Example for Race Condition

```
int called=0, *dat;
void not_thread_safe(n) {
    if(called==0) {
        called=1;
        dat=(int *)malloc(sizeof(int)*n);
    }
}
```

If different threads read variable `called` at the same time
$\rightarrow$ for all these threads the if-condition is satisfied and they all will allocate new memory for pointer `dat`!

Synchronisation via Semaphores

- A semaphore is a signal to coordinate processes, based on two operations:
  - `lock : P`
  - `unlock : V`
- Semaphore variable can have two possible values: locked or unlocked

Synchronisation via Semaphores

- A semaphore is a signal to coordinate processes, based on two operations:
  - `lock : P`
  - `unlock : V`
- Semaphore variable can have two possible values: locked or unlocked
- When semaphore is unlocked by the authorized process, other processes are free to access and lock it.

Synchronisation via Semaphores

- A semaphore is a signal to coordinate processes, based on two operations:
  - `lock : P`
  - `unlock : V`
- Semaphore variable can have two possible values: locked or unlocked
- When semaphore is unlocked by the authorized process, other processes are free to access and lock it.
- If process tries to lock an unlocked semaphore then semaphore gets locked until the process unlocks it.
- If process tries to lock an already locked semaphore it has to wait until the previous process unlocks the semaphore.

Synchronisation via Semaphores

- A semaphore is a signal to coordinate processes, based on two operations:
  - `lock : P`
  - `unlock : V`
- Semaphore variable can have two possible values: locked or unlocked
- When semaphore is unlocked by the authorized process, other processes are free to access and lock it.
- If process tries to lock an unlocked semaphore then semaphore gets locked until the process unlocks it.
- If process tries to lock an already locked semaphore it has to wait until the previous process unlocks the semaphore.
- In OpenMP: atomic instead of critical section has less overhead semaphores allow more freedom than critical section

Example for Semaphores

```
Semaphore s
parallel do i=1,n
    s.P             // lock
    x=x+1
    s.V             // unlock
end parallel do
```

Example for Semaphores

```
Semaphore s
parallel do i=1,n
    s.P             // lock
    x=x+1
    s.V             // unlock
end parallel do
```

Without lock, one process could try to read `x` while another one is incrementing `x`. This would lead to a classical race condition and a nondeterministic behaviour of the code.

The steps in the loop are:

```
Read x
Increment x
Write x
Increment loop
```

Which value will be read by the process depends on step in loop.

Example for Race Condition

```
c$omp parallel sections
      A = B + C
c$omp section
      B = A + C
c$omp section
      C = B + A
c$omp end parallel sections
```

Example for Race Condition

```
c$omp parallel sections
      A = B + C
c$omp section
      B = A + C
c$omp section
      C = B + A
c$omp end parallel sections
```

The result varies unpredictably depending on the actual order at runtime.

Wrong answers without warning!

Avoid race condition by synchronization!

Example for Dead Lock

```
call omp_init_lock(var)
c$omp parallel sections
c$omp section
    call omp_set_lock(var)
    ... (work)
    if (ival > tol) then
        call omp_unset_lock(var)
    else
        call error(ival)   // no unlocking here!
    end if
c$omp section
    call omp_set_lock(var) // var potentially never unlocked!
    ... (work2 depends on work)
    call omp_unset_lock(var)
c$omp end parallel sections
```

Section 1 first: blocks sec. 2, might never unlock and reach work2.
Section 2 first: blocks sec. 1 and work, so never reaches work2.

Critical Sections in OpenMP

A critical section is a section of code in which only one subcomputation is active

In our previous example only one statement

```
x = x+1
```

can be active between `lock(s)` and `unlock(s)`.

Therefore only one processor will be in a critical section at one time!

Critical Sections in OpenMP

Explicit synchronization in OpenMP via the

```
c$omp critical
```

directive, which acts like a guard (or mutex = mutual exclusion) at the beginning of a critical section (lock/unlock mutex).

The end is marked by `c$omp end critical`

```
c$omp parallel do
do i = 1, n
  c$omp critical
  x = x + 1
  c$omp end critical
end do
```

Barrier in OpenMP

- A barrier is a point in a parallel code where all processors stop in order to synchronize.
- All processors wait there until all others have arrived.
- Once they have all arrived, they are all released to continue in the code.

Barrier in OpenMP

- A barrier is a point in a parallel code where all processors stop in order to synchronize.
- All processors wait there until all others have arrived.
- Once they have all arrived, they are all released to continue in the code.

Typical example:

```
parallel do i=1,n
    "local part of comput., each processor independent in parallel"
    call barrier()
    "all using results of other processors from previous part"
end parallel do
```

- After all reached barrier one can be sure that all other processors have computed their partial results and it is safe to use these results.

Loop Tiling in OpenMP

- OpenMP directives provide instructions to the compiler, but the directives are not translated into code.
- Simple case: parallel do instruction implies that the loop following it is to be subdivided into as many separate threads as are avaliable.

```
c$omp parallel do
    do i = 2, n-1
        newx(i) = .5*( x(i-1) + x(i+1) - dx2*x(i) )
    enddo
c$omp end parallel do //Synchronizing barrier, can be omitted
c$omp parallel do
    do i = 2, n-1
        x(i) = newx(i)
    enddo
c$omp end parallel do //can be omitted
```

Private Variables in OpenMP

- `c$omp parallel do private (a,b,i,j,x,y)`: parallel statement includes a list of private variables that are local in loop.
- This is similar to adding an extra array index to the variable to make it different for each iteration of the loop.
- Private variables can be safely used if there is no write statement involved with the private variables.

```
do i = 1, 100
    if ( parity(i) .EQ. 1 )
        temp = i  // temp can not be privatized safely
    endif
    if ( parity(i) .EQ. 0 )
        temp = 0
    endif
enddo
```

Examples for Use of Private Variables

```
c$omp parallel do private(TEMP,I)
do I = 1, 100
    TEMP = I
    A(I) = 1.0 / TEMP
enddo
c$omp end parallel do

c$omp parallel do private(k)
do k = 0, P-1
    localsum(k) = 0.0
    do i = 1 + k*(N/P),  (k+1)*(N/P)
        localsum(k) = localsum(k) + x(i)
    enddo
enddo
c$omp end parallel do
```

Decomposition Choices in OpenMP

- `c$omp parallel do schedule(static)`
  indicates a static block decomposition of the iteration space
  blocksize = iteration space / number of processors

Decomposition Choices in OpenMP

- `c$omp parallel do schedule(static)`
  indicates a static block decomposition of the iteration space
  blocksize = iteration space / number of processors
- `c$omp parallel do schedule(static, chunksiz)`
  indicates a static block decomposition of the iteration space
  where the blocksize is given by `chunksiz`

Decomposition Choices in OpenMP

- `c$omp parallel do schedule(static)`
  indicates a static block decomposition of the iteration space
  blocksize = iteration space / number of processors
- `c$omp parallel do schedule(static, chunksiz)`
  indicates a static block decomposition of the iteration space where the blocksize is given by `chunksiz`
- `c$omp parallel do schedule(dynamic, chunksiz)`
  indicates that chunks of loop indices should be allocated to processors on a first-available basis

Decomposition Choices in OpenMP

- `c$omp parallel do schedule(static)`
  indicates a static block decomposition of the iteration space
  blocksize = iteration space / number of processors
- `c$omp parallel do schedule(static, chunksiz)`
  indicates a static block decomposition of the iteration space where the blocksize is given by `chunksiz`
- `c$omp parallel do schedule(dynamic, chunksiz)`
  indicates that chunks of loop indices should be allocated to processors on a first-available basis
- `c$omp parallel do schedule(guided, chunksiz)`
  starts with a large size of chunks each processor takes, and then exponentially reduces the chunk size down to `chunksiz`.

Reductions in OpenMP

```
#pragma omp parallel for reduction(+:x,y)
for (i = 0; i < n; i++) {
    x += b[i];
    y = sum(y,c[i]);
}
```

Syntax: `reduction(operator:list)`

reduction is not a subroutine!

Compare `MPI_Allreduce`

How Many Processes?

```
call omp_set_num_threads(inthreads)
```

Designates the number of processes via inthreads.

The number of threads can be determined with

```
nThreads = omp_get_num_threads( )
```

OpenMP Compilers

- GNU gcc, compile with `-fopenmp`
- IBM XL C/C++/Fortran, Sun C/C++/Fortran, compile with `-xopenmp`
- Intel C/C++/Fortran, compile with `-Qopenmp` (Windows) or `-openmp` on Linux

Test code:

```
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    printf("Hello from thread %d, nthreads %d\n",
            omp_get_thread_num(), omp_get_num_threads());
}
```

1.2.14. Important Factors of Communication

- Cost of communication:
  - Inter-task communication implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communication frequently requires some type of synchronization between tasks, which can result in tasks spending time ”waiting” instead of doing work.
  - Competing communication traffic can saturate available network bandwidth, further aggravating performance problems.
  - Hiding communication by overlap between computation and communication.

Latency vs. Bandwidth

- Latency:
  - Time to send a minimal (0 byte) message from point A to point B.
  - Commonly expressed in microseconds.

Latency vs. Bandwidth

- Latency:
  - Time to send a minimal (0 byte) message from point A to point B.
  - Commonly expressed in microseconds.
- Bandwidth:
  - Amount of data that can be communicated per unit of time.
  - Commonly expressed in megabytes/sec or gigabytes/sec.
  - Sending many small messages can cause latency to dominate communication overheads.
  - Often more efficient to package small messages into larger message, thus increasing the effective communications bandwidth.

Visibility of Communication

- With Message Passing Model, communication is
  - explicit
  - generally visible
  - and under the control of the programmer.
- With shared memory model, communication is
  - not transparent to programmer.
  - Programmer may not even be able to know how and when inter-task communications are being accomplished.

1.3. Numerical Problems

- Operations for vectors $x, y \in \mathbb{R}^n$
- Dot or inner product

$$x^T y = (x_1, \ldots, x_n) \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \sum_{i=1}^{n} x_i y_i$$

1.4. Numerical Problems

- Operations for vectors $x, y \in \mathbb{R}^n$
- Dot or inner product

$$x^T y = (x_1, \ldots, x_n) \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \sum_{i=1}^{n} x_i y_i$$

- Sum of vectors

$$x + \alpha y = \begin{pmatrix} x_1 + \alpha y_1 \\ \vdots \\ x_n + \alpha y_n \end{pmatrix}$$

1.5. Numerical Problems

- Operations for vectors $x, y \in \mathbb{R}^n$
- Dot or inner product

$$x^T y = (x_1, \ldots, x_n) \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \sum_{i=1}^{n} x_i y_i$$

- Sum of vectors

$$x + \alpha y = \begin{pmatrix} x_1 + \alpha y_1 \\ \vdots \\ x_n + \alpha y_n \end{pmatrix}$$

- Outer product

$$xy^T = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} (y_1, \ldots, y_m) = \begin{pmatrix} x_1 y_1 & \cdots & x_1 y_m \\ \vdots & \ddots & \vdots \\ x_n y_1 & \cdots & x_n y_m \end{pmatrix}$$

Matrix Product

$$\underbrace{\begin{pmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nk} \end{pmatrix}}_{A \in \mathbb{R}^{n\times k}} \underbrace{\begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{k1} & \cdots & b_{km} \end{pmatrix}}_{B \in \mathbb{R}^{k\times m}} = \underbrace{\begin{pmatrix} c_{11} & \cdots & c_{1m} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nm} \end{pmatrix}}_{C=A\cdot B \in \mathbb{R}^{n\times m}}$$

$$c_{ij} = \sum_{r=1}^{k} a_{ir} b_{rj}, \qquad i = 1, \ldots, n; \qquad j = 1, \ldots, m$$

$$c_{ij} = A(i,:)B(:,j) = \sum_{r=1}^{k} A(i,k)B(k,j)$$

Tensors: a_{ijk}, b_{rstj}. Product via summation of common index i.

Alternative Matrix Products

- Hadamard product:

$$A \in \mathbb{R}^{n\times k}, \quad B \in \mathbb{R}^{n\times k}, \quad C = A \circ B \in \mathbb{R}^{n\times k}$$

$$c_{ij} = a_{ij} \cdot b_{ij}, \qquad \text{for } i = 1,\ldots,n;\ j = 1,\ldots,k$$

Alternative Matrix Products

- Hadamard product:

$$A \in \mathbb{R}^{n\times k}, \quad B \in \mathbb{R}^{n\times k}, \quad C = A \circ B \in \mathbb{R}^{n\times k}$$

$$c_{ij} = a_{ij} \cdot b_{ij}, \qquad \text{for } i = 1,\ldots,n;\ j = 1,\ldots,k$$

- Kronecker (Tensor) product:

$$A \in \mathbb{R}^{n\times m}, \quad B \in \mathbb{R}^{s\times t}, \quad C = A \otimes B \in \mathbb{R}^{ns\times kt}$$

$$C = \begin{pmatrix} a_{11}B & \cdots & a_{1m}B \\ \vdots & \ddots & \vdots \\ a_{n1}B & \cdots & a_{nm}B \end{pmatrix}$$

$$(a_1 \;\; a_2) \otimes (b_1 \;\; b_2 \;\; b_3) = (a_1b_1 \;\; a_1b_2 \;\; a_1b_3 \mid a_2b_1 \;\; a_2b_2 \;\; a_2b_3)$$

Solving Linear Equations

- Triangular system:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ddots &= \vdots \\ a_{nn}x_n &= b_n \end{aligned}$$

Solving Linear Equations

- Triangular system:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ 0 & \cdots & 0 & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\ \ddots &= \vdots \\ a_{nn}x_n &= b_n \end{aligned}$$

- Solution:

$$x_n = \frac{b_n}{a_{nn}} \Rightarrow a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = b_{n-1}$$

$$\Rightarrow x_{n-1} = \frac{b_{n-1} - a_{n-1,n}x_n}{a_{n-1,n-1}}$$

- Algorithm: for $j = n, \ldots, 1$: $x_j = \frac{b_j - \sum_{k=j+1}^{n} a_{jk}x_k}{a_{jj}}$

Eigenvalues

- Eigenvalue λ with eigenvector $u \neq 0$: $Au = \lambda u$
- For an $n \times n$-symmetric (Hermitian) matrix A there exist n pairwise orthogonal eigenvectors u_j with real eigenvalues $\lambda_j,\ j = 1, \ldots, n$.

$$Au_j = \lambda_j u_j, \quad j = 1, \ldots, n$$

$$U := (u_1, \ldots, u_n), \qquad \Lambda := \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$$

$$AU = U\Lambda \;\Rightarrow\; U^H AU = \Lambda$$

- Matrix A describes linear mapping.
- Eigenvectors are fixed points of this mapping.
- Eigenvector basis describes optimal coordinate system for this mapping!
- More general: SVD and norms.

1.6. Data Dependency Graphs

General problem: Data dependency

Compute:

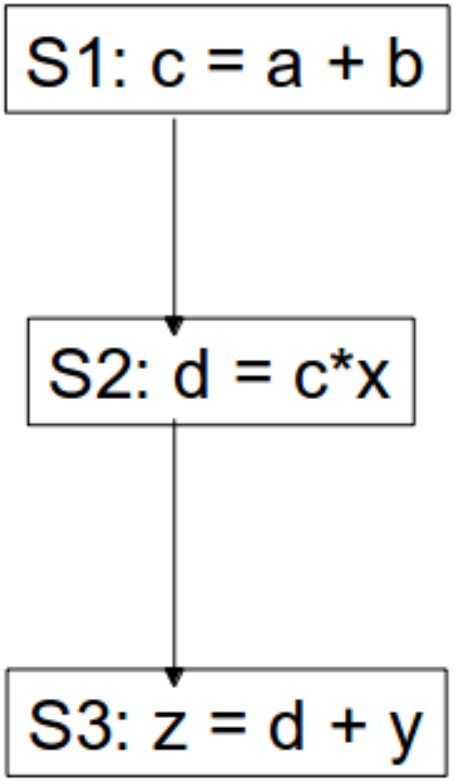
1. $c = a + b$
2. $z = c \cdot x + y$

Obviously, 2. can be computed only after 1.

Example: Self–referential loop:

```
for(i=1;i<=n;i++)
    a[i] = b[i] * a[i-1] + c[i]
```

Graphical Representation of Data Dependency

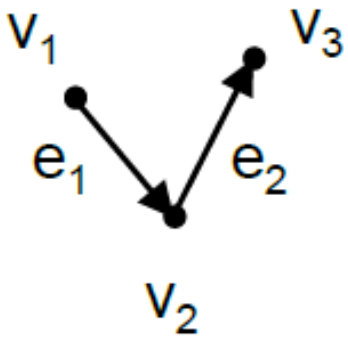


This will be further analysed in the following chapter.

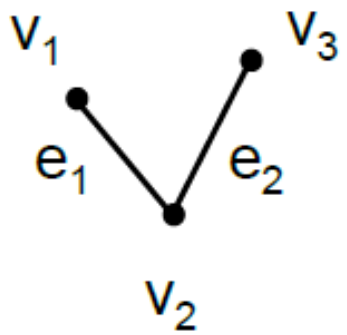
1.6.1. Graphs and Computations

Graph: $G = (V, E)$ with vertices v_i in V and edges $e_{ik} = (v_i, v_k)$ in E.

Directed graph: $e_{ik} \neq e_{ki}$



Undirected graph: $e_{ik} = e_{ki}$



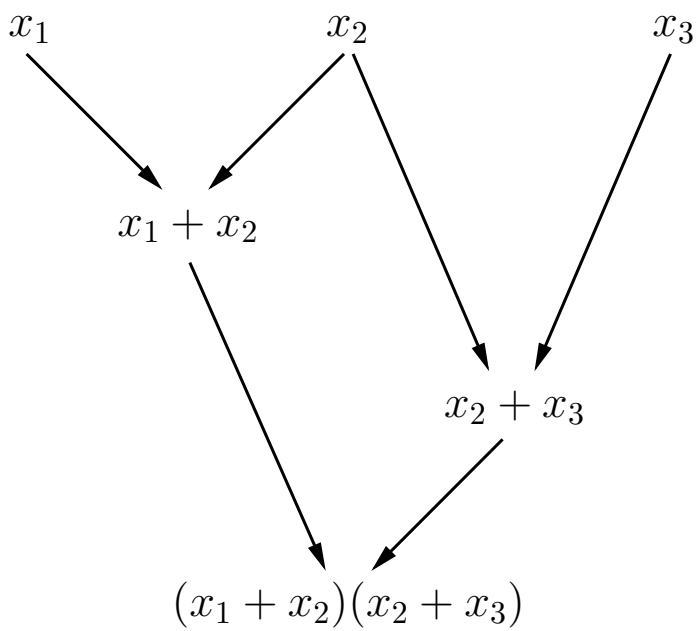
Example: Tree, star, ...
Modifying the graph by generating time levels.

Example: Nonlinear Evaluation

Computation of $(x_1 + x_2)(x_2 + x_3) = x_1x_2 + x_2^2 + x_1x_3 + x_2x_3$

Representation of the computational flow by a graph:

Sequential computation



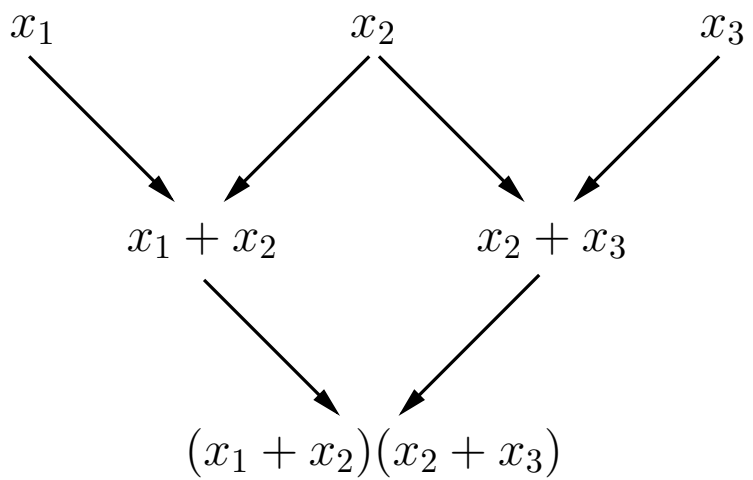
time step
0
1
2
3

Sequential computation takes 3 time steps

Example: Nonlinear Evaluation

Computation of $(x_1 + x_2)(x_2 + x_3) = x_1x_2 + x_2^2 + x_1x_3 + x_2x_3$

Parallel computation



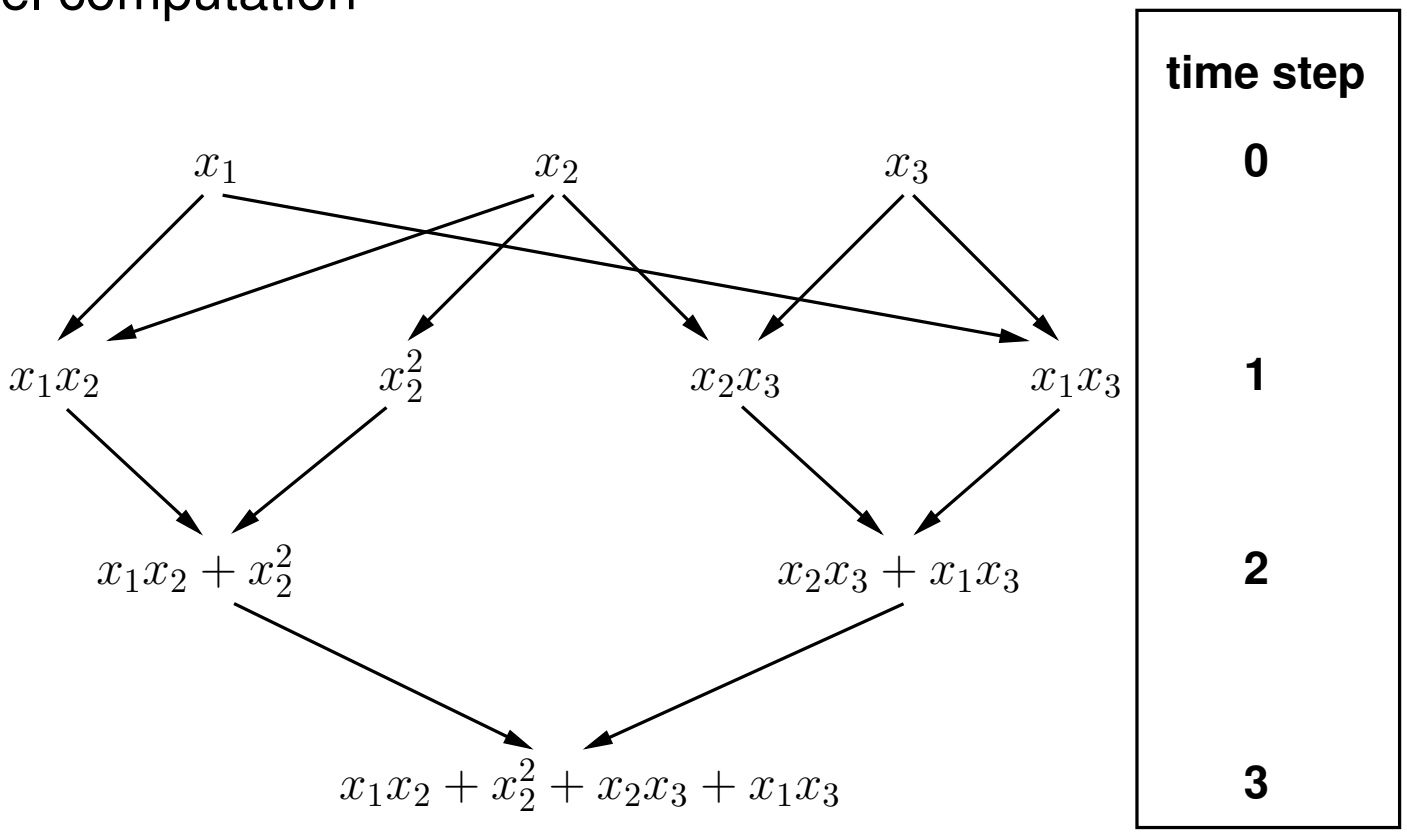
time step
0
1
2

Parallel computation takes 2 time steps

Example: Nonlinear Evaluation

Computation of $x_1 x_2 + x_2^2 + x_1 x_3 + x_2 x_3$

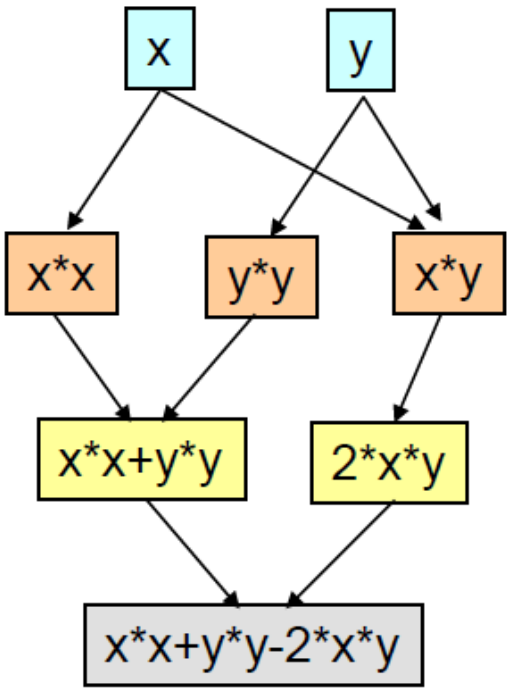
Parallel computation



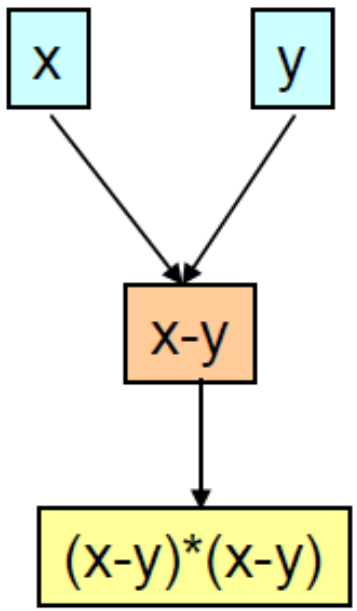
Parallel computation takes 3 time steps!

Further Example

Computation of $f(x, y) = x^2 - 2xy + y^2 = (x - y)^2$



3 time steps in parallel



2 time steps sequentially!

1.6.2. 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)

1.6.3. 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.

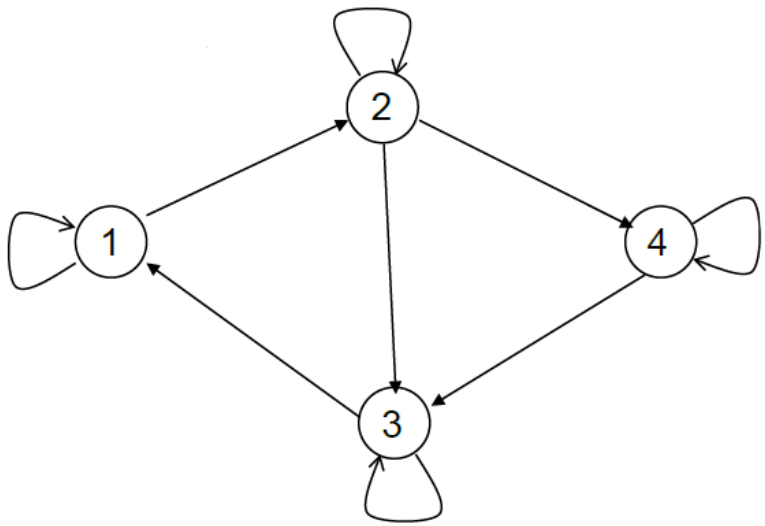
Iteration in Vector Form

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

Example:

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

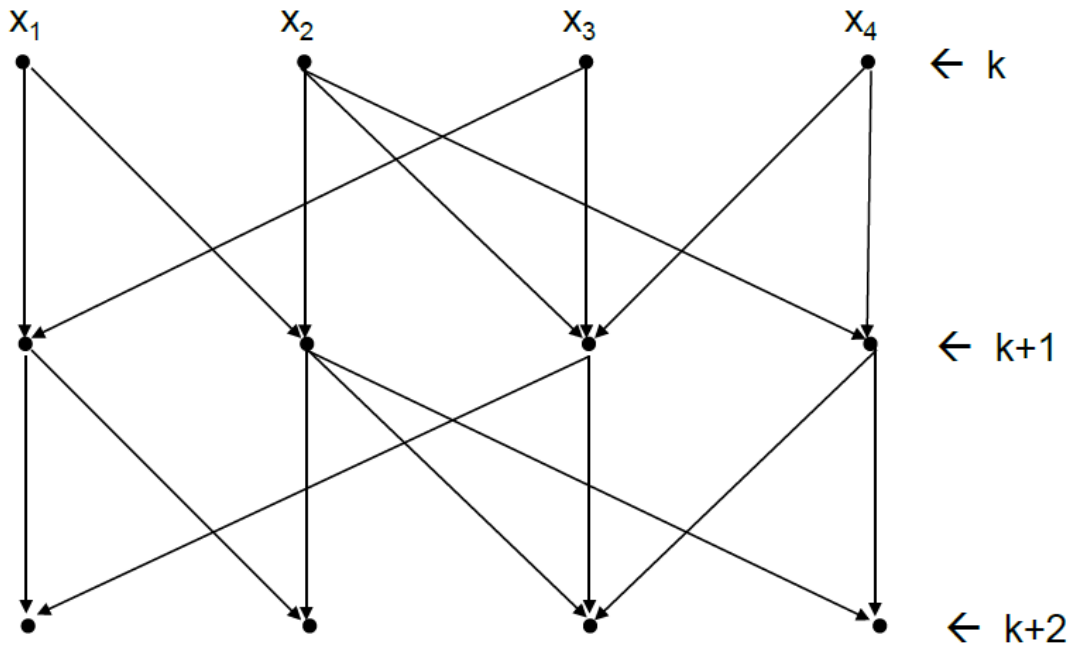
Dependency Graph:



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 sequentially:

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

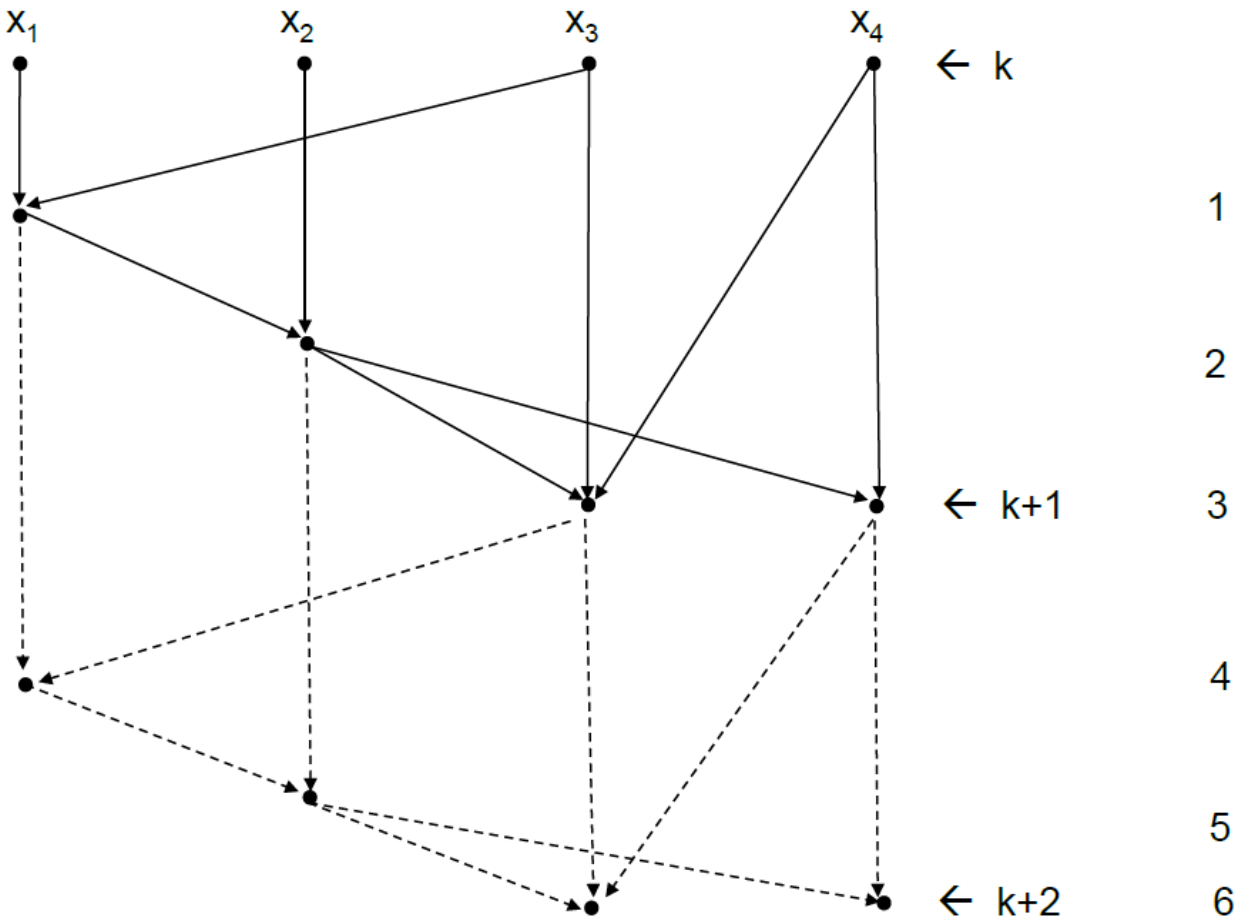
New information!

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

Computation from top to bottom:

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

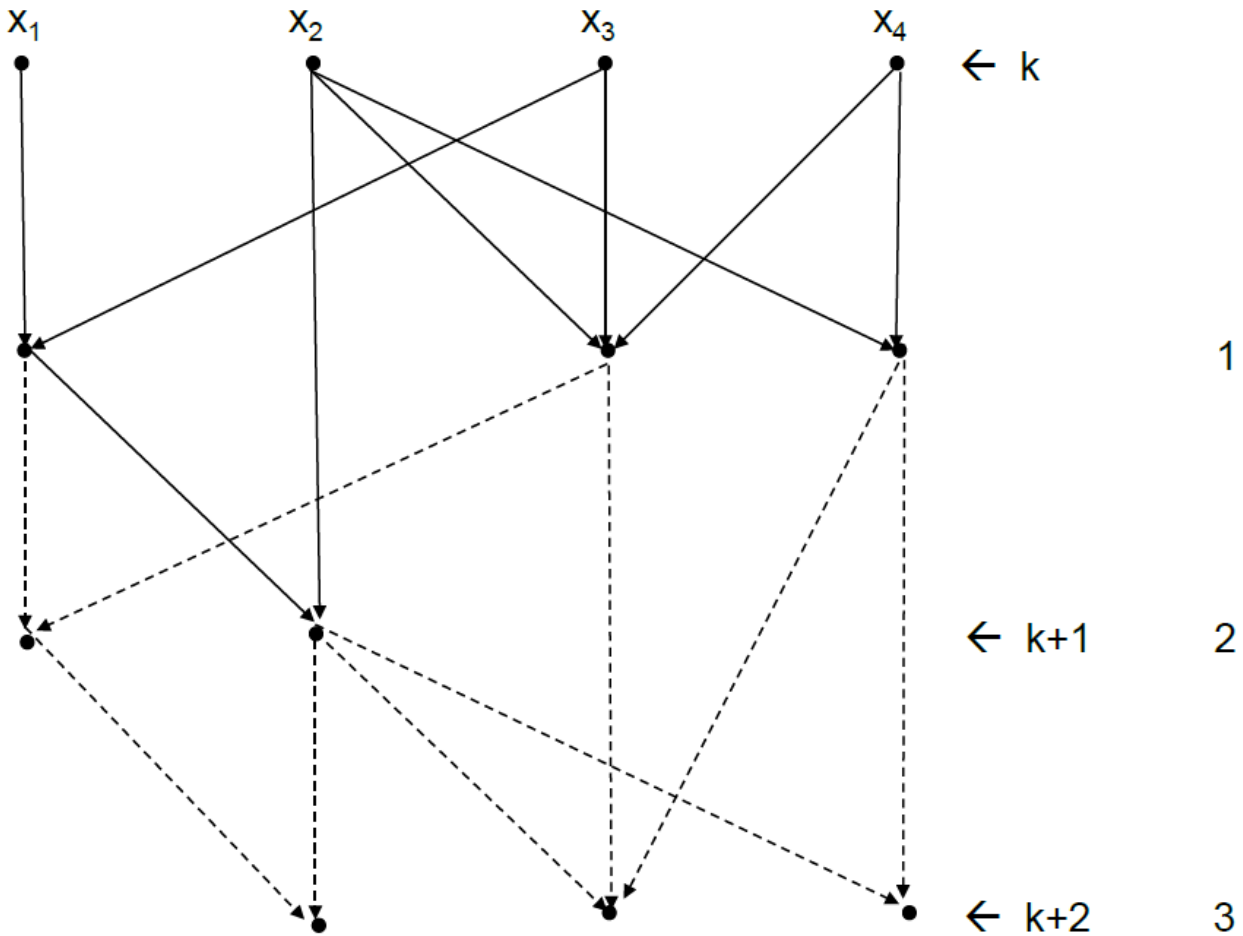
Equivalent to Jacobi, can be done in parallel!

New information

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?

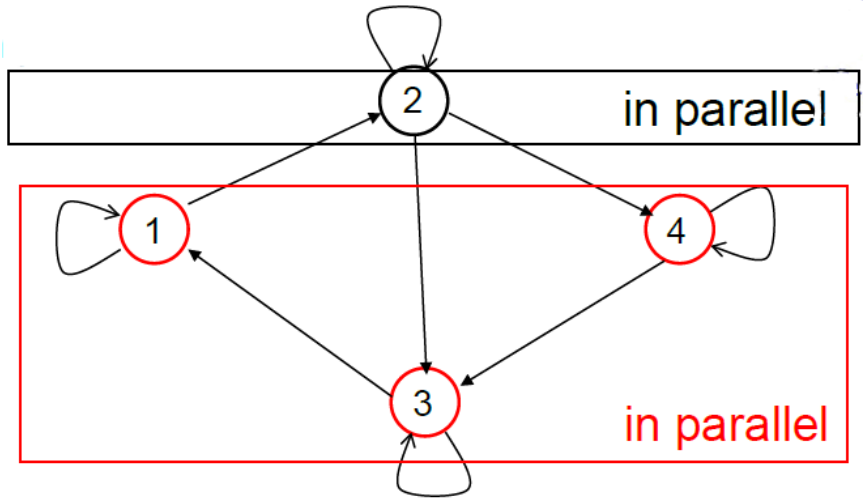
- Find 'optimal' ordering without losing parallel efficiency **and** with fast convergence!
- Use coloring algorithms for dependency graph to find efficient orderings:
  - use k colors for vertices of the graph
  - find minimal k but such that there are no cycles connecting vertices of same color! $\rightarrow$ assures ordering such that vertices with the same color are independent and can be updated in parallel!

Better Parallelism or Slower Convergence?

- Find 'optimal' ordering without losing 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 $4 \to 3 \to 1$.



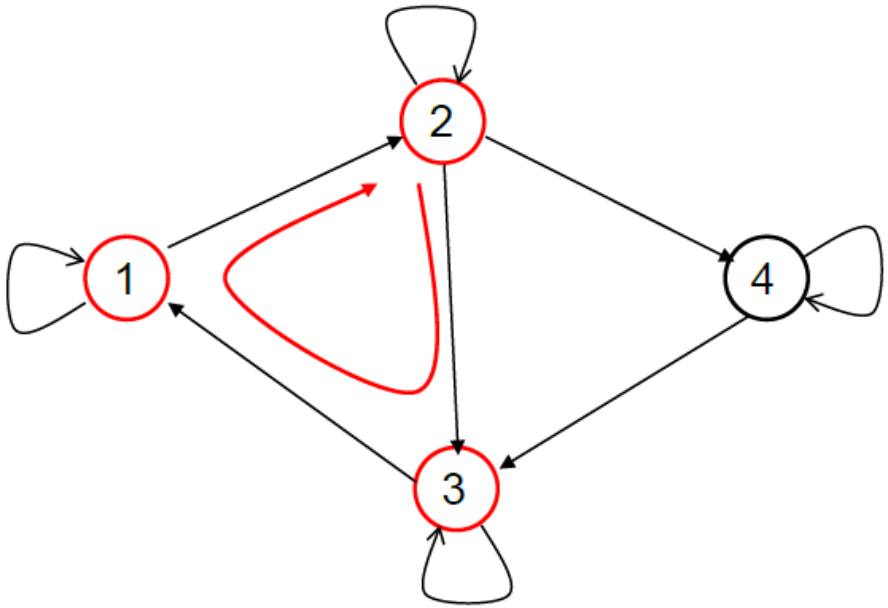
Induced numbering: (x_1, x_3, x_4) and (x_2)

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.

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

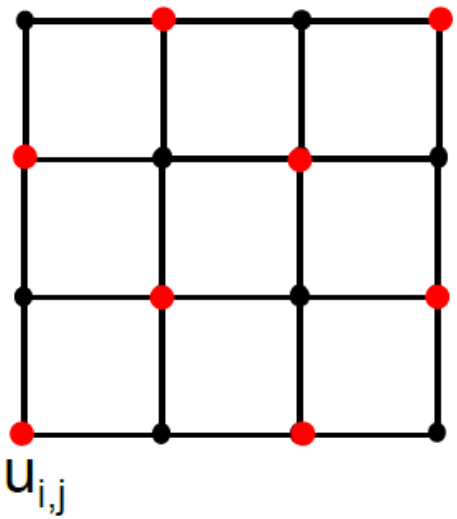
b) There exists a coloring of the graph with k colors such that there is no cycle of edges connecting vertices with the same color.

Proof:

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

Example from PDE

Discretization of rectangular region, Gauss-Seidel iteration:

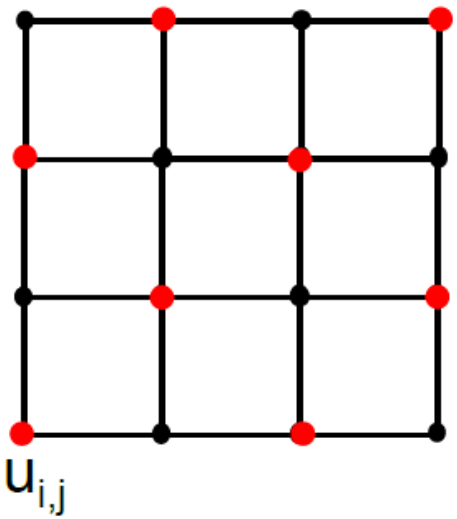


5-point discretization of 2d Laplacian

$$u_{xx} = \frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i,j-1} + u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}}{h^2}$$

Example from PDE

Discretization of rectangular region, Gauss-Seidel iteration:



5-point discretization of 2d Laplacian

$$u_{xx} = \frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i,j-1} + u_{i-1,j} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}}{h^2}$$

Odd components: red

Even components: black

k=2, two colors

Reorder discretization matrix according to the above red-black ordering. Gives 2×2 block matrix with diagonal matrices on the main diagonal.
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: $it_{\text{GS}} \cdot n$

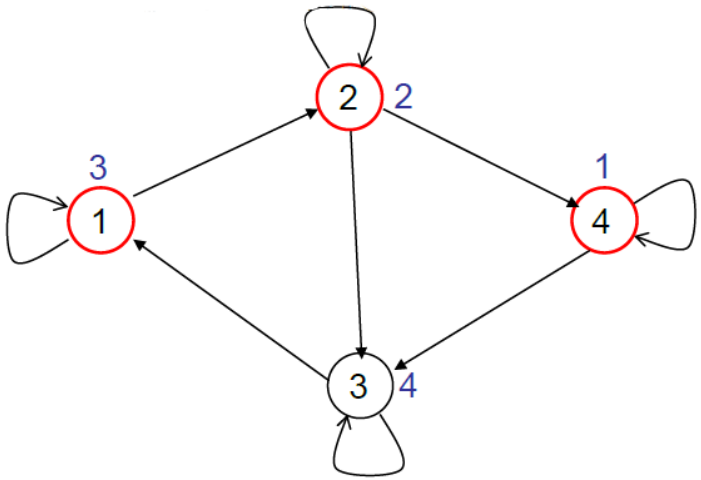
Example from PDE

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

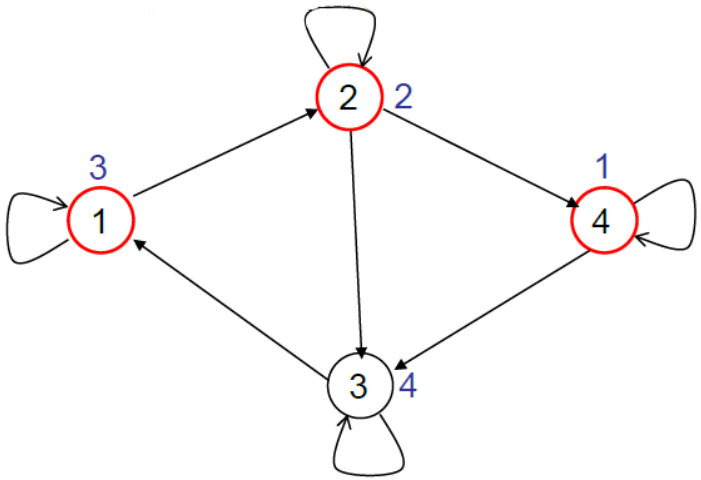
Newest information:
For x_3: new x_2 and x_4
For x_1: new $x_3 \rightarrow$ total: 3



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