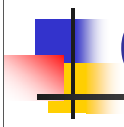
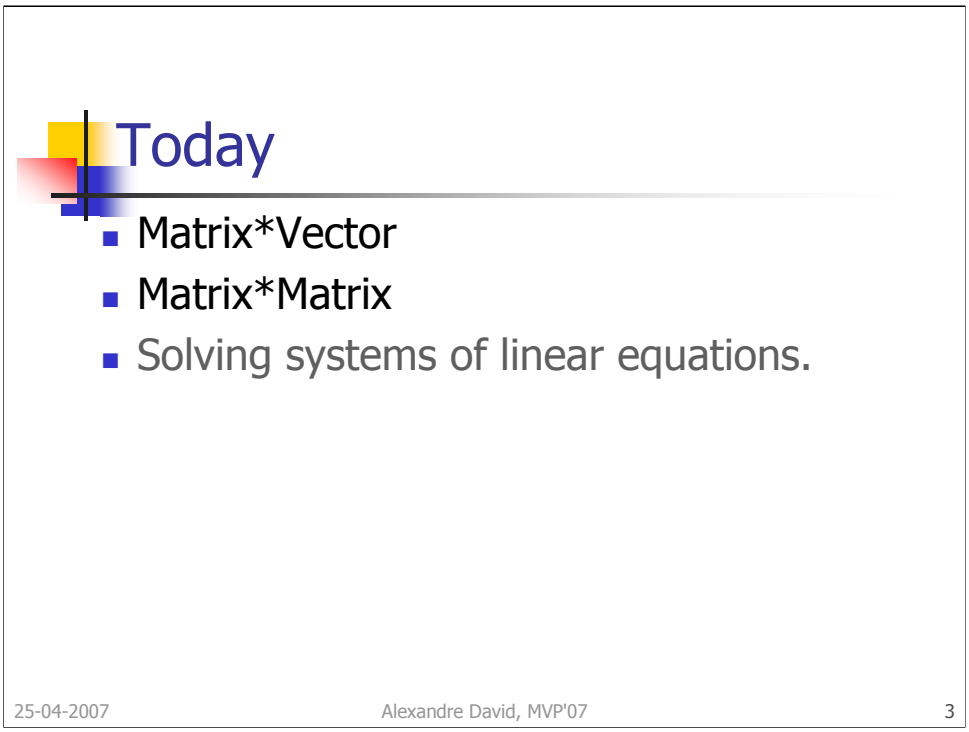


Dense Matrix Algorithms (Chapter 8)

Alexandre David
B2-206

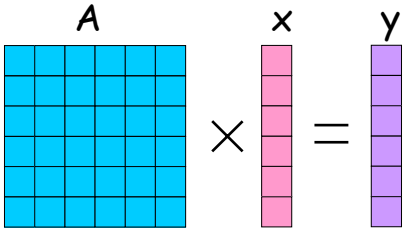
Dense Matrix Algorithm

- Dense or full matrices: few known zeros.
  - Other algorithms for sparse matrix.
- Square matrices for pedagogical purposes only – can be generalized.
- Natural to have data decomposition.
  - 3.2.2 input/output/intermediate data.
  - 3.4.1 mapping schemes based on data partitioning.



The chapter is not just about giving algorithms but also about analyzing them.

Matrix*Vector – Recall

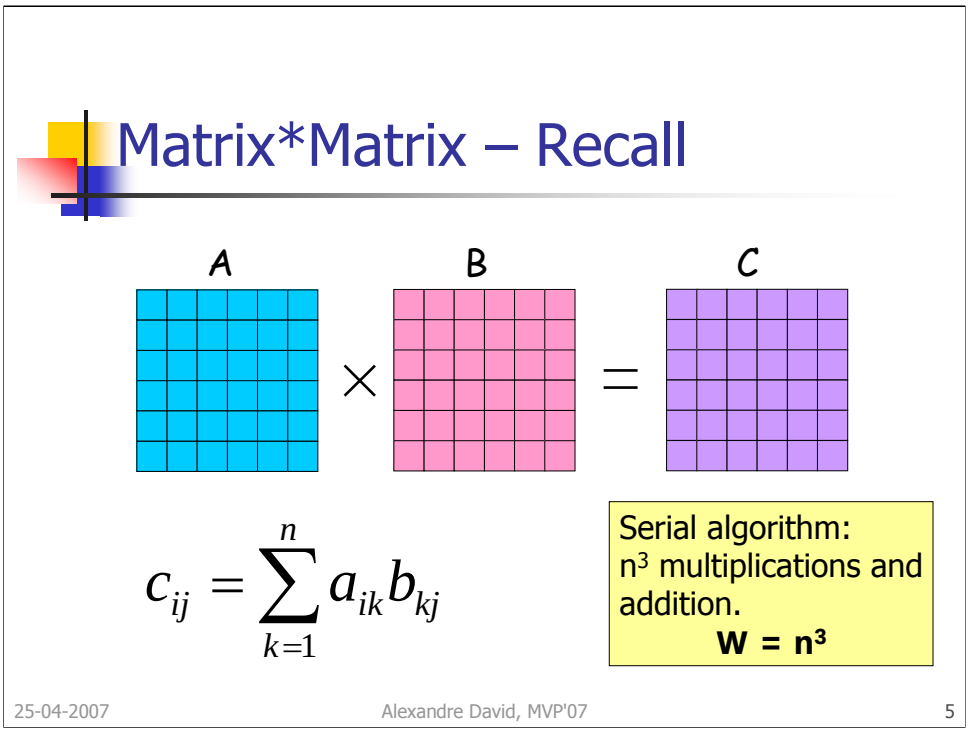


$$y_i = \sum_{k=1}^{n} a_{ik} x_k$$

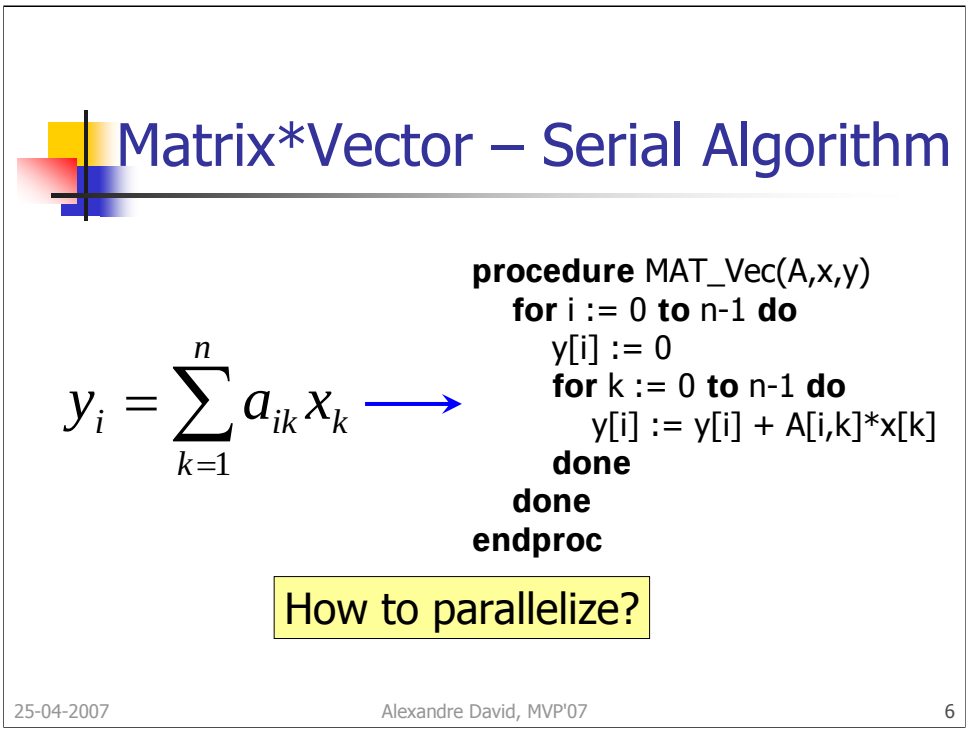
Serial algorithm:
n^2 multiplications and addition.

W = n^2

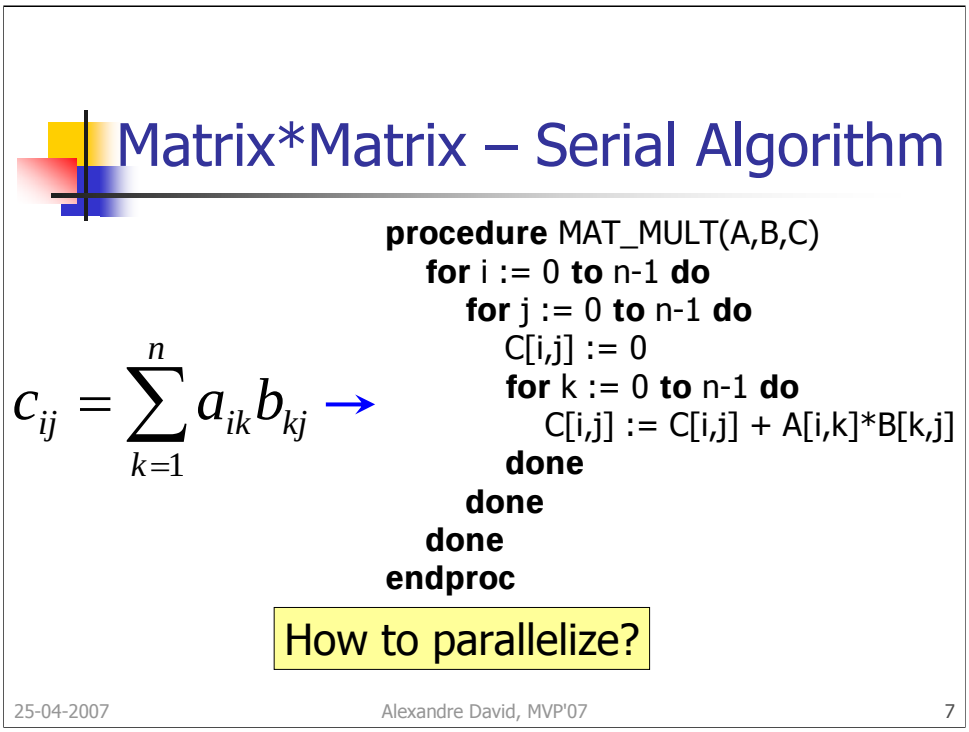
Recall: See chapter 5 on analysis. W = problem size = number of basic computation in the best sequential algorithm to solve the problem.



There are better algorithms but the sake of simplicity we consider the straight-forward computation, which is still a good algorithm, though not the best.



How to parallelize: Row-wise 1-D, column-wise 1-D, or 2-D partioning.

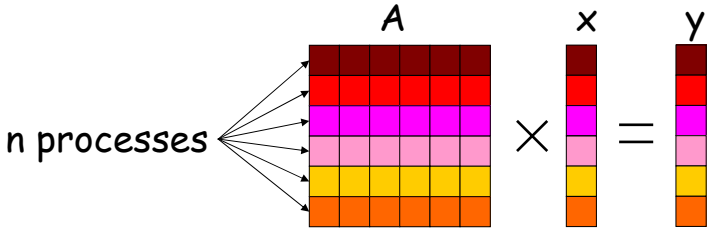


Similar partitioning as for the matrix*vector and block partitioning.

Matrix*Vector – Row-wise 1-D Partitioning

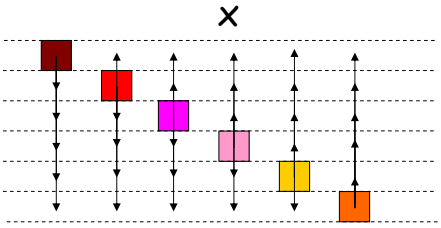
- Initial distribution:
  - Each process has a row of the *n*n* matrix.
  - Each process has an element of the *n*1* vector.
  - Each process is responsible for computing one element of the result.

Matrix*Vector – 1-D

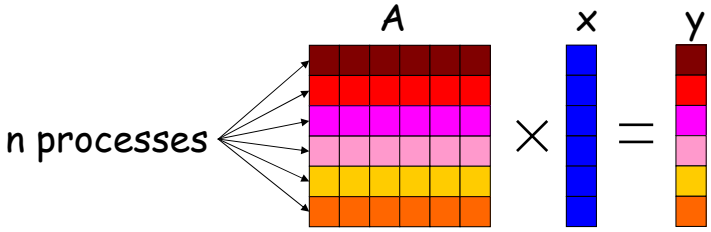


But every process needs the entire vector
⇒ all-to-all broadcast.

All-to-All Broadcast

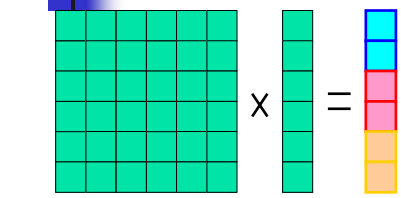


Parallel Computation

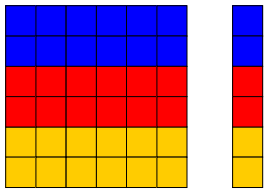


$$y_i = \sum_{k=1}^{n} a_{ik} x_k$$ in parallel on the n processes.

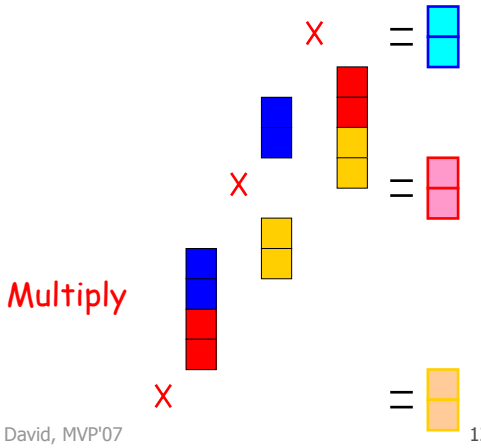
Example Matrix*Vector (Program 6.4)



Partition on rows.



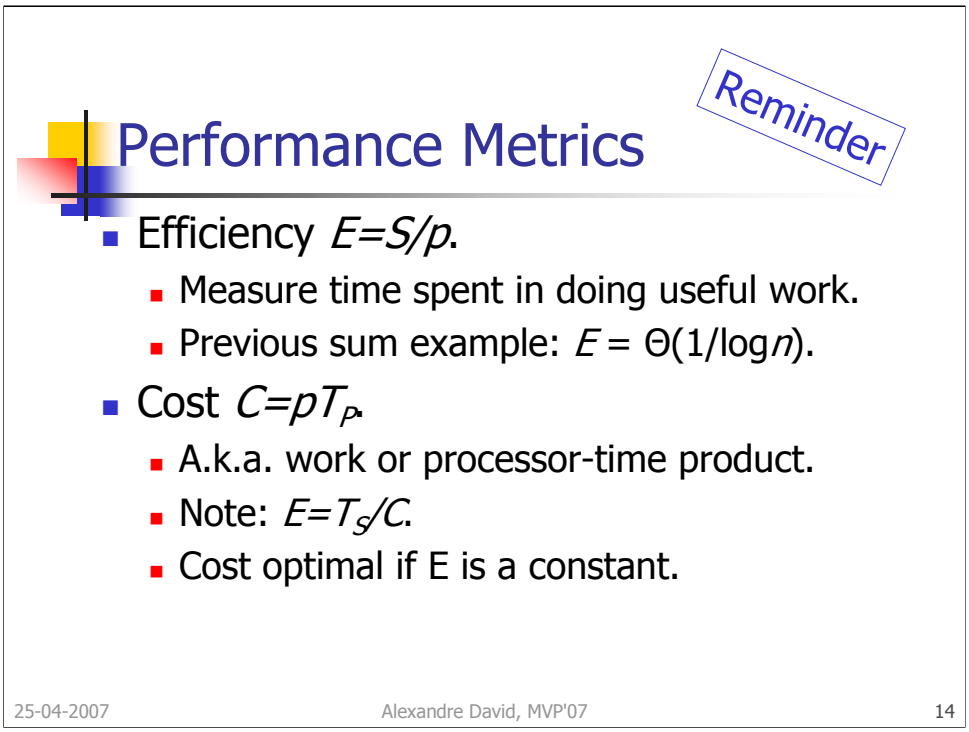
Allgather (All-to-all broadcast)



Analysis

- All-to-all broadcast & multiplications in $\Theta(n)$.
- For n processes $W=nT_p=n^2$.
  $\Rightarrow$ The algorithm is cost-optimal.

> A parallel system is cost-optimal iff $pT_p=\Theta(W)$.



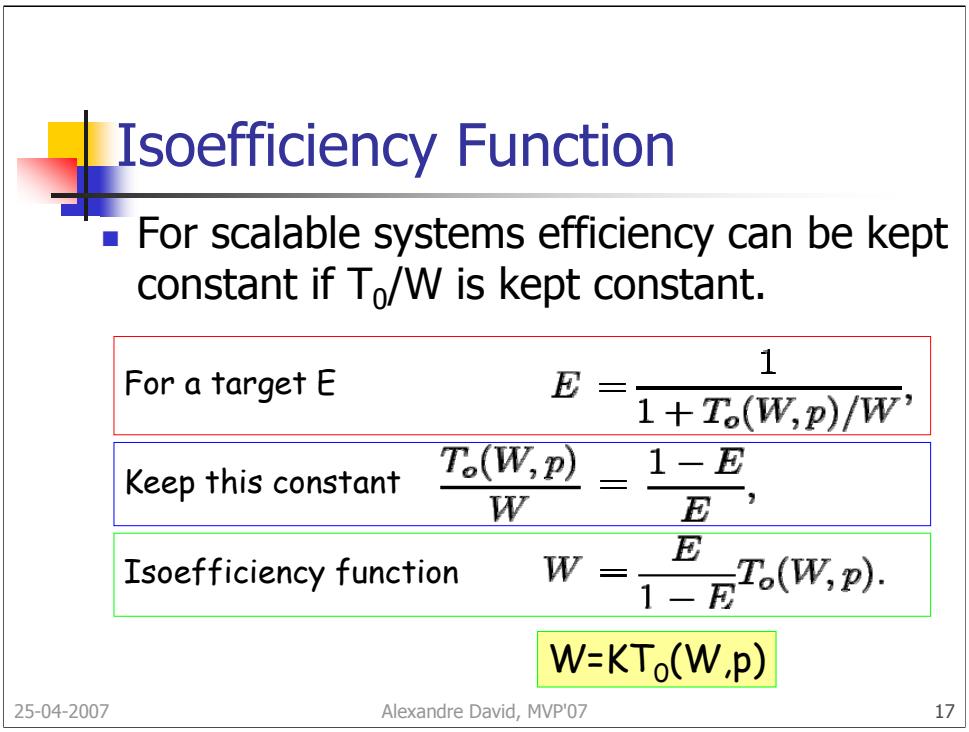
Reminder from lecture 8.

Using Fewer Processes

- Brent's scheduling principle: It's possible.
- Using p processes:
  - n/p rows per process.
  - Communication time = $t_s \log p + t_w (n/p)(p-1)$ $\sim t_s \log p + t_w n = \Theta(n)$.
  - Computation: n*n/p.
    $\Rightarrow pT_P = \Theta(n^2) = W \Rightarrow$ It is cost optimal.

Scalability – Recall

- Efficiency increases with the size of the problem.
- Efficiency decreases with the number of processors.
- Scalability measures the ability to increase speedup in function of p.



What it means: The isoefficiency function determines the ease with which a parallel system can maintain its efficiency in function of the number of processors. A small function means that small increments of the problem size are enough (to compensate the increase of p), i.e., the system is scalable. A large function means the problem size must be incremented dramatically to compensate p, i.e., the system is poorly scalable.

Unscalable system do not have an isoefficiency function.

Isoefficiency function is in function of *p*.

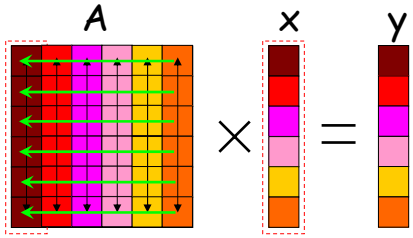
Is Our Algorithm Scalable?

- $T_0=pT_P-W \Rightarrow T_0=t_s p \log p+t_w np$. *Overhead*
- We want to determine $W=KT_0$. Try with both terms separately:
  - $W=Kt_s p \log p$.
  - $W=Kt_w np=n^2 \Rightarrow W=(Kt_w p)^2$.
  - Bound from concurrency: $p=O(n) \Rightarrow W=\Omega(p^2)$.
  - $W=\Theta(p^2)$: asymptotic isoefficiency function. Rate to increase the problem size (in function of p) to maintain a fixed efficiency: $p=\Theta(n)$.

Matrix*Vector – 2-D

- Matrix $n*n$ partitioned on $n*n$ processes.
- Vector $n*1$ distributed in the last (or 1st column).
- Similarly we want fewer processes: blocks of $(n/\sqrt{p})^2$ elements.

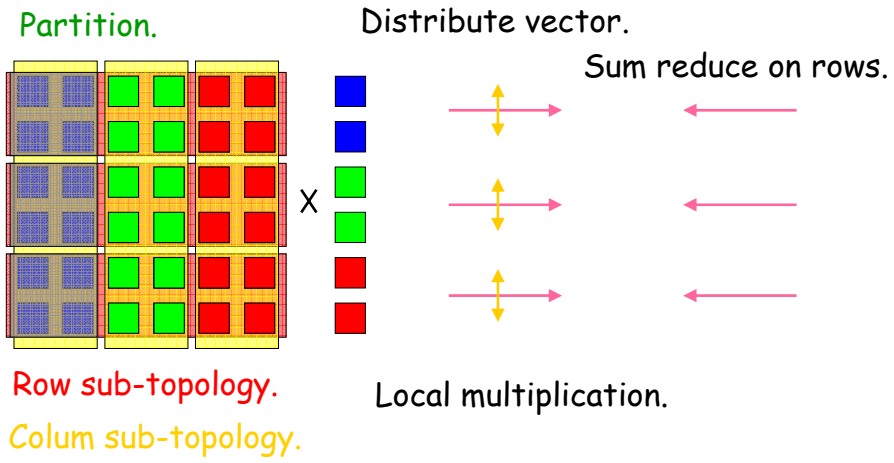
Matrix*Vector – 2-D



Processes in column i need element of the vector in row i.

1. Distribute on diagonal.
2. One-to-all broadcast on columns.
3. Multiplication.
4. All-to-one reduction (+).

Example Matrix*Vector (Program 6.8)



Which one is better? 1-D or 2-D?

Analysis

- Communications:
  - one-to-one $\Theta(1)$ +
  - one-to-all broadcast $\Theta(\log n)$ +
  - all-to-one reduction $\Theta(\log n)$.
- + multiplications $\Theta(1)$.
- $pT_p=\Theta(n^2 \log n) \Rightarrow$ not cost-optimal.
- Brent's scheduling principle?

Using Fewer Processes

- Blocks of $(n/\sqrt{p})^2$ elements. Costs:
  - one to one in $t_s+t_w n/\sqrt{p}$ +
  - one-to-all broadcast in $(t_s+t_w n/\sqrt{p})\log\sqrt{p}$ +
  - all-to-one reduction in $(t_s+t_w n/\sqrt{p})\log\sqrt{p}$ +
  - computations in $(n/\sqrt{p})^2$.
- Total $\sim n^2/p+t_s \log p+(t_w n/\sqrt{p})\log p$.
- $pT_P=\Theta(n^2) \Rightarrow$ cost-optimal if…

Scalability Analysis

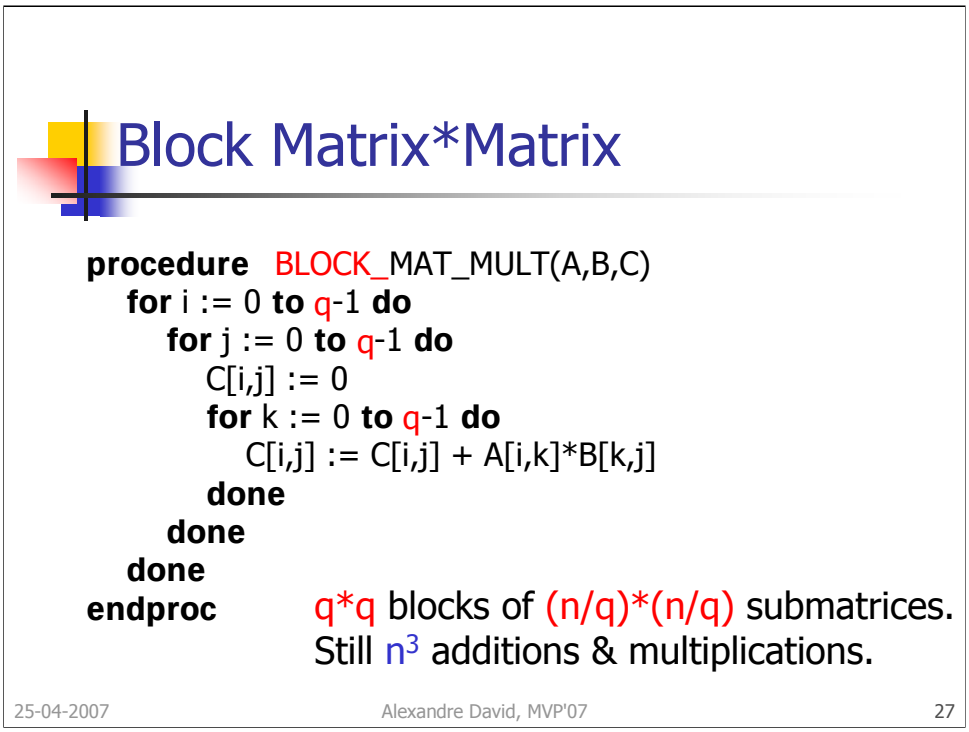
- $T_0 = pT_P - W = t_s \log p + t_w n\sqrt{p} \log p$.
- As before, isoefficiency analysis:
  - $W = Kt_s p \log p$.
  - $W = Kt_w n\sqrt{p} \log p = n^2 \Rightarrow W = (Kt_w \sqrt{p} \log p)^2$.
  - Bound from concurrency: $p = O(n^2) \Rightarrow W = \Omega(p)$.
  - $W = \Theta(p \log^2 p)$.
- $p = f(n)$? $p \log^2 p = \Theta(n^2)$ … $p = \Theta(n^2 / \log^2 n)$.

Which One Is Better?

- 1-D: $T_P \sim n^2/p + t_s \log p + t_w n$.
- 2-D: $T_P \sim n^2/p + t_s \log p + (t_w n/\sqrt{p}) \log p$.

- 1-D: $W = \Theta(p^2)$.
- 2-D: $W = \Theta(p \log^2 p)$.

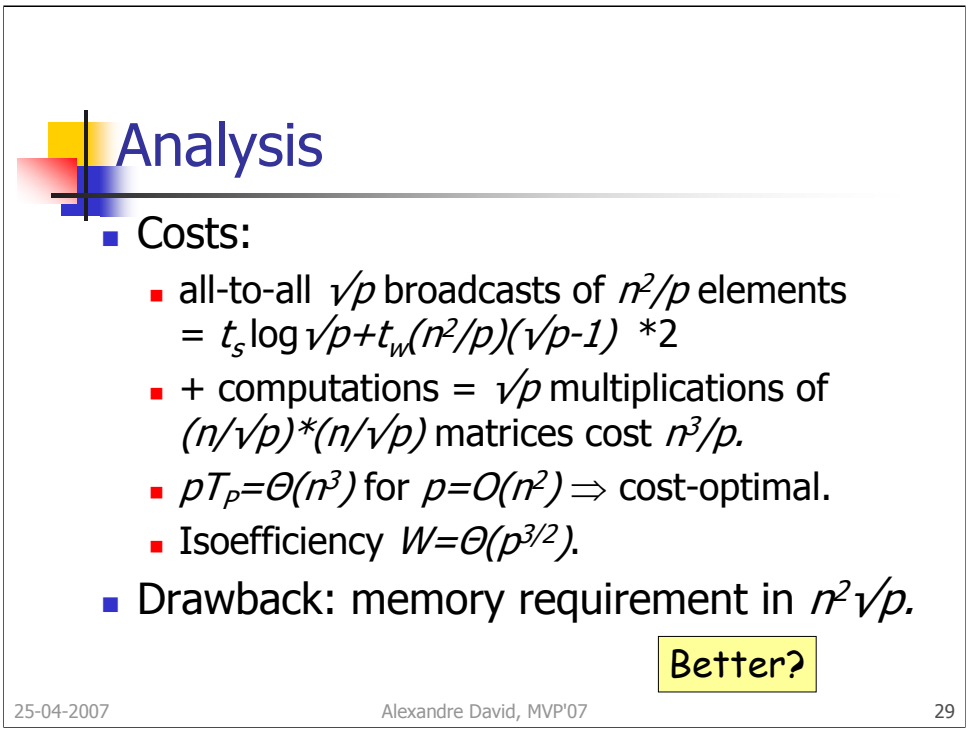
- Degree of concurrency…



Similar to conventional matrix multiplication.

A Simple Parallel Algorithm

- Map the algorithm to $p=q^2$ processes.
- We need all A[i,k] and B[k,j] to compute the C[i,j].
- Steps:
  - All-to-all broadcast of A[i,k] on rows.
  - All-to-all broadcast of B[k,j] on columns.
  - Local multiplications.

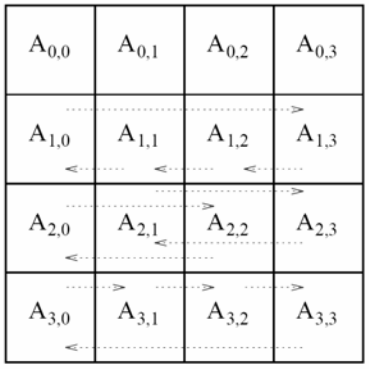


One multiplication of a $(n/\sqrt{p})*(n/\sqrt{p})$ matrix costs $(n/\sqrt{p})^3$ and we have $\sqrt{p}$ of them (per process).

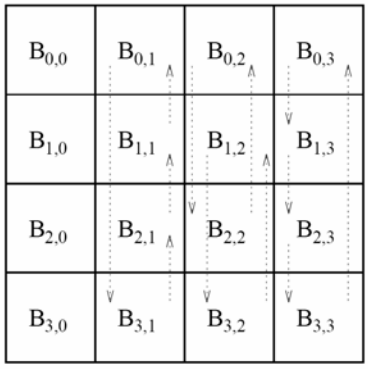
Cannon's Algorithm

- Idea: re-schedule computations to avoid contention.
  - Processes on rows i hold a different A[i,k].
  - Processes on columns j hold a different B[k,j].
  - Rotate the matrices $\Rightarrow$ we need only 2 sub-matrices per process at any time.
    $\Rightarrow$ memory efficient in $O(n^2)$.

Align A & B

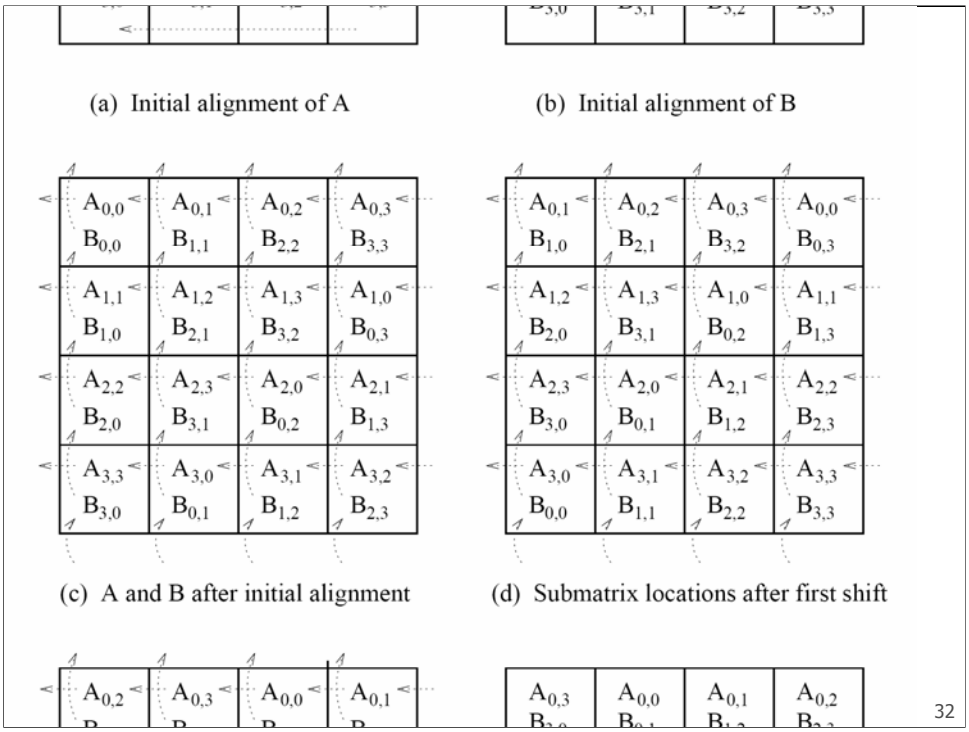


(a) Initial alignment of A



(b) Initial alignment of B

31


(a) Initial alignment of A
(b) Initial alignment of B
(c) A and B after initial alignment
(d) Submatrix locations after first shift

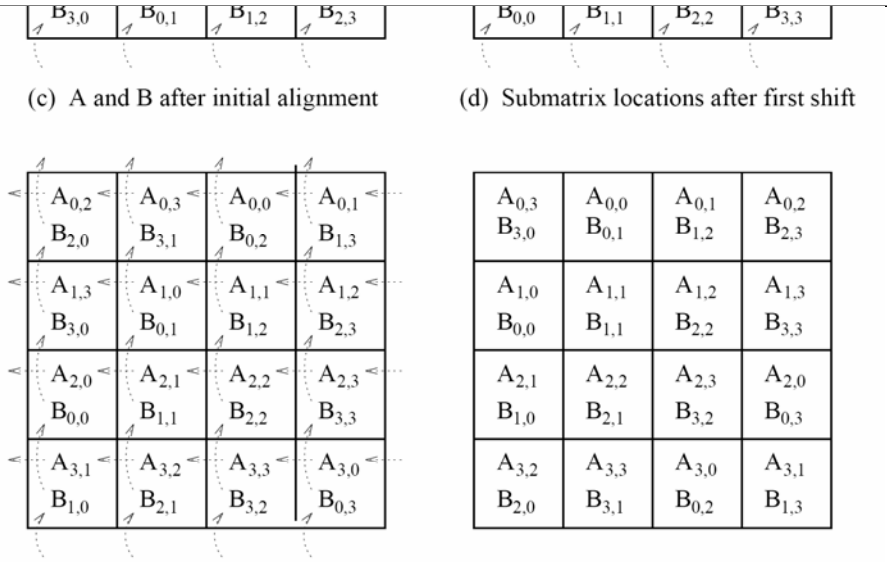
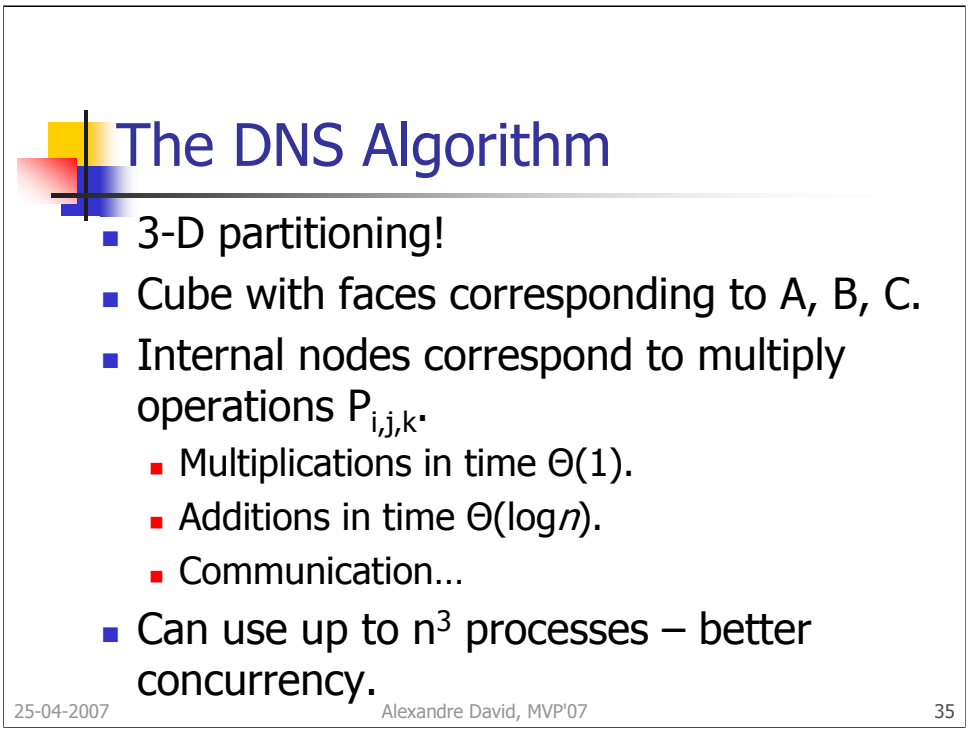


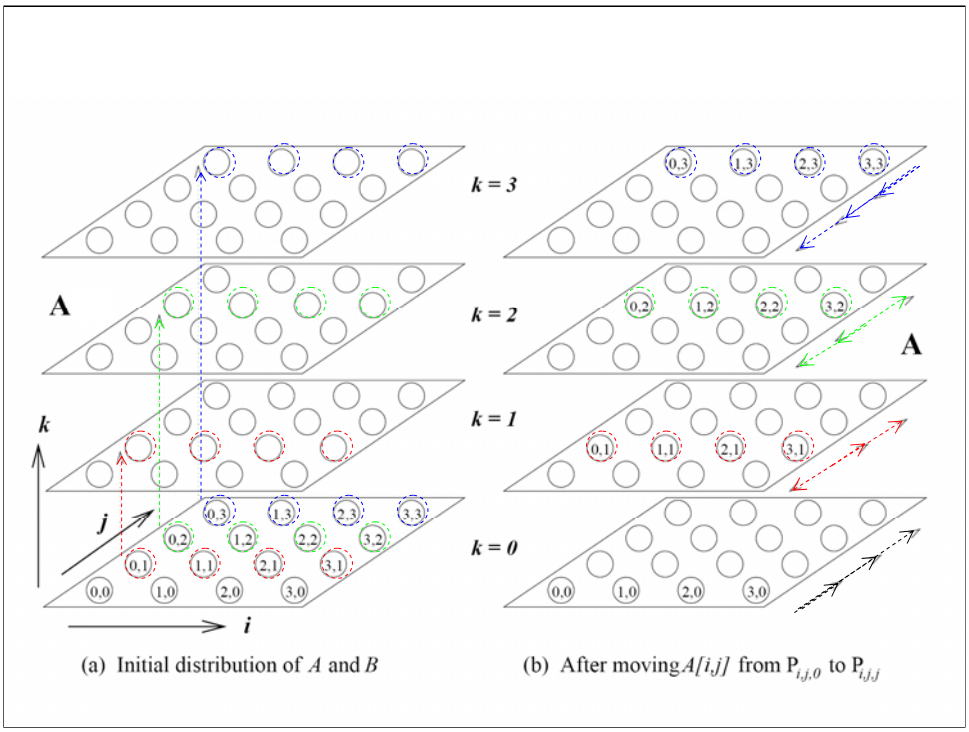
Figure 8.3 The communication steps in Cannon's algorithm on 16 processes.

Analysis

- Costs:
  - 2* (A & B) $\sqrt{p}$-single step shifts = $2(t_s+t_w n^2/p)\sqrt{p}$ +
  - $\sqrt{p}$ multiplications of $(n/\sqrt{p})*(n/\sqrt{p})$ sub-matrices = n^3/p.
  - Cost-optimal, same isoefficiency function as previously.



DNS: Dekel, Nassimi, and Sahni.



(a) Initial distribution of A and B

(b) After moving $A[i,j]$ from $P_{i,j,0}$ to $P_{i,j,j}$

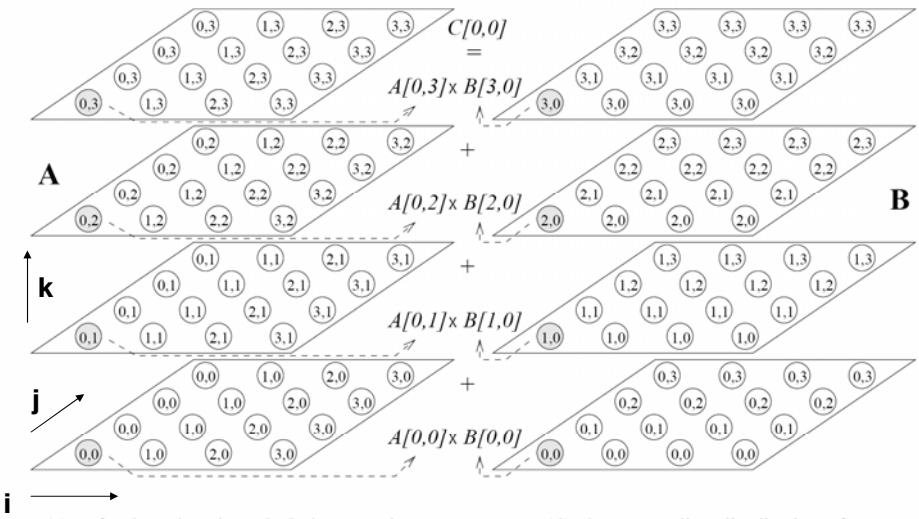
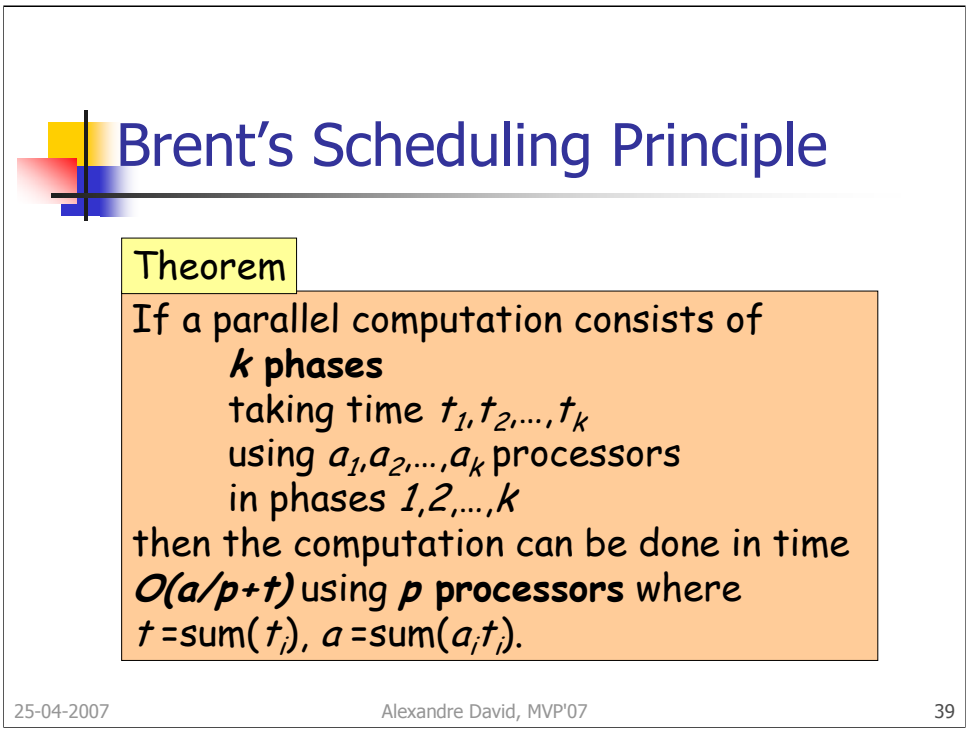


Figure 8.4 The communication steps in the DNS algorithm while multiplying 4 $\times$ 4 matrices A and B on 64 processes. The shaded processes in part (c) store elements of the first row of A and the shaded processes in part (d) store elements of the first column of B.

Communication Steps

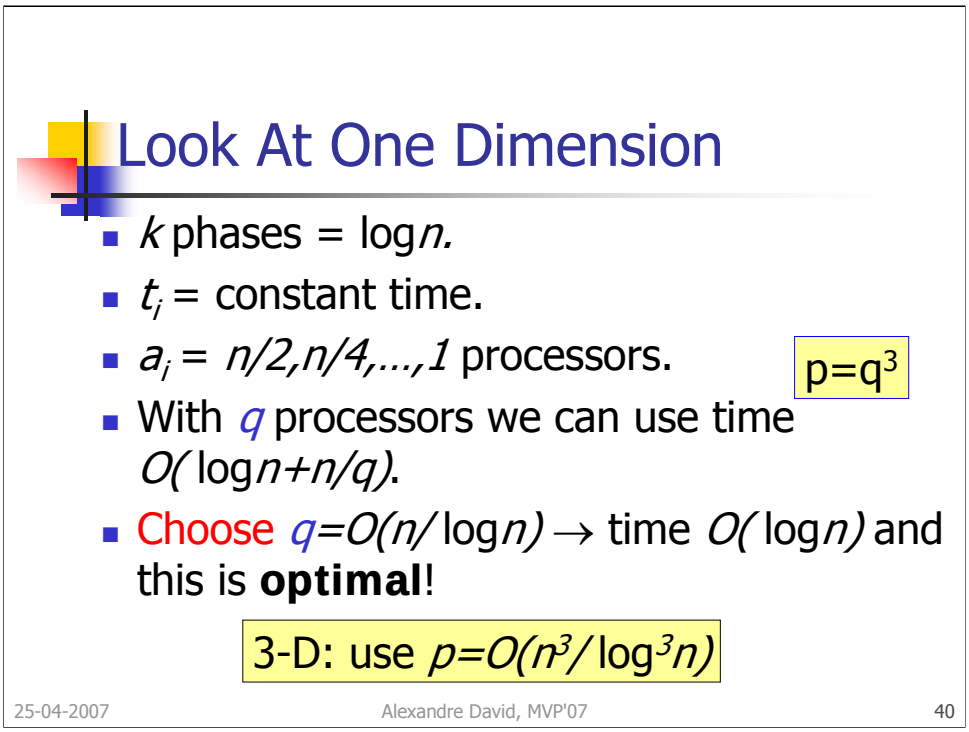
- Move the columns of A & rows of B.
- One-to-all broadcast along j & i axis.
- All-to-one reduction (+) along k axis.
- Communication on groups of n processes, in time $\Theta(\log n)$.
- Not cost optimal for n^3 processes.



What it means: same time as the original plus an overhead. If the number of processors increases then we decrease the overhead. The overhead corresponds to simulating the a_i with p. What it **really** means: It is possible to make algorithms optimal with the right amount of processors (provided that $t*p$ has the same order of magnitude of $t_{sequential}$). That gives you a bound on the number of needed processors.

It's a *scheduling* principle to reduce the number of physical processors needed by the algorithm and increase utilization. It does not do miracles.

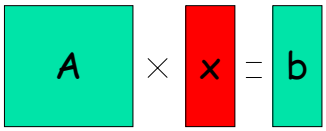
Proof: i'th phase, p processors simulate a_i processors. Each of them simulate at most $\text{ceil}(a_i/p) \leq a_i/p+1$, which consumes time t_i at a constant factor for each of them.



Note: *n* is a power of 2 to simplify. Recall the definition of optimality to conclude that it is optimal indeed. This does not gives us an implementation, but almost.

Divide and conquer same as compress and iterate for the exercise.

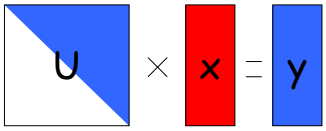
Systems of Linear Equations



$$a_{0,0}x_0+a_{0,1}x_1+\ldots+a_{0,n-1}x_{n-1}=b_0,$$
$$\ldots$$
$$a_{n-1,0}x_0+a_{n-1,1}x_1+\ldots+a_{n-1,n-1}x_{n-1}=b_{n-1}$$

Solving Systems of Linear Equations

- Step 1: Reduce the original system to



- Step2:
  Solve & back-substitute from x_{n-1} to x_0.

Technical Issues

- Non singular matrices.
- Numerical precision (is the solution numerically stable) $\rightarrow$ permute columns.
  - In particular no division by zero, thanks.
  - Procedure known as Gaussian elimination with partial pivoting.

Gaussian Elimination

```
1.   procedure GAUSSIAN_ELIMINATION (A, b, y)
2.   begin
3.       for k := 0 to n − 1 do                    /* Outer loop */
4.       begin
5.           for j := k + 1 to n − 1 do
6.               A[k, j] := A[k, j]/A[k, k];   /* Division step */
7.           y[k] := b[k]/A[k, k];
8.           A[k, k] := 1;
9.           for i := k + 1 to n − 1 do
10.          begin
11.              for j := k + 1 to n − 1 do
12.                  A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.              b[i] := b[i] − A[i, k] × y[k];
14.              A[i, k] := 0;
15.          endfor;                  /* Line 9 */
16.      endfor;                      /* Line 3 */
17.  end GAUSSIAN_ELIMINATION
```

$W=2n^3/3$

Parallel Gaussian Elimination

- 1-D partitioning:
  - 1 process/row.
  - Process i computes A[i,*].
  - Cost (+communication) = $\Theta(n^3 \log n)$ not cost optimal.
- All processes work on the same iteration.
  - k+1 iteration starts when k^{th} iteration is complete.
  - Improve: pipelined/asynchronous version.

Pipelined Version

```
1.  procedure GAUSSIAN_ELIMINATION (A, b, y)
2.  begin
3.      for k := 0 to n − 1 do                    /* Outer loop */
4.      begin
5.          for j := k + 1 to n − 1 do
6.              A[k, j] := A[k, j]/A[k, k];   /* Division step */
7.          y[k] := b[k]/A[k, k];
8.          A[k, k] := 1;
9.          for i := k + 1 to n − 1 do
10.         begin
11.             for j := k + 1 to n − 1 do
12.                 A[i, j] := A[i, j] − A[i, k] × A[k, j]; /* Elimination step */
13.             b[i] := b[i] − A[i, k] × y[k];
14.             A[i, k] := 0;
15.         endfor;              /* Line 9 */
16.     endfor;                  /* Line 3 */
17. end GAUSSIAN_ELIMINATION
```

P_k forwards & does not wait.

P_js forward & do not wait.

Pipelined Gaussian Elimination

- No log$n$ for communication (no broadcast) and the rest of the computations are the same.
- The pipelined version is cost-optimal.
- Fewer processes:
  - Block 1-D partitioning, loss of efficiency due to idle processes (load balance problem).
  - Cyclic 1-D partitioning better.

Gaussian Elimination – 2-D Partitioning

- Similar as before.
- Pipelined version cost-optimal.
- More scalable than 1-D.

Finally Back-Substitution

```
1.  procedure BACK_SUBSTITUTION (U, x, y)
2.  begin
3.      for k := n − 1 downto 0 do   /* Main loop */
4.          begin
5.              x[k] := y[k];
6.              for i := k − 1 downto 0 do
7.                  y[i] := y[i] − x[k] × U[i, k];
8.          endfor;
9.  end BACK_SUBSTITUTION
```

Algorithm 8.5 A serial algorithm for back-substit
entries of the principal diagonal equal to one, and all

Intrinsically serial algorithm.
Pipelined parallel version not cost optimal.
Does not matter because of lower order of magnitude.