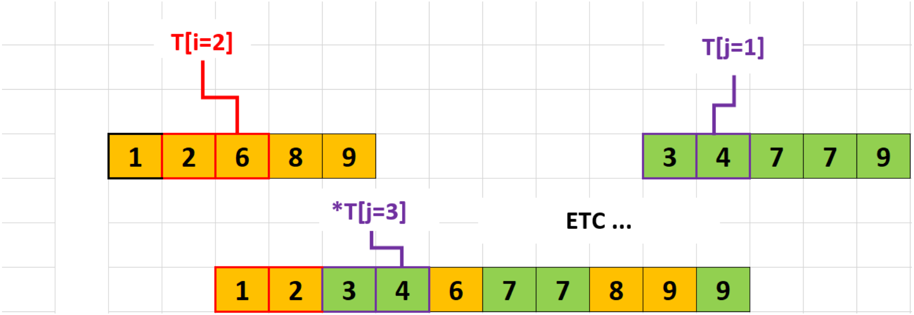
## Serial Merge Path



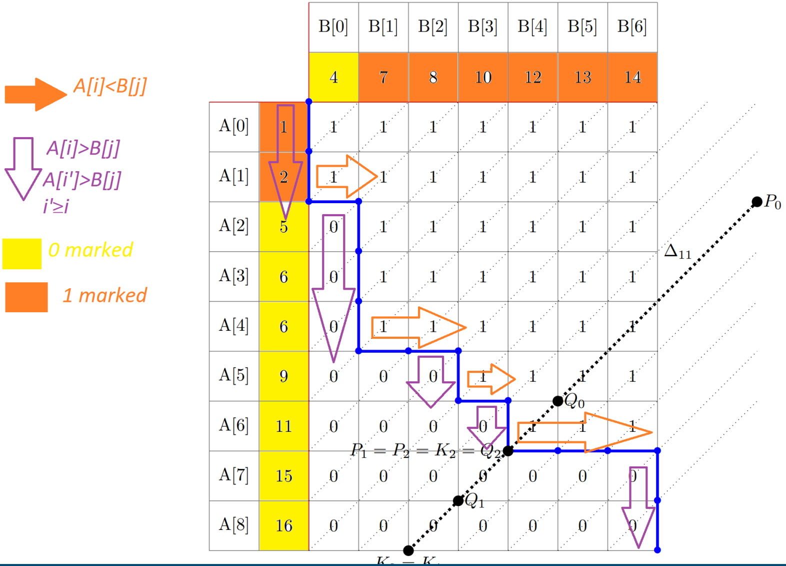
# Serial Merge Path



# Serial Merge Path



# On our example



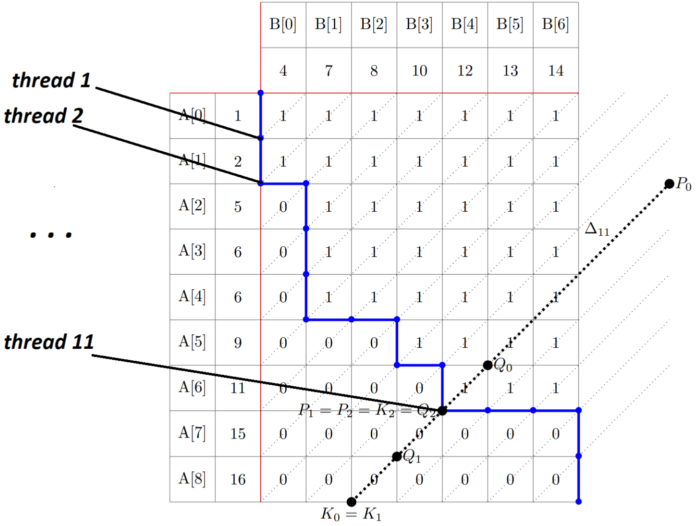
| | | B[0] | B[1] | B[2] | B[3] | B[4] | B[5] | B[6] |
|---|---|---|---|---|---|---|---|---|
| | | 4 | 7 | 8 | 10 | 12 | 13 | 14 |
| A[0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[1] | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[2] | 5 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[3] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[4] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[5] | 9 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| A[6] | 11 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| A[7] | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A[8] | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## On our example

- It is simple to implement
- It has always the same linear complexity $\Theta(|A| + |B|)$ : no bad surprise
- In its original form, it is impossible to parallelize the algorithm.
- An average linear complexity is not optimal

# Blocks and memories use

For one block :

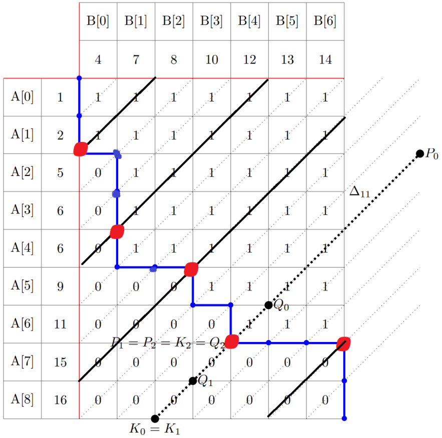


## Several Blocks and memories use

- **Parallel Partition**
  ⇒ Map exactly the necessary inputs into each tile.
- **Merge Path**
  ⇒ Cross diagonal binary search **GLOBAL MEMORY**
- **MergeSort of the Merge Path** ⇒ Placing the Merge Path on the **SHARED MEMORY** ⇒ Blocksort

## Blocks and memories use

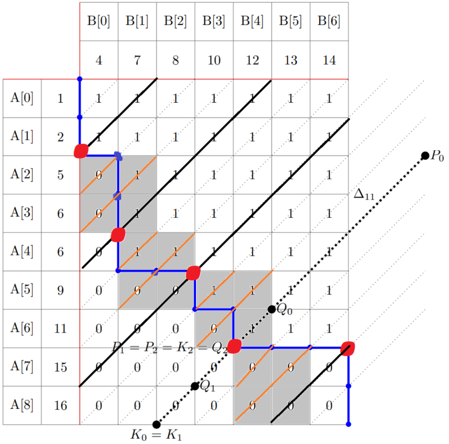
Diagonals divide the work among the blocks.Points of intersection using binary searches on the cross diagonals by comparing elements from A and B on **GLOBAL MEMORY**.



|  |  | B[0] | B[1] | B[2] | B[3] | B[4] | B[5] | B[6] |
|---|---|---|---|---|---|---|---|---|
|  |  | 4 | 7 | 8 | 10 | 12 | 13 | 14 |
| A[0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[1] | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[2] | 5 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[3] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[4] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[5] | 9 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| A[6] | 11 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| A[7] | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A[8] | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Blocks and memories use

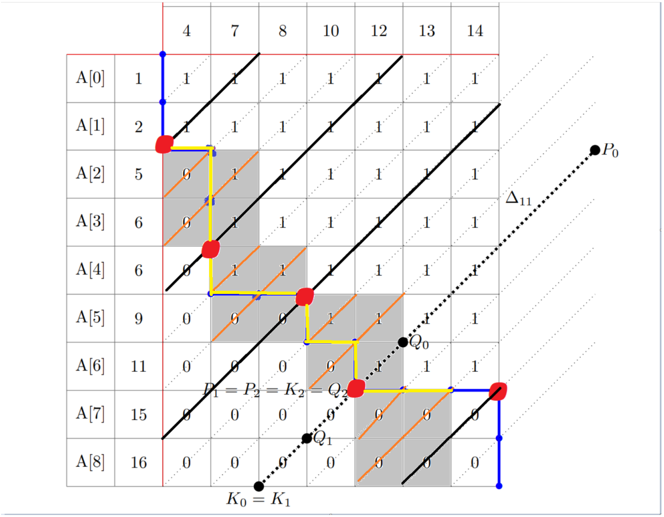
Take a window consisting of the Z largest elements of each of the partitions and place them in local **SHARED MEMORY**.



|  |  | B[0] | B[1] | B[2] | B[3] | B[4] | B[5] | B[6] |
|---|---|---|---|---|---|---|---|---|
|  |  | 4 | 7 | 8 | 10 | 12 | 13 | 14 |
| A[0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[1] | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[2] | 5 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[3] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[4] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[5] | 9 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| A[6] | 11 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| A[7] | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A[8] | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Blocks and memories use

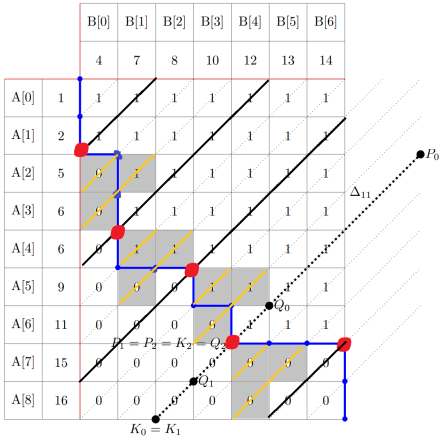
Write the Merge Path found on **GLOBAL MEMORY**.



| | | 4 | 7 | 8 | 10 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|
| A[0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[1] | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[2] | 5 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[3] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[4] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[5] | 9 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| A[6] | 11 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| A[7] | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A[8] | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Blocks and memories use

Several cycles in parallel and Place the Merge Path on the **SHARED MEMORY**$\Rightarrow$ MERGE.



|  |  | B[0] | B[1] | B[2] | B[3] | B[4] | B[5] | B[6] |
|---|---|---|---|---|---|---|---|---|
|  |  | 4 | 7 | 8 | 10 | 12 | 13 | 14 |
| A[0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[1] | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[2] | 5 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[3] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[4] | 6 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A[5] | 9 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| A[6] | 11 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| A[7] | 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A[8] | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Blocks and memories use

3 different cases illustration:

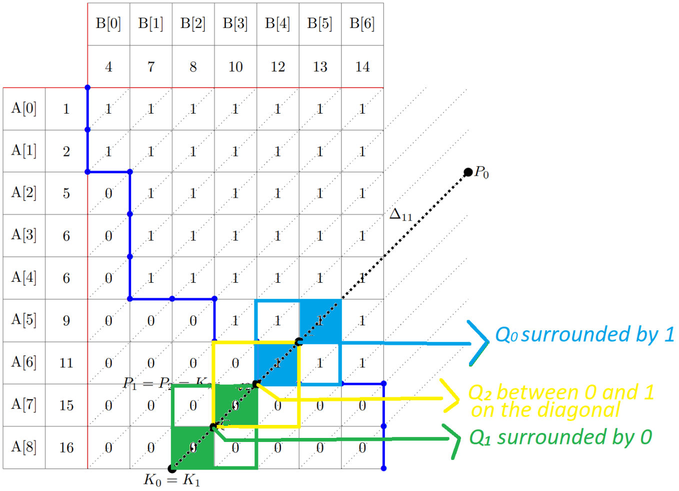
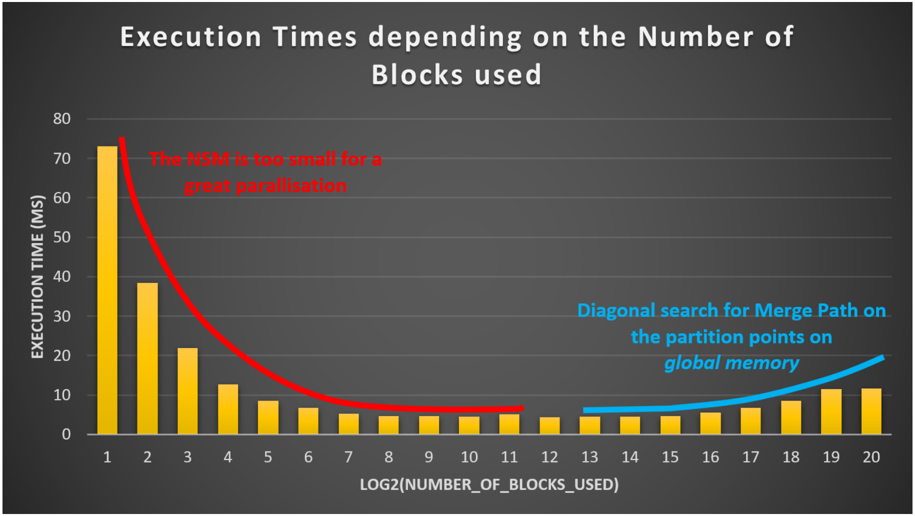


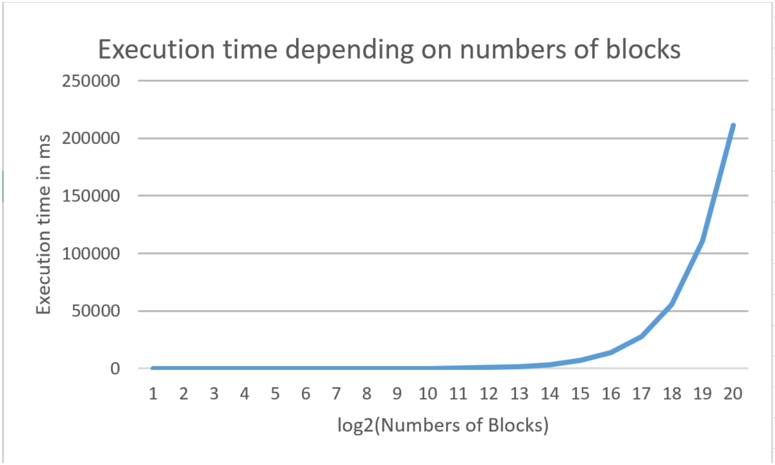
Figure 1: An example of Merge Path procedure

## Performances for our MergeSort

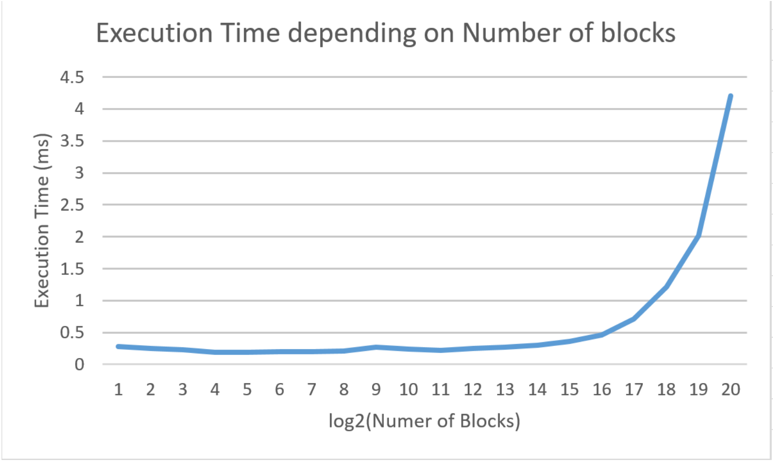
### Optimal number of blocks



## Performances of 1.3



## Performances of 1.2

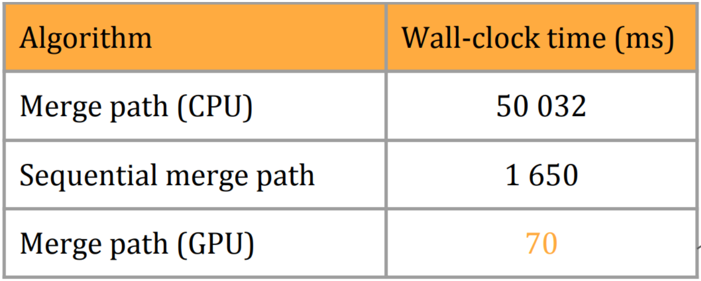


## Performances: Wall Clock Time

- We decided to compare the wall-clock time of merging randomly generated vectors of integers for the three algorithms. For the GPU algorithm, we also monitored the wall-clock time for sort initialization ⇒**CUDA EVEN TIMER**
- Sanity check We implemented a sanity check (on CPU for the three versions) to check if the array is really sorted in the end of the program.

## Performances: Wall Clock Time

We ran 100 simulations of the three algorithms, gathering the average wall-clock time of the merge part only. The task consisted in merging $2^{20} = 1{,}048{,}576$ arrays of size 2

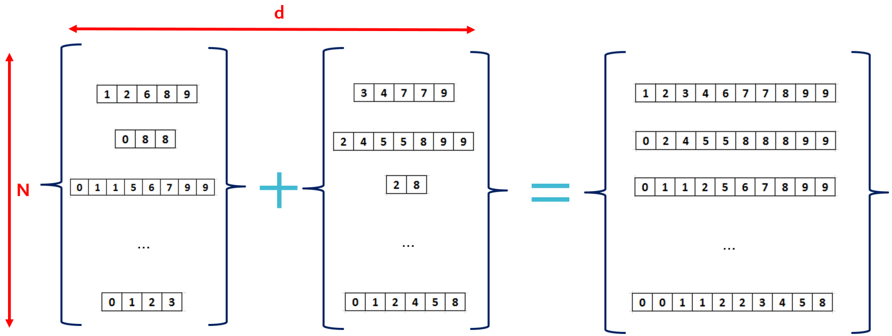


| Algorithm | Wall-clock time (ms) |
|---|---|
| Merge path (CPU) | 50 032 |
| Sequential merge path | 1 650 |
| Merge path (GPU) | 70 |

$\approx \times 20$

## Summary

## Definition


d
N
1 2 6 8 9
0 8 8
0 1 1 5 6 7 9 9
...
0 1 2 3
3 4 7 7 9
2 4 5 5 8 9 9
2 8
...
0 1 2 4 5 8
1 2 3 4 6 7 7 8 9 9
0 2 4 5 5 8 8 8 9 9
0 1 1 2 5 6 7 8 9 9
...
0 0 1 1 2 2 3 4 5 8

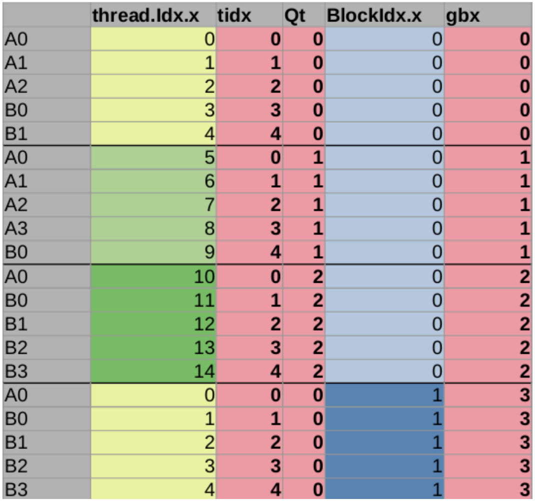
## Indexes Importance

d: number of elements in a pair of array

m: number of pairs of array mergesorted in a single block

1. int tidx = threadIdx.x%d ⇒ Enumeration from 0 to d-1 of the elements of each of the m pairs of arrays which are going to be sorted in a single block
2. int Qt = (threadIdx.x-tidx)/d ⇒ Local enumeration from 0 to m-1 of the pairs of arrays in a single block
3. int gbx = Qt + blockIdx.x*(blockDim.x/d) ⇒ Global enumeration from 0 to N-1 of the pairs of arrays in all the blocks

## Indices



| | thread.Idx.x | tidx | Qt | BlockIdx.x | gbx |
|---|---|---|---|---|---|
| A0 | 0 | 0 | 0 | 0 | 0 |
| A1 | 1 | 1 | 0 | 0 | 0 |
| A2 | 2 | 2 | 0 | 0 | 0 |
| B0 | 3 | 3 | 0 | 0 | 0 |
| B1 | 4 | 4 | 0 | 0 | 0 |
| A0 | 5 | 0 | 1 | 0 | 1 |
| A1 | 6 | 1 | 1 | 0 | 1 |
| A2 | 7 | 2 | 1 | 0 | 1 |
| A3 | 8 | 3 | 1 | 0 | 1 |
| B0 | 9 | 4 | 1 | 0 | 1 |
| A0 | 10 | 0 | 2 | 0 | 2 |
| B0 | 11 | 1 | 2 | 0 | 2 |
| B1 | 12 | 2 | 2 | 0 | 2 |
| B2 | 13 | 3 | 2 | 0 | 2 |
| B3 | 14 | 4 | 2 | 0 | 2 |
| A0 | 0 | 0 | 0 | 1 | 3 |
| B0 | 1 | 1 | 0 | 1 | 3 |
| B1 | 2 | 2 | 0 | 1 | 3 |
| B2 | 3 | 3 | 0 | 1 | 3 |
| B3 | 4 | 4 | 0 | 1 | 3 |

Figure: Indices for $d = 5$, $N = 4$, BlockDim.x $= 15$

# Execution time

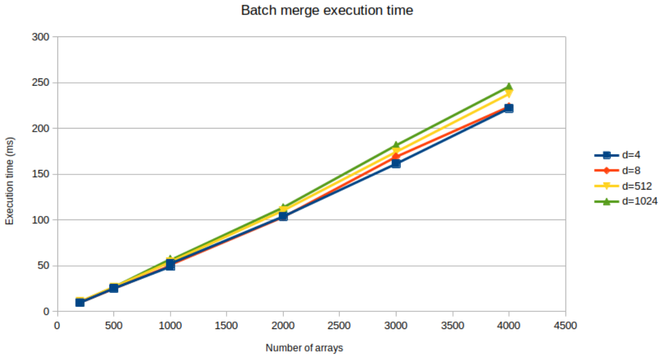


Figure: Batch merge execution times in function of N with various values of *d*

# Execution time

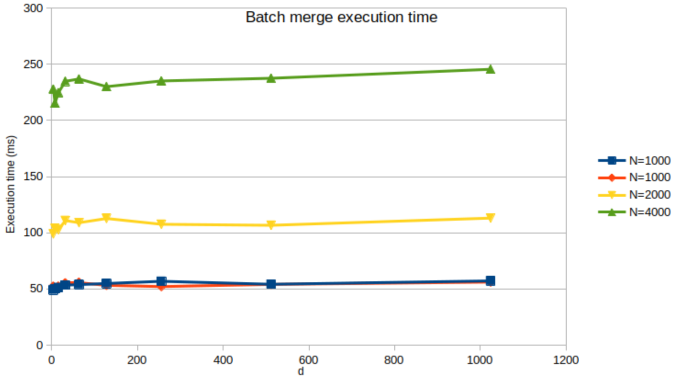


Figure: Batch merge execution times in function of *d* with various values of *N*

## Execution time

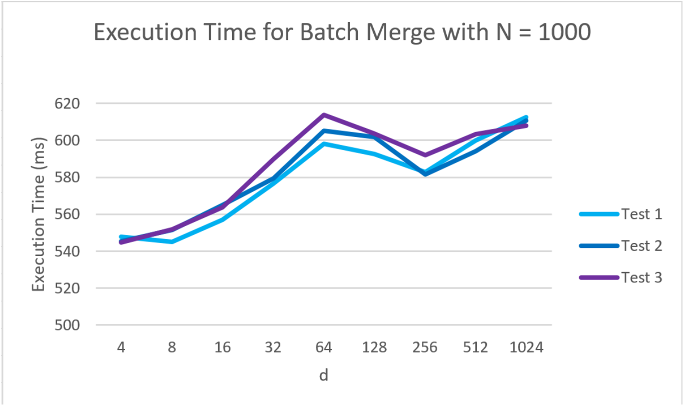


Figure: Batch merge execution times in function of $N = 10^4$ with various values of $d$

# Execution time

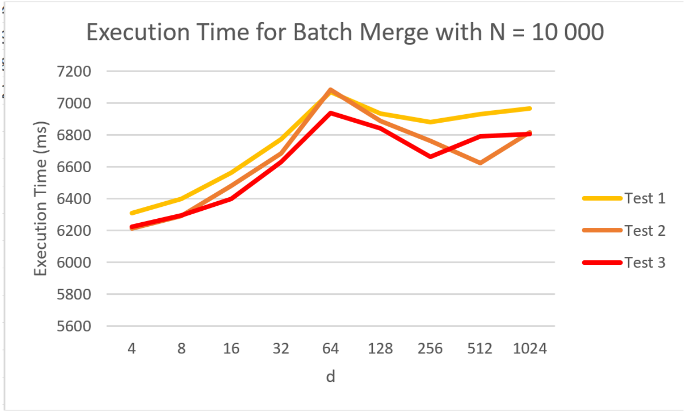


Figure: Batch merge execution times in function of $N = 10^5$ with various values of $d$

# Execution time

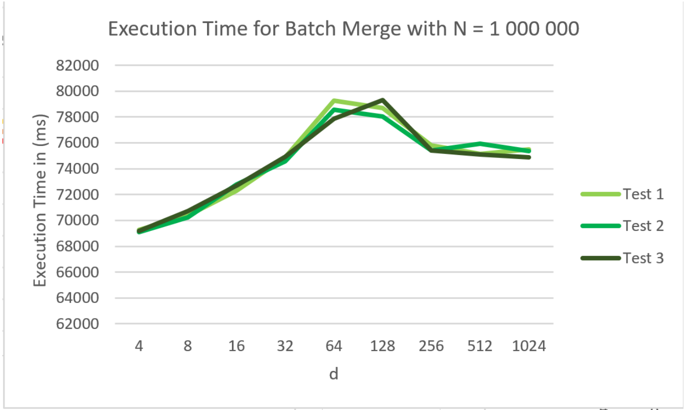


Figure: Batch merge execution times in function of $N = 10^6$ with various values of $d$

# Execution time

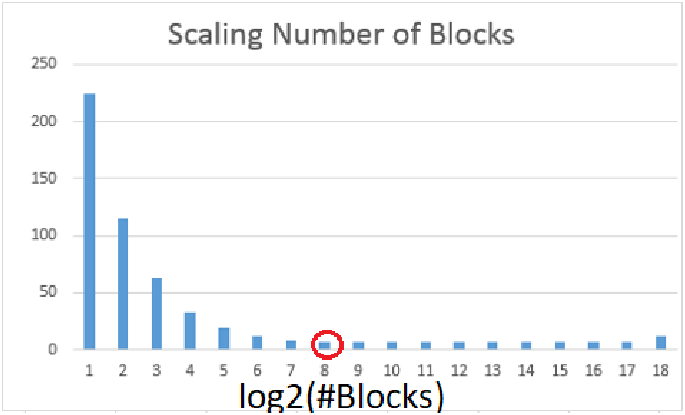


Figure: Scaling Numbers of blocks depending on the execution time in *ms*

## Summary

# Authentification protocol

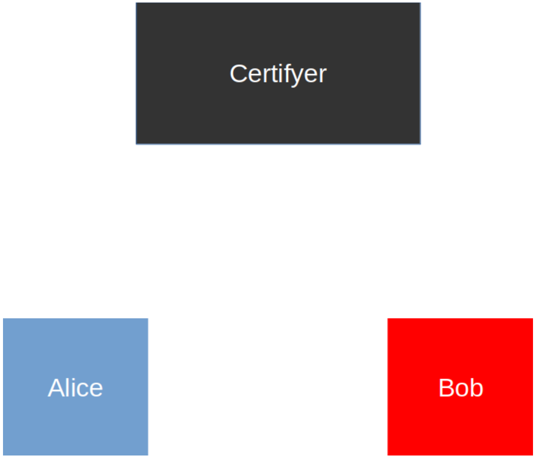


Figure: Alice and Bob wan't to get access to the certifyer

# Authentification protocol

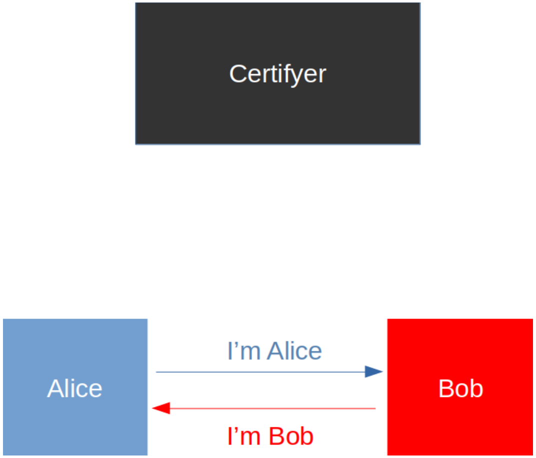


Figure: Alice and Bob recognise each other

## Authentification protocol

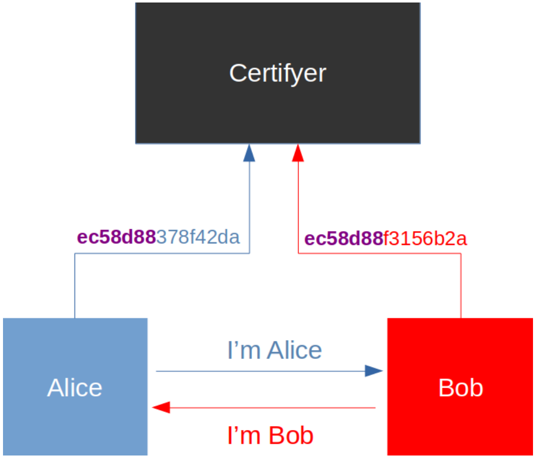
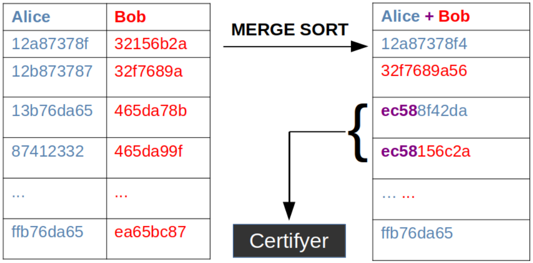


Figure: Alice and Bob send their hashed message

## Hackage method

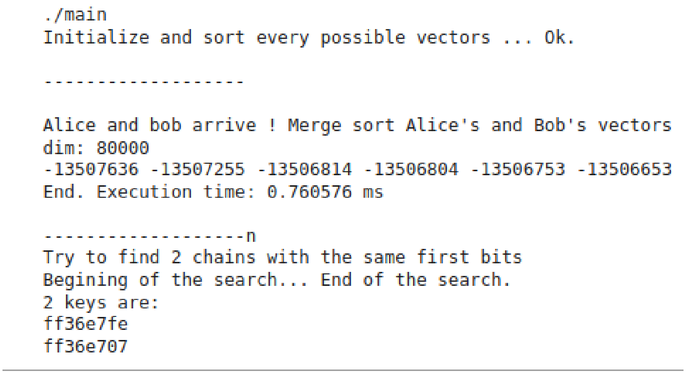


| Alice | Bob |
|---|---|
| 12a87378f | 32156b2a |
| 12b873787 | 32f7689a |
| 13b76da65 | 465da78b |
| 87412332 | 465da99f |
| ... | ... |
| ffb76da65 | ea65bc87 |

| Alice + Bob |
|---|
| 12a87378f4 |
| 32f7689a56 |
| ec588f42da |
| ec58156c2a |
| ... ... |
| ffb76da65 |

Figure: Merge sort and find keys to get access

## Result



```
./main
Initialize and sort every possible vectors ... Ok.

--------------------

Alice and bob arrive ! Merge sort Alice's and Bob's vectors
dim: 80000
-13507636 -13507255 -13506814 -13506804 -13506753 -13506653
End. Execution time: 0.760576 ms

--------------------n
Try to find 2 chains with the same first bits
Begining of the search... End of the search.
2 keys are:
ff36e7fe
ff36e707
```

Figure: Output

## Conclusion

THANK YOU
FOR
YOUR
ATTENTION!
ANY QUESTIONS?

*Reference: O. Green, R. McColl and D. A. Bader «GPU Merge Path, A GPU Merging Algorithm» 26th ACM International Conference on Supercomputing (ICS), San Servolo Islan, Venice, Italy, June 25-29, 2012*