

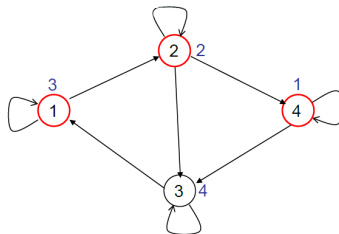
Comparison of Different Colorings

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)

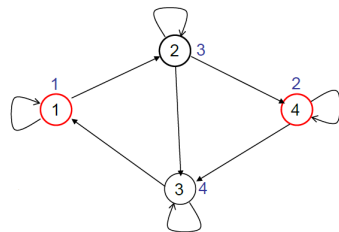
Newest information:

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)

Put $\frac{n}{2}$ function evaluations on each proc.

Parallel time: around $\frac{n}{2}$

$k = 2$, two colors with $\frac{n}{2}$ indices

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 $\frac{n}{2}$



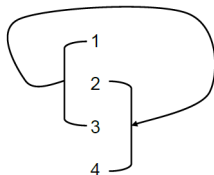
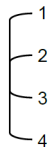
Comparison of Different k —Result

- #colors only important if rel. large #processors is available!
- n processors 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)
- White board example tridiagonal pattern
- Compare Block Jacobi and Block Gauss-Seid!



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:

$$x_1 = b_1/a_{11}$$

$$x_2 = (b_2 - a_{21}x_1)/a_{22}$$

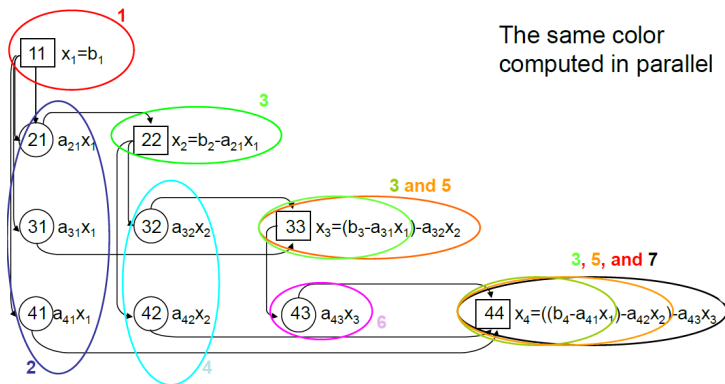
$$x_3 = (b_3 - a_{31}x_1 - a_{32}x_2)/a_{33}$$

$$x_4 = (b_4 - a_{41}x_1 - a_{42}x_2 - a_{43}x_3)/a_{44}$$

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



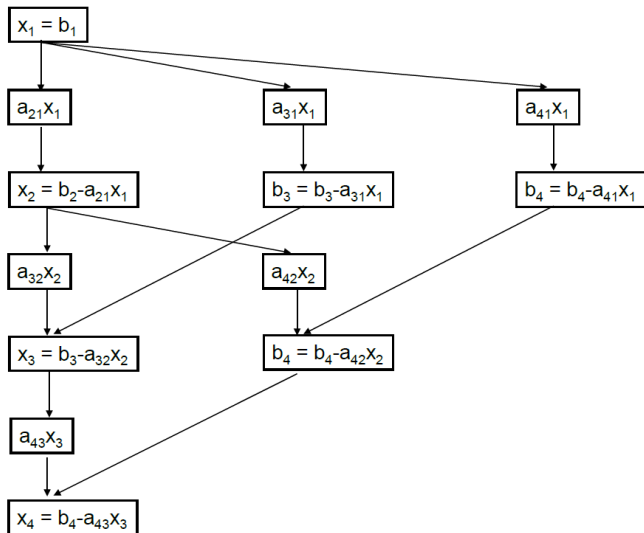
Graph for $a_{ij} = 1$



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



Data Dependency



1.7. Loop Manipulations

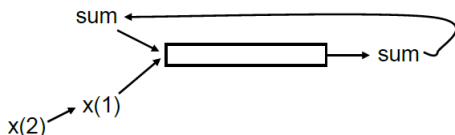
- Loops major computational tools in computing.
- For general efficiency important aspects:
 - data dependency inside loops
 - overhead of loop administration
 - work done in each loop



1.8. 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
```



```
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
```

→

```
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
```

→

```
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  
→  
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).



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).



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

```

→

```

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

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.



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
```

What is bad?



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
```

What is bad?

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
```

What is bad?



More Complicated Code with Less Divisions

```
dx1 = 1.0 / h(1)
For i = 1 : n
    dx0 = dx1
    dx1 = 1.0 / h(i+1)
    diag(i) = dx0 + dx1
    offdiag(i) = - dx1
End
```

What is bad?



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
```

What is bad?

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 2016/2017

2 Elementary Linear Algebra Problems



Contents

- 1 Introduction
 - 1.1 Computer Science Aspects
 - 1.2 Numerical Problems
 - 1.3 Graphs
 - 1.4 Loop Manipulations
- 2 Elementary Linear Algebra Problems**
 - 2.1 BLAS: Basic Linear Algebra Subroutines
 - 2.2 Matrix-Vector Operations
 - 2.3 Matrix-Matrix-Product
- 3 Linear Systems of Equations with Dense Matrices
 - 3.1 Gaussian Elimination
 - 3.2 Parallelization
 - 3.3 QR-Decomposition with Householder matrices
- 4 Sparse Matrices
 - 4.1 General Properties, Storage
 - 4.2 Sparse Matrices and Graphs
 - 4.3 Reordering
 - 4.4 Gaussian Elimination for Sparse Matrices
- 5 Iterative Methods for Sparse Matrices
 - 5.1 Stationary Methods
 - 5.2 Nonstationary Methods
 - 5.3 Preconditioning
- 6 Domain Decomposition
 - 6.1 Overlapping Domain Decomposition
 - 6.2 Non-overlapping Domain Decomposition
 - 6.3 Schur Complements



2.1. BLAS: Basic Linear Algebra Subroutines

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



2.2. 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	$\mathcal{O}(n^3)$

2.3. BLAS: Basic Linear Algebra Subroutines

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

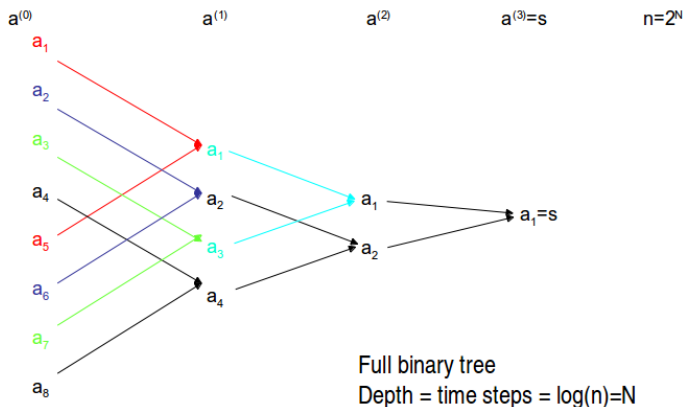
BLAS level	type of operation	complexity
BLAS-1	vector-vector	$\mathcal{O}(n)$
BLAS-2	matrix-vector	$\mathcal{O}(n^2)$
BLAS-3	matrix-matrix	$\mathcal{O}(n^3)$

- 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 $n = 2^N$

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

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



Vector update in the fan-in process with

$$n = 2^N$$

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

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

```
for(k=1;k<=N;k++)
  for(j=1;j<=2N-k;j++)
    aj = aj + aj+2N-k
  end
end
```

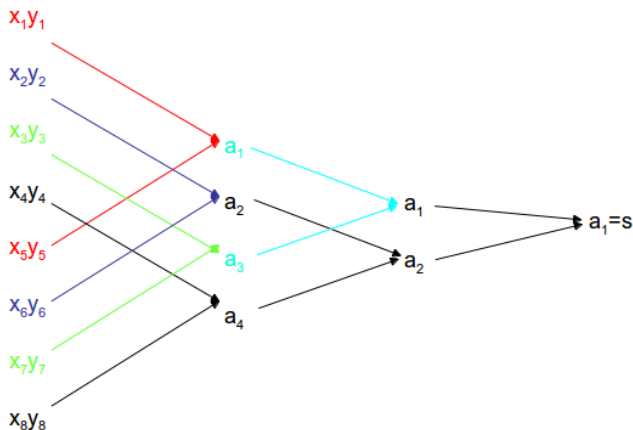
$\log(n) = N$ time steps in parallel for vector of length n . Pipelining?



Level-1 BLAS

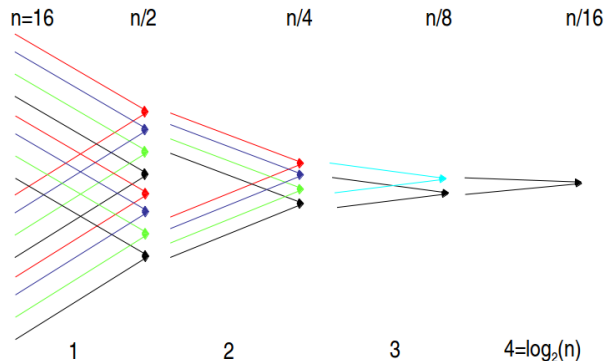
BLAS routines with $\mathcal{O}(n)$ problems (vectors only, $x, y \in \mathbb{R}^n$).

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



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: $y = \alpha x + y$



Level-1 BLAS: SAXPY

BLAS-Notation:

S single precision (D for double, C for complex)

A α scalar

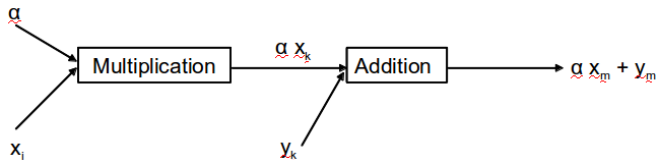
X vector

P plus operation

Y vector

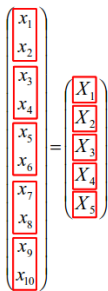
SAXPY: $y = \alpha x + y$

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



SAXPY Parallelization by Partitioning

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

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


- 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 $Y_j = \alpha X_j + Y_j$, $j = 1, 2, \dots, R$.
- 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)



Level-2 BLAS

- Matrix-Vector operations with $\mathcal{O}(n^2)$ operations (sequentially)

- BLAS-Notation:

S	single precision
G	} general matrix
E	
M	
V	vector

- defines SGEMV, matrix-vector product: $y = \alpha Ax + \beta y$
- Other Level-2 BLAS: solving triangular system $Lx = b$ with triangular matrix L .

Level-3 BLAS

- Matrix-Matrix operations with $\mathcal{O}(n^3)$ operations (sequentially)

- BLAS-Notation:

S single precision

G

E } general matrix

M

M matrix

- defines SGEMM, matrix-matrix product: $C = \alpha AB + \beta C$



Granularity for BLAS

BLAS level	operation	formula	memory	granularity
BLAS-1	AXPY: $2n$	$\alpha x + y$	$2n$	< 1
BLAS-2	GEMV: $2n^2$	$\alpha Ax + \beta y$	n^2	2
BLAS-3	GEMM: $2n^3$	$\alpha AB + \beta C$	$4n^2$	$\frac{n}{2}$

Note that the numbers in the table are $O(n)$ and memory means load+store.

BLAS-3 has best operations to memory ratio!



2.4. Analysis of the Matrix-Vector-Product

$$A = (a_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,m}} \in \mathbb{R}^{n \times m}, \quad b \in \mathbb{R}^m, \quad c \in \mathbb{R}^n$$

2.4.1. Vectorization

$$\begin{aligned} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} &= \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + \cdots + a_{1m}b_m \\ \vdots \\ a_{n1}b_1 + \cdots + a_{nm}b_m \end{pmatrix} = \\ &= \begin{pmatrix} \sum_{j=1}^m a_{1j}b_j \\ \vdots \\ \sum_{j=1}^m a_{nj}b_j \end{pmatrix} = \sum_{j=1}^m b_j \begin{pmatrix} a_{1j} \\ \vdots \\ a_{nj} \end{pmatrix} \end{aligned}$$



2.5. Analysis of the Matrix-Vector-Product

$$A = (a_{ij})_{\substack{i=1,\dots,n \\ j=1,\dots,m}} \in \mathbb{R}^{n \times m}, \quad b \in \mathbb{R}^m, \quad c \in \mathbb{R}^n$$

2.5.1. Vectorization

$$\begin{aligned} \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} &= \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + \cdots + a_{1m}b_m \\ \vdots \\ a_{n1}b_1 + \cdots + a_{nm}b_m \end{pmatrix} = \\ &= \begin{pmatrix} \sum_{j=1}^m a_{1j}b_j \\ \vdots \\ \sum_{j=1}^m a_{nj}b_j \end{pmatrix} = \sum_{j=1}^m b_j \begin{pmatrix} a_{1j} \\ \vdots \\ a_{nj} \end{pmatrix} \end{aligned}$$

n DOT-products of length m

m SAXPYs of length n (GAXPY)



Pseudocode: *ij*-form

```
c = 0;  
for i=1,...,n  
  for j=1,...,m  
     $c_i = c_i + a_{ij}b_j$  } DOT-product  
  end  
end
```

$c_i = A_{i\bullet}b$, DOT-product of *i*th row of *A* with vector *b*



Pseudocode: *ij*-form

```

c = 0;
for i=1,...,n
  for j=1,...,m
     $c_i = c_i + a_{ij}b_j$ 
  end
end

```

} DOT-product

$c_i = A_{i\bullet} \cdot b$, DOT-product of *i*th row of *A* with vector *b*

$$\boxed{c_i} = \boxed{A_{i\bullet}} \cdot \boxed{b}$$



Pseudocode: *ji*-form

```

c = 0;
for j=1,...,m
  for i=1,...,n
    ci = ci + aijbj
  end
end
end

```

$\left. \begin{array}{l} \text{SAXPY} \\ \downarrow \\ c = c + b_j \cdot A_{\bullet j} \end{array} \right\} \text{GAXPY}$

- SAXPY updating vector c with j th column of A
- GAXPY:
 - Sequence of SAXPYs related to the same vector
 - Advantage: vector c , that is updated, can be kept in fast memory
- No additional data transfer



GAXPY (repetition)

- SAXPY:

$$y := y + \alpha X$$

- GAXPY:

$$y = y_0$$

for $i = 1 : n$

$$y := y + \alpha_i X_i$$

end

- Series of SAXPYs regarding the same vector y .
- $\text{length}(\text{GAXPY}) = \text{length}(y)$
- Advantage: less data transfer!



2.5.2. Parallelization by Building Blocks

Reduce matrix-vector product on smaller matrix-vector products.

$$\{1, 2, \dots, n\} = \langle 1, n \rangle = I_1 \cup I_2 \cup \dots \cup I_R \text{ disjoint: } I_j \cap I_k = \emptyset \text{ for } j \neq k$$

$$\{1, 2, \dots, m\} = \langle 1, m \rangle = J_1 \cup J_2 \cup \dots \cup J_S \text{ disjoint: } J_j \cap J_k = \emptyset \text{ for } j \neq k$$



2.5.3. Parallelization by Building Blocks

Reduce matrix-vector product on smaller matrix-vector products.

$$\{1, 2, \dots, n\} = \langle 1, n \rangle = I_1 \cup I_2 \cup \dots \cup I_R \text{ disjoint: } I_j \cap I_k = \emptyset \text{ for } j \neq k$$

$$\{1, 2, \dots, m\} = \langle 1, m \rangle = J_1 \cup J_2 \cup \dots \cup J_S \text{ disjoint: } J_j \cap J_k = \emptyset \text{ for } j \neq k$$

Use 2-dimensional array of processors P_{rs} .

P_{rs} gets matrix block $A_{rs} := A(I_r, J_s)$, $b_s := b(J_s)$, $c_r := c(I_r)$.

$$I_r \left\{ \begin{array}{c} \left(\begin{array}{c|c} & \\ \hline & A_{rs} \\ \hline & \\ \hline \end{array} \right) \cdot \left(\begin{array}{c} \\ \hline b_s \\ \hline \end{array} \right) \Big\}_{J_s} = \left(\begin{array}{c} \\ \hline c_r \\ \hline \end{array} \right) \Big\}_{I_r}$$

$$c_r = \sum_{s=1}^S A_{rs} b_s =: \sum_{s=1}^S c_r^{(s)}$$



Pseudocode

```
for  $r = 1, \dots, R$   
  for  $s = 1, \dots, S$   
     $c_r^{(s)} = A_{rs} b_s$ ;  
  end  
end
```

Small, independent matrix-vector products. No communication necessary during computations!

```
for  $r = 1, \dots, R$   
   $c_r = 0$   
  for  $s = 1, \dots, S$   
     $c_r = c_r + c_r^{(s)}$ ;  
  end  
end
```

Blockwise collection and addition of vectors. Rowwise communication! Fan-in.



Blocking: Special Cases

$S = 1$: The computation of $A_i \bullet b$ is vectorizable by GAXPYs.

$$c = \begin{pmatrix} A_{1\bullet} \\ A_{2\bullet} \\ \vdots \end{pmatrix} \cdot b = \begin{pmatrix} A_{1\bullet} b \\ A_{2\bullet} b \\ \vdots \end{pmatrix}$$

No communication necessary between processor P_1, \dots, P_R



Blocking: Special Cases

$S = 1$: The computation of $A_{i\bullet}b$ is vectorizable by GAXPYs.

$$c = \begin{pmatrix} A_{1\bullet} \\ A_{2\bullet} \\ \vdots \end{pmatrix} \cdot b = \begin{pmatrix} A_{1\bullet}b \\ A_{2\bullet}b \\ \vdots \end{pmatrix}$$

No communication necessary between processor P_1, \dots, P_R

$R = 1$: $A_{\bullet j}b_j$ are independent.

$$c = (A_{\bullet 1} | A_{\bullet 2} | \dots) \cdot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \end{pmatrix} = A_{\bullet 1}b_1 + A_{\bullet 2}b_2 + \dots$$

Then collection of partial results from processor P_1, \dots, P_S . Fan-in.
Final sum in one processor: vectorizable by GAXPYs.



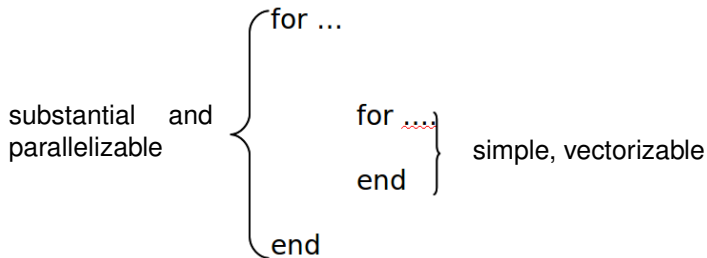
Rules

1. Inner loops of a program should be simple, vectorizable



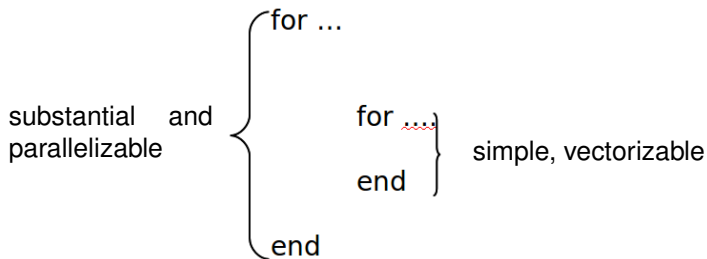
Rules

1. Inner loops of a program should be simple, vectorizable
2. Outer loop of a program should be substantial, independent, parallelizable.



Rules

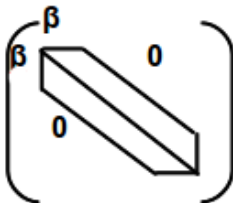
1. Inner loops of a program should be simple, vectorizable
2. Outer loop of a program should be substantial, independent, parallelizable.



3. Reuse of data (cache, minimal data transfer, blocking)



2.5.4. $c = Ab$ for Banded Matrix



- Bandwidth β (symmetric)
- $2\beta+1$ diagonals: main diag. + β subdiag. + β superdiag.
- $\beta = 1$: tridiagonal

Notation: Banded Matrices A and \tilde{A}

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1,\beta+1} & 0 & \cdots & 0 \\ \vdots & a_{22} & \ddots & \ddots & \cdots & \vdots \\ a_{\beta+1,1} & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & a_{n-\beta,n} \\ \vdots & \vdots & \ddots & \ddots & a_{n-1,n-1} & \vdots \\ 0 & \cdots & 0 & a_{n,n-\beta} & \cdots & a_{nn} \end{pmatrix} \rightarrow$$



Notation: Banded Matrices A and \tilde{A}

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1,\beta+1} & 0 & \cdots & 0 \\ \vdots & a_{22} & \ddots & \ddots & \cdots & \vdots \\ a_{\beta+1,1} & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & a_{n-\beta,n} \\ \vdots & \vdots & \ddots & \ddots & a_{n-1,n-1} & \vdots \\ 0 & \cdots & 0 & a_{n,n-\beta} & \cdots & a_{nn} \end{pmatrix} \rightarrow$$

$$\tilde{A} = \begin{pmatrix} \tilde{a}_{10} & \cdots & \tilde{a}_{1,\beta} & 0 & \cdots & 0 \\ \vdots & \tilde{a}_{20} & \ddots & \ddots & \cdots & \vdots \\ \tilde{a}_{\beta+1,-\beta} & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \tilde{a}_{n-\beta,\beta} \\ \vdots & \vdots & \ddots & \ddots & \tilde{a}_{n-1,0} & \vdots \\ 0 & \cdots & 0 & \tilde{a}_{n,-\beta} & \cdots & \tilde{a}_{n,0} \end{pmatrix}$$



$c = Ab$ for Banded Matrix

Storing entries diagonalwise: $n(2\beta + 1)$ matrix instead of n^2 .

$$\tilde{a}_{i,s} = a_{i,i+s} \quad \text{for row } i = 1, \dots, n$$
$$1 \leq i + s \leq n \quad \text{and} \quad -\beta \leq s \leq \beta \quad \text{and} \quad 1 \leq i \leq n$$



$c = Ab$ for Banded Matrix

Storing entries diagonalwise: $n(2\beta + 1)$ matrix instead of n^2 .

$$\tilde{a}_{i,s} = a_{i,i+s} \quad \text{for row } i = 1, \dots, n$$

$$1 \leq i + s \leq n \quad \text{and} \quad -\beta \leq s \leq \beta \quad \text{and} \quad 1 \leq i \leq n$$

$$1 - i \leq s \leq n - i \quad \text{and} \quad -\beta \leq s \leq \beta$$

↓ in row i

$$s \in [l_i, r_i] = [\max\{-\beta, 1 - i\}, \min\{\beta, n - i\}]$$



$c = Ab$ for Banded Matrix

Storing entries diagonalwise: $n(2\beta + 1)$ matrix instead of n^2 .

$$\tilde{a}_{i,s} = a_{i,i+s} \quad \text{for row } i = 1, \dots, n$$

$$1 \leq i + s \leq n \quad \text{and} \quad -\beta \leq s \leq \beta \quad \text{and} \quad 1 \leq i \leq n$$

$$1 - i \leq s \leq n - i \quad \text{and} \quad -\beta \leq s \leq \beta$$

↓ *in row i*

$$s \in [l_i, r_i] = [\max\{-\beta, 1 - i\}, \min\{\beta, n - i\}]$$

$$1 - s \leq i \leq n - s \quad \text{and} \quad 1 \leq i \leq n$$

↓ *in diag. s*

$$i \in [\tilde{l}_s, \tilde{r}_s] = [\max\{1, 1 - s\}, \min\{n, n - s\}]$$

