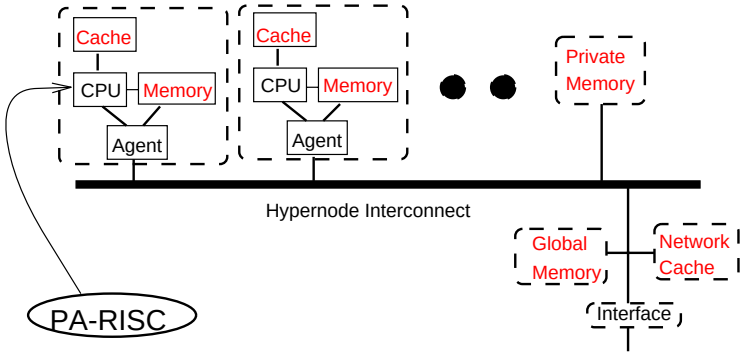
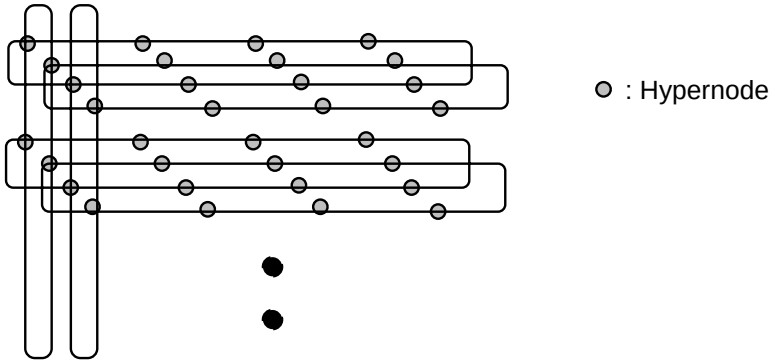
Overview of Parallel Programming

Architectural Issues in Parallel Processing

Convex Exemplar Architecture:



Hypernode



Network of hypernodes

Memory latencies:

Processor cache 10 ns
CPU private memory 500 ns
Hypernode private memory 500 ns
Network cache 500 ns
Interhypernode shared memory 2 microsec

Within hypernode: SMP
Across hypernodes: NUMA

Locality of reference is extremely important!!

Implications for Parallel Programming

- Algorithm must be parallel

  Without adequate parallelism, problem may run slower on parallel machine than on a sequential machine

  eg) solving triangular systems

- Locality of reference is critical

  Wherever possible, code and data should be colocated.

  Dynamic locality of reference is important for cache performance.

- Block transfers of data may be important

  Important if start-up latency of communication is large compared to cost of incremental data transfer

- Load balancing

  Time-of-completion of first and last processors to finish execution should be close (assuming there is adequate parallelism).

- Uniprocessor performance is critical

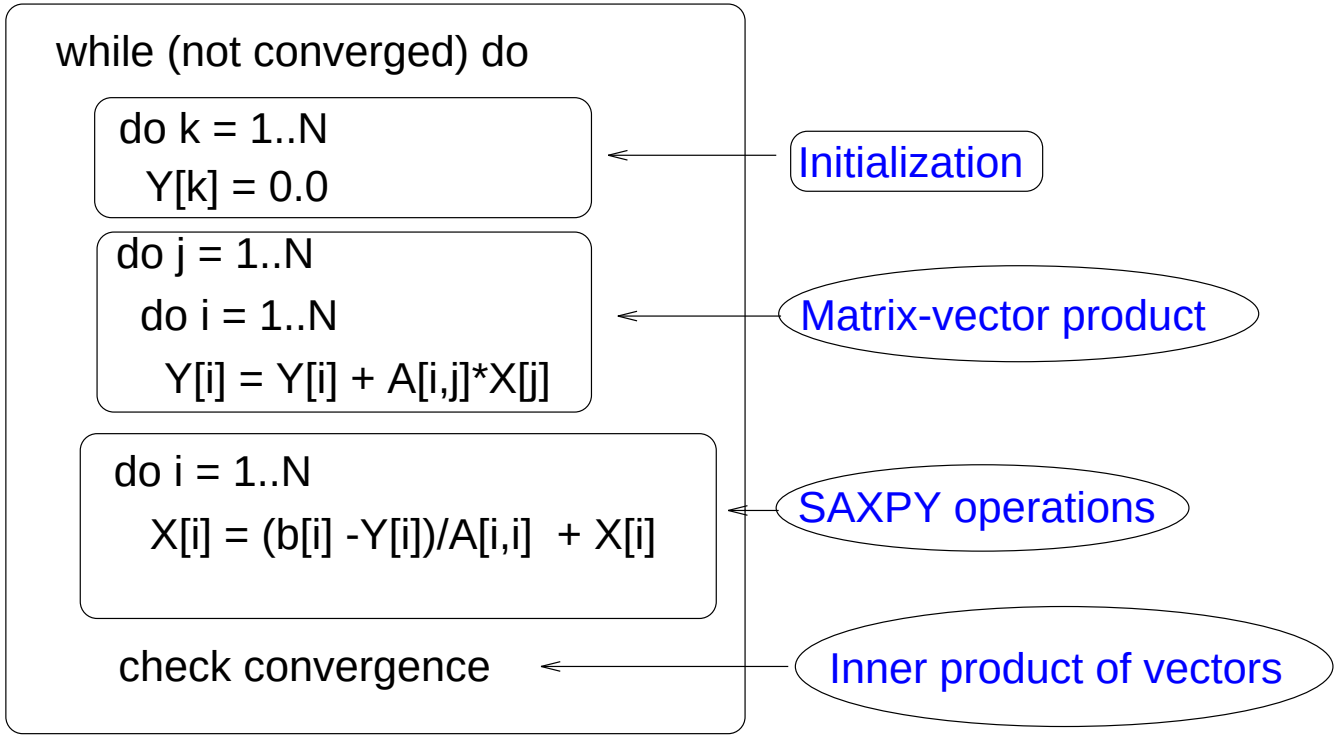
  Processor pipelines should be kept full.

  Data/instruction cache misses must be minimized

Parallelization Example:

Iterative methods for solving linear systems Ax = b

Jacobi method: $M * X_{k+1} = (M - A) * X_k + b$ (M is DIAGONAL(A))



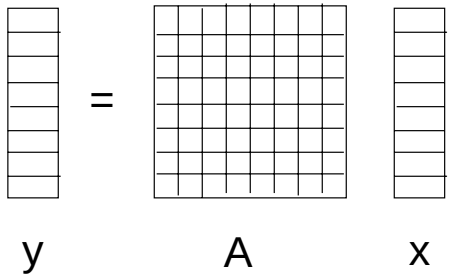
Matrix-vector product: $O(N^2)$ work

SAXPY, Inner product: O(N) work

Most of the time is spent in matrix-vector product.

Where is the parallelism in matrix vector product?

```
do j = 1..N
  do i = 1..N
    Y[i] = Y[i] + A[i,j]*X[j]
```



Each row of the matrix can be multiplied by x in parallel.
(ie., inner loop is a parallel loop)
If addition is assumed to be commutative and associative,
then outer loop is a parallel loop as well.

Question: How do we tell which loops are parallel?

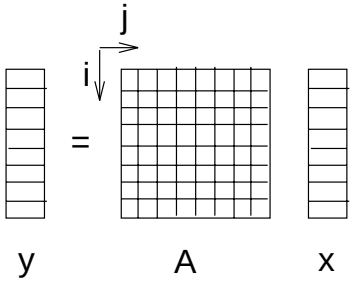
```
do i = 1 ..N
  x(2*i + 1) = ...x(2*i) ....
```

```
do i = 1 ..N
  x(i+1) = ....x(i) ....
```

One of these loops is parallel, the other is sequential!

Answer: Dependence analysis
Theory: Solution of linear equations for integer solutions

Loop Transformations:

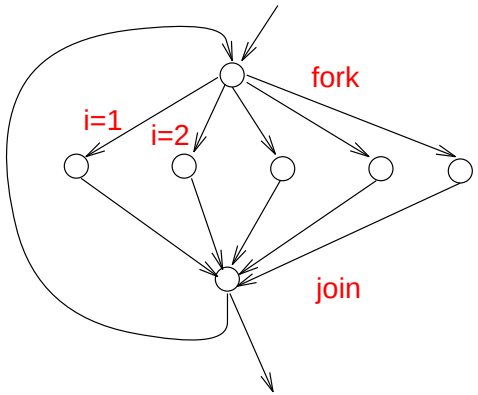


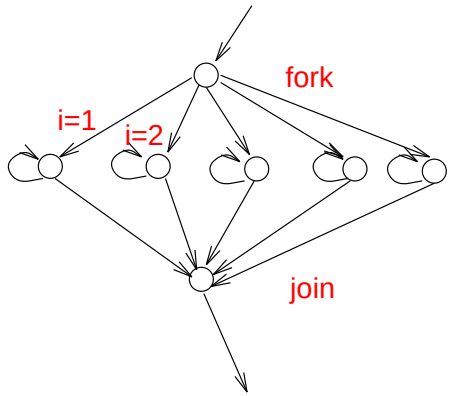

```
do j = 1..N
  doall  i = 1..N
    Y[i] = Y[i] + A[i,j]*X[j]
```

permutation ⟷

```
doall i = 1..N
  do j = 1..N
    Y[i] = Y[i] + A[i,j]*X[j]
```

Which order of loops is best for parallel execution?





If parallel loop is outermost, synchronization is reduced.

Question: What loop transformations are appropriate?

How do we know if a particular transformation is legal?

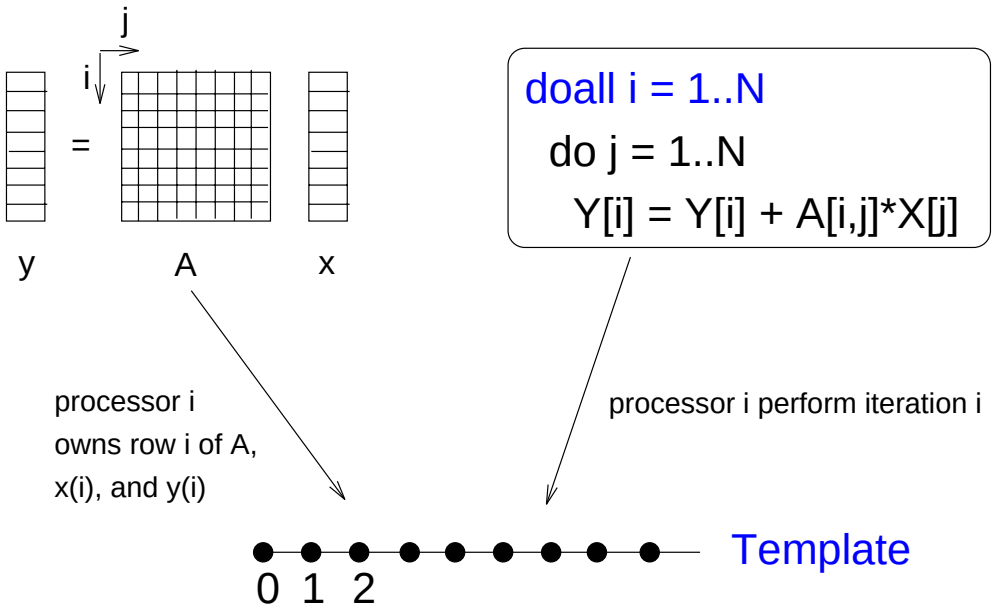
Alignment and Distribution:

How should iterations and data be assigned to processors
so that the data required for an iteration is local wherever possible?

Two steps: Template : Cartesian grid of logical processors (HPF)

Alignment: map from iterations and data arrays to template

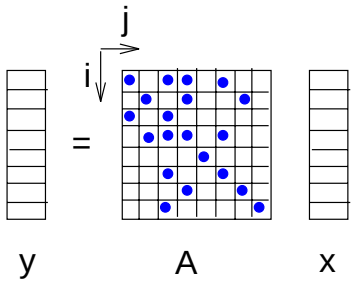
Distribution: map from template points to physical processors



Misaligned references = nonlocal data = communication

Question: How do we specify/deduce alignment?

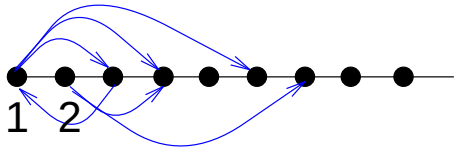
Distribution:




```
doall i = 1..N
  do j = 1..N
    Y[i] = Y[i] + A[i,j]*X[j]
```

processor i
owns row i of A,
x(i), and y(i)

processor i perform iteration i



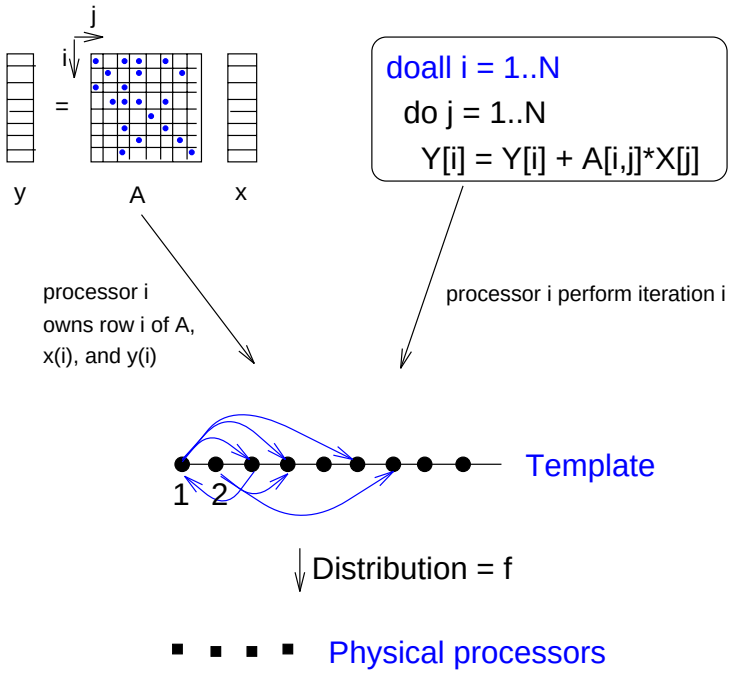
Template

Distribution

Physical processors

Question: How do we specify/deduce distributions?

Code Generation:



- select storage scheme for sparse matrix: Compressed Row Storage
- physical processor p performs iterations in set $f^{-1}(p)$
- communication required for misaligned references
- determination of communication schedule needs to be done just once
  because sparsity structure of A is fixed, and alignment/distribution are fixed
- communication can be blocked

Runtime Systems:

- Efficient communication

- Irregular, adaptive computations: AMR

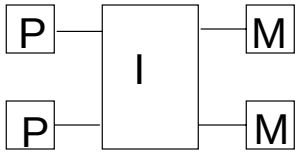
    - shock wave propagating through a medium
    - mesh must be fine near shock wave to capture physics
      but it can be coarse in the far field to avoid unnecessary computation
    - as shock wave propagates, mesh must continuously be refined
      and coarsened

- Runtime system to support repartitioning of iterations and data
- Dynamic load balancing
- Example: PREMA being built by Chrisochoides et al

Logical Abstractions of Multiprocessors

Physical Organization

- Uniform memory access (UMA) machines



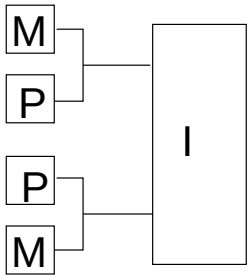
All memory is equally far away from all processors.

Early parallel processors like NYU Ultracomputer

Problem: why go across network for instructions? read-only data?

what about caches?

- Non-uniform memory access (NUMA) machines:



Access to local memory is usually 10-1000 times faster than access to non-local memory

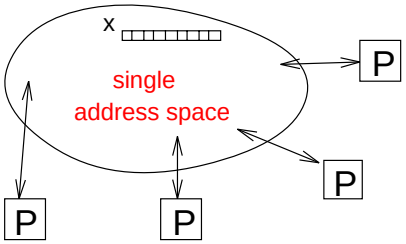
Static and dynamic locality of reference are critical for high performance.

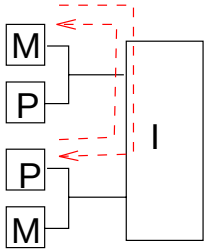
Compiler support? Architectural support?

Bus-based symmetric multiprocessors (SMP's): combine both aspects

Logical Organization

- Shared Memory Model

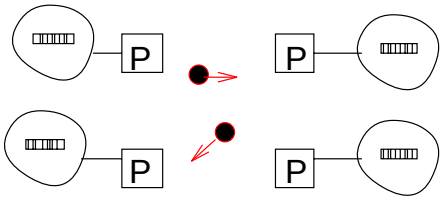


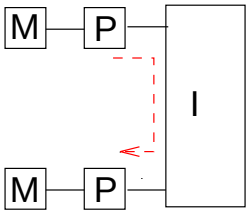


(conceptual picture)

- hardware/systems software provide single address space model to applications programmer
- some systems: distinguish between local and remote references
- communication between processors: read/write shared memory locations: put get

- Distributed Memory Model (Message Passing)





(conceptual picture)

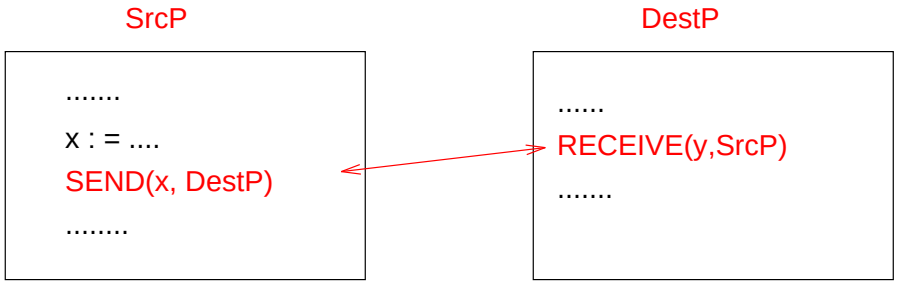
- each processor has its own address space
- communication between processors: messages (like e-mail)
- basic message-passing commands: send receive

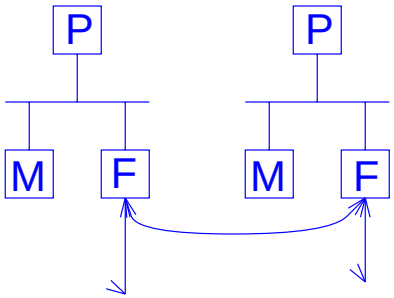
Key difference: In SMM, P1 can access remote memory locations w/o prearranged participation of application program on remote processor

Message Passing

Blocking SEND/RECEIVE : couple data transfer and synchronization

- Sender and receiver rendezvous to exchange data





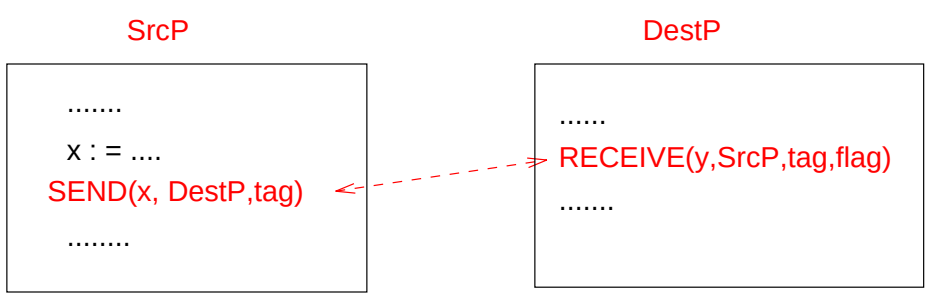
History: Caltech Cosmic Cube

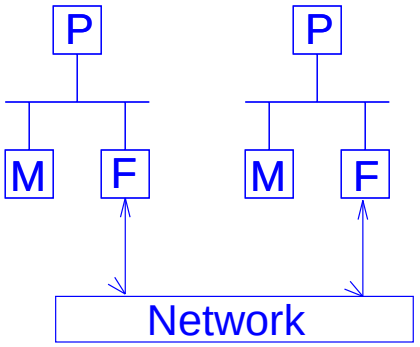
- SrcP field in RECEIVE command permits DestP to select which processor it wants to receive data from
- Implementation:
  - SrcP sends token saying ‘ready to send’
  - DestP returns token saying ‘me too’
  - Data transfer takes place directly between application programs w/o buffering in O/S
- Motivation: Hardware ‘channels’ between processors in early multicomputers
- Problem:
  - sender cannot push data out and move on
  - receiver cannot do other work if data is not available yet

    one possibility: new command TEST(SrcP,flag): is there a message from SrcP?

Overlapping of computation and communication is critical for performance

Non-blocking SEND/RECEIVE : decouple synchronization from data transfer



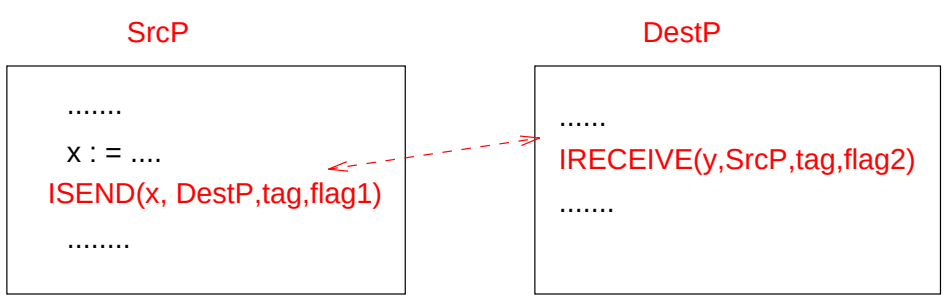


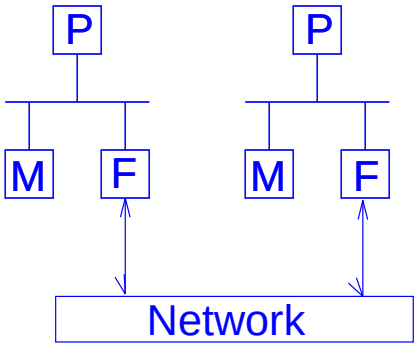
- SrcP can push data out and move on
- Many variation: return to application program when
  - data is out on network?
  - data has been copied into an O/S buffer?
- Tag field on messages permits receiver to receive messages in an order different from order that they were sent by SrcP
- RECEIVE does not block
  - flag is set to true by O/S if data was transfered/false otherwise
- Applications program can test flag and take the right action
- What if DestP has not done a RECEIVE when data arrives from SrcP?
- Data is buffered in O/S buffers at DestP till application program does a RECEIVE

Can we eliminate waiting at SrcP ?

Can we eliminate buffering of data at DestP ?

Asynchronous SEND/RECEIVE





- SEND returns as soon as O/S knows about what needs to be sent
- 'Flag1' set by O/S when data in x has been shipped out
- Application program continues, but must test 'flag1' before overwriting x
- RECEIVE is non-blocking:
    - returns before data arrives
    - tells O/S to place data in 'y' and set 'flag' after data is received
    - 'posting' of information to O/S
    - 'Flag2' is written by O/S and read by application program on DestP
- Eliminates buffering of data in DestP O/S area if IRECEIVE is posted before message arrives at DestP

So far, we have looked at point-to-point communication

Collective communication:

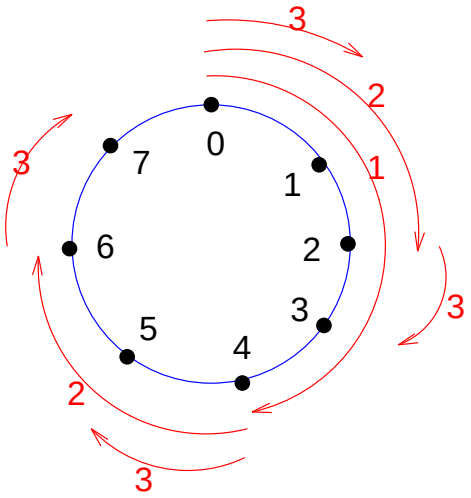
- patterns of group communication that can be implemented more efficiently than through long sequences of send's and receive's

- important ones:

  - one-to-all broadcast
    (eg. A*x implemented by rowwise distribution: all processors need x)
  - all-to-one reduction
    (eg. adding a set of numbers distributed across all processors)
  - all-to-all broadcast
    every processor sends a piece of data to every other processor
  - one-to-all personalized communication
    one processor sends a different piece of data to all other processors
  - all-to-all personalized communication
    each processor does a one-to-all communication

Example: One-to-all broadcast

(intuition: think 'tree')



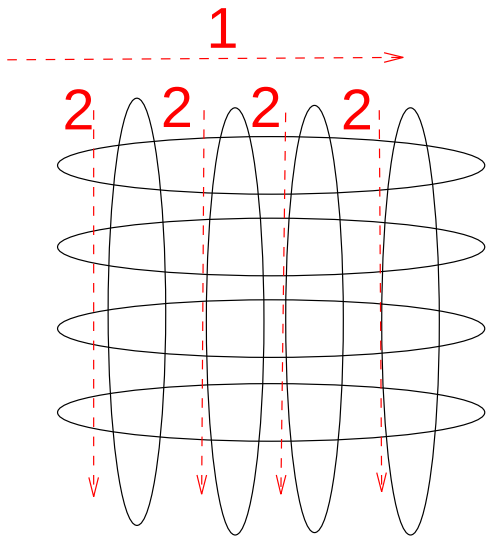
Messages in each phase
do not compete for links

Assuming message size is small, time to send a message = Ts + h*Th
where Ts = overhead at sender/receiver
Th = time per hop

Total time for broadcast = Ts + Th*P/2
+ Ts + Th*P/4
+ .....
= Ts * logP + Th*(P-1)

Reality check: Actually, a k-ary tree makes sense because processor 0 can send many messages by the time processor 4 is ready to participate in broadcast

Other topologies: use the same idea



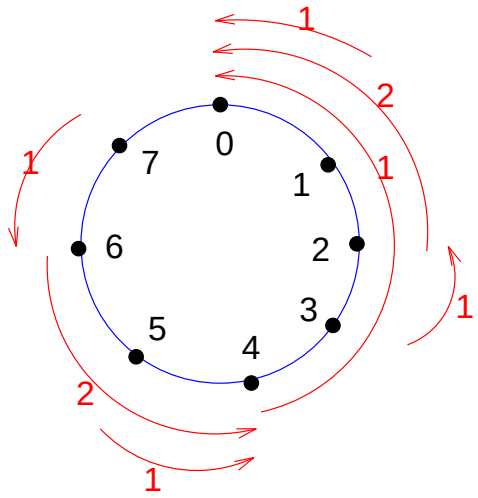
2-D Mesh

Step 1: Broadcast within row of originating processor

Step 2: Broadcast within each column in parallel

Time = Ts logP + 2Th*(sqrt(P) -1)

Example: All-to-one reduction



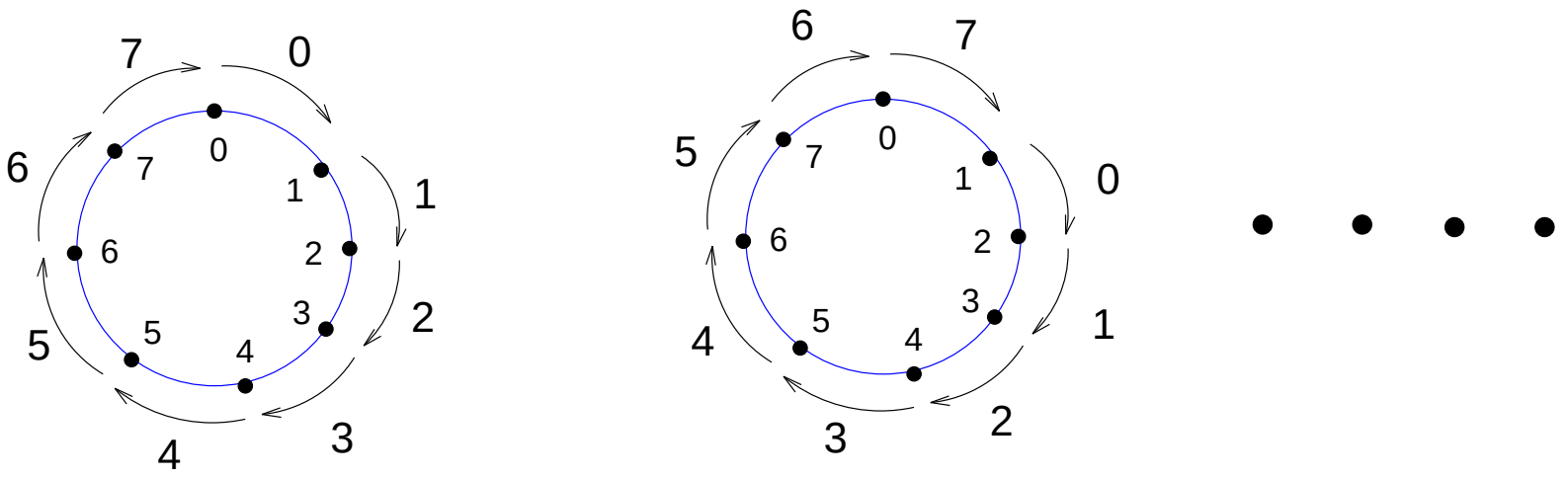
Messages in each phase
do not compete for links

Purpose: apply a commutative and associative operator
(reduction operator) like +,*,AND,OR etc to values
contained in each node

Can be viewed as inverse of one-to-all broadcast
Same time as one-to-all broadcast

Important use: determine when all processors are finished working
(implementation of ‘barrier’)

Example: All-to-all broadcast



- Intuition: cyclic shift register
- Each processor receives a value from one neighbor ,
  stores it away, and sends it to next neighbor in the next phase.
- Total of (P-1) phases to complete all-to-all broadcast

Time = (Ts + Th) *(P-1) assuming message size is small

- Same idea can be applied to meshes as well:
  - first phase, all-to-all broadcast within each row
  - second phase, all-to-all broadcast within each column

Message-passing Programming

MPI: Message-Passing Interface

Goal: Portable Parallel Programming for
Distributed Memory Computers

- Lots of vendors of Distributed Memory Computers:
  IBM,NCube, Intel, CM-5, ...

- Each vendor had its own communication constructs
  => porting programs required changing parallel programs
  even to go from one distributed memory platform to another!

- MPI goal: standardize message passing constructs syntax and semantics

- Mid 1994: MPI-1 standard out and several implementations available (SP-2)

Key MPI Routines we will use:

MPI_INIT : Initialize the MPI System
MPI_COMM_SIZE: Find out how many processes there are
MPI_COMM_RANK: Who am I?
MPI_SEND: Send a message
MPI_RECV: Receive a message
MPI_FINALIZE: Terminate MPI

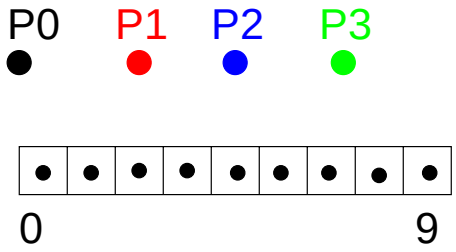
MPI_BCAST: Broadcast

Data Distributions

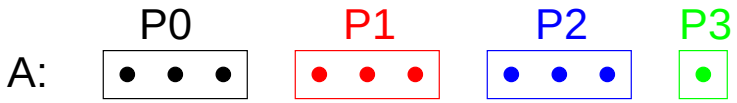
Goal:

- distribute arrays across local memories of parallel machine so that data elements can be accessed in parallel
- Standard distributions for dense arrays: (HPF, Scalapack)
  - block
  - cyclic
  - block cyclic(b)
- Block cyclic distribution subsumes other two

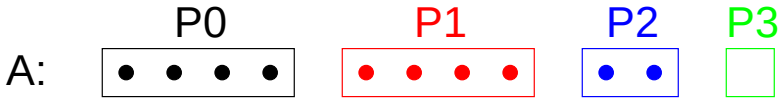
Block:



DISTRIBUTE A(BLOCK)

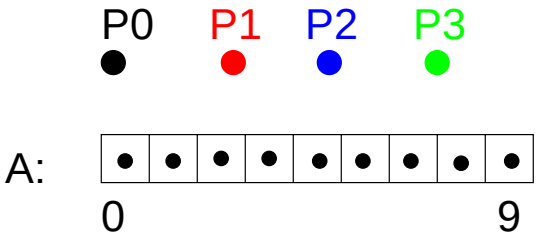


DISTRIBUTE A(BLOCK(4))



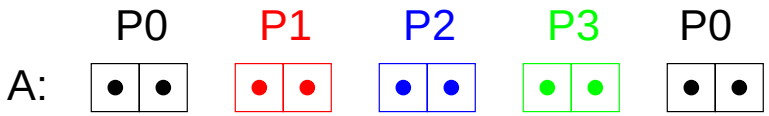
A(i) is mapped to processor $\lfloor i/b \rfloor$ if distribution is BLOCK(b)

Cyclic/Block Cyclic:



DISTRIBUTE A(CYCLIC)

DISTRIBUTE A(CYCLIC(2))

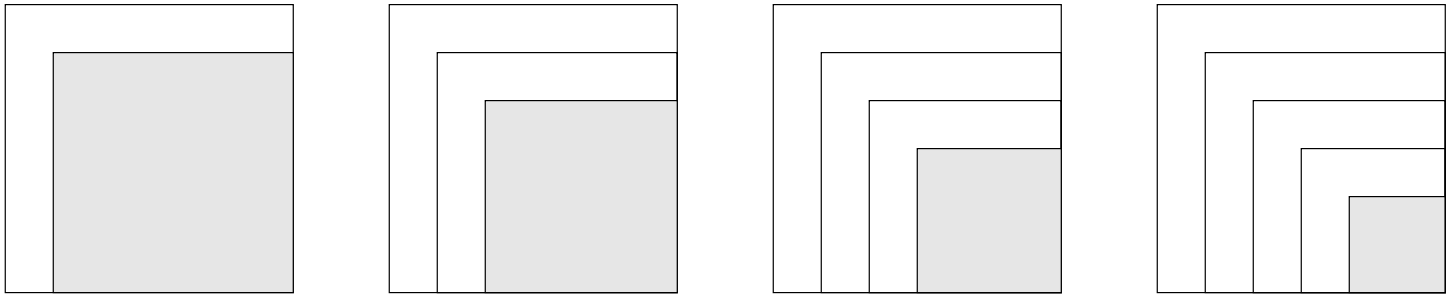


A(i) is mapped to processor $\lfloor i/b \rfloor \bmod P$

if distribution is CYCLIC(b)

Common use of cyclic distribution:

Matrix factorization codes



- BLOCK distribution: small number of processors end up with all the work after a while

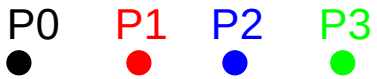
- CYCLIC distribution: better load balance

- BLOCK-CYCLIC: lower communication costs than CYCLIC

Distributions for 2-D Arrays:

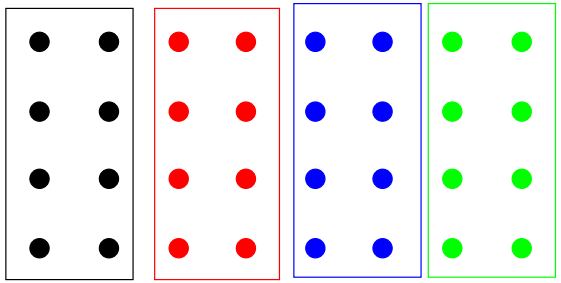
Each dimension can be distributed by

- block
- cyclic
- * : dimension not distributed

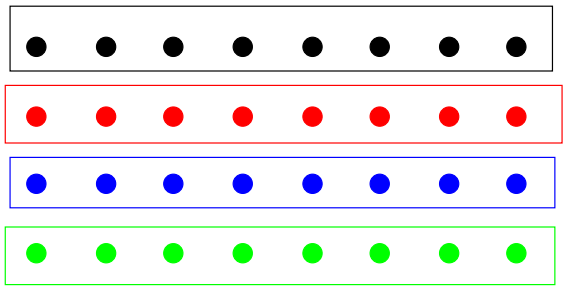


A (4,8)

DISTRIBUTE A (*, BLOCK)



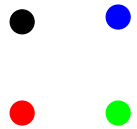
DISTRIBUTE A (CYCLIC,*)



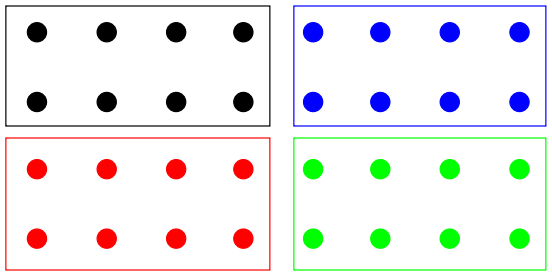
Distributing both dimensions:

- # of array distribution dimensions
  = # of dimensions of processor grid

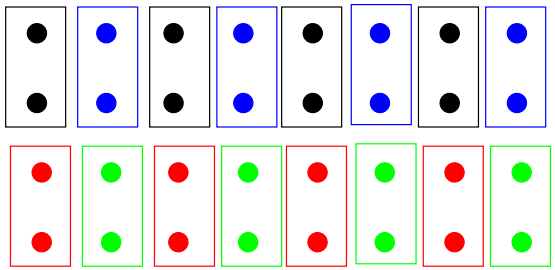
- 2-D processor grid



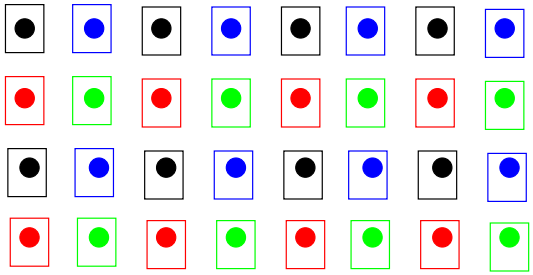
A (4,8)



DISTRIBUTE A (BLOCK, BLOCK)



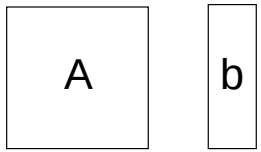
DISTRIBUTE A(BLOCK,CYCLIC)



DISTRIBUTE A (CYCLIC,CYCLIC)

Performance Analysis of Two MVM Programs

In the last lecture, we discussed the following MVM program :



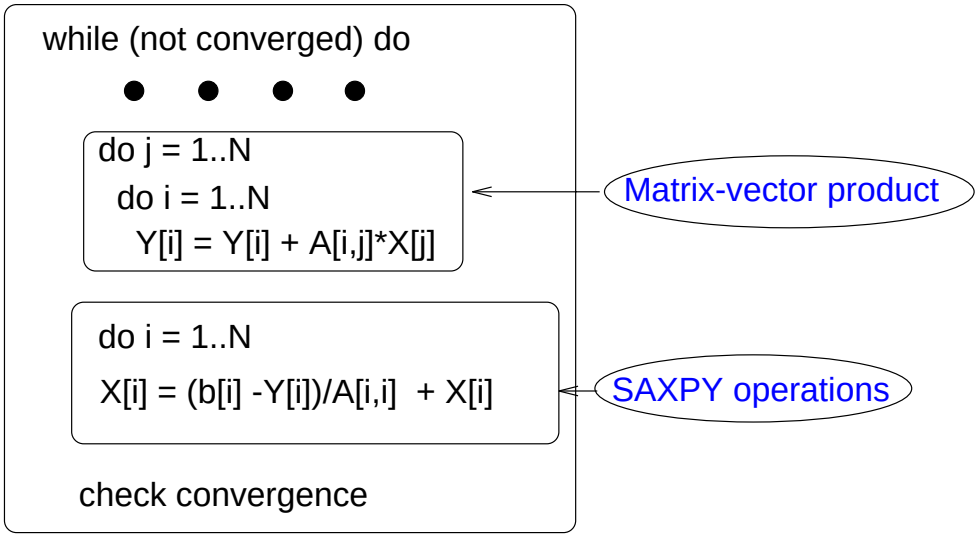
- Style of programming: Master-Slave
  - one master, several slaves
  - master co-ordinates activities of slaves
- Master initially owns all rows of A and vector b
- Master broadcasts vector b to all slaves
- Slaves are self-scheduled
  - each slave comes to master for work
  - master sends a row of matrix to slave
  - slave performs product, returns result and asks for more work

- Why is this program not useful in practice?

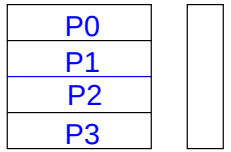
Parallelization Example: MVM

Iterative methods for solving linear systems Ax = b

Jacobi method: $M * X_{k+1} = (M - A) * X_k + b$ (M is DIAGONAL(A))



- Matrix A is usually sparse, not dense



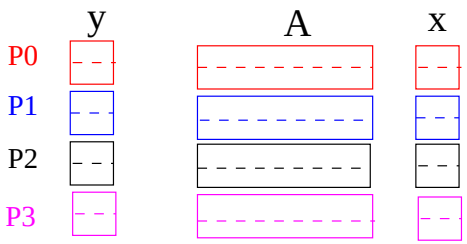
- While loop => MVM is performed many times with same A, many X's
  - => Why ship A from master to slaves in each iteration?
  - => Replace self-scheduling with static assignment of rows to processors
- Caveat: what happens to load balancing?
  - If each processor gets roughly same number of rows, load is balanced provided each rows has roughly same number of non-zeros which is true for dense matrices and most sparse matrices in practice

why?

- Computation of Y is distributed => computation of X_{k+1} can be distributed

=> not efficient to assume that X is broadcast from master every iteration

Matrix-vector Multiply: 1-D Alignment




```
do j = 1..N
  do i = 1..N
    Y[i] = Y[i] + A[i,j]*X[j]
```

- Each processor gets roughly the same number of contiguous rows of A before MVM starts
- If a processor owns rows (r,r+1,r+2,.....) of A, it gets elements (r,r+1,r+2....) of x

Step 1: All-to-all broadcast in which each processor broadcasts its portion of x to all other processors

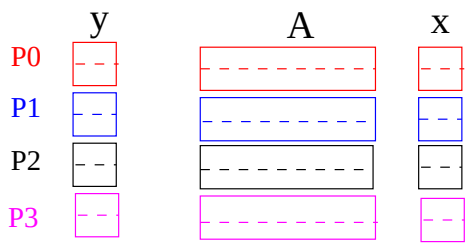
Step 2: Each processor computes the inner product of its rows with x to generate elements (r,r+1,r+2....) of y

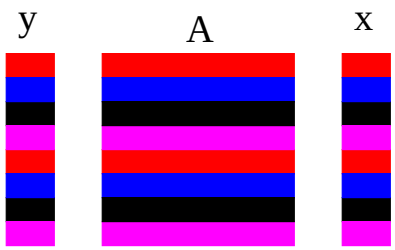
- If this was part of Jacobi iteration, each processor would use its portion of y to compute its portion of x for the next iteration
  (note: next x is mapped as required by Step 1 of MVM)

- Assignment of contiguous rows/columns of a matrix to processors is called 'block distribution'.
- Assignment of rows/columns in round-robin fashion: 'cyclic distribution'

Why did we choose block and not cyclic distribution for our MVM ?

Block vs cyclic Distribution for 1-D MVM:





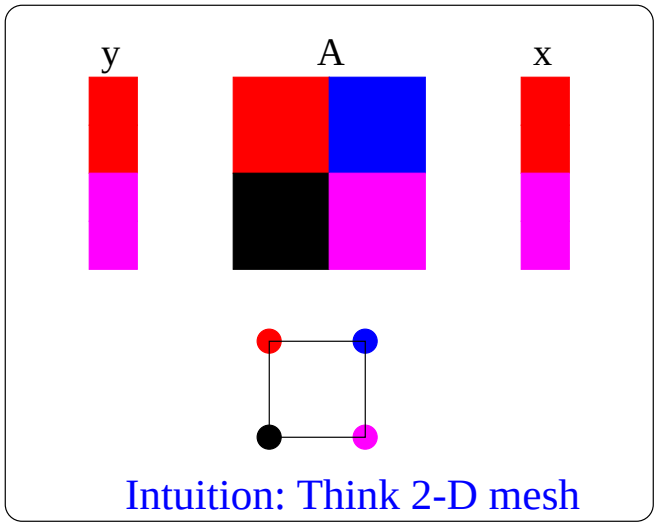
Block Distribution

Cyclic Distribution

- Each processor allocates space for entire x vector.
- It receives messages containing portions of x from all other processors.
- Values received in a message must be placed into storage for x.
- With block distribution of x, values from each message are written into contiguous memory locations (efficient).
- With cyclic distribution of x, values in each message must be written into non-contiguous (but distinct) memory locations (scatter operation).
- Scatters are usually not as efficient as block copies.

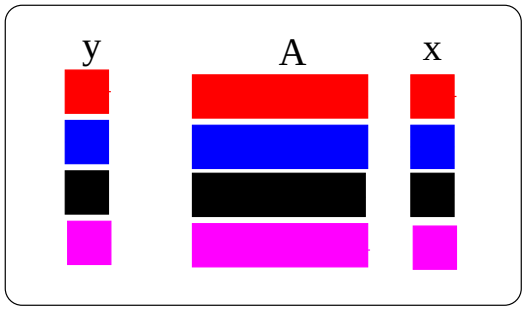
Question: Why not allocate A with cyclic distribution and x with block distribution?

Matrix-vector Multiply: 2-D Alignment



Intuition: Think 2-D mesh

2-D Alignment



1-D Alignment

```
do j = 1..N
  do i = 1..N
    Y[i] = Y[i] + A[i,j]*X[j]
```

- Matrix A is distributed in 2-D blocks to processors
- x is initially distributed to processors on the diagonal of the mesh

Step 1: In each column of mesh, diagonal processor broadcasts its portion of x to all other processors in its column.

Step 2: Each processor performs a mini-MVM with its block of the matrix and the portion of x it has.

Step 3: Processors along each row perform a reduction of their partial sums, using diagonal processors as roots for the reductions.

How do we evaluate different algorithms?

What are good performance models for parallel machines?

Very difficult problem: no clear answers yet.

Speed-up

- most intuitive measure of performance

$$\text{Speed-up}(N,P) = \frac{T_{seq}(N)}{T_{par}(N,P)}$$

T_{seq} : time to solve problem of size N on one processor

T_{par} : time to solve problem of size N on P processors

$$T_{par} = T_{comp} + \underbrace{T_{comm} + T_{synch}}_{\text{parallel overhead}}$$

$$\text{Speed-up}(N,P) = \frac{T_{seq}(N)}{T_{comp} + T_{overhead}(N,P)}$$

$$\text{Parallel efficiency}(N,P) = \text{Speed-up}(N,P)/P$$

(How effectively did we use P processors?)

Purists position:

Sequential time must be measured using 'best sequential algorithm'

Usually, we just use same algorithm on 1 processor, w/o parallel constructs

Sequential machine must have equivalent amount of cache & memory as P processors together

Usually, we do not bother (watch out for superlinear speedups!)

Bounds on speed-up:

$$\text{Speed-up}(N,P) = \frac{T_{seq}(N)}{T_{comp}(N,P) + T_{overhead}(N,P)}$$

- One extreme: Amdahl's Law

Assume that parallel overhead = 0
fraction of program executed in parallel = x
parallel part can be done infinitely fast

Speed-up = 1/ (1 -x)

=> even if 90% of program is parallel and processors are very fast, speed-up is only 10!

Good speed-up requires parallelization of very large proportion of program.

- Other extreme: communication/computation ratio

Assume that program is completely parallelized & perfectly load balanced

$$\text{Speed-up}(N,P) = \frac{T_{seq}(N)}{\frac{T_{seq}(N)}{P} + T_{overhead}(N,P)} = \frac{P}{1 + P * \frac{T_{overhead}(N,P)}{T_{seq}(N)}}$$

Communication-to-computation ratio:

$$\text{Speed-up}(N,P) = \frac{P}{1 + P * \frac{T_{overhead}(N,P)}{T_{seq}(N)}}$$

What happens to speed-up as N and P vary?

- If P is fixed and N (problem size) increases, speed-up usually increases.
  (some people call such algorithms 'scalable')
  Quick check: look at communication-to-computation ratio
  = volume of communication(N)/amount of computation(N)
- If N is fixed and P increases, parallel efficiency usually goes down.

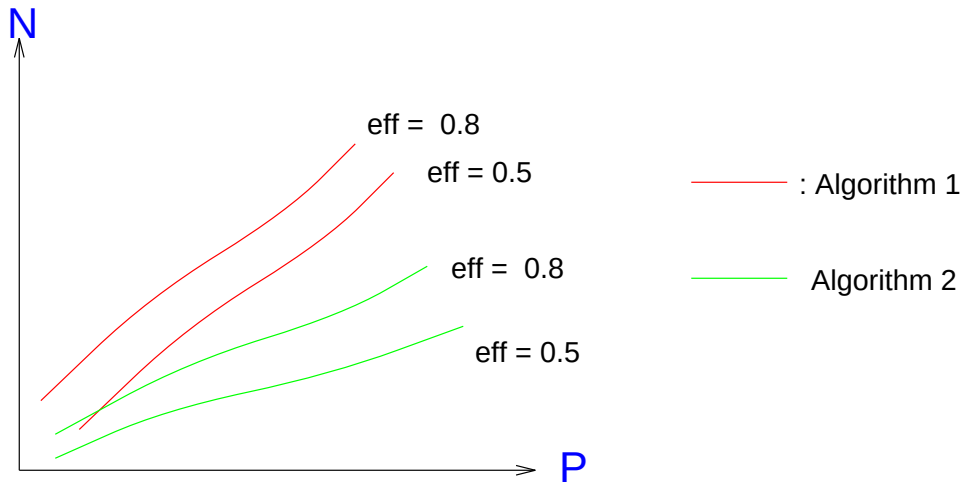
Examples:

- MVM : 1-D : Vector of size N is broadcast to P processors
  => communication volume = O(N*P)
  Computation = $O(N^2)$
  => Communication to computation ratio varies as P/N
  2-D: Communication volume = O(N* sqrt(P))
  => Communication to computation ratio varies as sqrt(P)/N
  Conclusion: 2-D scales better than 1-D

Iso-efficiency Curves:

$$\text{Efficiency}(N,P) = \frac{1}{1 + P * \frac{T_{overhead}(N,P)}{T_{seq}(N)}}$$

To keep parallel efficiency the same, how does problem size have to increase as the number of processors increases?



Iso-efficiency Curves for Two Algorithms

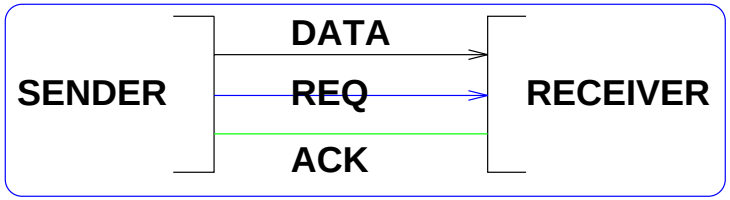
We can also answer questions like: If problem size is fixed, what is the maximum number of processors we can use and still have efficiency > e ?

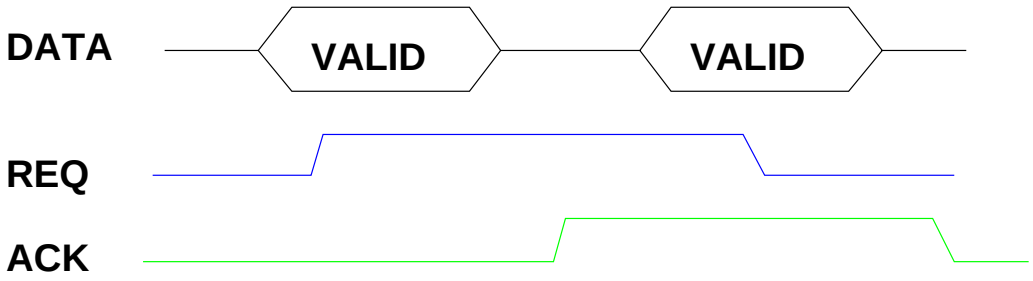
More detailed analysis:

Model communication time

Transmission

Simple protocol: use REQ/ACK wires





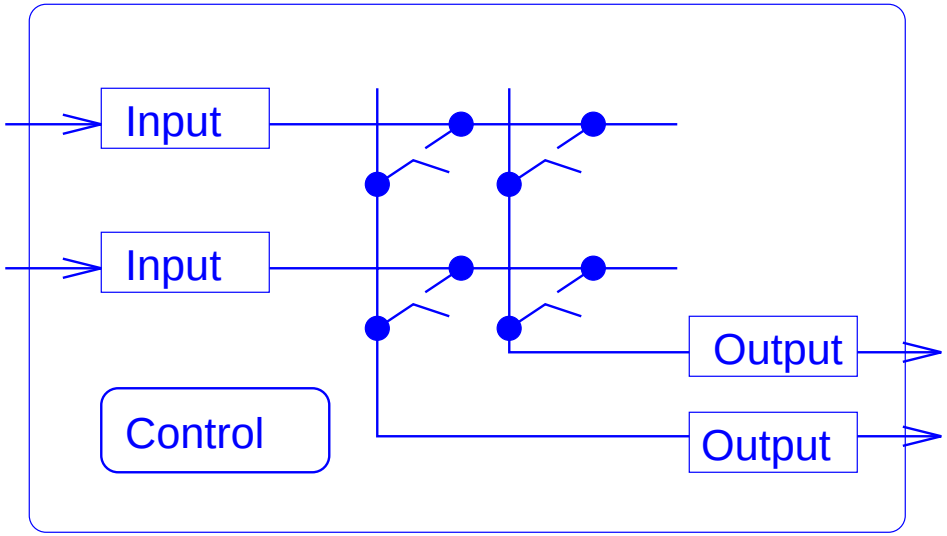
SENDER: sends if REQ = ACK & makes REQ = not(ACK)
RECEIVER: receives if REQ is not(ACK)

Other protocols: encode REQ with DATA

Delay on wire: Depends on RC time constant
Time constant affected by length of wire

Switches

Small cross-bars

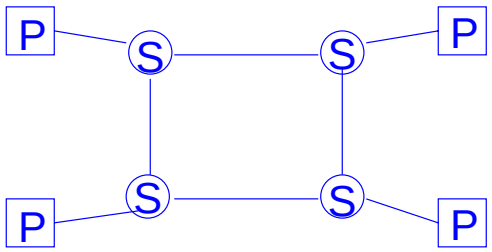


2x2 Cross-bar

As long as two or more inputs do not contend for same output, all inputs can be routes to their desired outputs.

Switch latency in modern parallel computers: < 100 nanosecs

Packetization



Message: unit of data transfer visible to programmer

Circuit switching:

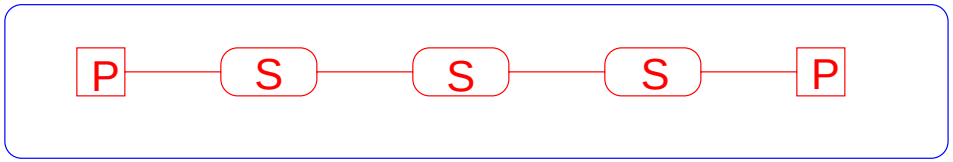
- establish end-to-end connection
- send message
- break connection down

Problems: - short messages

- long messages block out other traffic

Packet switching:

- break message into packets
- each packet travels independently through network
- message is reassembled at destination
- no end-to-end connection is made before data transmission



Store & Forward Networks

- Message is buffered at each switch
- Problem: excessive latency

$$\text{Latency} = L/W * T_{hop} * n$$

where L = size of packet
W = width of channel
n = number of hops from source to destination
T_{hop} = time per hop for W bits

Wormhole Routing

- Divide a packet into ‘flits’: unit of transfer between stages
- All flits in packet follows the same route, but flit transmission is pipelined
- Combines features of circuit and packet switching

$$\text{Latency} = (L/W + n) * T_{hop} = (L * T_{w} + n * T_{hop})$$

These are ‘transmission latencies’ (not counting s/w overhead at two ends).

Reducing Communication Latencies

Good because
- program spend less time waiting for data
- compiler needs to worry less about reducing communication

Reducing/alleviating latency:

- Applications program: should try to overlap communication with computation: send data out as soon as possible, use asynchronous receives to reduce buffering overheads,....

- O/S: reduce the software overhead at sender/receiver

- Hardware: design network to minimize time-of-flight

Let us look at reducing time-of-flight

- if all wires and switches are identical,

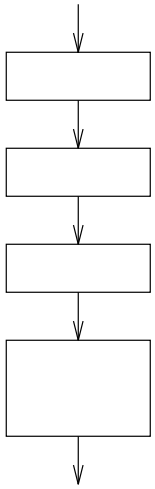
Expected time of flight = T_{w+s} * Expected number of hops

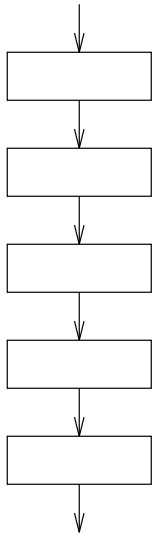
where T_{w+s} = delay through 1 wire + 1 switch

=> Goal of network topology selection is to minimize expected number of hops.

Or is it?

An intuitive picture:





Clock cycle = 2 ticks

Time to go from input to output

= 2 * 4 = 8 ticks

Clock cycle = 1 tick

Time to go from input to output

= 1 * 5 = 5 ticks

Conclusion: Pipeline latency is a function not just of the number of stages but also of clock speed.

Clock speed is determined by slowest stage.

Conclusions:

- It is important to take wire lengths into account in latency estimates.

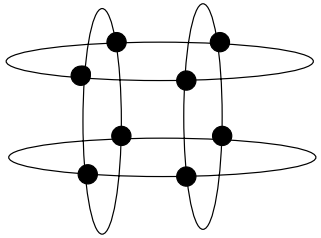
- Higher dimensional wires are longer than lower dimensional wires when embedded into 2/3 dimensions.

- If the length of the longest wire affects latency, higher dimensional networks are not necessarily better.

- One fix: permit several bits at a time to be ‘in-flight’ on a wire => treat wire itself as a pipeline.

Detailed Scalability Analysis of MVM:

2-D Alignment on mesh:



- Column broadcast of x

$$\text{Time} = \left(T_s + \frac{n}{\sqrt{P}} * T_w + \frac{\sqrt{P} * T_h}{2}\right) + \left(T_s + \frac{n}{\sqrt{P}} * T_w + \frac{\sqrt{P} * T_h}{4}\right) + \bullet\ \bullet\ \bullet\ \bullet\ \bullet$$

log(sqrt(P)) phases

$$= \left(T_s + \frac{n}{\sqrt{P}} * T_w\right) \log(\sqrt{P}) + \sqrt{P} * T_h$$

- Computation time: n^2/P

- Row summation : same time complexity as column broadcast

$$\text{Speed-up}(n,P) = \frac{n^2}{\frac{n^2}{P} + 2*\left(T_s + \frac{n}{\sqrt{P}} * T_w\right) \log(\sqrt{P}) + \sqrt{P} * T_h}$$

Some numbers for SP-1: $T_s = 15{,}000$ $T_h = 350$

Twelve Ways to Fool the Masses

David Bailey, NASA

1. Present performance figures for an inner kernel, and then represent these figures as the performance of the entire application.
2. Quietly employ assembly code and other low-level language constructs.
3. Scale up the problem size with the number of processors, but omit any mention of this fact.
4. Quote performance results projected to a full system.
5. Compare your results against scalar, unoptimized code on CRAYs.
6. When direct runtime comparisons are required, compare with an old code on an obsolete system.
7. If MFLOPS rates must be counted, base the operation count on the parallel implementation, not on the best sequential implementation.
8. Quote performance in terms of processor utilization, parallel speedups or MFLOPs/dollar.
9. Measure parallel runtimes on a dedicated system, but measure sequential runtimes on a busy system.
10. Use an algorithm with a slow convergence rate.
11. Give 32-bit performance numbers, not 64-bit numbers.
12. If all else fails, show pretty pictures, animated videos and don't talk about performance.