Advanced Algorithms and Parallel Programming

github.com/martinopiaggi/polimi-notes

2022-2023

Contents

1 Divide and Conquer paradigm

Divide and conquer is just one of several powerful techniques for algorithm design, which often leads to efficient algorithms. These algorithms can generally be easily analyzed using recurrences and the Master Theorem. A general DandC algorithm is built by:

1. Divide problem in sub-problems
2. Conquer sub-problems
3. Combine sub-problems to solve the main problem

1.1 Merge Sort

Merge Sort is the typical example of D&C algorithms.

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

Note that the number of subproblems are 2, $\frac{n}{2}$ is the size of each subproblem and n is the work necessary to combine the subproblems (merge part of the algorithm). We can solve this using the master theorem (case 2 of master theorem): $f(n) = \Theta(n \log(n))$.

1.2 Binary search

1. Divide problem in sub-problem cutting in the middle element of the array
2. Conquer sub-problems
3. Combine sub-problems to solve the main problem, in this case $\Theta(1)$ because we have to combine nothing

$$T(n) = 1T(\frac{n}{2}) + \Theta(1)$$

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow \text{CASE } 2(k = 0)$$

$$\Rightarrow T(n) = \Theta(\lg n).$$

1.3 Powering a number

The naïve algorithm is $\Theta(n)$ while here is the divide and conquer approach:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

So the complexity is:

$T(n) = T(\frac{n}{2}) + \Theta(1) = \Theta(log(n))$

Note that in front of $T(\frac{n}{2})$ there isn't 2:because the numbers of sub-problems is 1. You haven't to apply the sub-problem twice (it's just a multiplication of 1 recurrence).

1.4 Matrix Multiplication and Strassen's algorithm

Matrix multiplication is one of algebra's simplest operations and it's also one of the most fundamental computational tasks and one of the core mathematical operations in today's neural networks. Naïve implementation is $\Theta(n^3)$:

```
for(i from 1 to n){
    for(j from 1 to n){
        R[i,j] = 0;
        for(k from 1 to n){
            R[i,j] = R[i,j]+ a[i,k] × b[k,j];
        }
    }
}
```

The Strassen's idea is based on using 7 multiplications and 4 adds.

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$\begin{array}{ll} P_1 = a \cdot (f - h) & r = P_5 + P_4 - P_2 + P_6 \\ P_2 = (a + b) \cdot h & s = P_1 + P_2 \\ P_3 = (c + d) \cdot e & t = P_3 + P_4 \\ P_4 = d \cdot (g - e) & u = P_5 + P_1 - P_3 - P_7 \\ P_5 = (a + d) \cdot (e + h) & \\ P_6 = (b - d) \cdot (g + h) & \\ P_7 = (a - c) \cdot (e + f) & \end{array}$$

So the complexity is:

$$T(n) = 7T(n/2) + \Theta(n^2)$$

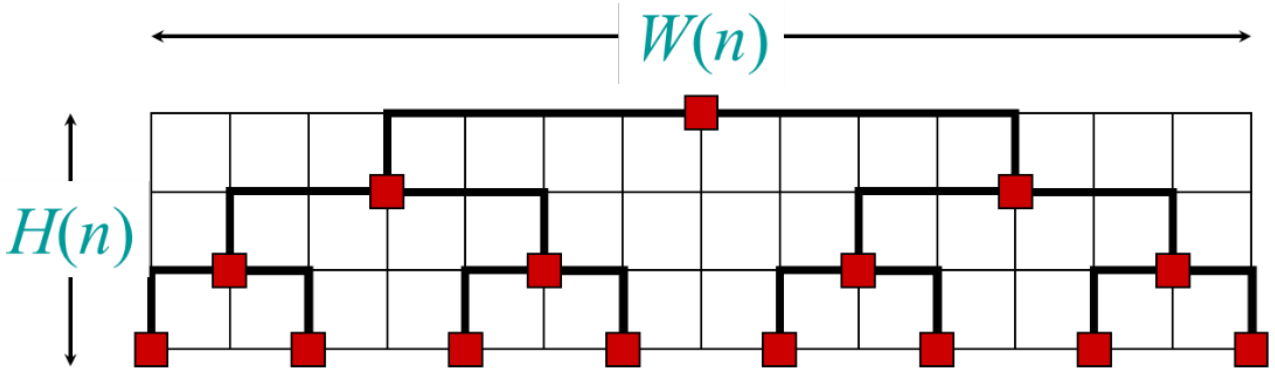
Using Master theorem, case 1:

$$n^{\log_{ba} a} = n^{\log_2 7} \approx n^{2.81} \Rightarrow \Theta\left(n^{\log_2 7}\right)$$

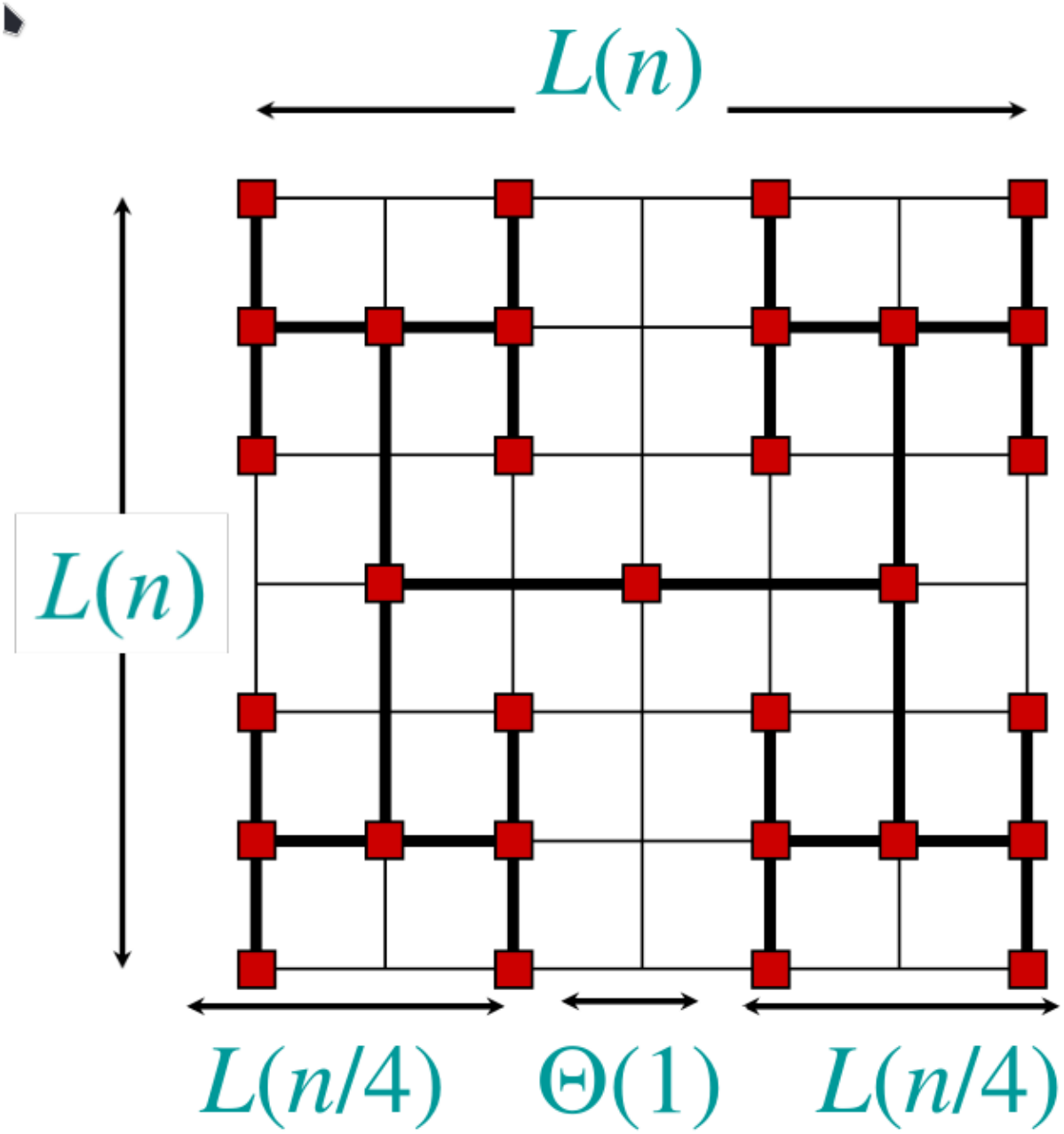
The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 32$ or so.

1.5 VLSI Layout

Embed a complete binary tree with n leaves in a grid using minimal area.



where $H(n) = H(\frac{n}{2}) + \Theta(1) = \log_2(n)$ and $W(n) = 2W(\frac{n}{2}) = \Theta(n)$ using respectively the second and first case of MT. So we obtain a total complexity (area) of $\Theta(n \log(n))$ that can be improved using this new arrangement :



This new solution based on Divide and Conquer has a total complexity of $\Theta(n)$ obtained by $L(n) \cdot L(n)$ where $L(n) = 2L(\frac{n}{4}) + \Theta(1)$ which is $\sqrt{n}$ using the MT (first case).

2 PRAM

A machine model is fundamental to reason about algorithms without considering the hardware. For algorithms applicable to parallel computers we use PRAM models. M' is a PRAM model and it's defined as system $< m, x, y, a >$ where $m_1, m_2 \ldots$ are different rams machines called processors, $x_1, x_2, \ldots$ are input memory cells, $y_1, y_2, \ldots$ are output memory cells and $a_1, a_2 \ldots$ are shared memory cells.

PRAM are classified based on their read/write abilities:

- exclusive read: all processors can simultaneously read from distinct memory locations
- exclusive write: all processors can simultaneously write to distinct memory locations
- concurrent read: all processors can simultaneously read from any memory location
- concurrent write: all processors can write to any memory location. Different criteria can apply in this case:
  - priority CW: processors have different priorities
  - common CW: processors complete writes iff the written values are equals
  - random CW

Some definitions:

- $T^*(n)$ is time to solve problem using best sequential algorithm
- $T_p(n)$ is time to solve problem using p processors
- $S_p(n) = \frac{T^*(n)}{T_p(n)}$ is the speedup on p processors
- $E_p(n) = \frac{T_1(n)}{pT_p(n)}$ is the efficiency
- $T_\infty(n)$ is the shortest run time on any p
- C(n) = P(n) · T(n) is the cost that depends on processors and time
- W(n) is the work, which is the total number of operations

There could be also some variants of PRAM, like for example:

- bounded number of shared memory cells
- bounded number of processor
- bounded size of a machine word
- handling conflicts over shared memory cells

Any problem that can be solved by a p-processors PRAM in time T_p can be solved by a p' (with $p' \leq p$) processors PRAM in time:

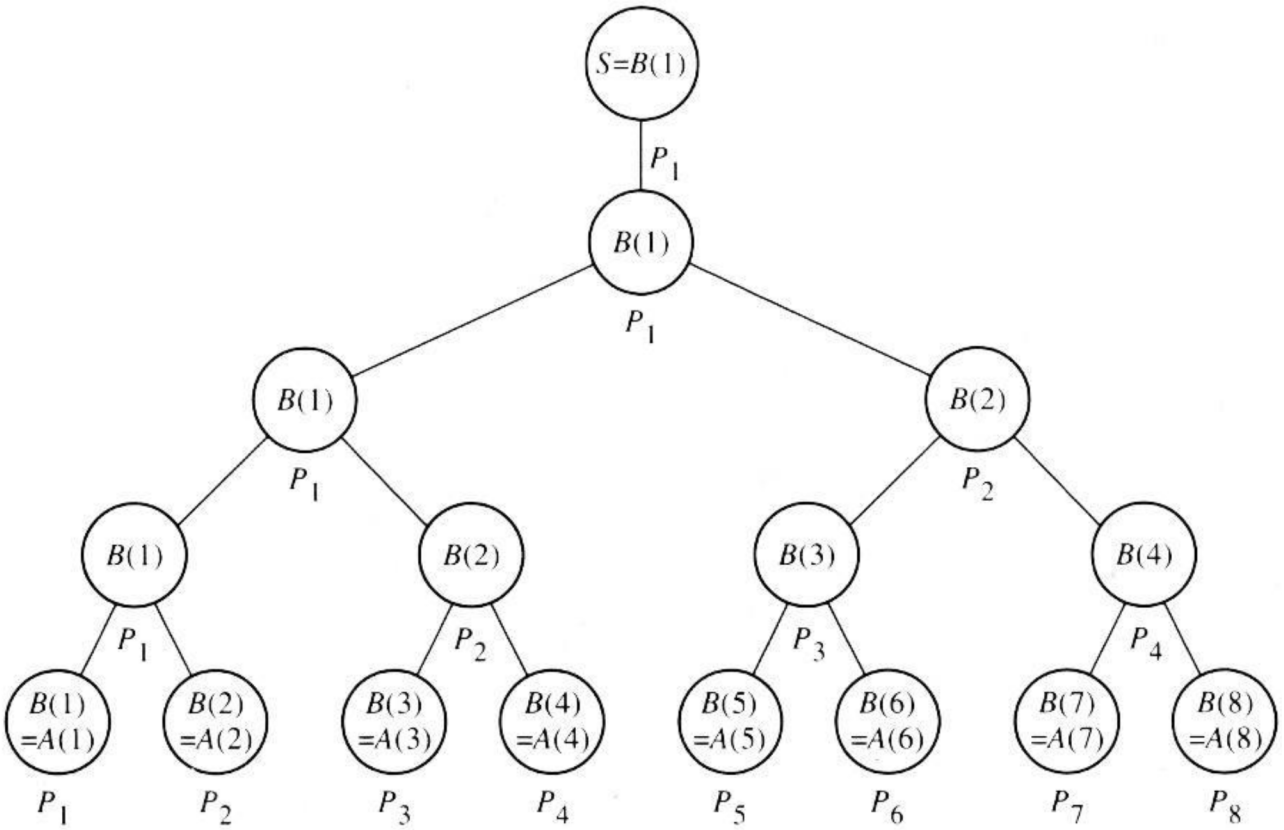
$$T_{p'} = \frac{pT_p}{p'}$$

And in a similar way (assuming $m' \leq m$) any problem that can be solved by a p-processor and m-cells PRAM in t steps can also be solved by a processors and m'-cells PRAM in

$$T_{p,m'} = \frac{mT_{p,m}}{m'}$$

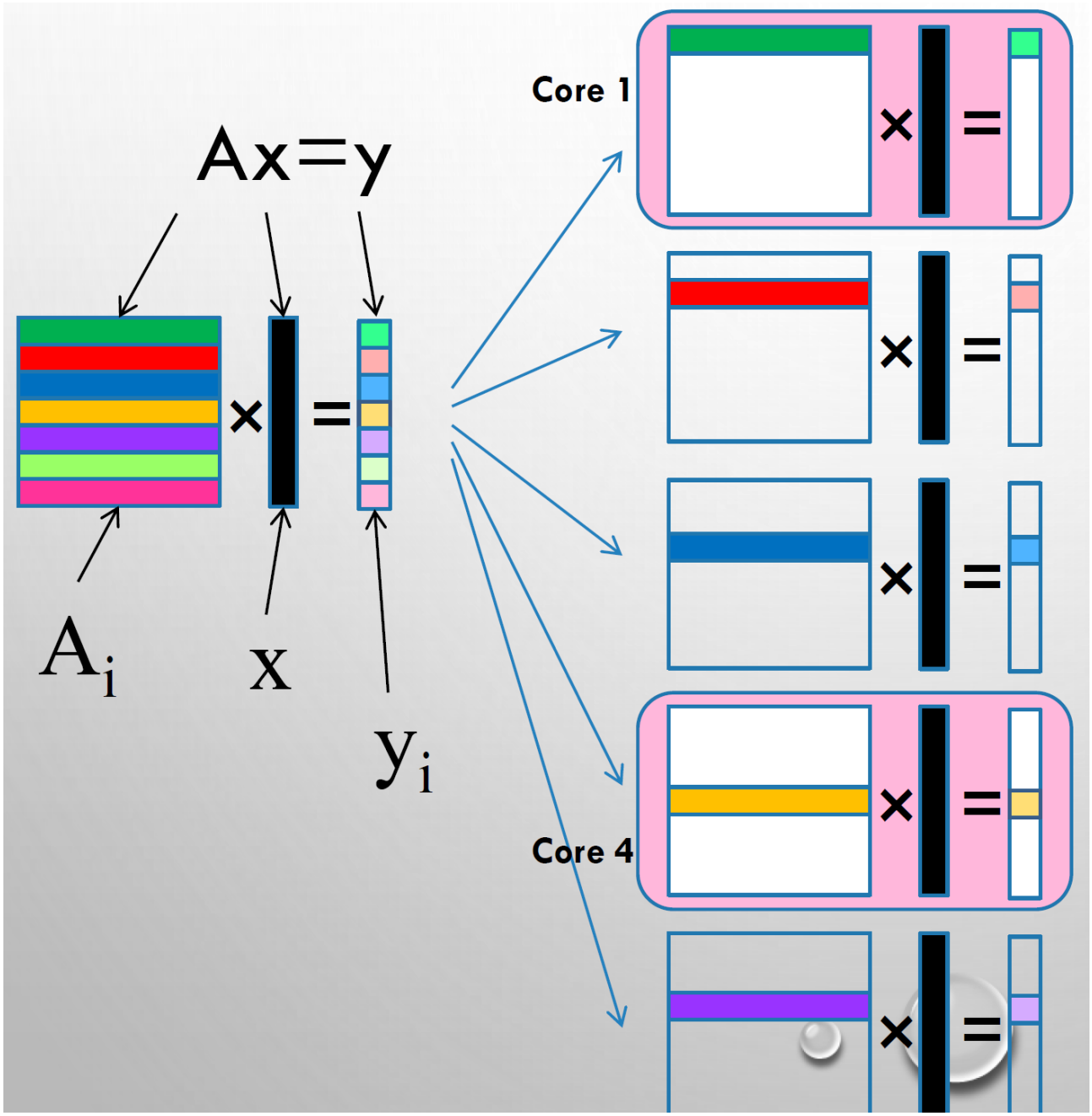
2.1 Sum of vector elements

Parallelization of a sum of vector elements with the naïve algorithm can be performed with $T*(n) = \Theta(n)$ while with a popular parallel pattern can be performed with $T_p = 2 + \log(n)$.



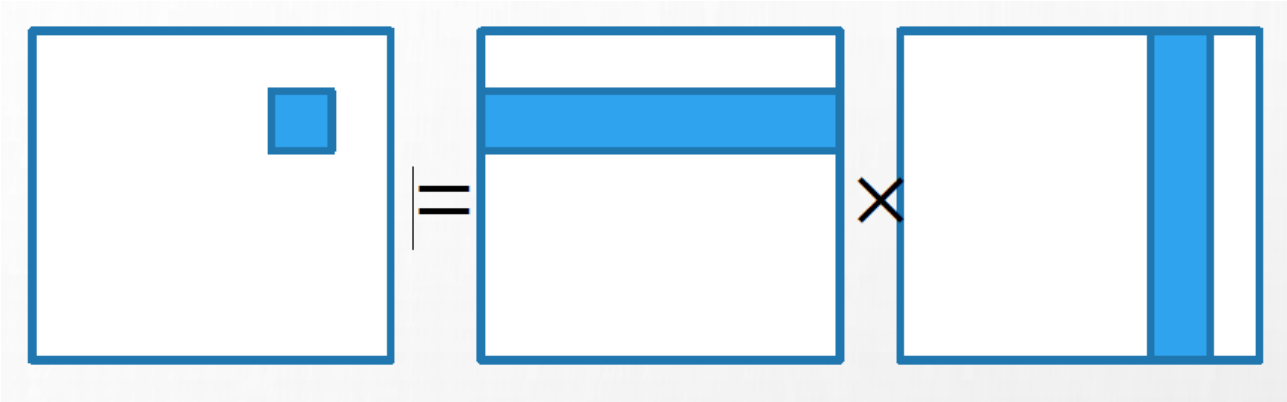
2.2 Matrix-vector multiplication

Embarrassingly parallel because there isn't cross-dependence (just the concurrent read over the vector x).



The matrix A can be subdivided in submatrices of size $\frac{n}{p}n$. This is so easy that from $T_1(n) = \Theta(n^2)$ we obtain $T_p(n) = \Theta(\frac{n^2}{p})$ (the ideal case is $p = n \to \Theta(n)$) so we have linear speedup and perfect efficiency $E_p = \frac{T_1}{pT_p} = 1$.

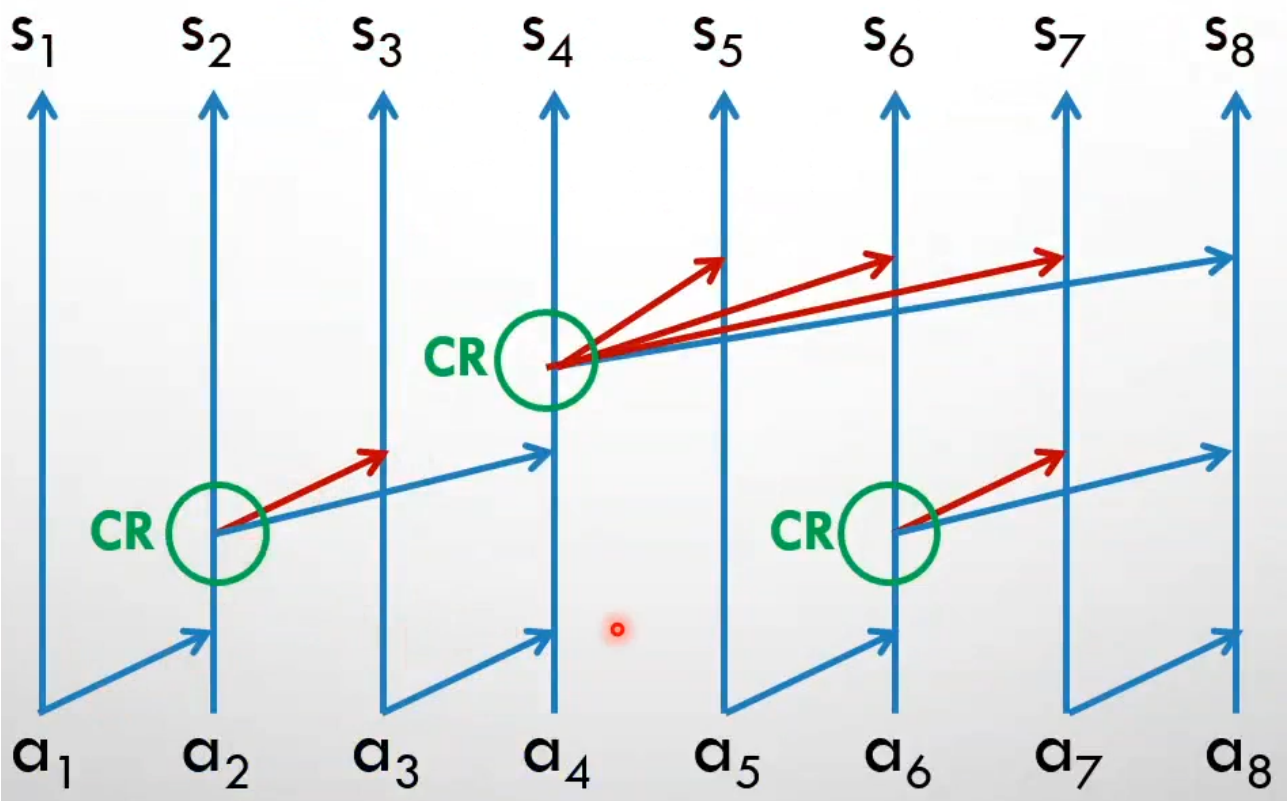
2.3 Matrix-matrix multiplications



This algorithm is characterized by concurrent read but only exclusive write so runs on CREW PRAM. Let $p = n^3$ processors we can compute the multiplication in $\log(n)$ assigning to each processor a row and a column and the corresponding cell in the result matrix. Note that $\log(n)$ is because you have to perform the sum of all elements of the vector (the vector obtained multiplying the row with the column).

2.4 Prefix Sum example

The previous PRAM algorithms make the same amount of work of the work done by a single processor, simply faster using parallelization. The prefix sum problem is basically the same of a sum of the vector elements but exploiting idle processors.



The idea is to make more work in same time taking advantage of idle processors in sum. Basically we used all

the processors all the time. Efficiency is 1 and the complexity is still $log(n)$. The goal of this algorithm (and the difference of a normal sum vector elements) is that we not only obtain the final sum but also the sum of all the prefixes. For example from a, b, c, d, e we obtain a , $a + b$, $a + b + c$, $a + b + c + d$ and $a + b + c + d + e$.

2.5 Boolean DNF (sum of product)

Example of CRCW where each processor is assigned to a product.

$$X = a_1b_1 + a_2b_2 + a_3b_3$$

Not all processors write the result X of the product but those that do, write $X = 1$. Very simple parallel solution where the time is $O(1)$. It works on CRCW PRAM with common, random and priority write.

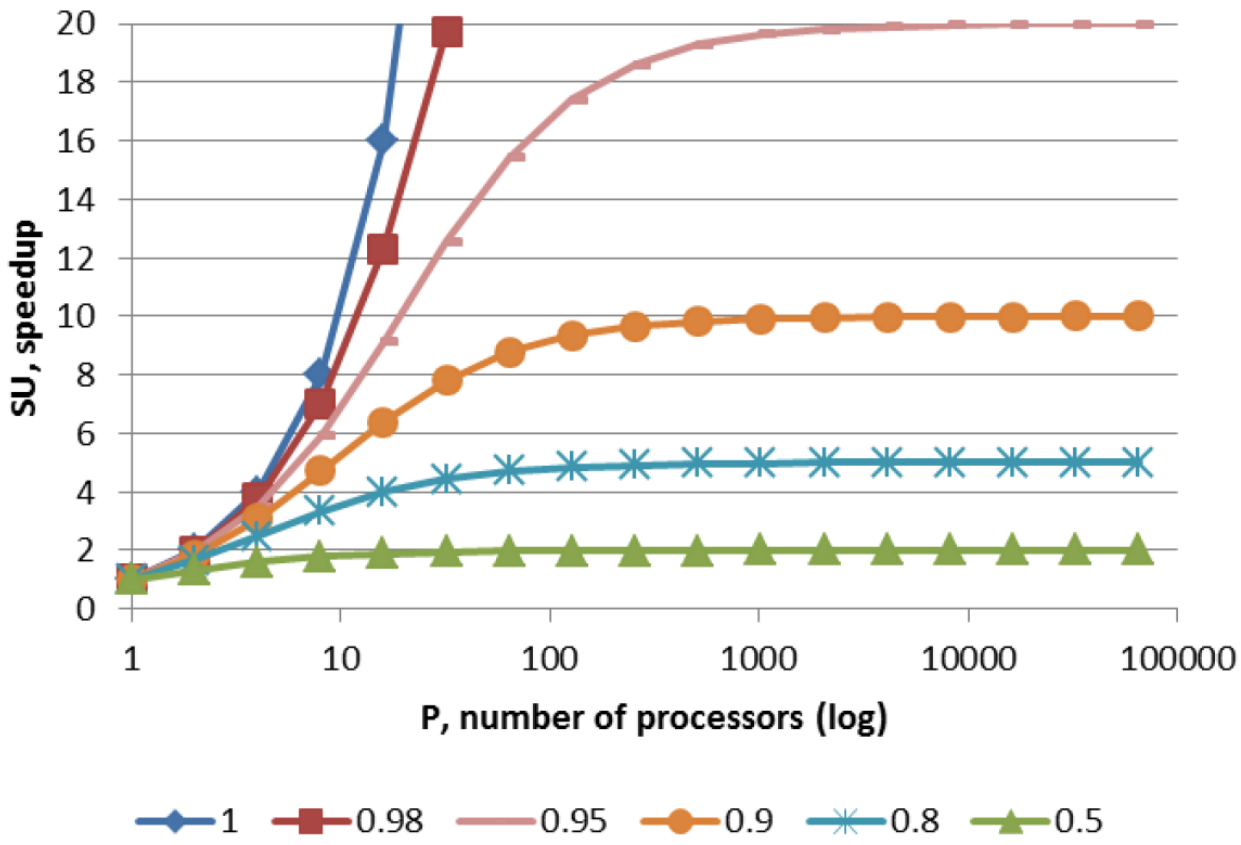
2.6 Amdahl's law

Gene Amdahl objected to parallelism saying that computation consists of interleaved segments of 2 types:

- a serial segments that cannot be parallelized
- parallelizable segments

The law is 'pessimist' since if the parallelized part is a fixed fraction f of the total computation, this will mean that given a problem with inherent $f = 0.9$, there will not sense to use more than 10 processors... showing the limits of parallelization.

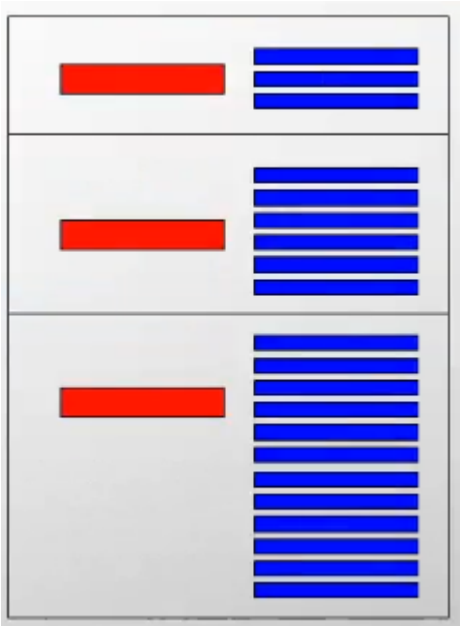
$$SU_p = \frac{T_1}{T_p} = \frac{T_1}{T_1(1-f) + \frac{f}{p}} = \lim_{p \to \infty} = \frac{1}{1-f}$$



2.7 Gustafson's law

> "We feel that it is important for the computing research community to overcome the"mental block" against massive parallelism imposed by a misuse of amdahl's speedup formula."

The key points of Gustafson are that portion f is not fixed and only absolute serial time is fixed. The end result of Gustafson's law is that we could always increase up the parallelized part of the computation **redoing** the sequential work more than one time. So basically we map the work to more processors and then we **repeat the initialization/sequential part** more than one time to permits an infinite theoretical speedup.



2.8 ISPC for implementing PRAM algorithms.

ISPC is a compiler for a variant of the C language that focuses on accelerating applications according to the SPMD paradigm. It parallelizes at the instruction level by distributing instructions over vectorized architectures (SSE and **AVX** units) for x86, ARM and GPUs. The documentation for ISPC can be found here: https://ispc.github.io/ispc.html . When a C/C++ function calls an ISPC function, the execution model instantly switches from a serial model to a parallel model, where a set of program instances called **gang** run in parallel. The parallelization is transparent to the OS and is managed entirely inside the program. Unless otherwise specified, variables are local to each program instance inside a gang. Doing so is memory-inefficient, and whenever possible variables should have the attribute `uniform` to signal that they are shared among all instances of the gang. This also opens the door to issued arising from concurrent accesses to the same `uniform` variable. Each program instance in a gang has knowledge about the gang's size and its own index within the gang. The gang's size is stored in the `programCount` variable, while the instance's index in the gang is stored in the `programIndex` variable. They can be used to distribute the computation over the gang members by manually assigning the data they should work on.

3 Randomization

3.1 Las Vegas vs Montecarlo algorithms.

An algorithm is randomized if it is in some way based on some random variable. There are mainly 2 different types of randomized algorithms:

- **Las Vegas** algorithms are **exact** : always provide correct solution.
- **Monte Carlo** algorithms are **approximate** : try to find an approximation of the solution in a given limited time (bound) and maybe it finds a wrong one.

Basically a Las Vegas algorithm is a MC algorithm with error probability of 0. Because of these characteristics we evaluate Las Vegas algorithms looking the **expected running time** , while we evaluate Monte Carlo algorithms looking the **maximum running time**.

3.1.1 Why the fuck we use MC algorithms if maybe they are wrong?

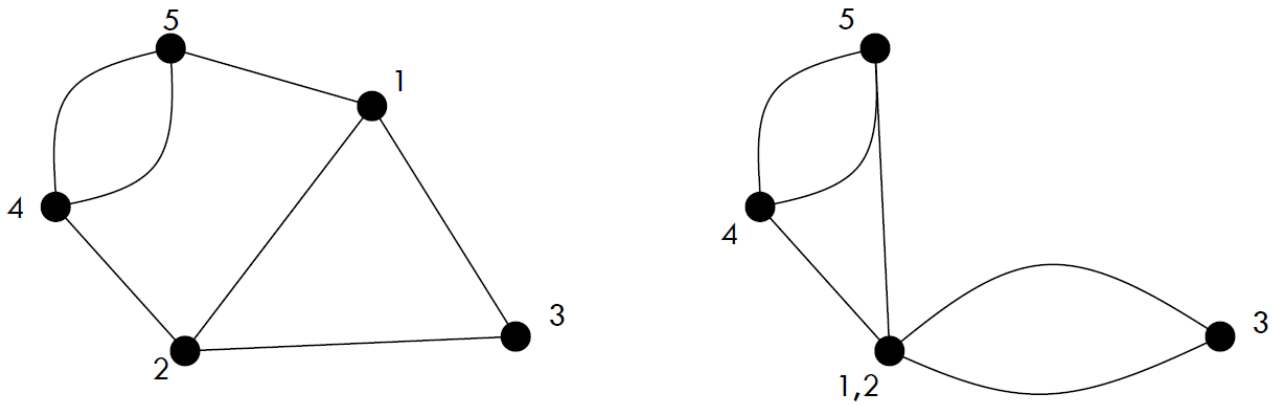
Monte Carlo algorithms are generally used for problems that are very very difficult, like for example the Karger's Min Cut algorithm (which is NP!). The MC solution for min-cut problem runs in polynomial time with the limit that we accept an error probability, that we can always tuning with the number of runs. In case MC algorithm is for a decision problem we can it in two classes:

- two-sided error: MC algorithm which has non-zero error probability for both two possible outputs.
- one-sided error: MC algorithm that has non-zero error probability only for one of the two possible outputs.

3.2 Karger's approach

A **minimum cut** of a graph is a partition of the vertices of a graph into two disjoint subsets with the minimal amount of deleted edges. Karger's algorithm is an algorithm to find a min-cut on a multi-graph using a randomized approach. Multigraphs are undirected graphs that do not allow for self-loops but allow for multiple edges between the same node pairs.

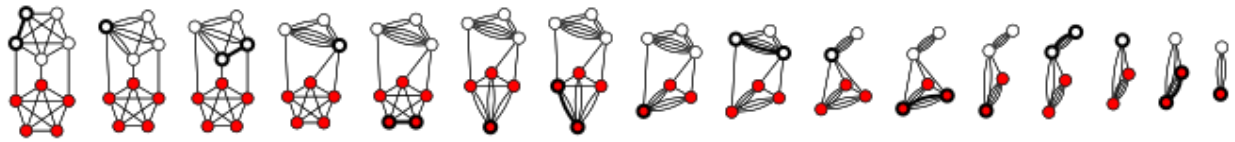
Contracting an edge means to collapse a vertex into another neighbor vertex (removing self loops):



The Karger's algorithm consists in contracting edges uniformly at random until only two vertices remain. These two vertices correspond to a partition of the original graph and the edges remaining in the two vertex graph correspond to a cut in the original input graph. The number of edges in the resultant graph is the cut produced by Karger's algorithm.

The key idea of the algorithm is that it is far more likely for non min-cut edges than min-cut edges to be randomly selected and lost to contraction, since min-cut edges are usually vastly outnumbered by non min-cut edges. Subsequently, it is plausible that the min-cut edges will survive all the edge contraction, and the algorithm will correctly identify the min-cut edge.

Karger's algorithm is a Monte Carlo algorithm and cut produced by it may not be minimum. For example, the following diagram shows that a different order of picking random edges produces a min-cut of size 3. The probability can be increased by repeated runs of basic algorithm and return minimum of all cuts found.



Each run of the algorithm is $O(n^2)$ and since the possibility of pick one of the edges of the min-cut during an iteration is $\binom{n}{2}$ we have to minimize $l\binom{n}{2}$, keeping in mind that we would like to have minimize probability of error $\frac{1}{\text{poly}(n)}$ increasing the number of runs. The best optimization is with $l = \log(n)$. So the final complexity is $O(n^4 \log(n))$.

3.2.1 Faster version of Karger and Stein

Karger's algorithm can be refined by observing that the probability of picking an edge belonging to the cut of minimum size is low for the first contractions and grows progressively as we go towards the end of the execution. This motivates the idea of switch to a faster algorithm during the first contraction steps and do exactly like Karger's algorithm in the last phase. From a multi-graph G, if G has at least 6 vertices, repeat twice: run the original algorithm down to $\frac{n}{\sqrt{2}} + 1$ vertices **recursively** on the resulting graph and each time return the minimum of the cuts found in the two recursive calls. When there multigraph has less than 6 vertices, it simply runs the normal Karger's algorithm. Since the recursivity, the faster version has this complexity:

$$T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + \Theta(n^2)$$

so (using master theorem, case 2) we have $T(n) = n^2 \log(n)$. Since we want to minimize the error of probability of $\frac{1}{\text{poly}(n)}$, we find that with $\log^2(n)$ runs we can do so the final complexity is $O(n^2 \log^3(n))$.

Check if an integer is prime is a good application of Divide and Conquer pattern. The naïve algorithm is $\Theta(\sqrt{n})$ but we can use a randomized method using a Monte Carlo algorithm that is false-biased (one side error):

- If answer is "not prime", then n is surely composite (not prime)
- If answer is " prime", then **probably** n is prime

The randomized algorithm is based on Fermat's little theorem: p prime iff **for each** $a < p - 1$ $a^{p-1} \mod p = 1$. Remember that there are numbers that are pseudo-primes, this means that some a fool the Primality Test. If during the computation we discover that the number is not prime we return false (and we are sure about that 100%) otherwise we return true but we are not completely sure about that (we should make more tests as possible using the function to reach some confidence about the result). During the computation of the Primalntiation technique: $a^n = a^{n/2} * a^{n/2}$ (if n is even, otherwise we obviouslyity Test, for computing the power we use fast expone multiply for a another time).
During the fast exponentiation we can do the nontrivial square roots test: iff p is prime and $0 < a < p$ then the only solutions to the equation $a^2 \mod p = 1$ are $a = 1$ and $a = p - 1$. During the fast exponentiation computation we already check if it's prime making every time $a^2 \mod p$.

Youtube video with good explanation about this

4 RSA algorithm

Possible application of the Primality Test is used in public-key cryptosystems for example, which is commonly used for secure data transmission. The idea of asymettric encryption is:

1. P (public key) and S (secret key) can be computed efficiently
2. With a P , S :
3. S is not computable from P (without a quantum computer, during the life of this universe)

RSA algorithm steps (simplified, toy example):

1. randomly select two primes p and q of similar size, each with $l + 1$ bits ($l \geq 500$)
2. let $n = p \cdot q$
3. let e an integer co-prime with $(p - 1)(q - 1)$
4. $d = (\frac{1}{e}) mod((p - 1)(q - 1))$

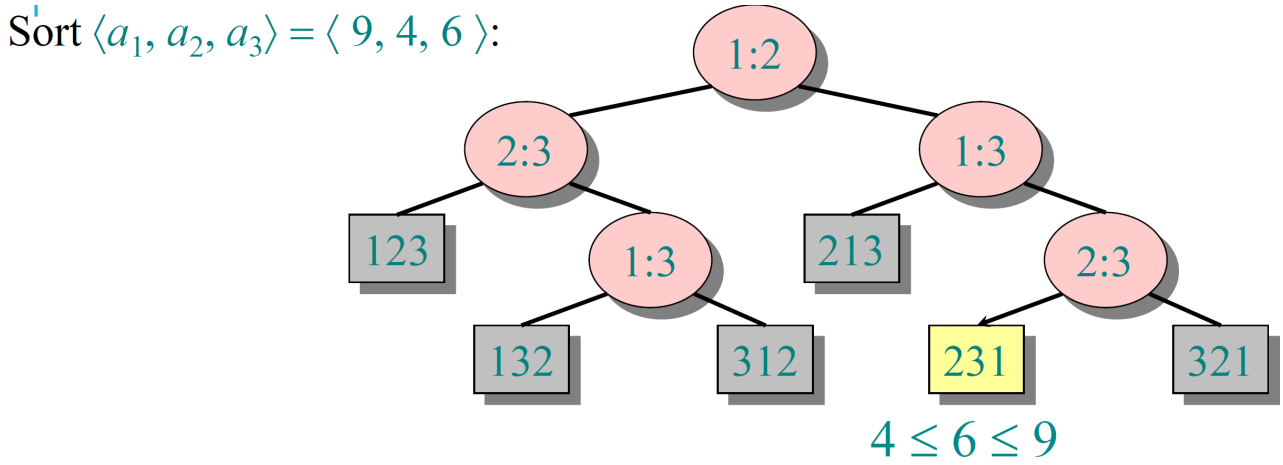
5. $P = (e, n)$ and $S = (d, n)$

6. let $M = \text{message}, C = M^e mod(n) \rightarrow M = C^d mod(n)$

Sorting(../../../BSc([/Algoritmi%20e%20Principi%20dell'Informatica/src/10.Sorting.md)

5 Sorting

All the sorting algorithms we have seen so far are comparison sorts : they use comparisons to determine the relative order of elements. We can prove using a decision-tree view of the problem that the the tree represents the sorting algorithm and it splits whenever it compares two elements.



Some properties about this kind of tree:

- The tree contains the comparisons along all possible instruction traces
- The running time of the algorithm is the length of the path taken
- Worst-case running time is the height of tree
- Since the leaves are $n!$ because there are $n!$ possible permutations of the initial array and a binary tree has $\leq 2^h$ we can solve $n! \leq 2^h$ to find $h \geq n \log(n)$.

So Heapsort and merge sort are asymptotically optimal comparison sorting algorithms and it's not possible to build better sorting algorithms **based on comparisons**. Since Counting Sort is not based on comparison, it can perform better results than others sorting algorithms. It is also stable: it preserves the input order.

5.1 Quicksort randomized

The worst case occurs every time the 2 partition are not balanced. For example if the input is already sorted or reverse sorted, using the Lomuto's partition, the pivot will always on min or max element. Because of this one of the two partition will be empty. In case of this unlucky event repeated all the times, we have $O(n^2)$. We can 'keep away' this unlucky case randomizing the original vector or partition around a random pivot.

5.2 Radix Sort

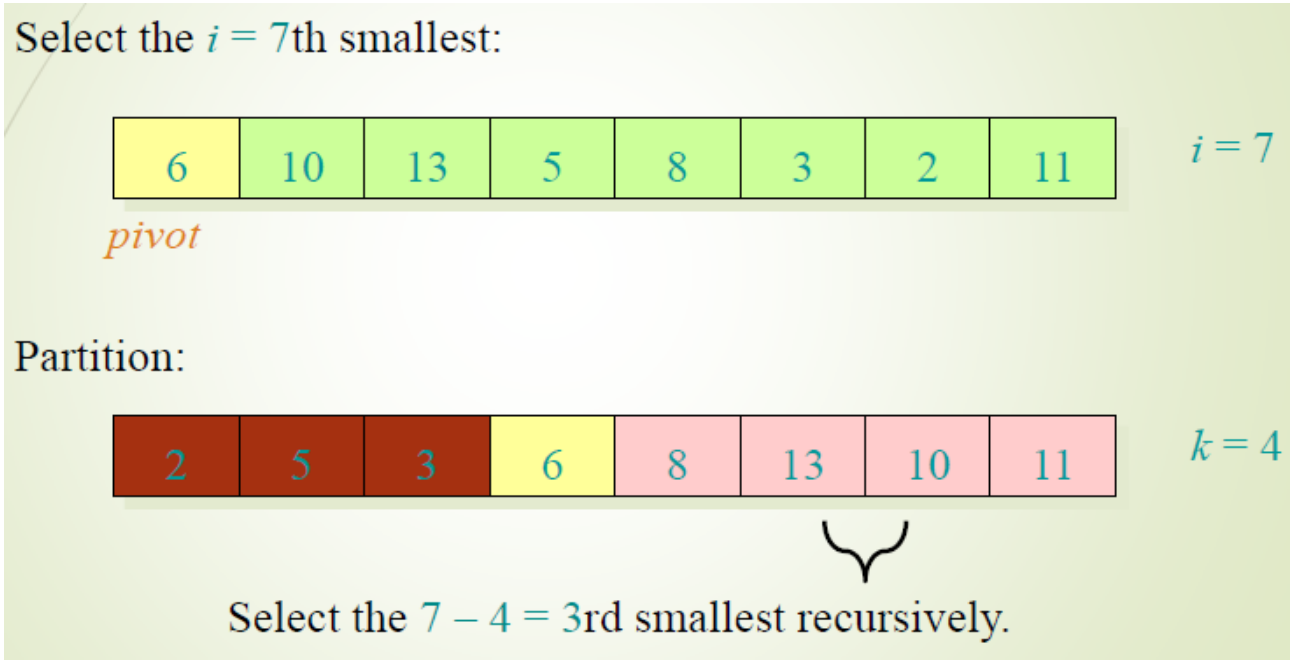
Radix sort is fast for large inputs but quicksort is still a viable option due to its locality that allows to better exploit the steep cache hierarchies of modern processors and result in better overall performances for most uses.

Radix sort is a non-comparative sorting algorithm: it avoids comparison by sorting on least significant digits first. It is also called **bucket sort** and **digital sort**. The cool thing about this algorithm is that we can use Counting Sort as the auxiliary stable sort because the limit domain of each sorting (0-9). Correctness of Radix

Sort proved by induction on digit position. Also we can split each word of b in group of digits, so that we can divide a word in r groups. The best way to divide each word is in $r = \log n$ where n is the number of words. In this way, the total complexity is $T(n) = \Theta(dn)$ where n is the number of words and d the length of each word. In practice, radix sort is fast for **large inputs**, as well as simple to code.

5.3 Randomized selection algorithm

In statistics, the k_{th} **order statistic** of a statistical sample is equal to its k_{th}-smallest value. Quickselect is the best-known selection algorithm and it's basically Quicksort. We don't do recursive calls to both the partitions of quick-sort but just to the one where we will know there is the $i-th$ element. How do we know where there is the $i-th$ element? Simply remember that in quicksort the position of the pivot after the iteration is the final one, even if the algorithm has not finished!



So we can proceed searching the $i-th$ element recursively without ordering all the array.
Average case: $\Theta(n)$ but worst case is $\Theta(n^2)$ that is worst than sorting! The demonstration of the complexity is similar to quicksort but with the difference that the recursion it will be applied only 1: Lucky case:

$$T(n) = T\left(\frac{1}{10}n\right) + \Theta(n) = \Theta(n)$$

Not so lucky case:

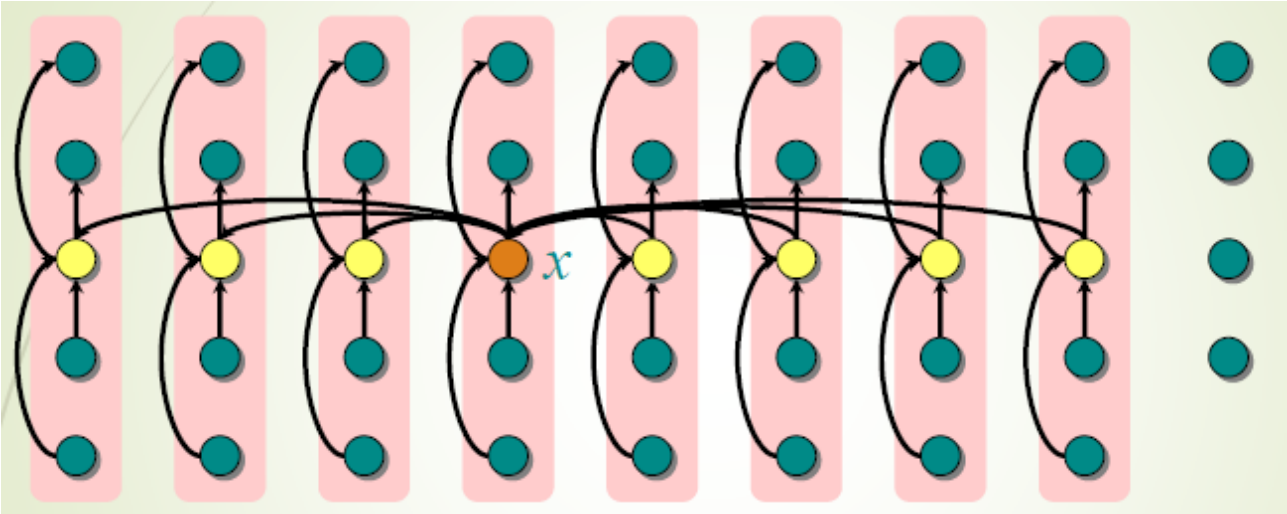
$$T(n) = T\left(\frac{9}{10}n\right) + \Theta(n) = \{\text{Master theorem case 3: } \log_{\frac{10}{9}} 1 = 0 = \Theta(1)\} \text{ so } T(n) = \Theta(n)$$

Unlucky case:

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

5.3.1 Deterministic version

A possible way to guaranteed that the smallest partition is not empty we could group all the elements in groups of 5 each and find the median of each group. Then select as pivot the median of the median group.



Developing the recurrence :

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. **Recursively** select the median x of the medians to be the pivot.
3. Partition around the pivot x. Let $k = rank(x)$
4. if $i = k$ then return x else if repeat step *2* recursively search the i_{th} element in the lower or the $(i-k)_{th}$ element in the upper part

Looking visually the structure created using groups of 5 we can find some interesting properties over the array and we can say that the recurrence can assume that step 4 takes time $T(\frac{3}{4}n)$ in the worst case. So the complexity is:

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{3}{4}n\right) + \Theta(n)$$

Since the work at each level of recursion is a constant fraction ($\frac{19}{20}$) smaller, the work per level is a geometric series dominated by the linear work at the root (this is always true, it's not the average case ... it's deterministic). So for this fact, it's theoretically better than quicksort but in practice, this algorithm runs slowly, because the constant in front of n is large. So the randomized algorithm is far more practical. # Randomized data structures

5.4 Disjoint sets

Disjoint sets is a data structure very useful for operations on graphs. A collection of sets of objects, not intersected and each identified by a representative element. A representative is some member of the set (often it often does not matter which element is the representative irrelevant). A couple of operations over the disjoint sets:

```
make(x);
union(x,y);
find-set(x);
```

The most efficient way is using 'forests' (rooted trees, where each tree is a set) with 2 optimizations:

- Union by Rank: Each node is associated with a rank, which is the upper bound on the height of the height of subtree rooted at the node. Then when `union(x,y)`, let the root with smaller rank point to the root with larger rank.
- Path Compression: used in `find_set(x)` operation, make each node in the path from x to the root directly point to the root.

```
public class DisjointSet
{
    private int[] _set;
    private int[] _rank;

    public DisjointSet(int size)
    {
        _set = new int[size];
        _rank = new int[size];
    }

    public void MakeSet(int x)
    {
        _set[x] = x;
        _rank[x] = 0;
    }

    public int FindSet(int x)
    {
        if (x != _set[x]) return FindSet(_set[x]);
        return x;
    }

    public void UnionSet(int x, int y)
    {
        var parentX = FindSet(x);
        var parentY = FindSet(y);
        if (_rank[x] > _rank[y]) _set[parentY] = parentX;
        else
        {
            _set[parentX] = parentY;
            if (_rank[x] == _rank[y]) _rank[y]++;
        }
    }
}
```

https://github.com/martinopiaggi/Unity-Maze-generation-using-disjoint-sets

5.5 Treaps

Treaps are binary trees where each child has a greater priority to his parent, like max heap. To do this, each element x is assigned a priority chosen uniformly at random before the insertion. With this magic trick, a treap achieve essentially the same time bounds of balanced trees with an expected number of rotations performed of 2 rotations for each operation. Also they are very simple to implement compared to AVL or RB trees for example. If n elements are inserted in random order into a binary search tree, the expected depth is always $1.39log(n)$.

Explained in spaghettata mode: I have a binary tree (so very good to search) but I have always the problem to balance it. So I assign to each key a random priority and then I consider my tree not only as a tree but also as a heap and I want to preserve the property of the heap using rotations. Note that the heap can be min heap or max heap without problems.

Ops:

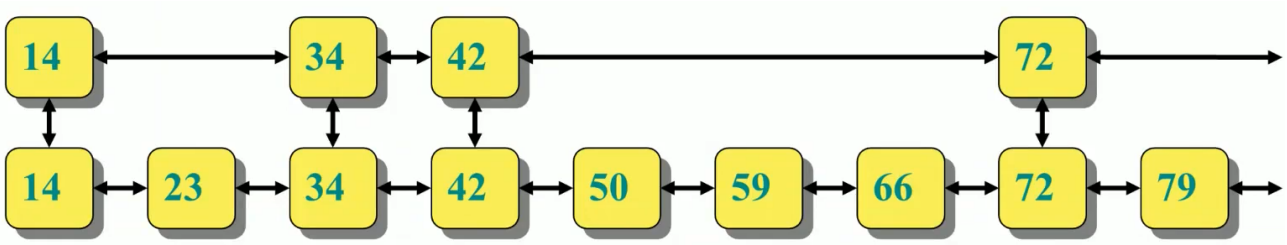
- insert operation follows the logic of the insert in a BST.

- delete operation is dual. The element is searched and once the node to be deleted is found we rotate the node left/right until the node to be deleted is a leaf. Once the element to be deleted is a leaf we simply remove it.
- min and max can be found following the left/right subtree of all encountered nodes from root to leaf.
- Merge of two treaps can be performed finding a key greater than the max of T_1 and smaller than the min T_2 . Then we will add this key with lower probability (in max heap version) as root with T_1 as left subtree and T_2 as right subtree. The algorithm then deletes the root from the treap. The expected running time is $O(\log(n))$.
- The split of two treaps can be performed adding a key that doesn't exists in the tree and giving it maximum priority (in case of max heap) so that it will end up as the root.

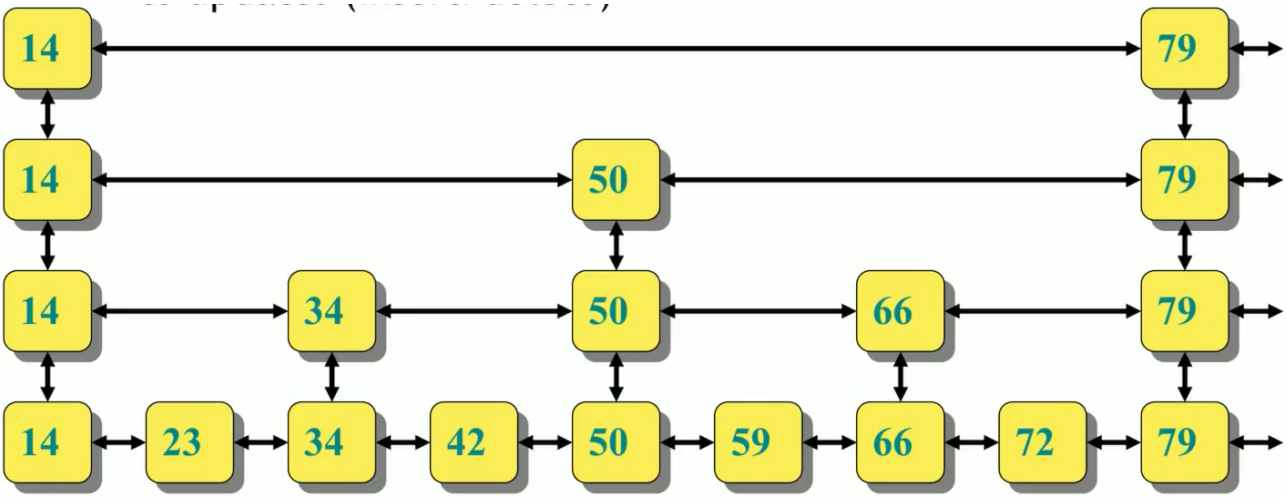
All these operations have $O(\log(n))$ time complexity.

5.6 Skip Lists

Skip lists similar benefits to treaps but are based on an alternative randomized data structure. The starting point is a **sorted** linked list, a dynamic structure with $O(n)$ for search elements. The next step is adding a second sorted linked list shorter to 'skip' some elements of the list.



Let's expand the idea further:



We randomly decide to promote an element to upper level. It's proved that the there will be $\log(n)$ levels/lists and at the end there will be a 'balanced' structure.

Ops:

- search is going to the 'right element' in the highest-level list until it is found an element greater than the one searched. If it's not already found the searched one and the element selected is lower than the one searched, it will be necessary to go to the lower list using the 'down' pointer. This process is repeated until the element is found or the last list is reached and there are not 'down pointers'.

- insert (after the 'search operation') is like in a normal linked list but at the end you will randomly choose if 'promote' it .
- deletion (after the 'search operation') is like in a normal linked list but at the end you will check to delete the element from all the lists.

All these operations have $O(\log(n))$ time complexity. # Dynamic programming

Dynamic programming is an algorithm design technique for problems where the solution of the problem contains the solution to subproblems which computation is overlapping. A couple of definitions:

- Optimal substructure means that the optimal solution to a problem (instance) contains optimal solutions to subproblems.
- Overlapping subproblems means that a recursive solution contains a "small" number of distinct subproblems repeated many times.

Generally redoing previous done computations solutions for the overlapping subproblems or the substructure feature of a problem should hint us at using a dynamic programming solution.

5.7 Memoization

To avoid re-computations we save the results of the subproblems in memory: when we face a certain subproblem, we simply look up the solution. We can proceed bottom-up or top-down:

- The bottom-up approach computes all the computations before the current computation so when we face a subproblem we know we have already solved all the subproblems that are 'previous' to that one. We avoid recursion overheads.
- The top-down approach computes at the moment the subproblem and when we face a sub-problem, we first look if the solution has already been computed (but it's not guaranteed as in bottom-up approach), if not we solve it and save it in memory; this way we solve subproblems only when necessary.

Climbing Stairs - Dynamic Programming - Leetcode 70 - Python - YouTube

6 Longest Common Subsequence

The **longest common subsequence** (**LCS**) is the most popular dynamic programming problem. It's the problem of finding the longest subsequence common to all sequences in a set of sequences. **It differs from the longest common substring problem**. LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.
LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4. The LCS is the basis the usual `diff` tool. The naïve solution is exponential : the complexity is 2^n with n length of the string. The running time of the dynamic programming approach is $O(nm)$ with n and m the lengths of the two sequences. The LCS has an optimal substructure: the LCS(z) contains also the LCS of the prefix of Z. Since the algorithm builds a matrix to store the partial results we can called this algorithm a '2D dynamic program'. We can define the LCS algorithm:

- LCS of 2 empty sequences is the empty sequence.
- LCS of "{prefix1}A" and "{prefix2}A" is LCS({prefix1}, {prefix2}) + A
- LCS of "{prefix1}A" and "{prefix2}B" is the **longest** of LCS({prefix1}A, {prefix2}) and LCS({prefix1}, {prefix2}B)

So it's the solution iteratively starting from the simple base cases. Note that the algorithm can work like this because the problem has so-called "optimal" structure, meaning that it can be built by reusing previous memoized steps. The algorithm is 'divided' in two parts: the first one is focused on find the length of the longest

common subsequence.

	∅	**A**	**G**	**C**	**A**	**T**
∅	0	0	0	0	0	0
G	0	←↑ 0	↖ 1	← 1	← 1	← 1
A	0	↖ 1	←↑ 1	←↑ 1	↖ 2	← 2
C	0	↑ 1	←↑ 1	↖ 2	←↑ 2	←↑ 2

This matrix will be used later to reconstruct LCS by tracing backwards:

	∅	**A**	**G**	**C**	**A**	**T**
∅	∅	∅	∅	∅	∅	∅
G	∅	← ∅	↖ (G)	← (G)	← (G)	← (G)
A	∅	↖ (A)	←↑ (A)&(G)	←↑ (A)&(G)	↖ (GA)	← (GA)
C	∅	↑ (A)	←↑ (A)&(G)	↖ (AC)&(GC)	←↑ (AC)&(GC)&(GA)	←↑ (AC)&(GC)&(GA)

Super useful video: Longest common subsequence algorithm

(Analisi complessità)

7 Amortized Analysis

Average time complexity takes a mean over many possible inputs to a single operation. The bound may be exceeded for a few "hard" inputs, but not for a sufficiently large number of inputs. For example think about the quick-sort algorithm where the average time complexity is $O(n \log n)$ while for some hard inputs (already sorted or reverse sorted) is $O(n^2)$. Amortized time complexity takes a mean over many iterations of the same operation. The bound may be exceeded for single operations, but not for a sustained period of operation: if the total amortized time of a series of operations is $O(f(n))$, some individual operations could take more time than $O(f(n))$ but they will be 'amortized' by 'lighter' operations . Generally we use AA over a data structure to evaluate it: often a data structure has some cheap operations and an expensive one. The goal of amortized analysis is to find an average cost of all of them without using any kind of probability.

8 Aggregated analysis

The aggregate method is a simply method of amortized analysis, not very precised that it's based on these relations:

$$\text{amortized cost} = \frac{\text{Total cost}}{\text{number of operations}}$$

$$\sum_{operations} \text{amortized cost} \geq \sum_{operations} \text{actual cost}$$

This two equations say that we can find an upper bound on total cost by adding the cost of all the operations (cheap and expensive ones) and then divide by the number of ops.

9 Accounting analysis

The concept is that you can 'store' some value during the analysis of the operations. For example an insertion uses a coin and store another coin. And then during the elimination I can consume the coins stored. Obviously all of this is.

$$\text{amortized cost} = \text{actual cost} + \text{deposits} - \text{withdraws}$$

An example for this could be the **table doubling** .

10 Potential analysis

Taking the Accounting method and evolving it we obtain the 'Potential method' that basically it's the same but in 'physics style'. It consists in :

- First find the potential function which fits best
- the amortized cost of each operation is the actual cost + the difference in potential $\Delta(\phi)$ that that operation caused

$$\hat{c} = c + \phi(i+1) - \phi(i)$$

So:

$$\sum (c + \phi(i+1) - \phi(i)) \geq \sum (c)$$

We would like to have a 0 potential energy at the "start" of the data structure (when for example it's empty) and then (based on the operations that we do) different potentials associated with the energy that we have 'stored' in the data structure and the energy we have used and subtracted from it. For example the potential function ϕ for a dynamic table which doubles its size every time is almost full could be: $\phi = 2(D.num - \frac{D.size}{2})$ so that when the table is just ' 're spawned" the potential is zero (we don't want 0 potential when the data structure is empty but when the data structure has just re-allocated.

10.0.1 Binary counter

The potential function could be the number of 1s. More there are and more you are near the carry which will cause a 'big change of the system'.

(Analisi complessità)

11 Competitive analysis

Difference between online (real-time) and offline algorithms:

- an online algorithm A executes the given operation without any knowledge of the future incoming operations
- an offline algorithm knows the whole sequence in advance

An online algorithm is α-competitive if exists a constant k such that for any incoming sequence of operations S we have that $C_A(S) \leq \alpha C_{\text{offline}} + k$ where C_{offline} is the optimal offline algorithm.

11.1 Move to front heuristic

Self-organizing lists. Elements that are accessed frequently will be in front of the list. MTF algorithm is 4-competive with an offline algorithm (that's good). In this video MTF is used to cache voxels in a rendering engine .

Parallel Patterns PThread OpenMP MPI

12 Parallel Programming

Parallel programming can be a powerful tool for improving the performance of code and for taking advantage of the capabilities of modern hardware.

Main advantages:

- performance
- cheaper than sequential implementation
- basically the ''big problems" can only solved by parallel algorithms

12.1 Brief history of parallel programming

PP has continued to increase during time driven by the development of new hardware architectures and the need for more computational power. Some of the main steps of PP evolution include:

- (80s and 90s): the development of high-level languages and parallel libraries made it possible applying PP to a wider range of applications.
- (2000s): the widespread adoption of multicore microprocessors began to drive the development of new parallel programming techniques and tools.
- (10s): the rise of manycore architectures, such as GPUs, led to the development of new parallel programming approaches, such as data parallelism and task parallelism.

TECHNOLOGY	TYPE	YEAR	AUTHORS
Verilog/VHDL	Languages	1984//1987	US Government
MPI	Library	1994	MPI Forum
PThread	Library	1995	IEEE
OpenMP	C/Fortran Extensions	1997	OpenMP
CUDA	C Extensions	2007	NVIDIA
OpenCL	C/C++ Extensions + API	2008	Apple
Apache Spark	API	2014	Berkeley

Manycore architectures are becoming increasingly common in modern computing, and are often used in applications that require a lot of computational power, such as data analysis, scientific simulations, and machine learning. A GPU, or graphics processing unit, is a type of manycore architecture that is specifically designed for high-performance graphics and parallel computation. GPUs are typically composed of hundreds or thousands of small, simple cores that are optimized for performing the parallel computations required for rendering graphics.

12.2 Independency from architecture and automatic parallelization

The design of parallel algorithms should focus on the logical steps and operations that are required to solve a particular problem, rather than the details of how those steps will be executed on a specific hardware platform. **But** the performance of a parallel algorithm can vary significantly depending on the hardware on which it is run. Therefore design a "good" parallel algorithm is not enough: which parallelism is available on the considered architecture is more important since non suitable parallelism can introduce overhead.

TECHNOLOGY	Target Independent Code?	Development Platforms
Verilog/VHDL	Yes (behavioral) No (structural)	Mainly Linux
MPI	Yes	All
PThread	Yes	All - Windows through a wrapper
OpenMP	Yes	All - Different compilers

TECHNOLOGY	Target Independent Code?	Development Platforms
CUDA	Depend on CUDA capabilities	All
OpenCL	Yes	All - Different compilers
Apache Spark	Yes	Mainly Linux

12.3 Automatic Parallelization

Automatic parallelization refers to the process of using a compiler or other tool to automatically transform sequential code into parallel code without requiring explicit parallelization directives from the programmer. In practice, it is not always possible for the compiler to accurately parallelize code. For example the compiler is not able to infer if 2 pointers of 2 arrays are pointing different region of RAM and are not overlapping while the programmer knows how to design the parallel algorithm. So parallelization by hand is predominant and the programmer needs to give hints to the tool: the concept is that the programmer needs to describe the parallelism to the compiler to make it exploitable.

12.4 Dependency Analysis

To determine if a set of statements can be executed in parallel, we have to do an analysis since not everything can be executed in parallel.

In general to execute in parallel: - statement order must not matter - statements must not have dependencies

The Dependency Analysis is performed over IN and OUT sets of the statements. The IN set of a statement is the set of memory locations (variables) that may be used in the statement. The OUT set of a statement is the set of memory locations that may be modified in the statement.

Often loops can be parallelized (we have to "unroll" the iterations), but there are also **loop-carried** dependencies, which often prevent loop parallelization. A loop-carried independence is a independence between two statements that is present only if this two statements are in two different iterations of the same loop.

12.5 Code extensions and languages for parallelism

The advantages of using extensions (such as OpenMP or MPI) of sequential languages instead of native parallel languages are many:

- Familiarity and ease of use
- Interoperability: this can be especially useful for codebases that use a mix of sequential and parallel code.
- Portability: Extensions of sequential languages are often designed to be portable across different platforms and architectures.

Overall, the advantages of using extensions of sequential languages instead of native parallel languages depend on the specific needs and goals of the code being developed.

Mainly the 3 macro paradigms of parallelism are:

- Single instruction, multiple data: most of the modern GPUs
- Multiple instruction, single data: experimental
- Multiple instruction, Multiple data: threads running in parallel on different data

TECHNOLOGY	SIMD	MISD	MIMD
Verilog/VHDL	Yes	Yes	Yes
MPI	Yes	Yes	Yes
PThread	Yes	(Yes)	Yes
OpenMP	Yes	Yes	Yels

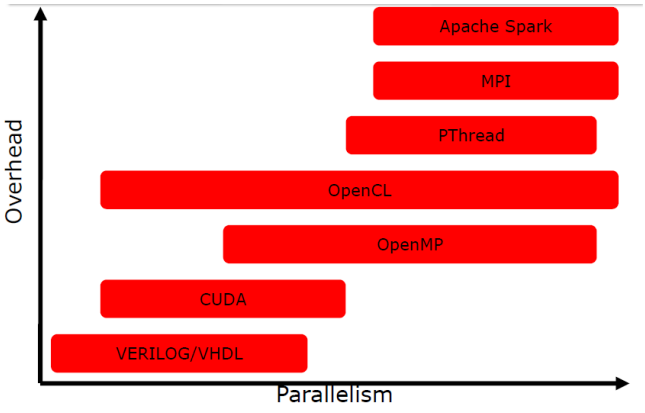
TECHNOLOGY	SIMD	MISD	MIMD
CUDA	Yes	No	Yes)
OpenCL	Yes	(Yes)	Yes
Apache Spark	Yes	No	No

We can classify parallelism over different levels:

- bits level: it is very relevant in hardware implementation of algorithm.
- instructions level: different instructions executed at the same time on the same core. This type of parallelism can be easily extracted by compilers.
- tasks level: a logically discrete section of computational work.

TECHNOLOGY	Bit	Instruction	Task
Verilog/VHDL	Yes	Yes	No
MPI	(Yes)	(Yes)	Yes
PThread	(Yes)	(Yes)	Yes
OpenMP	(Yes)	(Yes)	Yes
CUDA	(Yes)	No	(Yes)
OpenCL	(Yes)	No	Yes
Apache Spark	(Yes)	No	(Yes)

TECHNOLOGY	Parallelism	Communication
Verilog/VHDL	Explicit	Explicit
MPI	Implicit	Explicit
PThread	Explicit	Implicit
OpenMP	Explicit	Implicit
CUDA	Implicit(Explicit)	Implicit(Explicit)
OpenCL	Explicit/Implicit	Explicit/Implicit
Apache Spark	Implicit	Implicit



12.6 Main features of studied languages

12.6.1 PThread

PROS: - different architectures - explicit parallelism and full control / freedom CONS: - task management overhead - low level API - not scalable

12.6.2 OpenMP

PROS: - easy to learn - scalable - parallel applications could be executed also sequentially without modifications CONS: - mainly focused on shared memory homogeneous systems - require small interaction between tasks

12.6.3 MPI

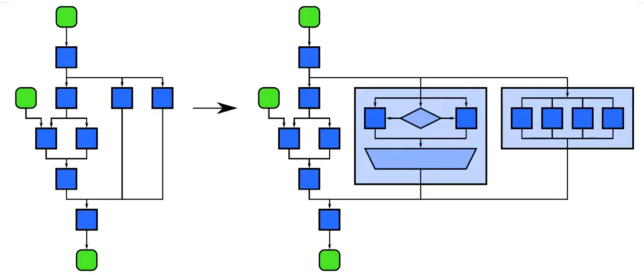
PROS: - different architectures - scalable solutions - communication explicit CONS: - communication can introduce overhead - programming paradigm more difficult

13 Parallel Patterns

A Parallel Pattern is a recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design. Patterns are universal, they can be used in any parallel programming system. Parallel patterns will be classified under these macro-classes:

- nesting pattern
- parallel control patterns
- parallel data management patterns
- other patterns

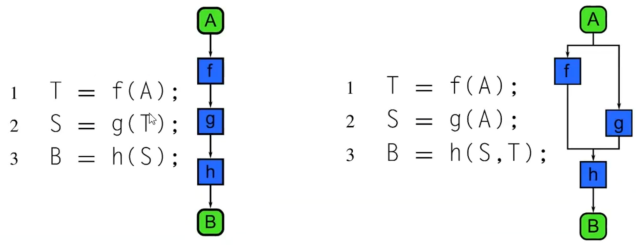
13.1 Nesting Pattern



13.2 Parallel control patterns

Recap of **Serial** Control Patterns: - sequence pattern - iteration pattern - selection pattern - recursion pattern

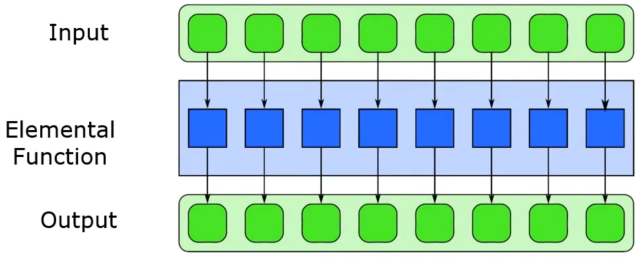
13.2.1 Fork-Join pattern



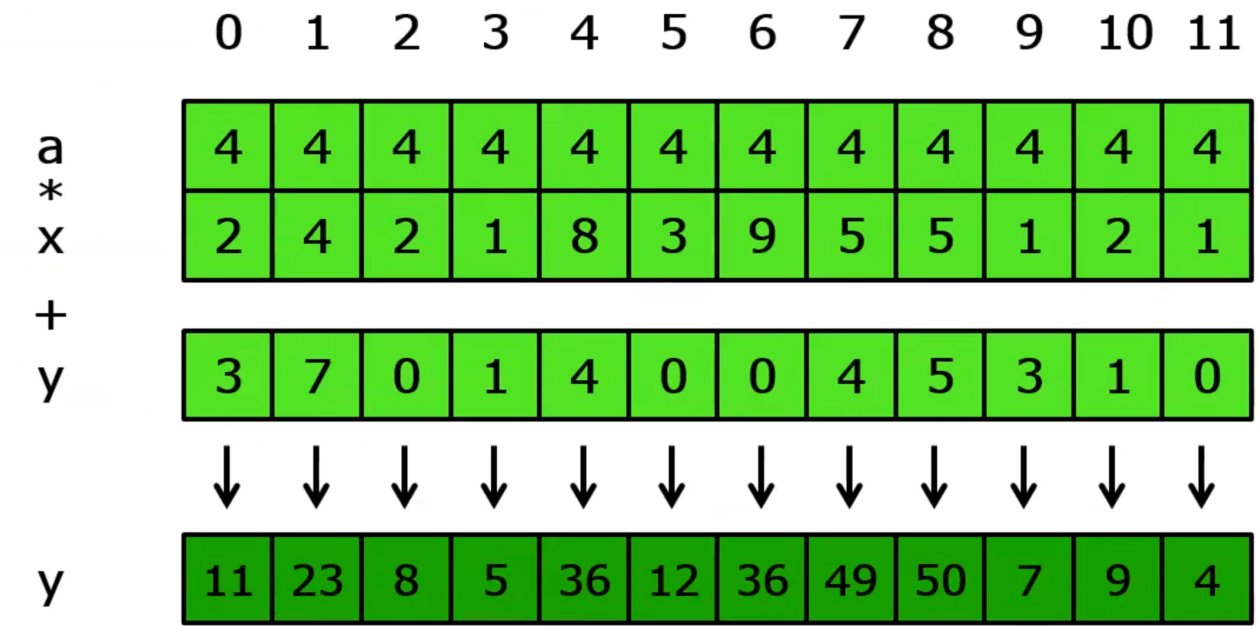
The fork-join pattern is a common parallelization technique used to decompose a sequence pattern into smaller subproblems that can be solved in parallel. The basic idea is to fork the original problem into multiple tasks and then combine the results (join) of these tasks later to obtain the final solution.

13.2.2 Map

The map is a foreach loop where each iteration is independent. It's embarrassingly parallel. The key concept is that a map is an operation applied to each element without knowledge of neighbors elements. Also a map function should be pure and should not modify any shared state.

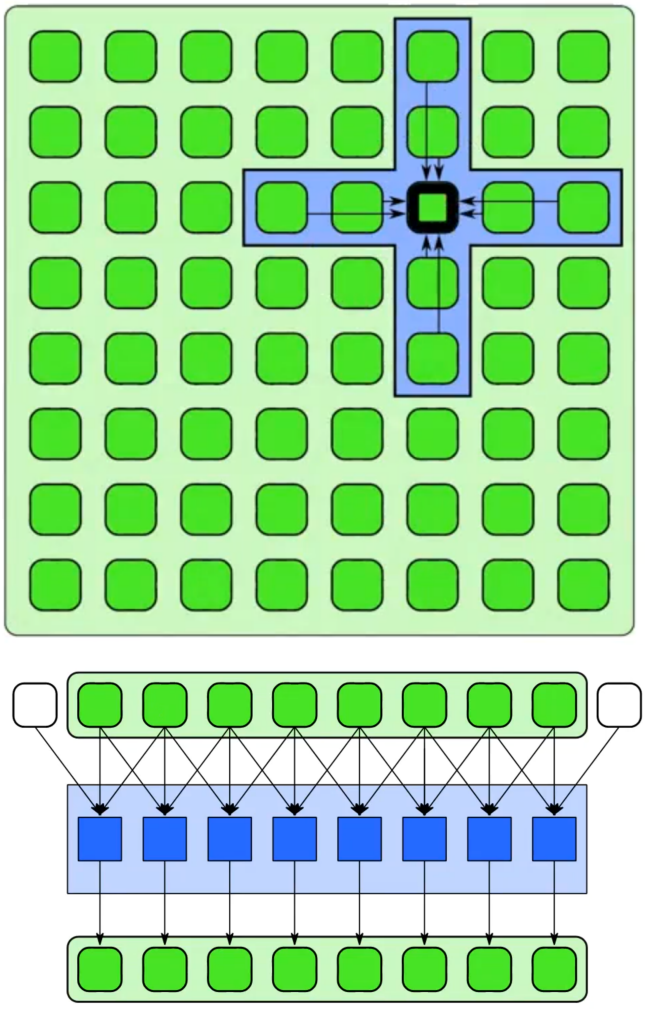


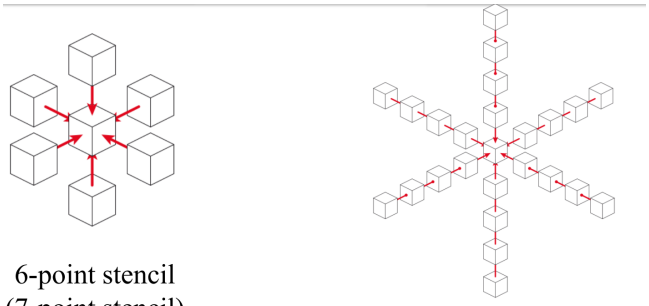
Purism means perfect independence and determinism, no data-races and no segfaults.



13.2.3 Stencil

An generalization of the map: a stencil is a function which accesses a set of "neighbors": the neighbors are a set of cells obtained by a fixed "offsets/distance" relative to the output position. Stencils can operate on one dimensional and multidimensional data so the neighborhoods can range from compact to sparse, square to cube, and anything else.





6-point stencil
(7-point stencil)

24-point stencil
(25-point stencil)

13.2.3.1 Stencil and ghost cells.

Since computing the value of each point requires the values of other points these computations are **not** embarrassingly parallel. Specifically, the points at the borders of a chunk require the values of points from the neighboring chunks. In order to apply the stencil operation consistently to all points in a computational domain, additional space is allocated for a series of **ghost cells** around the edges of each chunk. These ghost cells form a **halo** around each chunk, containing replicates of the borders of all immediate neighbors. **The ghost cells are not updated locally, but provide the necessary stencil values when updating the borders of the**

chunk.

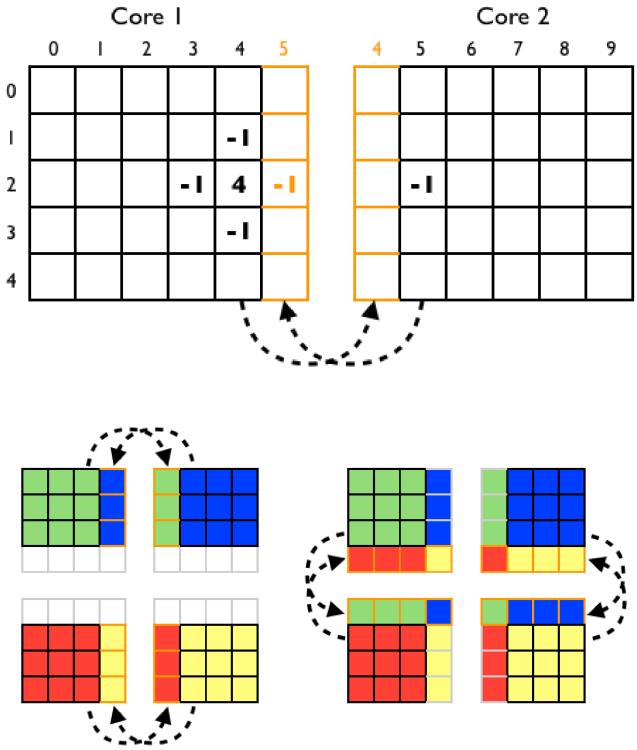
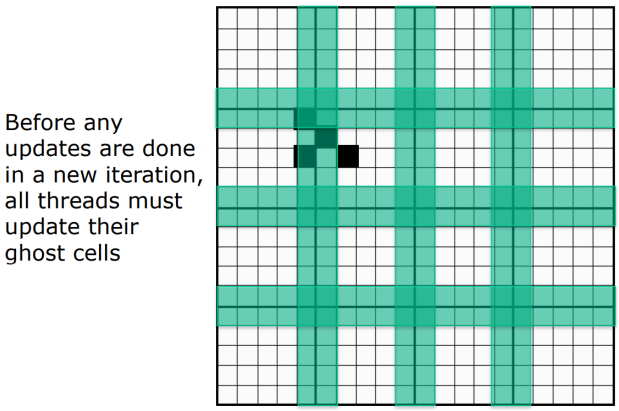


Figure 8: Two Dimensional Border Exchanges in two waves

The use of ghost cells in stencil parallel applications can improve the accuracy and efficiency of computations but also involves a **trade-off** between **computation** and **communication**.

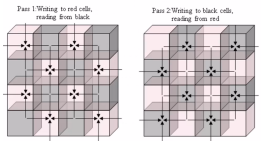


The ghost cells to total cells ratio will rapidly increase causing a greater demand on memory if we increase the number of threads.

13.2.3.2 Possible optimizations in Stencil pattern

- We could double or triple our **ghost cell boundary**: we do **extra computation** and we reduce the number of border exchanges, decreasing the associated communication overhead. This allows us to perform several iterations without stopping for a ghost cell update.

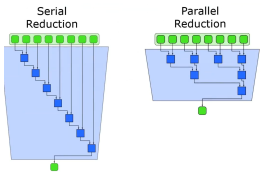
- A possible **latency hiding** could be to compute interior of stencil while waiting for ghost cell updates.
- In some specific cases (iterative codes in computer simulations of physical systems) **SOR**, or **Successive Over-Relaxation**, could be useful. **Red/Black SOR** is a variant of the SOR method that uses a special ordering of the grid points to speed up the process. Fully parallelizable since in the first step we only read from black and we write on red. Vice versa on second step. We actually "predict" the values of the next iteration when we are in a step... but at least they are completely parallelizable. It consists on an **interpolation** and **approximation**.



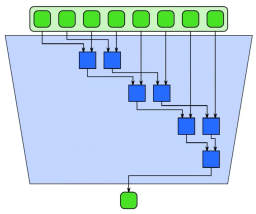
- Cache optimizations are based on assumptions over cache-lines: for example we can assume that rows of a 2D array are contiguous in memory. This means that horizontally related data will tend to belong to the same cache line while vertical offset accesses will most likely result in cache misses. **Strip-mining** is an optimization which is based on considerations over cache lines: dividing the overall grid of data that needs to be processed into smaller **strips** . Strips are groups elements in a way that avoids redundant memory accesses and aligns memory accesses with cache lines.

13.2.4 Reduction

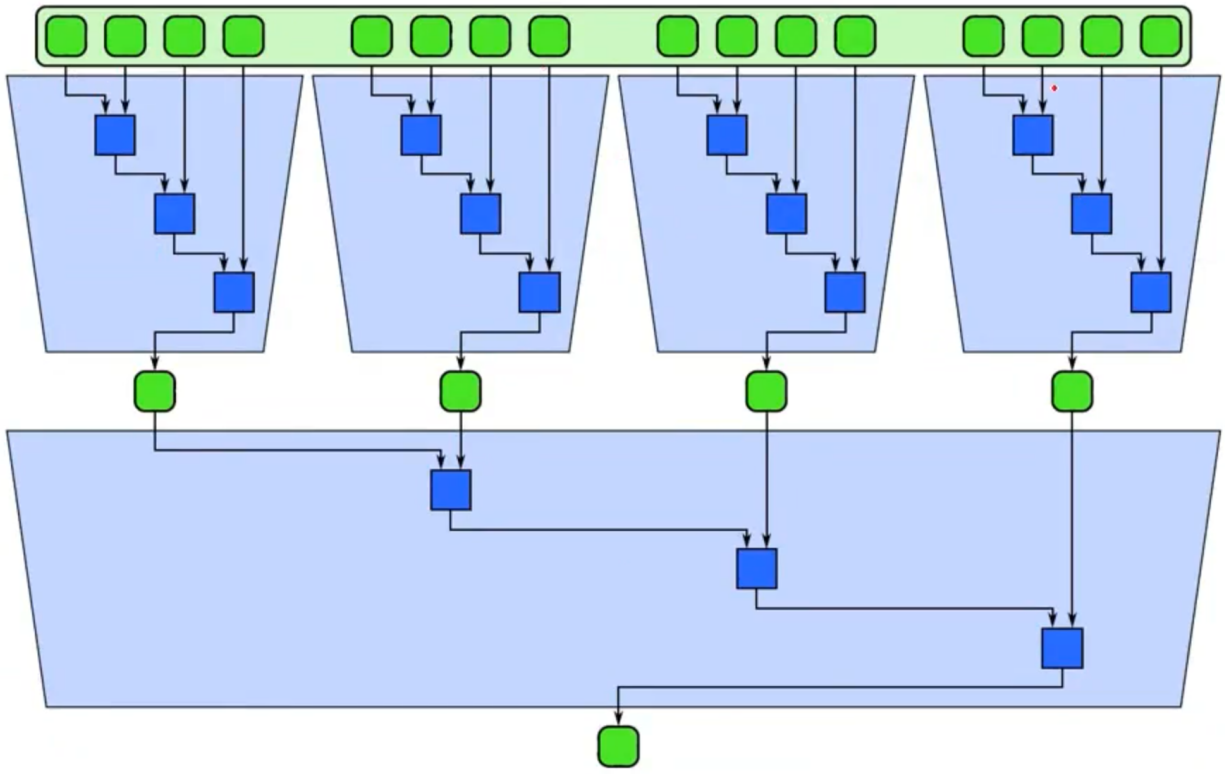
Reduction combines every element in a collection using an associative function. If the function isn't associative is not possible to parallelize a reduction operation. The associative property allows us to ''split" and change the order of operations of the reduction. Note that addition, multiplication, maximum, minimum and boolean AND, OR, XOR are all associative.



Even a single processor can perform ''vectorization". For example without doing any parallelization we can still have a speed up because we can make an operation with 2 elements in a cycle (so we have speedup of 2):



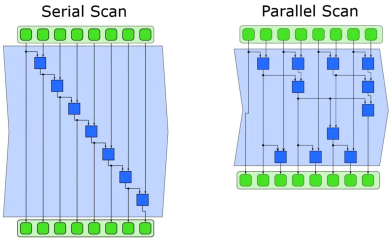
The main concept of reduction parallelization is **tiling** : breaking chunks of work to reduce serially.



Reduce example is the dot product is an essential operation in physics, graphics and videogames. The dot production of $\vec{a}$ and $\vec{b}$ is $|\vec{a}||\vec{b}|cos(\alpha)$ but also $\sum_i a_i b_i$ (the vector components) , so it's easy to parallelized it into

13.2.5 Scan

Scan computes all partial reductions of a collection. Like the reduction, if the function is **associative**, the scan can be parallelized. Scan is not obvious at first, because of the dependencies to previous iterations.



Note that in the parallel scan is necessary to perform more work than the serial version (look previous image).

Two types of scan:

- Inclusive scan: includes current element in partial reduction.
- Exclusive scan: excludes current element in partial reduction, partial reduction is of all prior elements prior to current element.

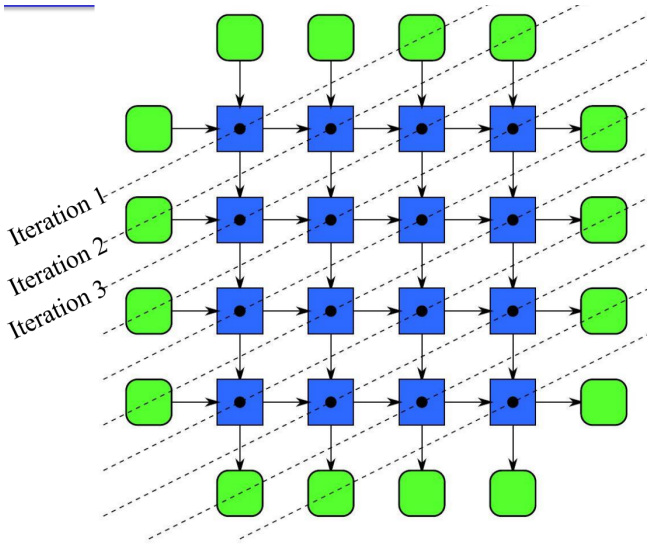
We can also identify two phases of the algorithm:

- Up sweep: compute reduction
- Down sweep: compute intermediate results

13.2.6 Recurrence

The recurrence parallel pattern is a more complex version of the map parallel pattern, in which the loop iterations can depend on one another. This means that, unlike in the map pattern, the elements of the input data are not necessarily independent of each other, and the outputs of some elements may be used as inputs to other elements. As a result, the order in which the elements are computed may be important, and the pattern typically involves the use of a serial ordering to ensure that the elements can be computed correctly. This pattern is used to structure the parallel execution of code that involves recursion.

This can still be parallelized! Trick: find a plane that cuts through grid of intermediate result



13.3 Parallel Data Management Patterns

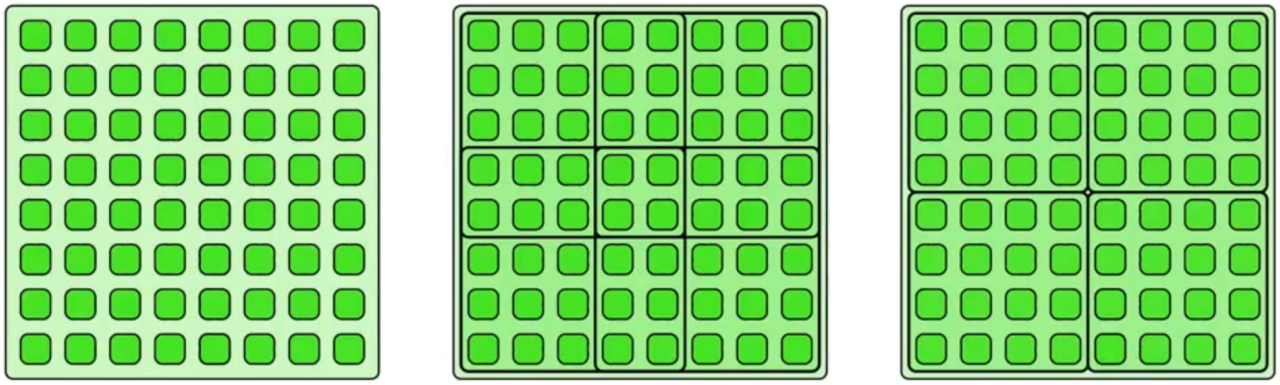
Serial Data Management Patterns are the classical ones:

- stack allocation
- heap allocation
- objects
- random read and write

Often the **bottleneck** is not the computation but the **data movement**. Considering the Principle of Locality is very important: it's better from an **hardware** perspective to keep data "local" and closer to the CPU to exploit the **cache**. **Transferring data across memory layers is costly since it can take many cycles.

13.3.1 Geometric decomposition

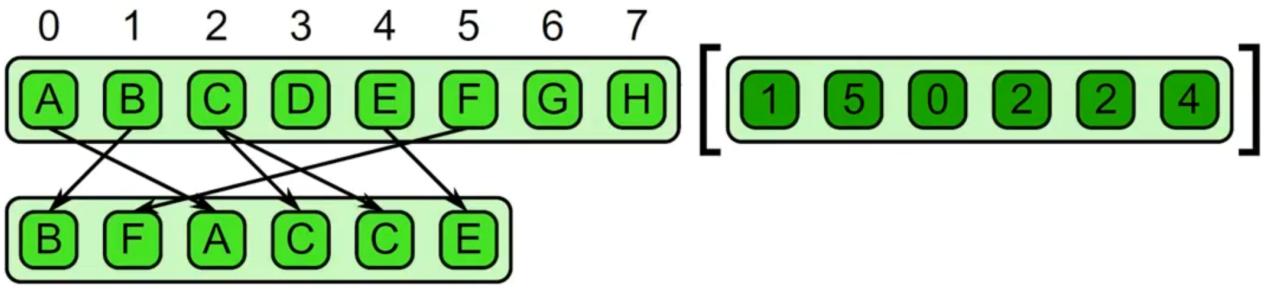
Geometric Decomposition is a common pattern to arrange data into subcollections (chunks) which can be overlapped or not. This pattern doesn't necessarily move data, it just gives us another view of it



Special cases are:
- partitioning pattern: sub-collections are same-sized and not-overlapping - segmentation pattern: non-uniformly sized partitions

13.3.2 Gather

Gather reads a collection of data given a collection of indices. Think of a combination of map and random serial reads. The output collection shares the same type as the input collection, but it share the same shape as the indices collection.



Read locations provided as input. Remember that the output will have the same size of the index array.

13.3.2.1 Zip

It's a special case of Gather, we start from two arrays and combine them.

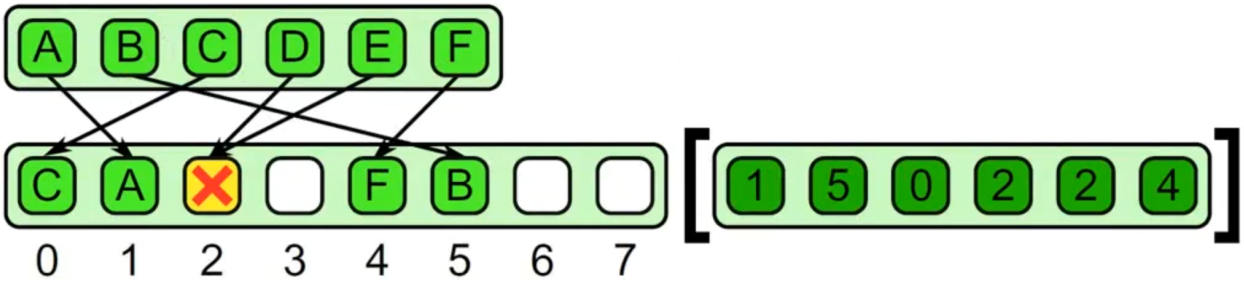
13.3.2.2 Unzip

Reverses a zip: extracts sub arrays at certain offsets from a given input.

13.3.3 Scatter

Scatter is the inverse of gather. A set of input and indices is required, but each element of the input is written to the output at the given index instead of read from the input at the given index.

This is different from Gather! Race conditions because write of same location are possible. Race conditions can occur when we have two writes to the same location!



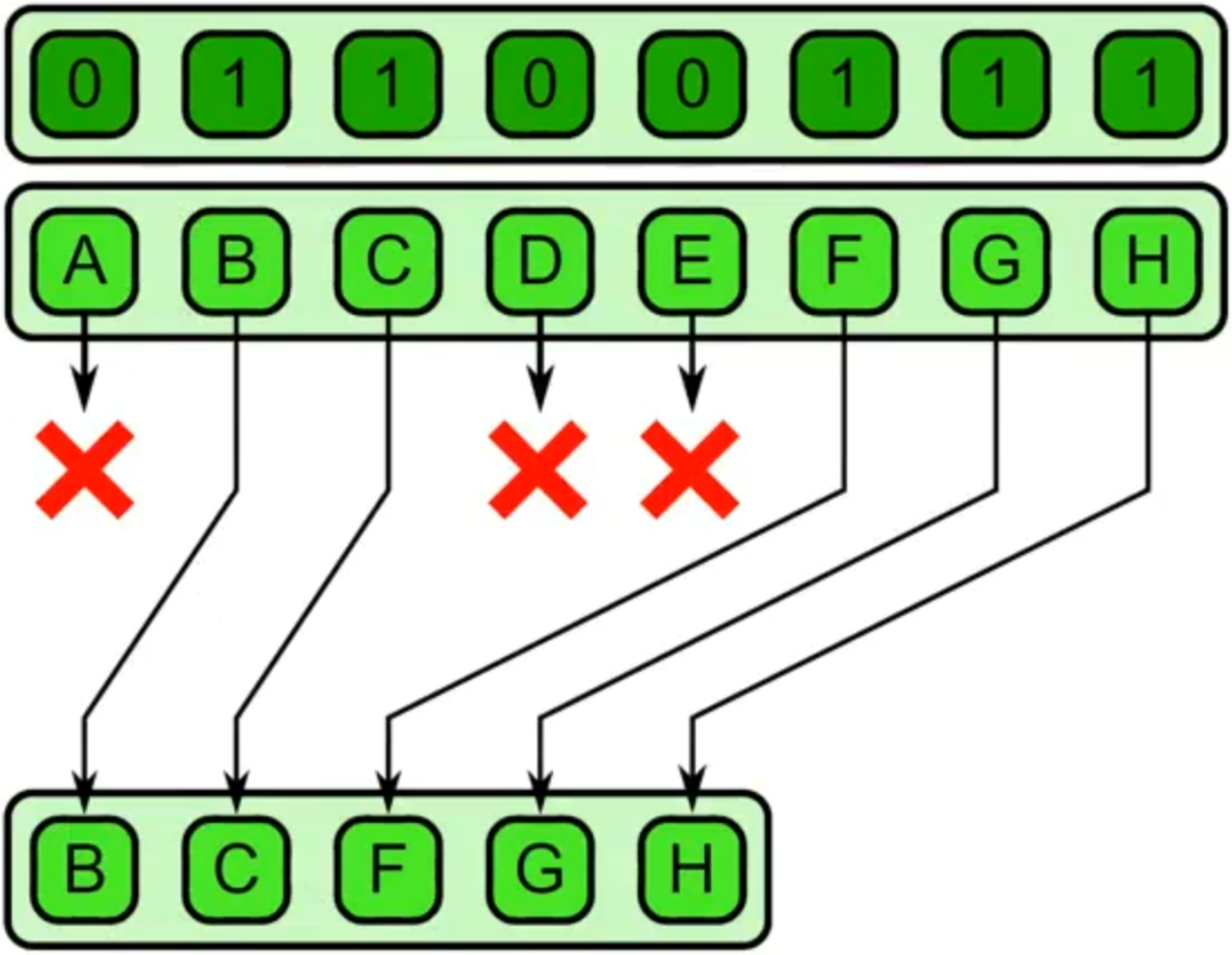
Write locations provided as input.

In case of collision we can have some rules like:

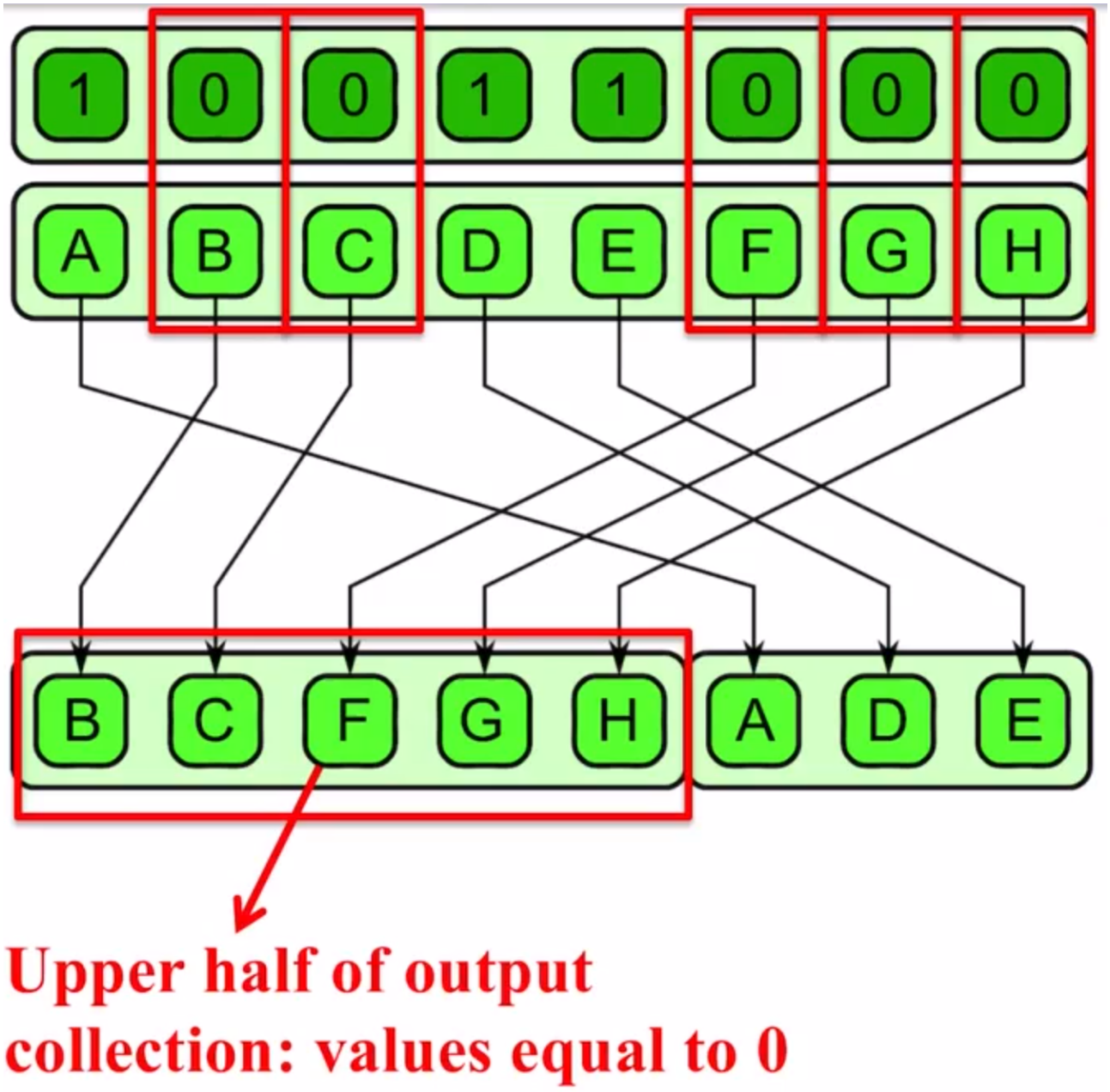
- in case of associative and commutative operators can merge colliders.
- we could associate to each value a priority. Example of this case in 3D graphics rendering.
- In case there aren't collisions the output is just a permutation, so no problem.

13.3.4 Pack

Pack is used to eliminate unused space in a collection. Elements marked false are discarded, the remaining elements are placed in a contiguous sequence in the same order.


0 1 1 0 0 1 1 1
A B C D E F G H
B C F G H

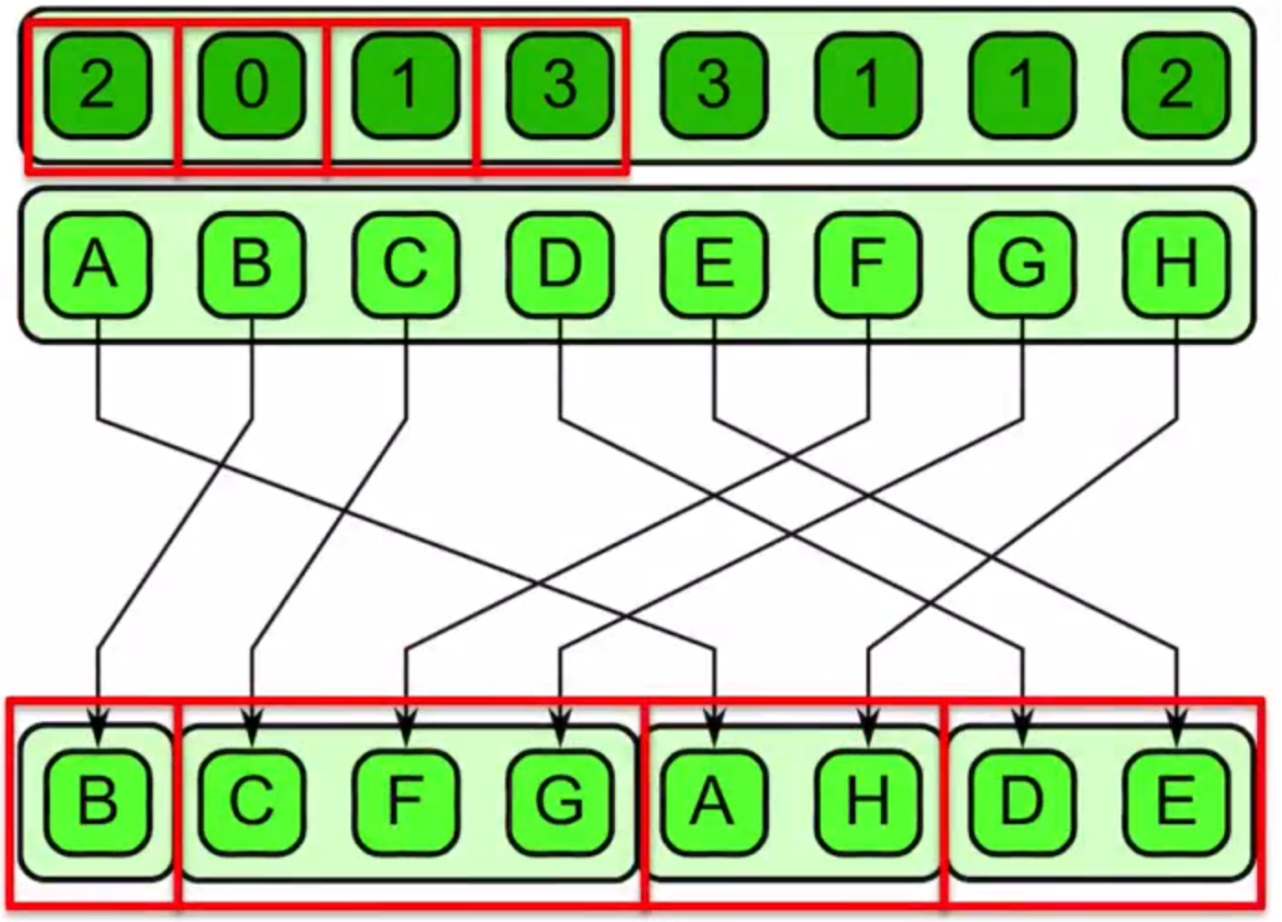
13.3.5 Split



Generalization of **pack** pattern, where the isn't information lose. There is also the ''inverse operation" pattern: unsplit.

13.3.6 Bin

There is also the **Bin** parallel pattern which is the generalization of split: simply split which support more categories.

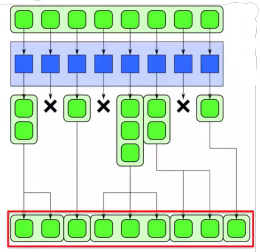


13.3.7 Pipeline

Pipeline connects tasks in a producer-consumer manner, which is very common. A linear pipeline is the basic pattern idea, but a pipeline in a DAG is also possible.

13.4 Other Patterns

- Expand: a combination of pack and map, where each element can output multiple elements.



- Superscalar Sequences: write a sequence of tasks, ordered only by dependencies
- Futures: similar to fork-join, but tasks do not need to be nested hierarchically
- Speculative Selection: general version of serial selection where the condition and both outcomes can all run in parallel

- Workpile: general map pattern where each instance of elemental function can generate more instances, adding to the "pile" of work
- Search: finds some data in a collection that meets some criteria
- Category Reduction: Given a collection of elements each with a label, find all elements with same label and reduce them

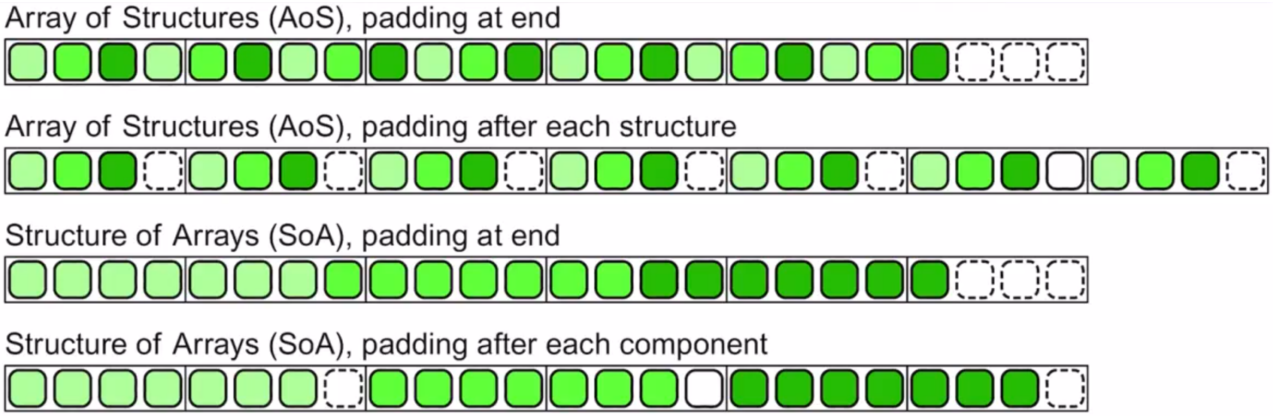
14 Different way to store things

14.1 Array of structures AoS

An array containing the different instances of a data structure. Extremely difficult to access memory for reads (gathers) and writes (scatters) but it can be useful if data is accessed randomly.

14.2 Structure of arrays SoA

A single data structure where in each property/attribute is stored all the values of all the different instances (using an array). Typically better for vectorization and avoidance of false sharing. Separate arrays for each structure-field, keeps memory access contiguous when vectorization is performed over structure instances.



The padding at the end indicates which is the size of a data structure.

15 Pthread

The POSIX thread (pthread) is based on the POSIX Threads standard, which defines a set of functions and data types for creating and controlling threads in a portable and consistent manner. PThread is a widely-used implementation of this standard, and is commonly used in multi-threaded programming on Linux and other Unix-like operating systems.

PThread level of control can be useful in some cases, but it can also make the programming of concurrent applications more complex and error-prone.

15.1 Basic recap of threads

A thread is a unit of execution within a process that can be scheduled and executed by the operating system. In the Shared Memory Model, threads have their own local resources, such as stack and register state, but they also have access to the shared resources of the process in which they run. This allows multiple threads to cooperate since they can communicate with each other implicitly by reading and writing to shared system

memory locations. However, this can lead to sync issues and the programmer must use mechanisms to ensure that threads access shared resources in a safe and orderly manner.

Threads created in a pthread application are not hierarchically organized. In the pthread library, threads are considered to be independent entities that can be scheduled and executed by the operating system in an arbitrary order.

15.2 Main Pthread functions

15.2.1 pthread_create

```
int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void *(*start_routine)(void *), void
```

1) `thread`, is a pointer to a `pthread_t` variable that will be set to the ID of the newly created thread.
2) `attr`, is a pointer to a `pthread_attr_t` structure that specifies optional attributes for the new thread. This parameter can be set to `NULL` to use the default values for all attributes.
3) `start_routine`, is a pointer to a function that will be executed by the new thread when it is created. This function must have a return type of `void *` and take a single `void *` argument.
4) `arg`, is a pointer to data that will be passed as an argument to the `start_routine` function when the new thread is created.

```c
struct thread_args {
    int arg1;
    char *arg2;
};

void * my_thread_func(void *args) {
    //remember the cast
    struct thread_args * my_args = (struct thread_args *) args;
    int arg1 = my_args->arg1;
    char *arg2 = my_args->arg2;
    // Use the arguments in the thread function...
}

int main() {
    pthread_t my_thread;
    struct thread_args my_args;

    my_args.arg1 = 10;
    my_args.arg2 = "hello";

    pthread_create(&my_thread, NULL, my_thread_func, &my_args);

    // Other code...
}
```

Note that an important step in pthreads to communicate variables is how `void *args` works: in this example the struct `thread_args` is defined to contain the two arguments that will be passed to the thread function. The generalization of this approach is to pass arguments of **any type** to a thread function as long as you define a struct that can hold the necessary data and pass a pointer to this struct to `pthread_create`.

15.2.2 pthread_join

Pthread_join is a function used in the POSIX thread (pthread) library to synchronize the execution of threads. It allows one thread to wait for the completion of another thread. When a thread calls pthread_join, it is suspended until the target thread (the thread being joined) completes its execution. The pthread_join function returns the exit status of the target thread, which can be used to determine whether it terminated successfully or not.

```c
#include <pthread.h>
#include <stdio.h>

void *my_thread_func(void *args) {
    // Do some work in the thread...
    return NULL;
}

int main() {
    pthread_t my_thread;

    pthread_create(&my_thread, NULL, my_thread_func, NULL);

    // Do some work in the main thread...

    pthread_join(my_thread, NULL);

    // Continue execution after the thread has completed...

    return 0;
}
```

15.2.3 pthread_barrier_t

```c
pthread_barrier_init(pthread_barrier_t barrier, pthread_barrierattr_t * attr, unsigned int count);
```

`count` is the number of variable to wait. `attr` is the number of variable to wait.

```c
#include <pthread.h>
#include <stdio.h>

pthread_barrier_t my_barrier;

void *my_thread_func(void *args) {
    // Do some work in the thread...

    pthread_barrier_wait(&my_barrier);

    // Continue execution after the barrier is reached...

    return NULL;
}

int main() {
    pthread_barrier_init(&my_barrier, NULL, 2);
```

```
    pthread_t my_thread;
    pthread_create(&my_thread, NULL, my_thread_func, NULL);

    // Do some work in the main thread...

    pthread_barrier_wait(&my_barrier);

    // Continue execution after the barrier is reached...

    pthread_join(my_thread, NULL);

    return 0;
}
```

Both the main thread and the new thread do some work, and then each calls the `pthread_barrier_wait(&barrier)` function to wait for the other thread to reach the barrier. When both threads have reached the barrier, they are released and can continue execution.

15.2.4 Cond and mutex

A **mutex** variable is a special type of lock that allows only one thread to hold it at any given time. When a thread acquires a mutex, it prevents other threads from acquiring the same mutex, ensuring that only one thread can access the protected resource at a time.

The `pthread_cond_wait` function release the mutex that is associated with the condition variable. This allows other threads to acquire the mutex and access the shared resource or data structure that is protected by the mutex.

```
#include <pthread.h>
#include <stdio.h>

pthread_mutex_t my_mutex;
pthread_cond_t my_cond;
int my_shared_var = 0;

void *my_thread_func(void *args) {
    pthread_mutex_lock(&my_mutex);

    // Wait for the condition to be signaled
    while (my_shared_var == 0) {
        pthread_cond_wait(&my_cond, &my_mutex);
    }

    // Do some work with the shared variable...

    pthread_mutex_unlock(&my_mutex);
    return NULL;
}

int main() {
    pthread_mutex_init(&my_mutex, NULL);
    pthread_cond_init(&my_cond, NULL);
```

```
    pthread_t my_thread;
    pthread_create(&my_thread, NULL, my_thread_func, NULL);

    // Do some work in the main thread...
    pthread_mutex_lock(&my_mutex);
    my_shared_var = 1;
    pthread_cond_signal(&my_cond);
    pthread_mutex_unlock(&my_mutex);

    pthread_join(my_thread, NULL);

    return 0;
}
```

`pthread_cond_signal(&cond)` signals only 1 thread! `pthread_cond_broadcast(&cond)` signals all the threads that are "listening" that condition variable. # OpenMP

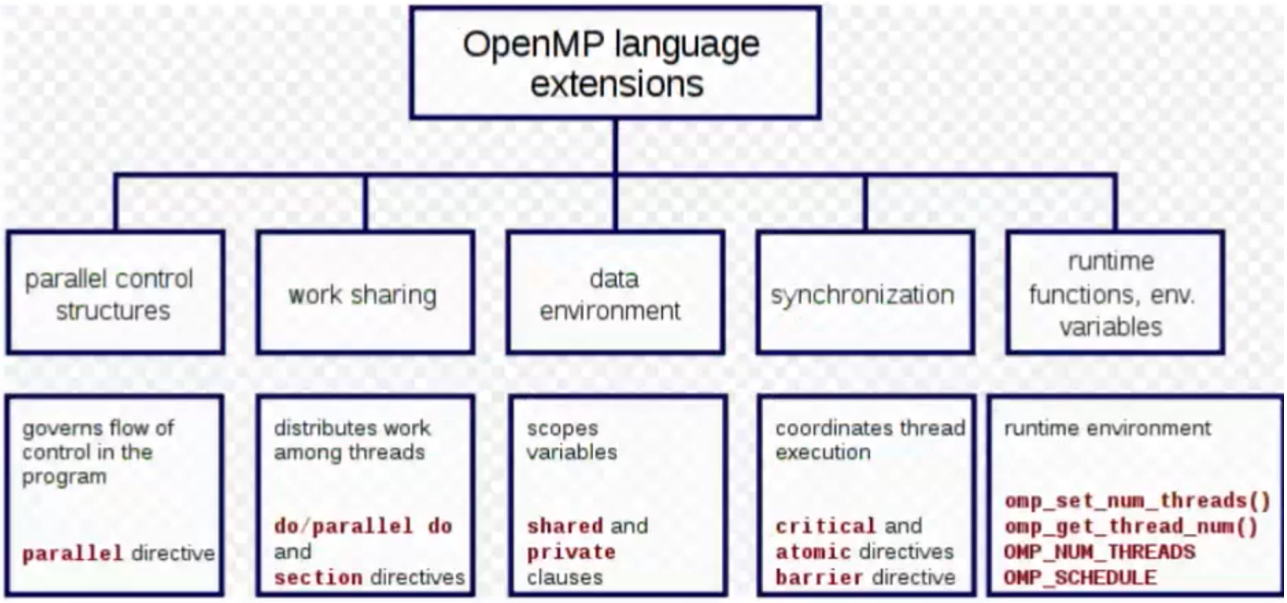
- OpenMP main goal is standardization and portability across a variety of **shared memory model** architectures.
- OpenMP uses a fork-join model, where a master thread forks a specified number of slave threads (so there is a hierarchy of threads).
- OpenMP provides directives to spawn threads, assign tasks, and manage synchronization.
- Variables defined outside a parallel region are visible to all threads in the team, while variables defined inside are private to each thread unless otherwise specified.

Note that OpenMP is an abstraction layer that provides a high-level interface for specifying parallelism, but the actual implementation of the parallelism may vary depending on the compiler and the target platform.

```
//this:
#pragma omp parallel num_threads(8) {...}
//might become:
for(i=0;i<8;i++){
    pthread_create (...);
}
for(i=0;i<8;i++){
    pthread_join (...);
}
```

Under the hood, a compiler that supports OpenMP may replace the OpenMP directives with code that uses another programming model, such as Pthreads, to implement the parallelism. For this it's also important to know that if the compiler does not support OpenMP, the directives will be treated as comments and ignored. This is because OpenMP directives are preprocessor directives, and they are only processed by the compiler if the compiler supports OpenMP. Nested parallelism is also possible in OpenMP, where a team of threads can create its own sub-teams of threads to perform additional parallel tasks: to this is important to specify the `nested` clause after the directive.

15.3 Overview of OpenMP



15.3.1 Clauses

In OpenMP, clauses are used within directives to control how the directive behaves. For example, the `private` clause can be used to make a variable private to each thread, and the `schedule` clause can be used to control loop iteration distribution among threads. There are many different types of clauses that can be used to control various aspects of behavior, such as thread scheduling, memory allocation, and synchronization.

15.3.2 Directives

The directives allow the programmer to specify which loops and regions of code can be executed concurrently by multiple threads.

```
#pragma omp <name> [list of clauses]
```

Preprocessor directives called pragma (pragmatic information).

15.4 Main directives

15.4.1 Parallel directive

```
#pragma omp parallel [clause, ...]
{
/* parallel section */
}
```

The main thread spawns a team of slave threads and becomes the master (thread number 0 in the within the team. Some important clauses that can be used with this directive are:

- `if(condition)`: This clause allows to specify a condition that must be satisfied in order for the region of code to be executed in parallel.

```
#pragma omp parallel if(n > 1)
{
```

```
    // Code to be executed in parallel
}
```

- `num_threads(n)`: This clause specifies the number of threads that should be spawned to execute the code in the parallel region. If this clause is not used, the number of threads will be determined by the OpenMP runtime.

```
#pragma omp parallel num_threads(4)
{
    // Code to be executed in parallel by 4 threads
}
```

The number of threads in a parallel region is determined evaluating the following factors, in order of precedence:

- Evaluation of the `if` clause: obviously if it's false, no threads will be instanced
- Value of the `num_threads` clause
- Use of the `omp_set_num_threads()` library function
- Setting of the `OMP_NUM_THREADS` environment variable
- Implementation default, example the number of CPUs on a node.

15.4.2 For directive

```
#pragma omp parallel {
    #pragma omp for [clauses...]
    <for_loop>
}
```

Parallelize execution of iterations of the cycle with the assumption that iterations number are static (they are not modified during runtime) and there are no data dependencies in the loop. Most important clauses: `schedule (type [ , chunk])` describes how iterations of the loop are divided among the threads in the team:

```
#pragma omp parallel {
    #pragma omp for schedule (type [ ,chunk])
    <for_loop>
}
```

Where: - `type` specifies the type of scheduling to be used: - `static`: specifies that iterations should be divided into equal-sized chunks. Eventually it is possible to divide into chunks of size `chunk` using the respective field. - `dynamic`: the loop iterations dynamically scheduled among threads. The default chunk size is 1 but you can specify. - `chunk` specifies the size of the chunk of iterations that should be assigned to each thread.

```
#pragma omp parallel {
    #pragma omp for schedule(static, 10)
    for (int i = 0; i < n; i++) {
      // loop body
    }
}
```

Note that for directive **needs to be enclosed in a parallel section**. The for directive will run the loop serially otherwise.

15.4.3 Sections

The directive to achieve MIMD parallelism of the application. Closed sections of code that are divided among the threads and executed concurrently.

```
#pragma omp sections [clause...]{
    #pragma omp section
        { / * code section 1 * / }
    #pragma omp section
        { / * code section 2 * / }
}
```

15.4.4 Single and master directive

```
#pragma omp single [clauses...]
{ / * code section   * / }

#pragma omp master
{ / * code section   * / }
```

`single` specifies that a section of a code is executed only by a single thread; the choice on the thread is implementation dependent. `master` specifies that a section of a code is executed only by the master thread, no implied barrier at the end.

15.4.5 Critical Directive

```
#pragma omp parallel
{
  // instructions to be executed by each thread

  // critical directive
  #pragma omp critical
  {
    // instructions to be executed by one thread at a time
  }

  // after the critical section
}
```

The critical directive specifies a **section of code** that must be executed by only one thread at time. The name is used as as global identifier. Different critical sections with the same name are considered as the same region. It's a concept very similar to the mutex stuff.

15.4.6 Atomic directive

```
#pragma omp parallel
{
    // instructions
    #pragma omp atomic
    shared_variable += 1;
    // instructions
}
```

The atomic directive, on the other hand, is used to specify that a particular **memory location** should be updated atomically. Different aspects of the statements are performed atomically depending on the clauses: `read` , `write` , `update` are possible clauses.

15.4.7 Barrier directive

```
#pragma omp parallel
{
  // instructions to be executed by each thread

  // barrier directive
  #pragma omp barrier

  // instructions to be executed by each thread after the barrier
}
```

It's a directive used to synchronized all threads. All threads will wait the others before proceed over the barrier.

15.4.8 Scope of variables

The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables (shared by default) should be scoped. The use of scope variables is an essential part of writing effective parallel code with OpenMP, since they can help to improve the performance and efficiency of the parallel code.

- private: `#pragma omp <directive name> private (list)` to have vars private for each thread. Each variable is uninitialized, we need `firstprivate` to specify that each var is initialized with the value of the variable, because it exists before the parallel construct. `lastprivate` specifies that the "final" variable is set equal to the private version of the thread which executes the final iteration or last section.
- shared: `#pragma omp <directive name> shared (list)` declares variables in its list to be shared among all threads in the team. A shared variable exists in only one memory location and all threads can read or write to that address. **It is the programmer's responsibility** to ensure that multiple threads properly access shared variables, for example by guarding the access to shared variable using the **critical directive** .

- default: `#pragma omp <directive name> default(shared|none)` specifies the default data scope for variables in the parallel region. If `default(shared)` is used, all variables in the region will be shared by default, meaning that they can be accessed and modified by all threads. If `default(none)` is used, no variables will be shared by default, and the programmer must explicitly specify which variables should be shared using the `shared` clause.

15.4.9 Reduction

reduction: `#pragma omp for reduction (operator:<variable_name>)` in a loop this kind of variables aggregates a value that depends on each iteration of the loop but not by their order. Possible operators are: `+ * - ^ && || min max` In this example, the `reduction` clause specifies that the values of the `sum` variable computed by each thread should be added together to produce the final result. Note that this can be more efficient than having each thread update a shared `sum` variable, as it avoids the need for synchronization:

```
#pragma omp for reduction(+:sum)
for (int i = 0; i < n; i++)
{
    sum += a[i];
}
```

15.4.10 Task directive

```
#pragma omp task [clause, ...]
{  /* structured block */ }
```

The task directive specifies a work unit which **may** be executed by another thread in the same team. Tasks are composed of code + data environment, which is initialized at creation time. The `depend` clause enforces constraints on the scheduling of tasks by modeling data dependencies:

```
#pragma omp task depend(in|out|inout:<variable_name>)
{ /* structured block */ }
```

The tasks are put into a task pool and picked by idle threads. Using dependencies we can explicit an execution order.

```
#include <stdio.h>
#include <omp.h>

int main()
{
    int data[];
    // Initialize data array

    // Create a task with a dependency on data[
    #pragma omp task depend(in:data[)
    {
        // Modify data[ in some way
        data[ = some_function(data[);
    }

    return 0;
}
```

Task directive can be use in conjunction with:

```
#pragma omp taskwait  //barrier for tasks completion
#pragma omp taskyield //used inside a task, interrupts execution
```

Task directive useful for implement the "Pipeline" parallel pattern.

15.4.11 Runtime functions

- `int omp_get_num_threads ()`
- `void omp_set_num_threads (int num_threads)` Sets the number of threads that will be used in the next parallel region
- `int omp_get_thread_num()`
- `double omp_get_wtime()` Provides a wall clock timing routine, use it in pairs to see how much time passes between calls
- `double omp_get_wtick()` Returns the precision of the timer used by the previous function.

16 MPI, Message Passing Interface

MPI is a library-based specification for parallel and distributed computing, similar to OpenMP and focused on Fortran and C languages. MPI is designed to explicitly implement parallelism in distributed memory environments using processes and a library-based syntax, while POSIX threads and OpenMP are used for implementing shared memory parallel programming with threads. MPI supports many parallel architectures and not just distributed systems. Compilation tools for MPI are typically wrappers that set the required flags and environment on top of existing compilers.

16.1 Basic functions

```c
#include <mpi.h>

int main(int argc, char argv[])
{
    /* No MPI call before this line */
    MPI_Init(&argc, &argv);

    /* MPI calls go here */

    MPI_Finalize();
    /* No MPI call after this line */

    return 0;
}
```

`MPI_init(int * argc_p, char *** argv_p)` is used to initialize the MPI environment and must be called by all processes in a parallel program before any other MPI function can be called. MPI_Init is responsible for setting up the communication infrastructure that allows the processes to communicate and coordinate their work. `argc_p` and `argv_p` are the parameters of the main function. In case of multithreaded programs it is used `MPI_Init_thread` when the program has to run on threads and not on a cluster. Dually `int MPI_Finalize(void)` is to de-allocates the message buffers and cleans up all.

The `MPI_COMM_WORLD` is the main communicator is a collection of processes that can communicate with each other, but there are many communicators. The `MPI_Comm_size(MPI_Comm , int * size)` function returns the number of processes in the communicator, while the `MPI_Comm_rank(MPI_Comm , int * rank)` function returns the rank of a particular process in the communicator. These functions are used to identify and distinguish processes within the communicator. The return value of these functions is an error code, while the relevant data is inserted into a pointer provided as an argument.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);

  // Get the number of processes
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // Get the name of the processor
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int name_len;
  MPI_Get_processor_name(processor_name, &name_len);

  // Print off a hello world message
  printf("Hello world from processor %s, rank %d out of %d processors\n",
         processor_name, world_rank, world_size);
```

```
  // Finalize the MPI environment.
  MPI_Finalize();
}
```

16.1.1 MPI_Send and MPI_Recv

The `MPI_Send` routine is used to write a message to a communicator.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

The `MPI_Recv` routine is used to read a message from a communicator.

```
int MPI_Recv(const void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_S
```

Both are a blocking routine: the function will not return until the targeted processes reads the message or have written the message.

- `buf` is the array ready to receive data or it's the data ready to send.
- `count` states how many replicas of the data type will be sent, or the maximum allowed in the buffer
- The `status` parameter is used to store information about the message.
- the `source` and `dest` argument specifies the rank of the process in the communicator `comm` from which the message is received. This value can be a specific rank, such as `0` or `1`, or it can be the special value `MPI_ANY_SOURCE`, which indicates that the message can be received from any process in the communicator `comm`. This can be useful if you don't know the rank of the process that will be sending the message.
- `tag` is used to distinguish messages travelling on the same connection (nonnegative int).
- Note that sending and receiving messages introduces synchronization between processes.
- Where there is synchronization, deadlocks may arise.

16.1.2 MPI_Datatype

We use this type to define what kind of message MPI is delivering.

Possible datatypes:

- MPI_INT
- MPI_CHAR
- MPI_BYTE
- MPI_FLOAT
- MPI_DOUBLE
- MPI_SHORT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_UNSIGNED
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED_LONG

Typically al processes can access the standard output, while only the process 0 can access the standard input.

16.2 Communication functions

16.2.1 Data Distribution

16.2.1.1 Broadcast

```
int MPI_ Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

16.2.1.2 Gather

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, M
```

Join even chunks of data from all processes in a communicator. The result stored in `recvbuf` is a single vector of elements in `dest` process, with `size = communicator size * recvcount` .

16.2.1.3 Scatter

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount,
```

Scatters data evenly across all processes in the communicator. `sendcount` is the amount of element to be stored in each `recvbuf` structure.

16.2.1.4 Scatterv

```
int MPI_Scatterv(const void *sendbuf, int sendcounts[k], const int displs[k], MPI_Datatype sendtype, void
```

Scatters data across all processes according to a user defined distribution. Process with rank `k` will receive `sendcounts[k]` elements, starting from the position `displs[k]` of the sendbuf.

16.2.2 Data collection

16.2.2.1 Reduce

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int dest,
```

Reduces the values in the `sendbuf` using the specified operation and stores the result in the receive buffer. Note that `count` specifies how many elements of send/input buffer should be reduced.

MPI_Op	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND (Between booleans)
MPI_BAND	Bit-wise AND
MPI_LOR	Logical OR
MPI_BOR	Bit-wise OR
MPI_LXOR	Logical XOR
MPI_BXOR	Bit-wise XOR
MPI_MAXLOC	Maximum value and location
MPI_MINLOC	Minimum value and location

16.2.2.2 Allreduce

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Co
```

MPI_Allreduce is a collective operation that combines the values of all the processes in a communicator and broadcasts the result to all the processes, while MPI_Reduce is a collective operation that combines the values of all the processes in a communicator and returns the result to a single process (the root one). It's the same of Reduce but the result of the reduction is stored in the `recvbuf` of all processes in the communicator.

16.2.3 Miscellaneous functions

16.2.3.1 Barrier Barrier explicit synchronization for all processes in a communicator.

```
// Synchronize all processes with a barrier MPI_Barrier(MPI_COMM_WORLD);
```

16.2.3.2 Wtime `double MPI_Wtime(void)` provides a processor dependent time measurement to evaluate the computing time of a portion of program.

16.2.3.3 Comm_split Partitions the group associated with comm into disjoint subgoups , one for each value of color.

```
MPI_Comm new_comm;
int color = 0;
int key = 0;

if (rank < n/2)
    color = 0;
else
    color = 1;

MPI_Comm_split(MPI_COMM_WORLD, color, key, &new_comm);

/*
If `rank` is less than `n/2`, the process belongs to the 'global' communicator, otherwise to the new comm
*/
```

17 Halide

Halide is a framework for fast optimizations in image processing.

17.1 Halide main features

- Halide is designed to make it easier to write efficient code that can be optimized for a variety of hardware architectures, including CPUs, GPUs, and specialized image processing hardware (with a single codebase).
- The Halide approach is to **decouple the algorithm from the schedule** where "schedule" means how the algorithm is computed.
- Halide uses a data-flow model, which means that computations are represented as directed graphs of operations (called "stages") that take inputs and produce outputs. The scheduling across loops needs to be consumer driven and is done manually by the programmer.
- Halide framework supports only computation over regular grids.
- Halide doesn't support feedback pipelines: a feedback pipeline is a pipeline where the output of one stage is feedback as input to a previous stage creating a loop in the data flow graph.
- Halide only supports bounded depth recursion.
- An example of schedule specified by user is: $f(x, y, z)$ performed over three dimensions x ,y and z . While the default schedule loops serially over the three dimensions, it is possible to compute the specified dimension in parallel with the others or to vectorize one dimension, or again split, reorder and tile the loops.
- Halide provides tools for profiling, tracing, and debugging, as well as optimization passes that can improve the performance of the generated code.