HUST-USYD Summer School on Parallel Programming Practice – Lecture 3

Bing Bing Zhou (bing.zhou@sydney.edu.au)

School of Computer Science, University of Sydney

THE UNIVERSITY OF SYDNEY

Outline

- Review of Homework 1
- Speedup and efficiency
- Amdahl's law and overheads
- General design process
  - Machine independent part
    - Partitioning, Communication/synchronization
    - Task dependency graph
    - Math associative law
  - Machine dependent part
    - Assignment and load balancing
- Lab exercise: Gaussian elimination with partial pivoting

Speedup and Efficiency

- Parallel computing is for high performance – high speed
- Speedup of a parallel algorithm is a measure of relative performance improvement over sequential algorithms for solving a given problem
  - Defined as the ratio of the compute time of a fastest sequential algorithm over the time of a parallel algorithm
- Let T_s be the compute time using a single processor and T_p be the time using p processors. The speedup is then defined as

$$S = \frac{T_s}{T_p}$$

- Efficiency of a parallel algorithm is the proportion of processors usefully utilized by the computation and defined as

$$E = \frac{S}{p} = \frac{T_s}{pT_p}$$

Speedup and Efficiency

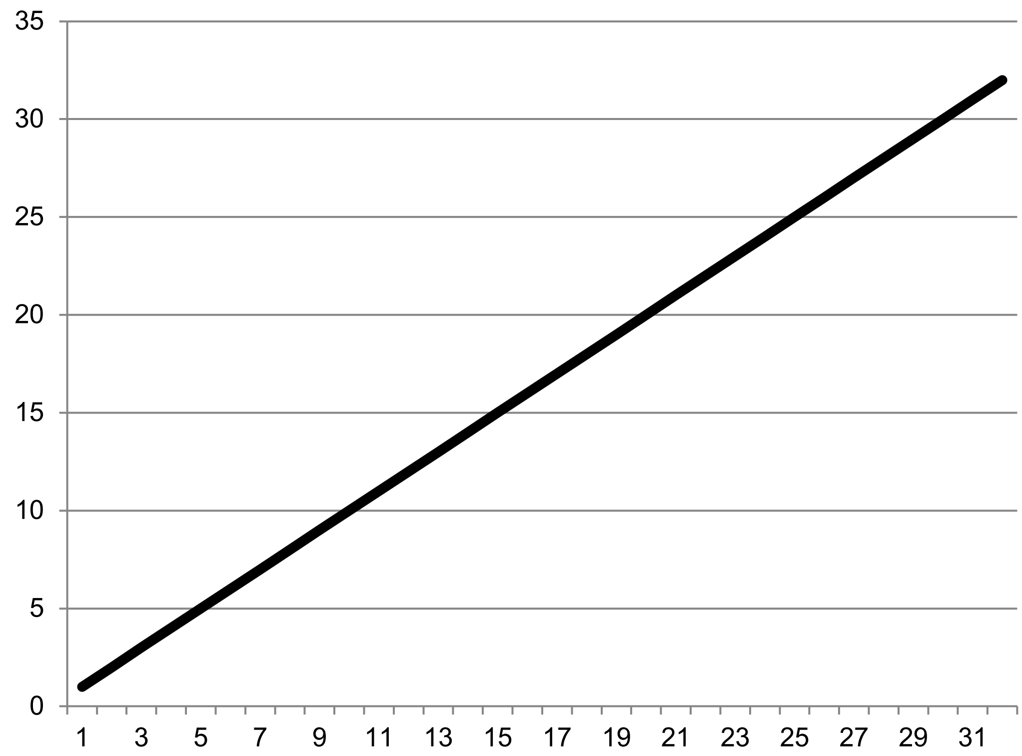
- When using p processors to solve a problem, do we expect p speedup, i.e., high efficiency?
  - Probably not!
  - Overheads in addition to the computation will be introduced in most parallel programs and they include
    - Process/thread communication or synchronization
    - Workload imbalance among available processors/threads
    - Extra work introduced to manage the computation and increase parallelism
    - …
- Let $T_o = pT_p - T_s$ be the total overhead. Then

$$E = \frac{T_s}{pT_p} = \frac{T_s}{T_o + T_s} = \frac{1}{1 + \frac{T_o}{T_s}}$$

- $0 \leq E \leq 1$ and E will be small when T_o is large

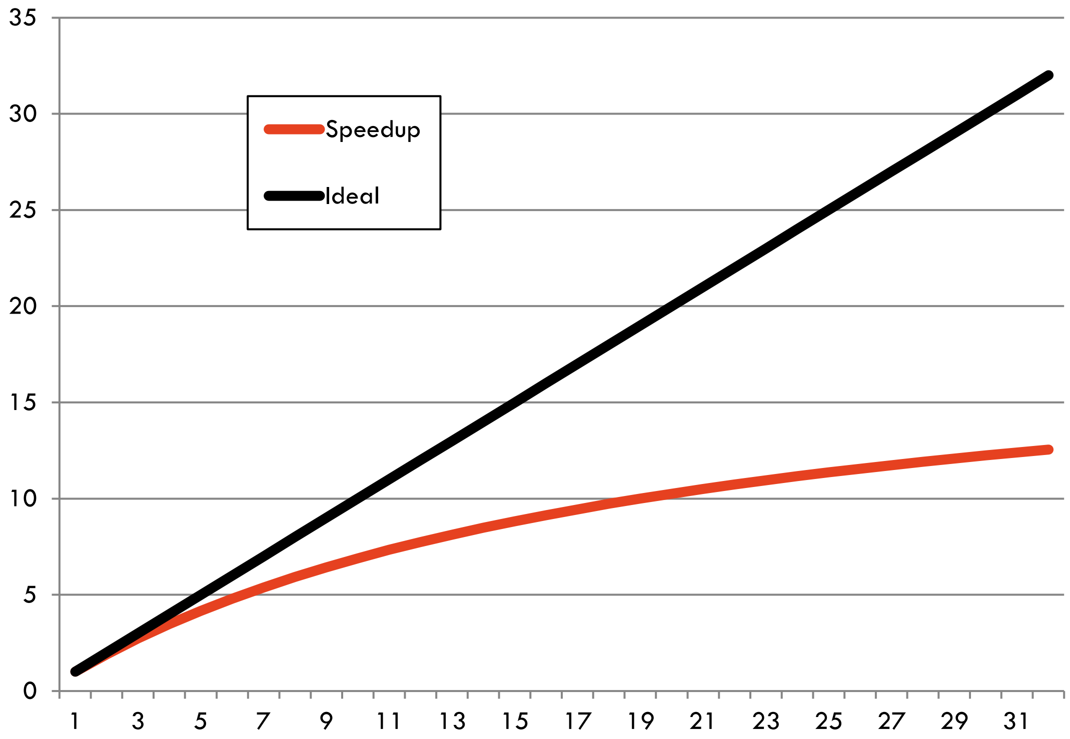
Ideal Speedup

- Speedup: $S = \frac{T_s}{T_p}$
- Ideally, we expect the speedup is increased by a factor of p when using p processors



Actual Speedup

– In practice, the actual speedup for solving a problem is often smaller than the ideal speedup and worse as the number of processors increases



Amdahl's Law

- Amdahl's law is used to predict the theoretical speedup when using multiple processors for parallel computing
- It shows that the serial parts of a parallel program impose a hard limit on potential speedup
- Assume the total amount of operations for solving a problem can be divided into two parts:
  - One part β is purely sequential
  - The other part $1-\beta$ is perfectly parallelizable
- The parallel time using p processors will be

$$T_p = \beta T_s + (1-\beta)T_s/p$$

Amdahl's Law

- The speedup is

$$S = \frac{T_s}{T_p} = \frac{T_s}{\beta T_s + (1-\beta)T_s/p} = \frac{p}{1+\beta(p-1)}$$

- When p is very large, we have

$$S \rightarrow \frac{1}{\beta}$$

- Then β becomes a limiting factor
  - E.g., when only 5% of the program are sequential , i.e., $\beta = 0.05$, no matter how many processors are used, the speedup cannot be greater than 20!

Overheads

- We know

$$E = \frac{T_s}{pT_p} = \frac{T_s}{T_o + T_s} = \frac{1}{1 + \frac{T_o}{T_s}}$$

- Main cause of Inefficiency in addition to poor single processor performance is overheads
- **Therefore, we must try the best to minimize all unnecessary overheads in designing and implementing parallel algorithms**

Designing Parallel Algorithms

- How a problem specification is translated into an algorithm that displays concurrency, scalability, and locality
- Parallel algorithm design is not easily reduced to simple recipes
- Rather, it requires the sort of integrative thought that is commonly referred to as ``creativity“
- May need new ideas that have not been studied before

General Design Process

- General design process involves two stages:
  - Machine independent stage
    - Recognize opportunities for parallel execution based on the characteristics of a given problem
      - Partitioning: divide a large task into multiple smaller ones which can be executed concurrently
      - Communication/synchronization: coordinate the execution of concurrent tasks and establish appropriate communication/synchronization structures
  - Machine dependent stage
    - Assignment: reorganize tasks and assign them to multiple processes/threads based on characteristics of a specific machine
      - minimize overheads and balance workload across processors

Partitioning

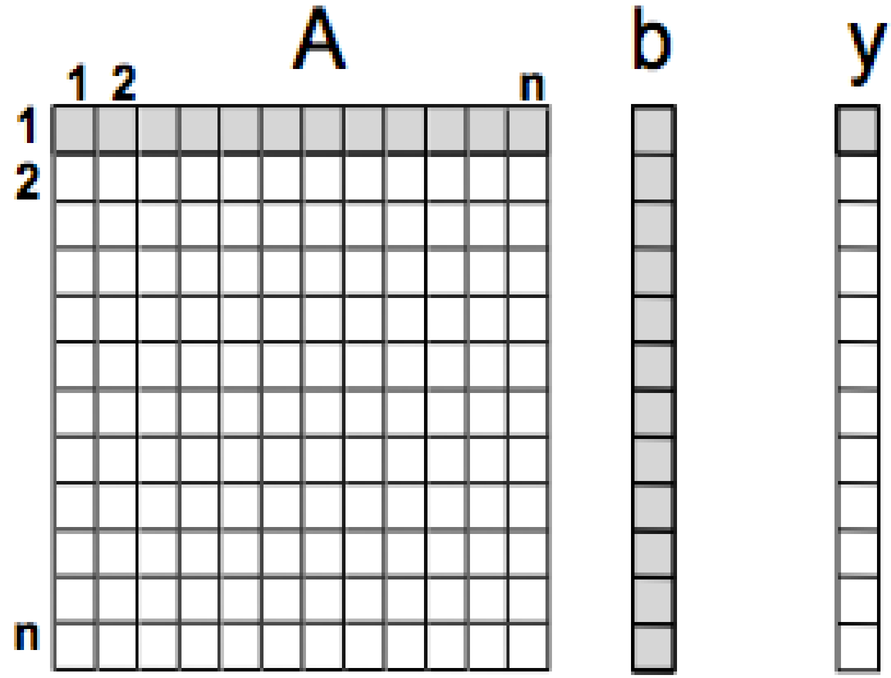
- Expose opportunities for parallel execution
- The focus is on defining a large number of small tasks, each of which consists of the computation and the data on which this computation operates
- Typical types of partitioning:
  - Task partitioning
    - Divide the computation into pieces first
    - Then associate data with the computations
  - Data partitioning
    - Divide data into pieces first
    - Then associate computations with the data
- Which one should be applied depends on the actual problem

Communication/Synchronization

- The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently
  - Data must then be transferred between tasks so as to allow computation to proceed
- Communication/synchronization is then required to manage the data transfer and/or coordinate the execution of tasks
- Organizing communication in an efficient manner can be challenging

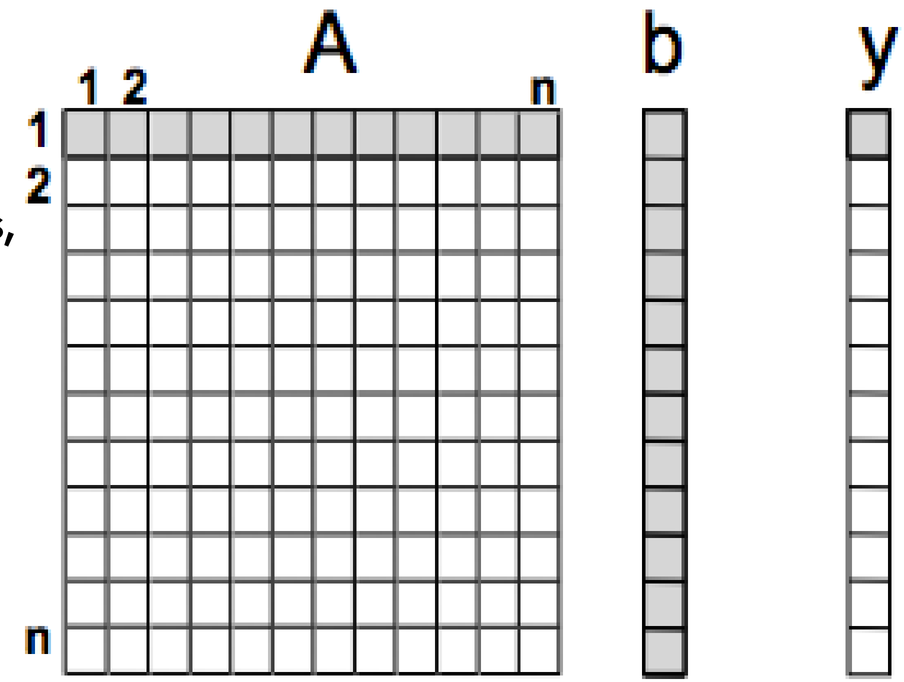
Example: Matrix-vector multiplication

- Task partitioning:
  - Partition vector y, one element per task
  - Computing each element of y involves one row of A and vector b
- Observations:
  - Task size is uniform
  - No dependences between tasks
  - Embarrassingly parallel



Example: Matrix-vector multiplication

- Data partitioning:
  - We can partition matrix A and vector b and then associate one multiplication with each pair of data items, one from A and one from b as a task
  - Note the results from these small tasks are just intermediate results, which will be considered as new data for further new task construction
  - This is equivalent to further partition each inner product into n smaller tasks
- Observations:
  - Task size is uniform
  - Good for fine-grained parallelism
  - But dependences between tasks for inner product of two vectors!
    - Require comm/sync



Parallel Structure of Algorithm

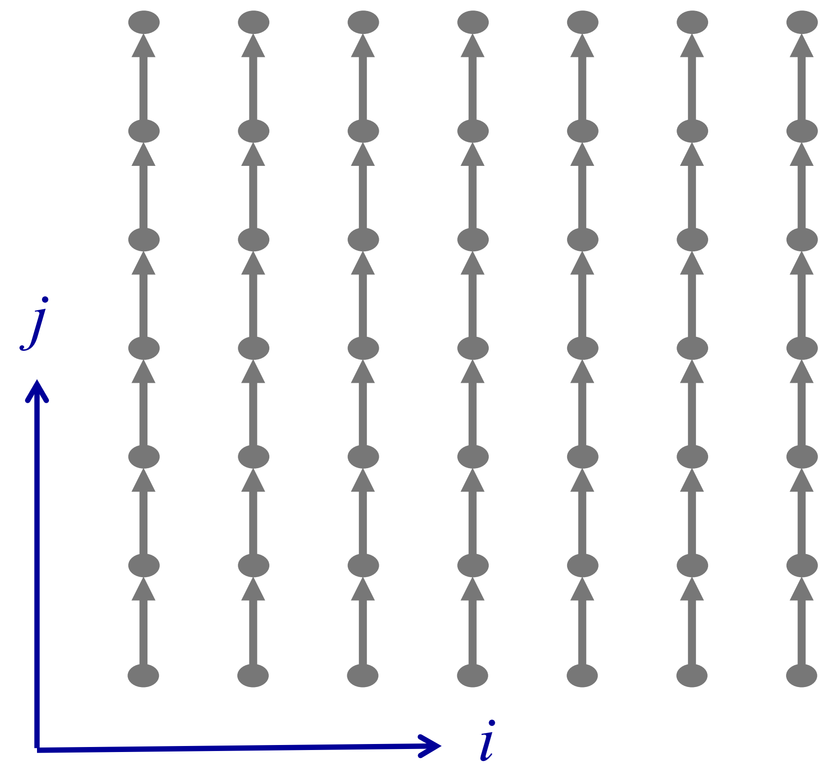
- To design parallel algorithms for solving a given problem, typically the first step is to detect parallel structures of the sequential algorithms
- If the algorithm's parallel structure is determined, many subsequent decisions become obvious
- Task-dependency graph is one of the good techniques to identify program's parallel structures

Task-Dependency Graph

– In a task-dependency graph each node represents a task and the arrows between nodes indicate dependencies between tasks
– The main purpose is to show the parallel structure to demonstrate various properties of the algorithm, e.g.,
  – parallel structure patterns – regularity
  – edge set features - dependency
  – structure dependent on input data – work prediction
– It is often enough to present one small graph
– However, it is challenging to construct the graph for complex algorithms

Task-Dependency Graph

- Matrix-vector multiplication:
  - Each node represents a multiplication of a_{ij} and b_j
  - Arrows indicate sequential additions (data dependency)
  - Each vertical arrow-node line represents an output of dot product y_i



Math Associative Law

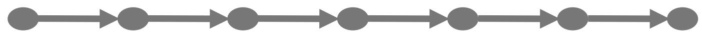
- The parallel structure of an algorithm is an important concept, but it should not be used alone
- To evaluate the parallelism potential of an algorithm, the mathematics behind the algorithm plays an equally important role
- Knowing the mathematical basics of an algorithm can increase the degree of parallelism – very important in practice

Math Associative Law

- Consider summation of n integers: $s = \sum_{i=0}^{n-1} a_i$
- For a typical sequential program:

```
s = 0;
for (i=0; i<n; i++)
    s = s + a[i];
```

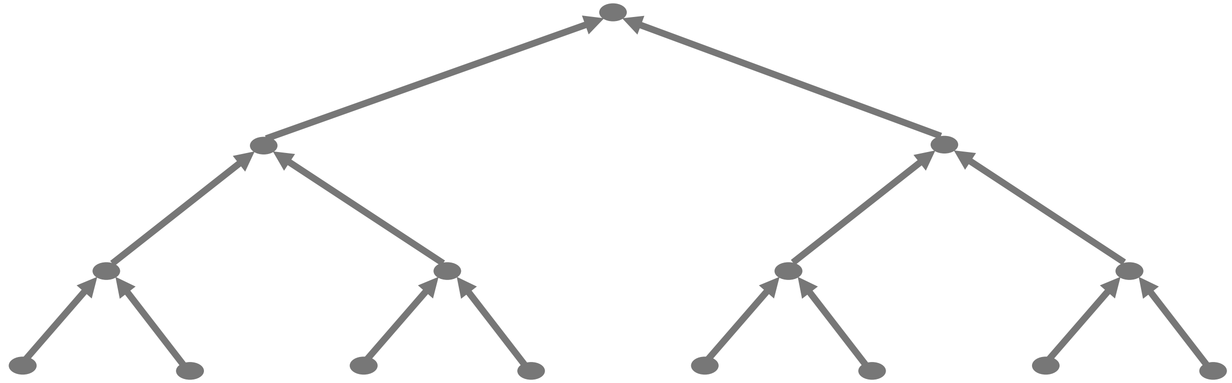
- The structure of the algorithm:



- It is completely sequential in nature!

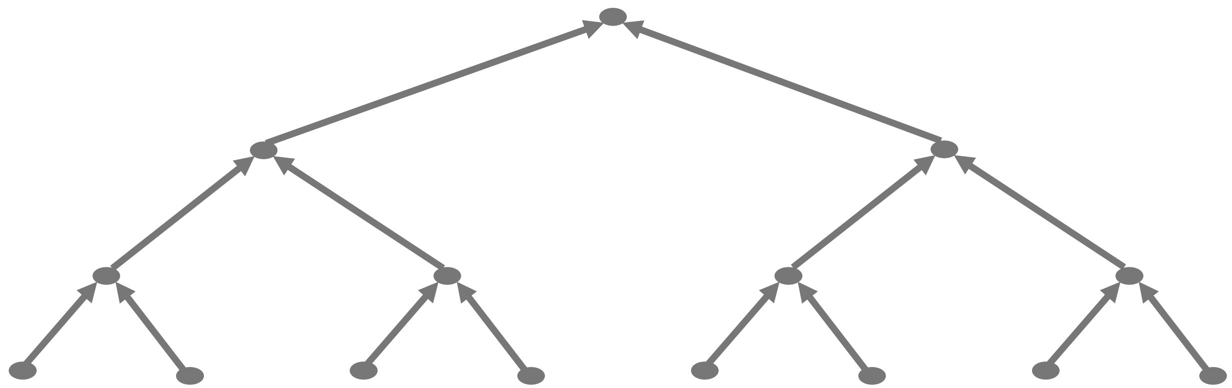
Math Associative Law

- Addition operation obeys the associative law
- For the summation, additions can be done in any order
- We can then have a new algorithm: $\sum_{i=0}^{2n-1} a_i = \sum_{i=0}^{n-1} a_i + \sum_{i=n}^{2n-1} a_i$
  - Pairwise additions recursively
- The parallel structure forms a binary tree
  - Degree of parallelism changes from n to 1



Math Associative Law

- Divide and conquer technique:
  - Divide: partition data into n groups, each having one element
  - Conquer: pairwise addition recursively
- Technique is widely used in practice for reduction operations
- The operator not necessarily addition, any operation which obeys associative law



Assignment

- After the structure and degree of parallelism for a problem are identified, we can construct tasks based on the characteristics of specific machines and assign tasks to processors
- Question: What constitutes a task?
- Three levels of parallelism:
  - Instruction level
    - Fine granularity for ILP
  - Thread level
    - Coarse granularity for shared-memory machines to reduce overhead for synchronization
    - Fine granularity for GPU to make all cores busy
  - Process level
    - Coarse granularity partitioning for distributed-memory machines to reduce communication costs
- The main purpose of assignment is to
  - minimize communication /synchronization costs and
  - balance workload across the processes/threads

Assignment

- The quality of task assignment is directly related to the quality of parallel computation
  - Some problems, e.g., dense matrix computation, regular meshes, can be easily decomposed into a number of tasks of equal size with regular data dependency pattern
    - task assignment relatively easy
  - Some other problems, e.g., sparse matrices, unstructured meshes, and graphs, are more complicated, during the computation we need to consider how to
    - balance workload
    - minimize communication/synchronization overhead
    - redundant computation

Lab exercise: Gaussian Elimination with Unrolling

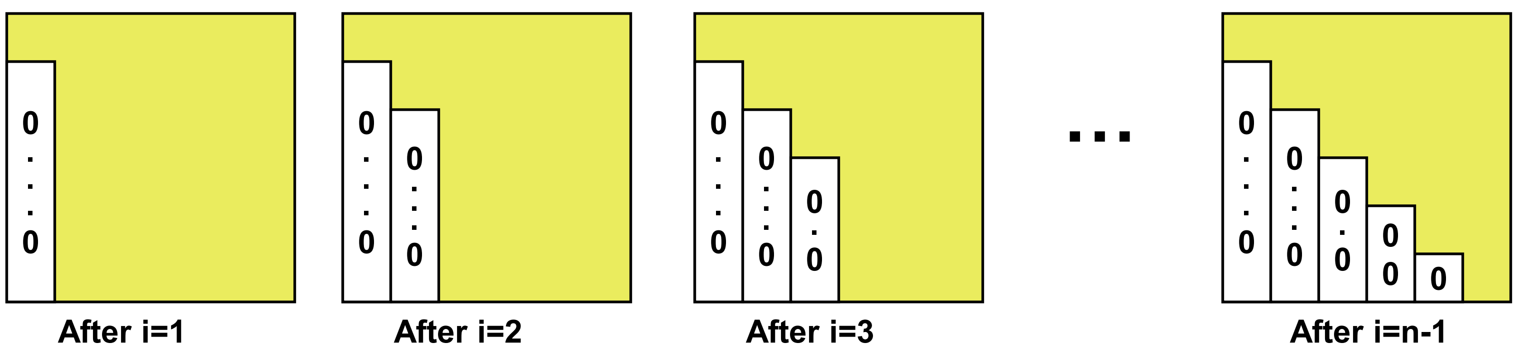
- Revise program gepp_0.c to add loop unrolling with unrolling factor = 4 for general cases, i.e., n may not be divisible by 4
- Test the correctness of your program and check the performance

- If unable to complete, take it home as Homework 2

Gaussian Elimination

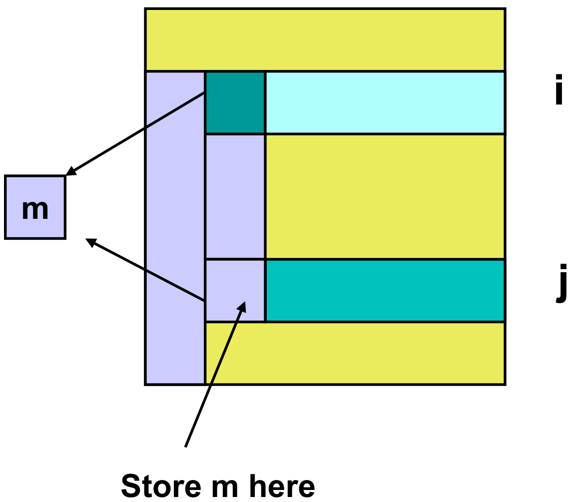
– For a given matrix A of size N by N, add multiples of each row to later rows to make A upper triangular:

```
//for each column I zero it out below the diagonal by adding
//multiples of row i to later rows
for i = 1 to n-1
    // for each row j below row i
    for j = i+1 to n
        // add a multiple of row i to row j
        tmp = A(j,i);
        for k = i to n
            A(j,k) = A(j,k) - (tmp/A(i,i)) * A(i,k)
```



Gaussian Elimination

– Store multipliers m below diagonal in zeroed entries for later use




```
for i = 1 to n-1
    for j = i+1 to n
        A(j,i) = A(j,i)/A(i,i)
        for k = i+1 to n
            A(j,k) = A(j,k) - A(j,i) * A(i,k)
```

– Call the strictly lower triangular matrix of multipliers M, and let L = I+M

– Call the upper triangle of the final matrix U

– *Lemma (LU Factorization):* If the above algorithm terminates (does not divide by zero) then A = L*U

Gaussian Elimination with Partial Pivoting

– A = [0 1] fails completely because can't divide by A(1,1)=0
       [1 0]

– But solving Ax=b should be easy!

– When diagonal A(i,i) is tiny (not just zero), algorithm may terminate but get completely wrong answer
  - Numerical instability
  - Roundoff error is cause

– Cure: Pivot (swap rows of A) so A(i,i) large

Gaussian Elimination with Partial Pivoting

– Partial Pivoting: swap rows so that A(i,i) is largest in column

```
for i = 1 to n-1
    find and record k where |A(k,i)| = max{i ≤ j ≤ n} |A(j,i)|
        … i.e. largest entry in rest of column i
    if |A(k,i)| = 0
        exit with a warning that A is singular, or nearly so
    elseif k ≠ i
        swap rows i and k of A
    end if
    A(i+1:n,i) = A(i+1:n,i) / A(i,i)        … each |quotient| ≤ 1
    A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```


Q&A