1.3.2. Dependency Graphs for Iterative Algorithms

Given: vector function $f(x) : x \to y = f(x), \mathbb{R}^n \to \mathbb{R}^n$

Iteration: start with given vector $x^{(0)}$, $x^{(k+1)} := f(x^{(k)})$
Defines sequence $x^{(k)} \in \mathbb{R}^n, k = 0, 1, 2, \ldots$

Often we are interested in the limit of this sequence with fixed point $\bar{x} = f(\bar{x})$ (if this limit exists)

Example: Newton iteration for solving nonlinear equations:

$$x^{(k+1)} = x^{(k)} - \text{inv}(J(f)(x^{(k)}))f(x^{(k)})$$

$$x^{(k)} \to \bar{x}, \quad f(\bar{x}) = 0$$

with J the Jacobi matrix of the derivatives of f.

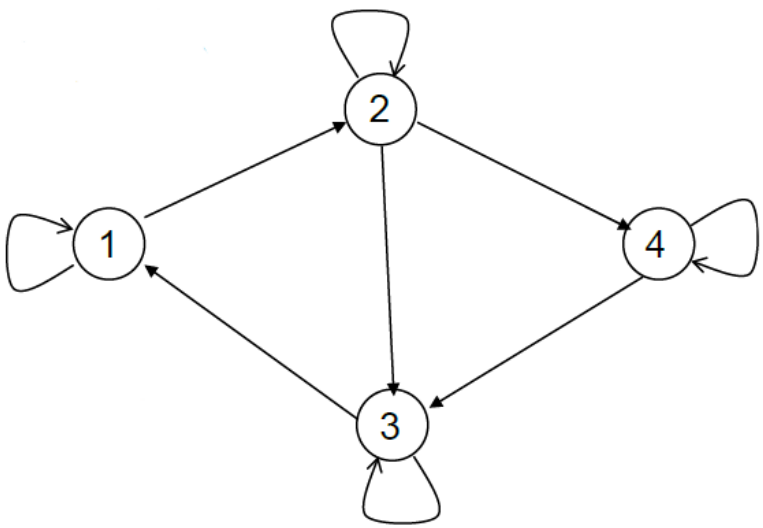
Iteration in Vector Form

$$\begin{pmatrix} x_1^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{pmatrix} = x^{(k+1)} := f(x^{(k)}) = \begin{pmatrix} f_1(x_1^{(k)}, \ldots, x_n^{(k)}) \\ \vdots \\ f_n(x_1^{(k)}, \ldots, x_n^{(k)}) \end{pmatrix}$$

Example:

$$\begin{aligned} x_1^{(k+1)} &= f_1(x_1^{(k)}, x_3^{(k)}) \\ x_2^{(k+1)} &= f_2(x_1^{(k)}, x_2^{(k)}) \\ x_3^{(k+1)} &= f_3(x_2^{(k)}, x_3^{(k)}, x_4^{(k)}) \\ x_4^{(k+1)} &= f_4(x_2^{(k)}, x_4^{(k)}) \end{aligned}$$

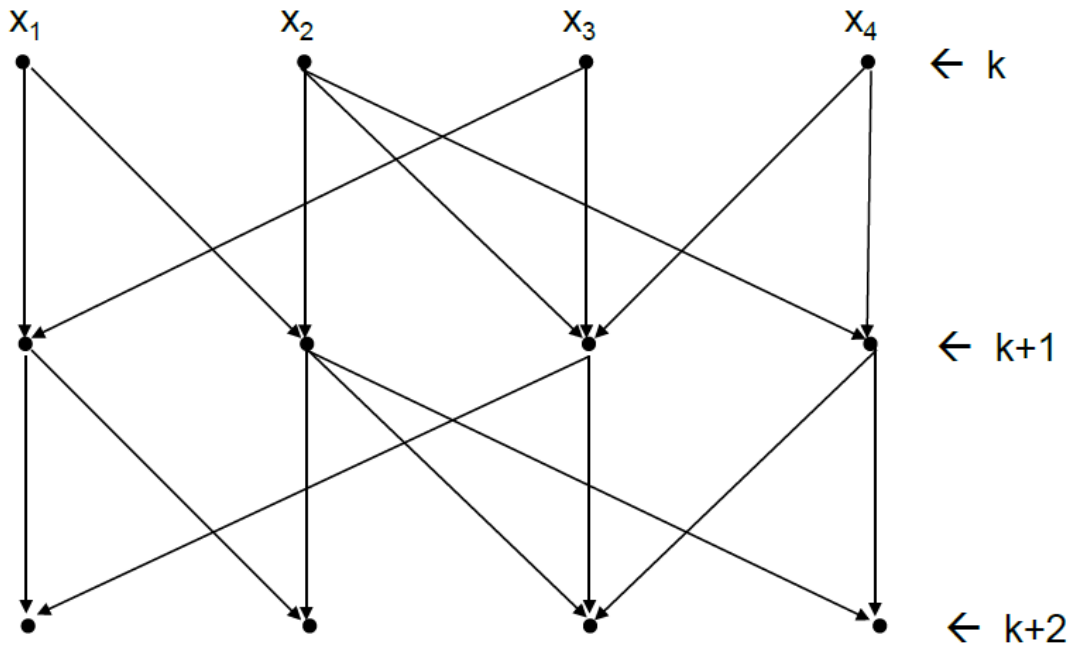
Dependency Graph:



Edge from m to i iff $x_m \in f_i$.

Parallel Computation of the Iteration

Full-step or Jacobi-Iteration



Each iteration step fully parallel.

But slow convergence $x^{(k)} \overset{k\to\infty}{\to} \bar{x}$.

Idea: Accelerating the Convergence

- Always use the newest available information:

$$x_1^{(k+1)} = f_1(x_1^{(k)}, x_3^{(k)})$$
$$x_2^{(k+1)} = f_2(x_1^{(k+1)}, x_2^{(k)})$$
$$x_3^{(k+1)} = f_3(x_2^{(k+1)}, x_3^{(k)}, x_4^{(k)})$$
$$x_4^{(k+1)} = f_4(x_2^{(k+1)}, x_4^{(k)})$$

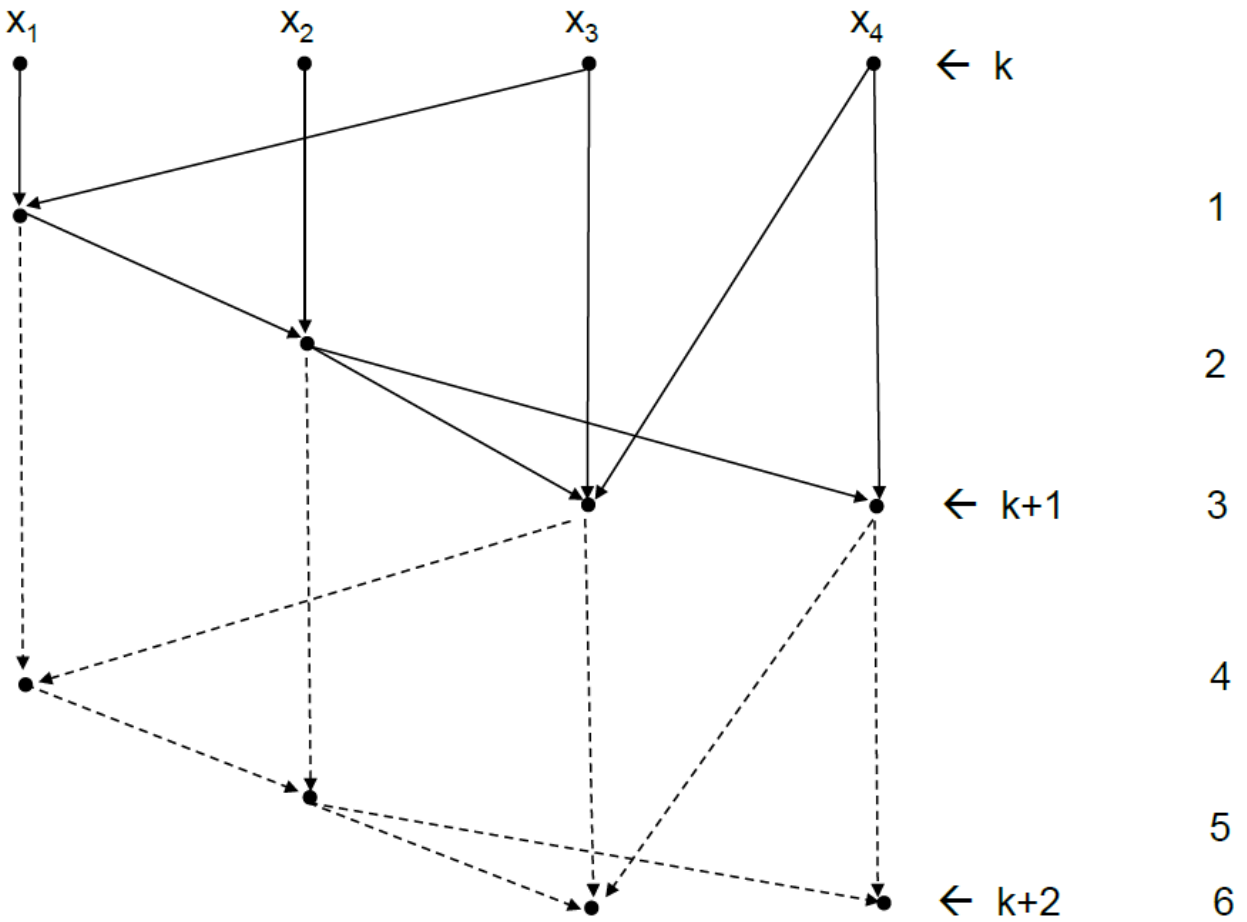
New information!

Ordering of function updates: First f_1, then f_2, …

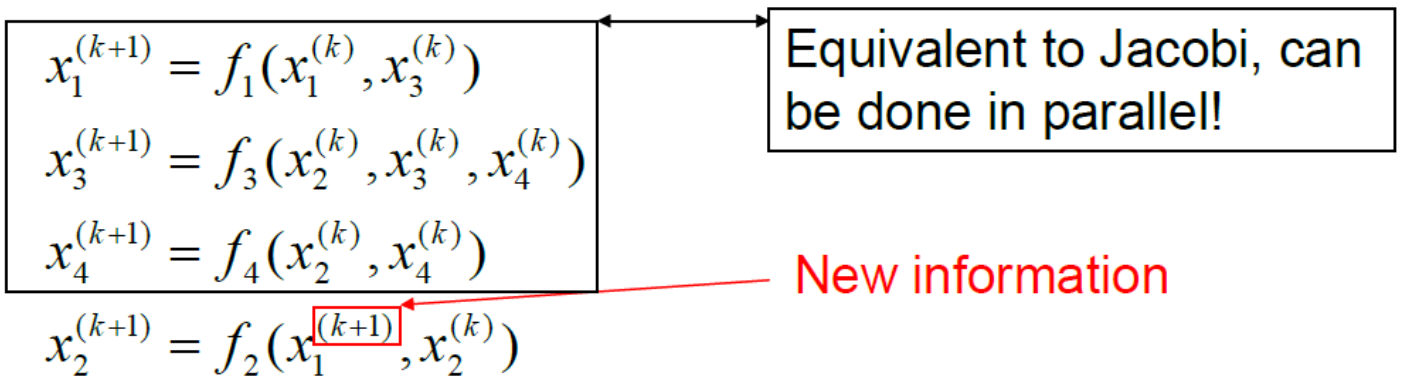
- Leads to faster convergence, but loss of parallelism.
- Updating the next component x_i in one iteration step can be done only after updating all the previous components $x_1, \ldots, x_{i-1}$!
- Single-step or Gauss-Seidel-Iteration.
- In this form the iteration depends on the ordering of the components of the vector x.

Parallel Computation of the Iteration

Single-step or Gauss-Seidel-iteration:



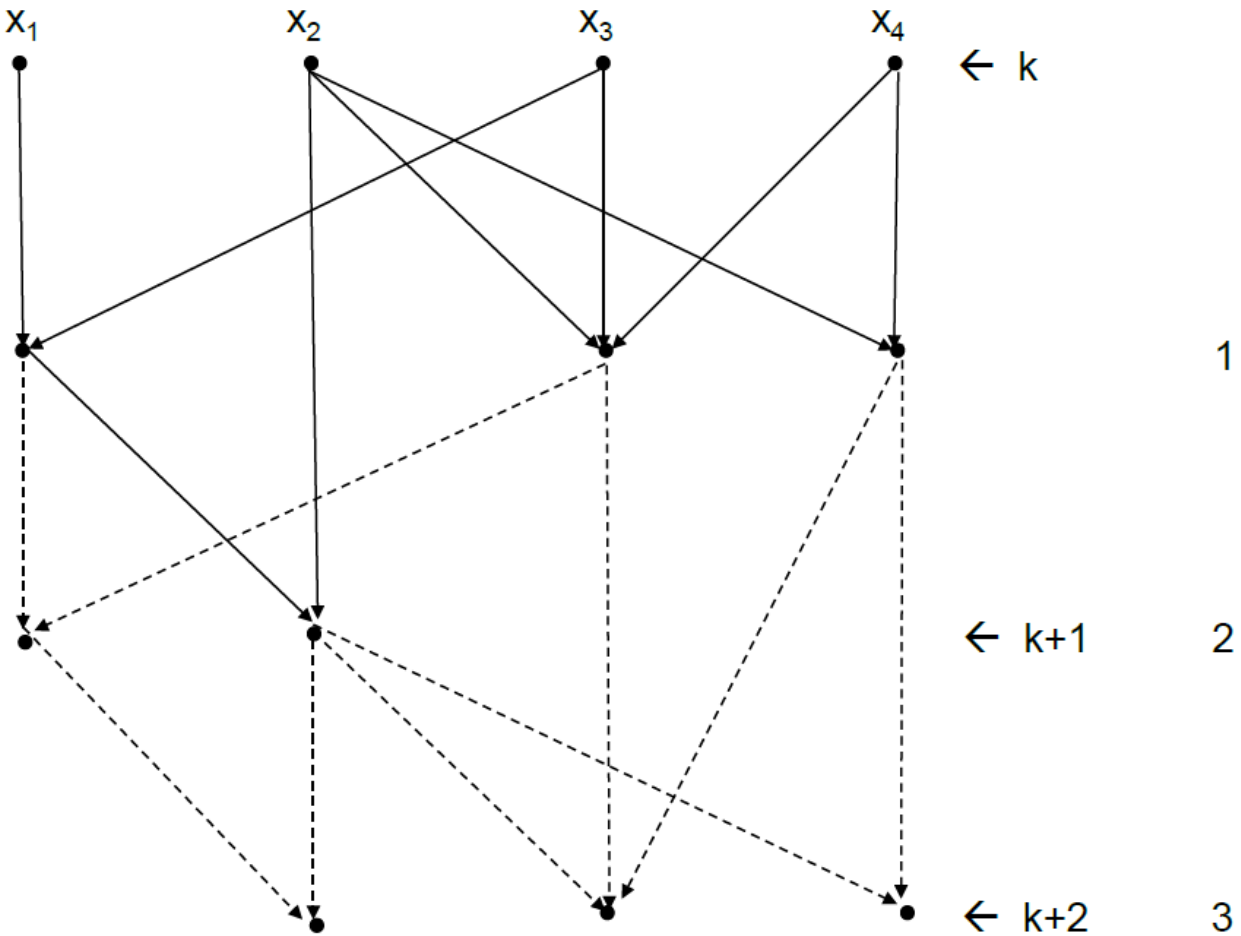
Different ordering by updating x_2 last



By reordering compromise between convergence speed and parallelism

Exact form of f is not important, only the data dependency!

Parallel Computation of the Iteration



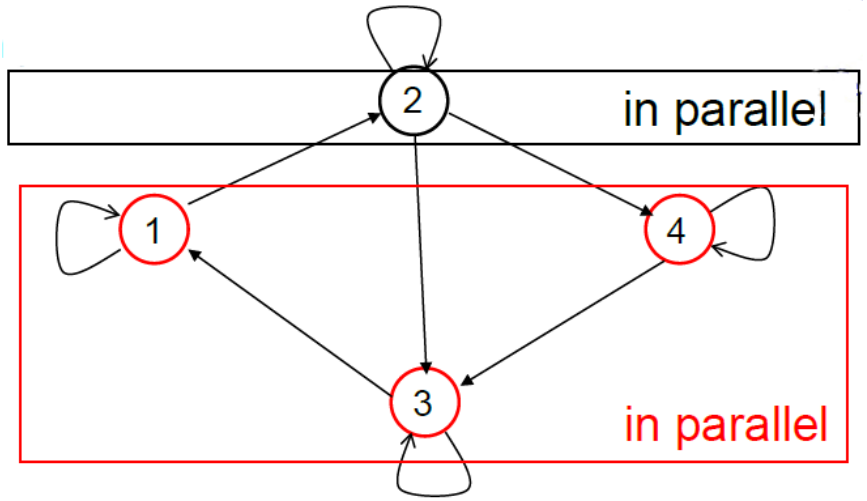
After 2 timesteps: k+1; after 3 timesteps: k+2

Better Parallelism or Slower Convergence?

- Find 'optimal' ordering without loosing parallel efficiency **and** with fast convergence!
- Use coloring algorithms for dependency graph to find efficient orderings:
  - use k colors for vertices of the graph
  - find minimal k but such that there are no cycles connecting vertices of same color! $\rightarrow$ assures ordering such that vertices with the same color are independent and can be updated in parallel!
- First step: Find coloring without cycles
- Second step: Apply ordering for each color based on subgraph of this color. This subgraph is a tree. So we can start the numbering with the leaves and end with the root.

Allowed coloring

No cycles in red: Only path is $4 \to 3 \to 1$.



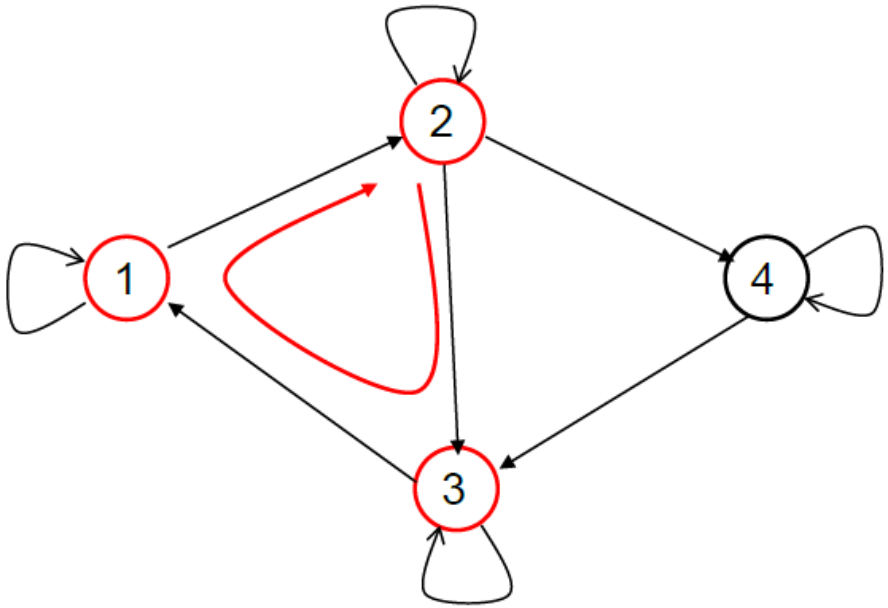
Induced numbering: (x_1, x_3, x_4) and (x_2)

For computing x_2 new we only need the newest red indices
For computing x_1 we need x_3, which is not updated already
For computing x_3 we need the black x_2 and x_4 which is not updated
For computing x_4 we need the black x_2.

Therefore, the red indices can be updated in parallel

Update sequence: $(x_2), (x_1, x_3, x_4), (x_2), (x_1, x_3, x_4), \ldots$

Wrong Coloring



Cycle in red vertices!

There is no suitable ordering in the red vertices that avoids the data dependency in updating the red components.

Theorem 1

Two statements are equivalent:

a) There exists an ordering of the graph such that one Gauss-Seidel-iteration step takes k (time) levels

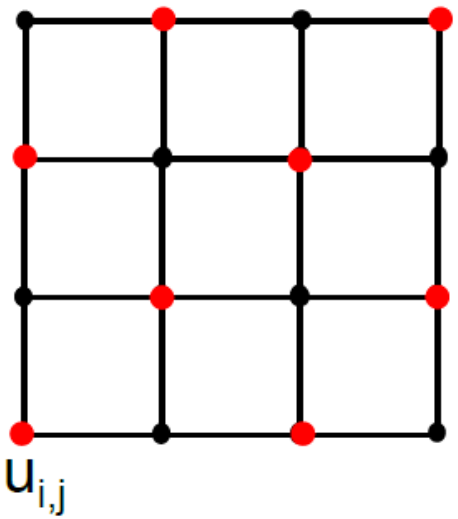
b) There exists a coloring of the graph with k colors such that there is no cycle of edges connecting vertices with the same color.

Proof:

Coloring in subgraph with no cycles
$\rightarrow$ subgraph is tree
$\rightarrow$ ordering from leaves to root
$\rightarrow$ no data dependency in subgraph
$\rightarrow$ computations relative to subgraph can be done in parallel
$\rightarrow$ k steps in parallel because of k colors

Example from PDE

Discretization of rectangular region:



5-point discretization of 2d Laplacian

$$u_{xx} = \frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2}$$

Odd components: red

Even components: black

k=2, two colors

Reorder discretization matrix according to the above red-black ordering.
Red/black components can be updated in parallel.

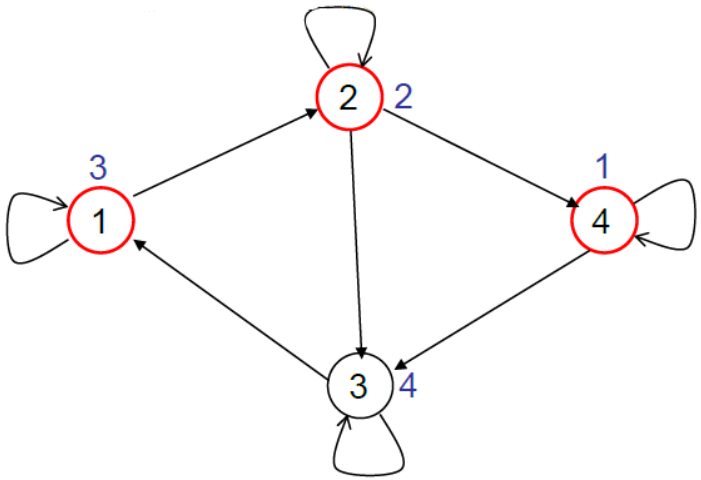
Example from PDE

- Aim:
  - As few colors as possible (#colors $=$ #time steps)
  - Use as much new information as possible (accelerate convergence)
- Total time:
  - Jacobi in parallel: it_{Jac}
  - Coloured Gauss-Seidel in parallel: $it_{\text{GS}} \cdot k$
  - Uncoloured GS: $it_{\text{GS}} \cdot n$
- $k = 2$ is the minimum number of colors for Gauss-Seidel!
- Hence, the number of iterations for GS has to be less then half of the number of iterations of the Jacobi method. Otherwise Jacobi is faster in parallel!

Comparison of Different Colorings

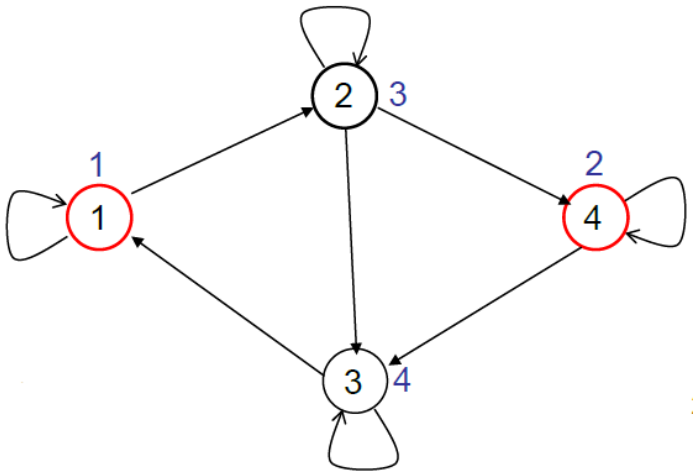
Ordering: $(4, 2, 1), (3)$

Newest information:
For x_3: new x_2 and x_4
For x_1: new $x_3 \rightarrow$ total: 3



Ordering: $(1, 4), (3, 2)$

Newest information:
For x_1: new x_3
For x_4: new x_2
For x_3: new x_4
For x_2: new $x_1 \rightarrow$ total: 4



Comparison of Different k

Assumption:

- 2 processors are available
- all function evaluations are similar
- the distribution on colors is balanced

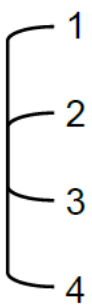
k=1 (Jacobi)	$k = 2$, two colors with $\frac{n}{2}$ indices
Put $\frac{n}{2}$ function evaluations on each proc.	Put $\frac{n}{4}$ from each color on p_1, and $\frac{n}{4}$ on p_2 p_1 and p_2 compute result for $n/4$ of the first color, then $\frac{n}{4}$ for second color, in parallel.
Parallel time: around $\frac{n}{2}$	Parallel time: around $\frac{n}{2}$

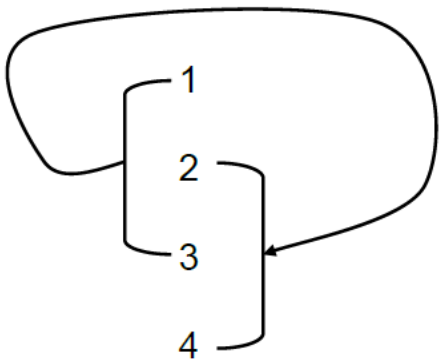
Comparison of Different k—Result

- #colors only important if large #processors is avaliable!
- n processsors for n functions $\rightarrow$
  - Jacobi takes 1 time step
  - Gauss-Seidel with $k = 2$ takes 2 time steps.
- `MATLAB` example `it_gs` (different Gauss-Seidel forms, convergence)

Compromise

Jacobi	Colored Gauss-Seidel	Gauss-Seidel
Fully parallel	k sequential groups that can be internally computed in parallel	Fully sequential
1 time step in parallel	k time steps in parallel	n time steps in parallel





Data Dependency for Solving Triangular System

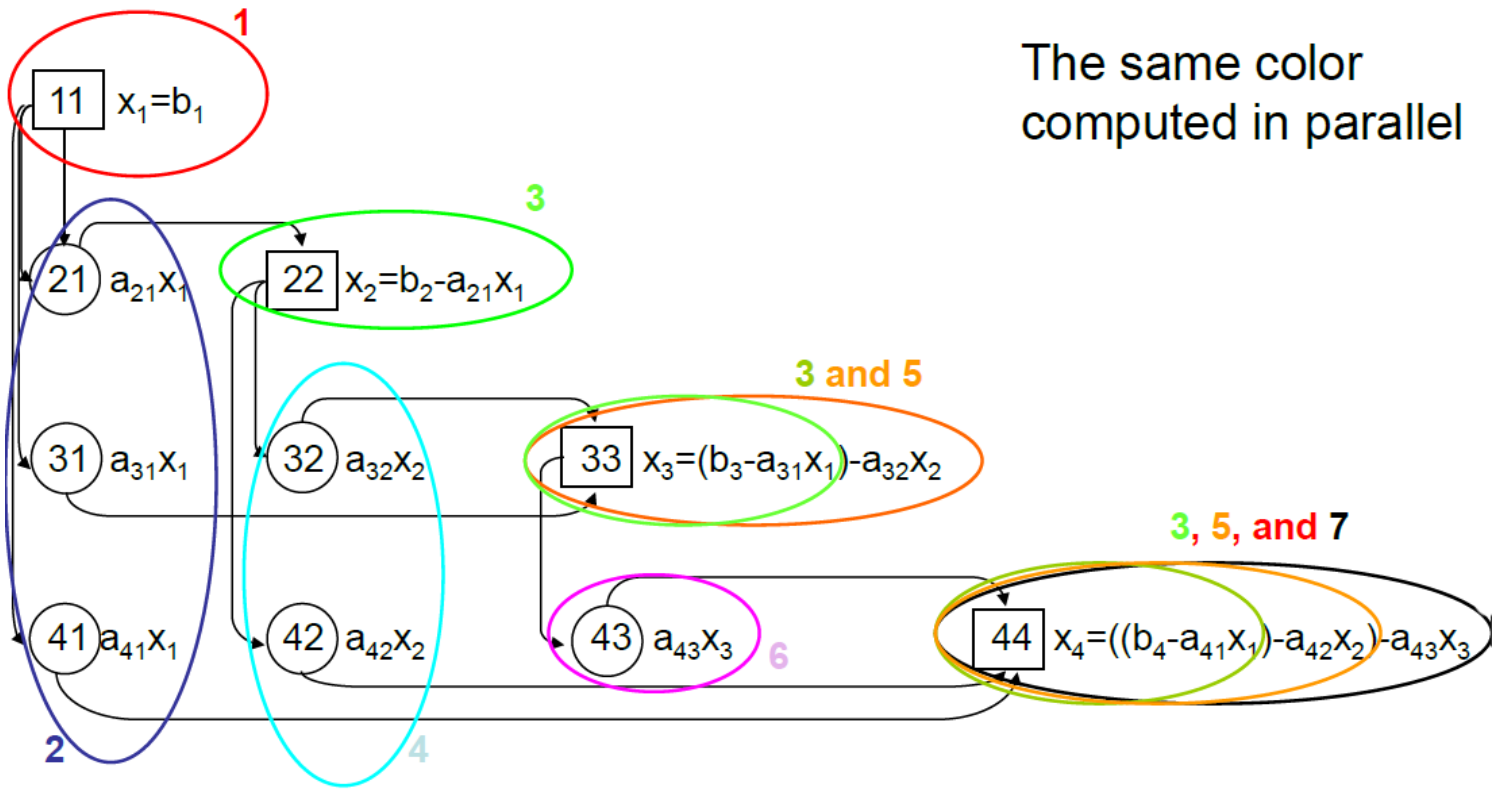
$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n1} & \cdots & a_{n,n-1} & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Strongly sequential:

$$\begin{aligned} x_1 &= b_1/a_{11} \\ x_2 &= (b_2 - a_{21}x_1)/a_{22} \\ x_3 &= (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33} \\ x_4 &= (b_4 - a_{41}x_1 - a_{42}x_2 - a_{43}x_3)/a_{44} \end{aligned}$$

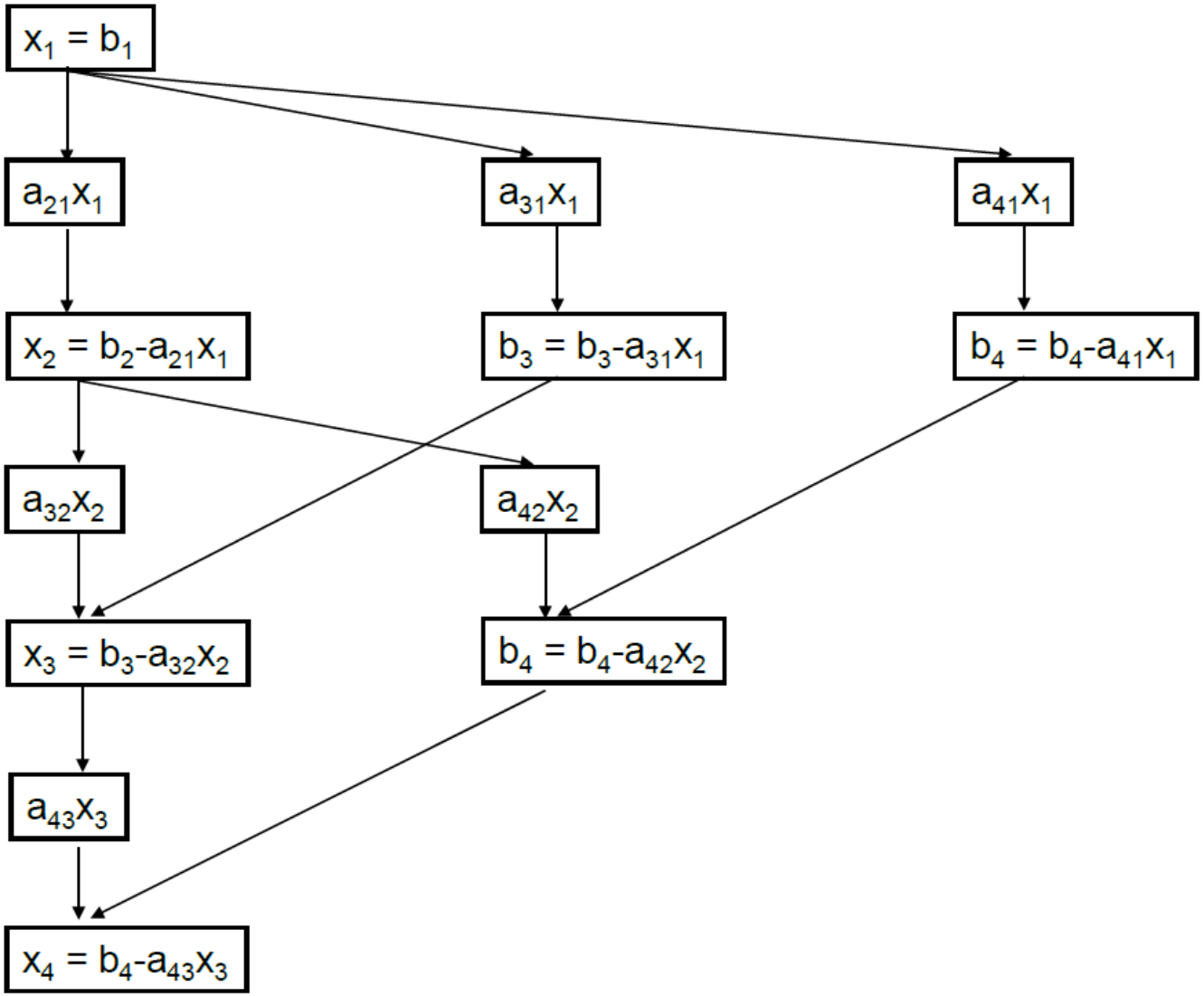
In general: for $k = 1, \ldots, n$: $x_k = \dfrac{b_k - \sum_{j=1}^{k-1} a_{kj}x_j}{a_{kk}}$

Graph for $a_{ij} = 1$



7 time steps in parallel. In general: $2n - 1$ time steps.

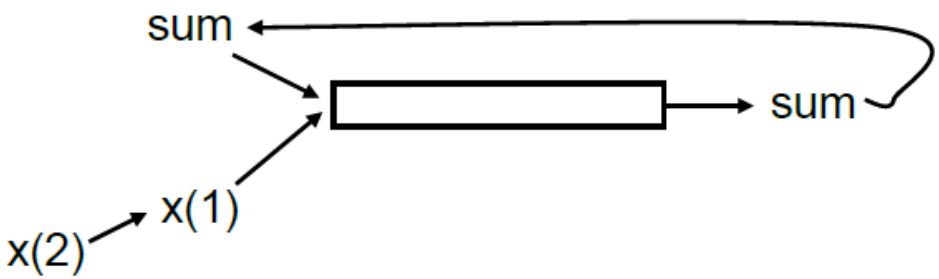
Data Dependency



1.4. Loop Manipulations

- Loops major computational tools in computing.
- For general efficiency important aspects:
  - data dependency inside loops
  - overhead of loop administration
  - work done in each loop
- Example:

```
sum = x(1)*x(1)
For i = 2 : N
    sum = sum + x(i)*x(i)
End
```



- Typical problem: pipeline for updating sum has to wait until partial result for sum has left pipeline

Loop Splitting

```
sum1 = x(1)*x(1)
sum2 = x(2)*x(2)
For i = 2 : n/2
    sum1 = sum1 + x(2*i-1)*x(2*i-1)
    sum2 = sum2 + x(2*i)*x(2*i)
End
sum = sum1 + sum2
```

To avoid empty pipeline!

Loop Fusion

Replace multiple loops by one loop:

```
For i = 1 : n
   a(i) = i
End
For i = 1 : n
    b(i) = i^2
End
```

$\longrightarrow$

```
For i = 1 : n
   a(i) = i
   b(i) = i^2
End
```

Possible advantages: less overhead for loop administration.

Loop Fission

Break one loop into multiple loops:

```
For i = 1 : n
   a(i) = i
   b(i) = i^2
End
```

$\longrightarrow$

```
For i = 1 : n
   a(i) = i
End
For i = 1 : n
   b(i) = i^2
End
```

Possible advantages:

- Better locality
- Independent jobs can be done independently

Loop Peeling

Simplify loop:

```
p = 10
For i = 0 : 10
   y(i) = x(i) + x(p)
   p = i
End
```

$\longrightarrow$

```
y(0) = x(0) + x(10)
For i = 1 : 10
   y(i) = x(i) + x(i-1)
End
```

Possible advantage: simpler loop that is executed very often.

Loop Unrolling (Unwinding)

Write loop with more code but faster execution type.

```
For i = 1 : n
   a(i) = i
End
```

$\longrightarrow$

```
For i = 1 : 5 : n
   a(i) = i
   a(i+1) = i+1
   a(i+2) = i+2
   a(i+3) = i+3
   a(i+4) = i+4
End
```

Possible advantages:

- Less loop overhead ('end of loop' test)
- Statements inside loop can be done in parallel (if independent)

Further Manipulations

- Loop Interchange: change inner and outer loops (compare (ijk) forms).

- Loop Inversion: replace `while`-statement by `do`-while wrapped in an `if`-condition.

  Advantage: Reducing number of jumps by two (in last iteration).

- Loop Reversal: reverse the direction of loop:

  ```
  For i = 1 : 1 : n
  ```

  is replaced by

  ```
  For i = n : -1 : 1
  ```

Loop Unswitching

Moves a conditional inside the loop outside by duplicating the loop's body.

```
For i = 1 : n
  x(i) = x(i) + y(i)
  if (w) then y(i) = 0
End
```

$\longrightarrow$

```
If (w) then
   For i = 1 : n
     x(i) = x(i) + y(i)
     y(i) = 0
   End
Else
    For i = 1 : n
      x(i) = x(i) + y(i)
    End
End
```

Possible advantages:

- Both loops are easier to parallelize
- Less operations inside loops

Loop Tiling or Blocking

Additional loop by substructuring or blocking:

```
For i = 1 : n
  For j = 1 : m
    c(i) = c(i)+a(i,j)*b(j)
  End
End
```

$\rightarrow$

```
For i = 1 :2 : n
  For j = 1 : 2 : m
    For ii = i : min(i+1,n)
      For jj = j : min(j+1,m)
        c(ii) = c(ii) + a(ii,jj)*b(jj)
      End
    End
  End
End
```

Possible advantages:

- Loop uses block data
- stays in the cache

Data Dependencies Inside Loop

Example:

```
For i = 1 : n
   temp = i
   a(i) = 1.0 / temp
End
```

Problem with parallelization: variable temp appears in each loop with different value.

Improvement:

```
For i = 1 : n
   temp(i) = i
   a(i) = 1.0 / temp(i)
End
```

Further Example

```
For i = 1 : n
   diag(i) = (1.0 / h(i)) + (1.0 / h(i+1))
   offdiag(i) = - (1.0 / h(i+1))
End
```

Reduce the number of superfluous divisions!

```
For i = 1 : n
   dxo = 1.0 / h(i)
   dxi = 1.0 / h(i+1)
   diag(i) = dxo + dxi
   offdiag(i) = - dxi
End
```

More Complicated Code with Less Divisions

```
dxi = 1.0 / h(1)
For i = 1 : n
   dxo = dxi
   dxi = 1.0 / h(i+1)
   diag(i) = dxo + dxi
   offdiag(i) = - dxi
End
```

Introducing a lot of loop-carried dependencies that hamper parallelization.

Explicit Parallelization

```
For ip = 0 : P-1
   dxi = 1.0 / h(ip*(n/P) + 1)
   For i = ip*(n/P) + 1  : ip*(n/P) + n/P
      dxo = dxi
      dxi = 1.0 / h(i+1)
      diag(i) = dxo + dxi
      offdiag(i) = - dxi
   End
End
```

Less division, explicitly parallel by tiling/blocking.

Parallel Numerics, WT 2013/2014

2 Elementary Linear Algebra Problems

Contents

2.1. BLAS: Basic Linear Algebra Subroutines

- First published 1979
- Library of basic linear algebra operations
- Organized in levels according to complexity:

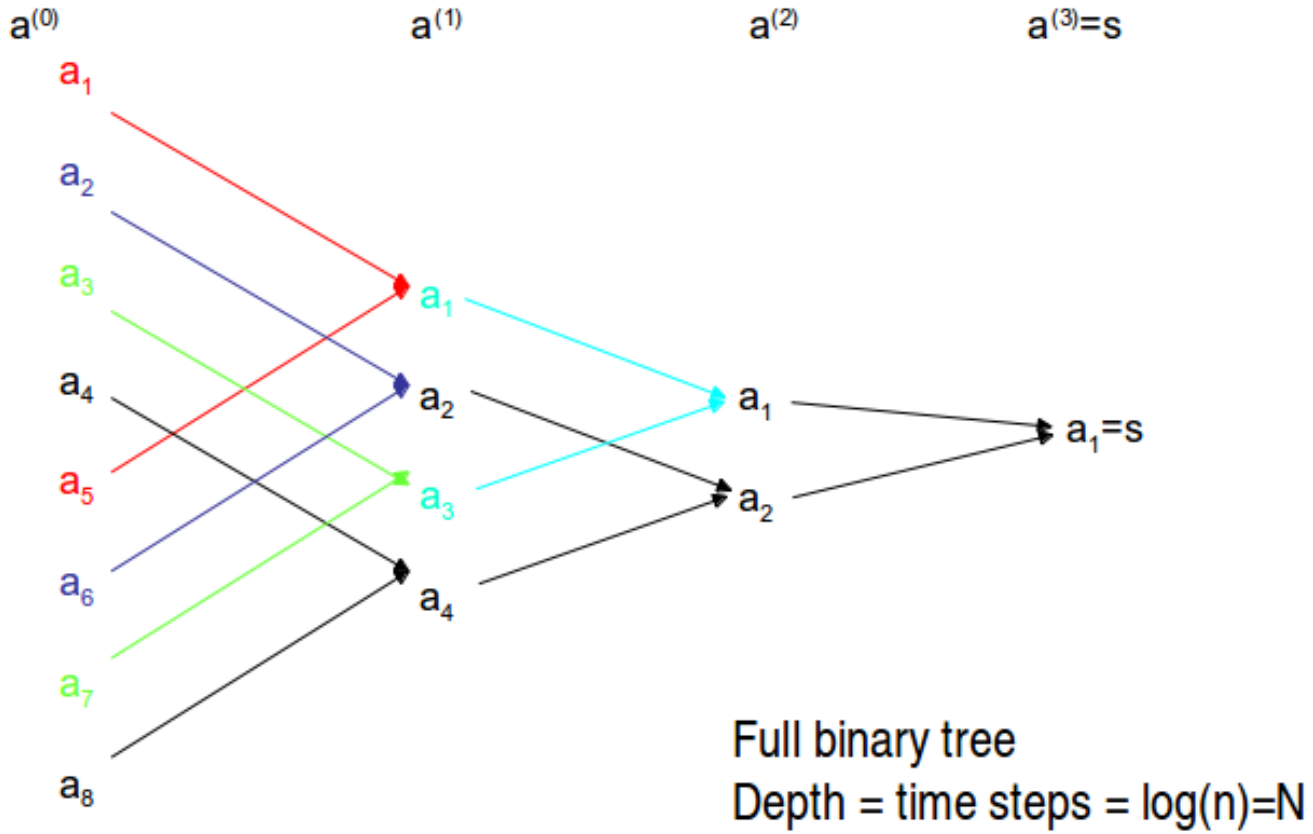
BLAS level	type of operation	complexity
BLAS-1	vector-vector	$\mathcal{O}(n)$
BLAS-2	matrix-vector	$\mathcal{O}(n^2)$
BLAS-3	matrix-matrix	$\mathcal{O}(n^3)$

- There are machine-specific optimized BLAS libraries
- Basis of other libraries, e.g., LAPACK (Linear Algebra Package: for solving linear equations, least squares problems, QR-decomposition, eigenvalues, singular values...)

Computation of Sum in Parallel

Sum of vector components: $s = \sum_{j=1}^{n} a_j$.

Computation by fan-in process:



Vector update in the fan-in process with $n = 2^N$

$$a^{(k)} = \begin{pmatrix} a_1^{(k)} \\ \vdots \\ \vdots \\ a_{2^{N-k}}^{(k)} \end{pmatrix} = \begin{pmatrix} a_1^{(k-1)} \\ \vdots \\ \vdots \\ a_{2^{N-k}}^{(k-1)} \end{pmatrix} + \begin{pmatrix} a_{2^{N-k}+1}^{(k-1)} \\ \vdots \\ \vdots \\ a_{2^{N-k+1}}^{(k-1)} \end{pmatrix}$$

$$a_1 + \ldots + a_8 = [(a_1 + a_5) + (a_3 + a_7)] + [(a_2 + a_6) + (a_4 + a_8)]$$

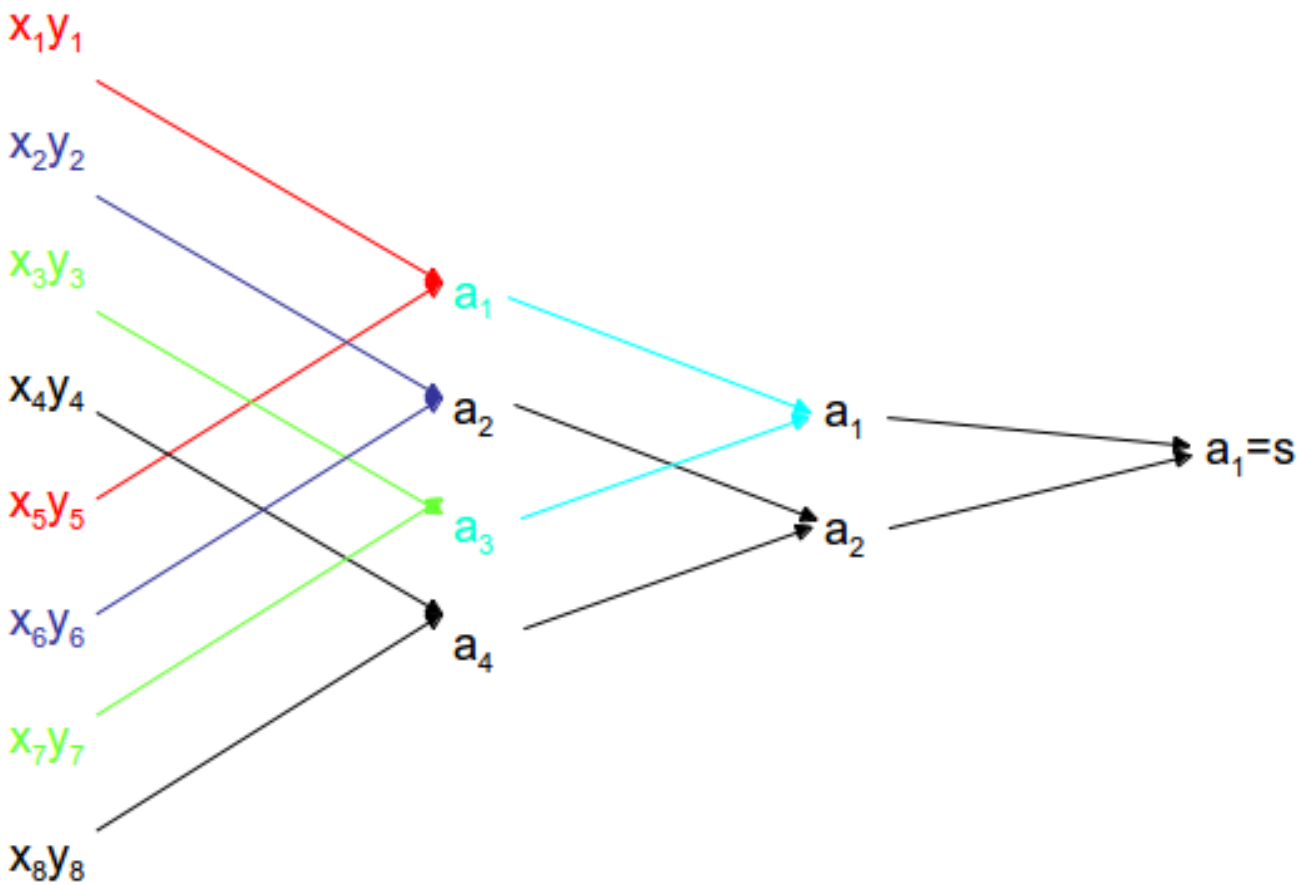
```
for(k=1;k<=N;k++)
   for(j=1;j<=2^(N-k);j++)
      a_j = a_j + a_(j+2^(N-k))
   end
end
```

$\log(n) = N$ time steps in parallel for vector of length n

Level-1 BLAS

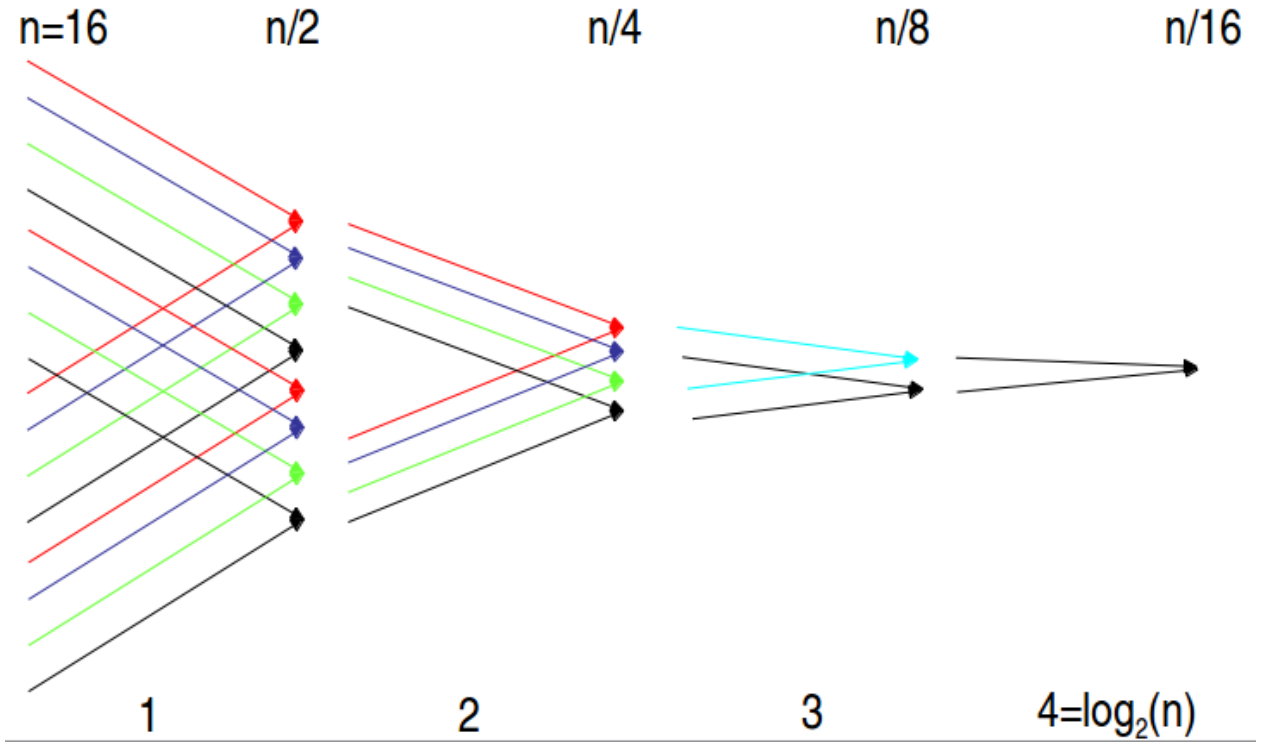
BLAS routines with $\mathcal{O}(n)$ problems (vectors only, $x, y \in \mathbb{R}^n$).

First example: DOT-product by fan-in: $s = x^T y = \sum_{i=1}^{n} x_j y_j$



DOT-Product in parallel

- Time steps in parallel of the DOT-product cannot be better than $\log(n)$.
- Every computation involving fan-in will take $\log(n)$ time steps in parallel.



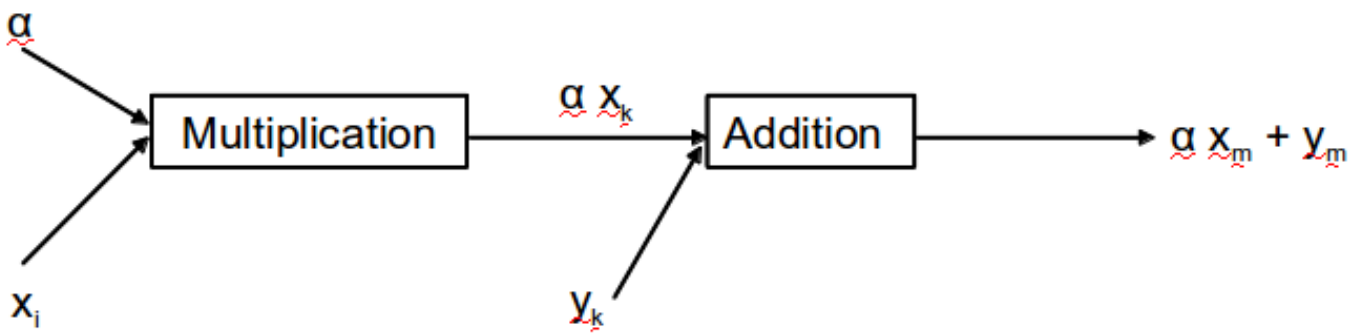
Level-1 BLAS: SAXPY

BLAS-Notation:

- S single precision (D for double, C for complex)
- A α scalar
- X vector
- P plus operation
- Y vector

SAXPY: $y = \alpha x + y$

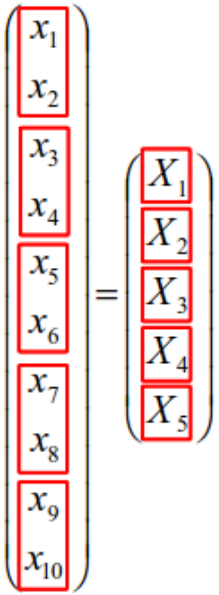
Vectorization of SAXPY ($\alpha x + y$) by pipelining:



SAXPY Parallelization by Partitioning

$$\{1, 2, \ldots, n\} = \langle 1, n \rangle = I_1 \cup I_2 \cup \ldots \cup I_R$$

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} X_1 \\ \vdots \\ X_R \end{pmatrix}, \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} Y_1 \\ \vdots \\ Y_R \end{pmatrix}$$



- Using short vectors X_j and Y_j of length $\frac{n}{R}$.
- Each processor P_j gets partial vector X_j and Y_j and computes $Y_j = \alpha X_j + Y_j,\ j = 1, 2, \ldots, R$.
- Result: SAXPY very good vectorizable and parallelizable.

Further Level-1 BLAS Routines

- SCOPY: $y = x$ or $y \leftarrow x$ (compare SAXPY)
- DOT-product $x^T y$: $\sum_i x_i y_i$
- norm: $\|x\|_2 = \sqrt{\sum_{j=1}^{n} x_j^2} = \sqrt{x^T x}$ (compare DOT-product)