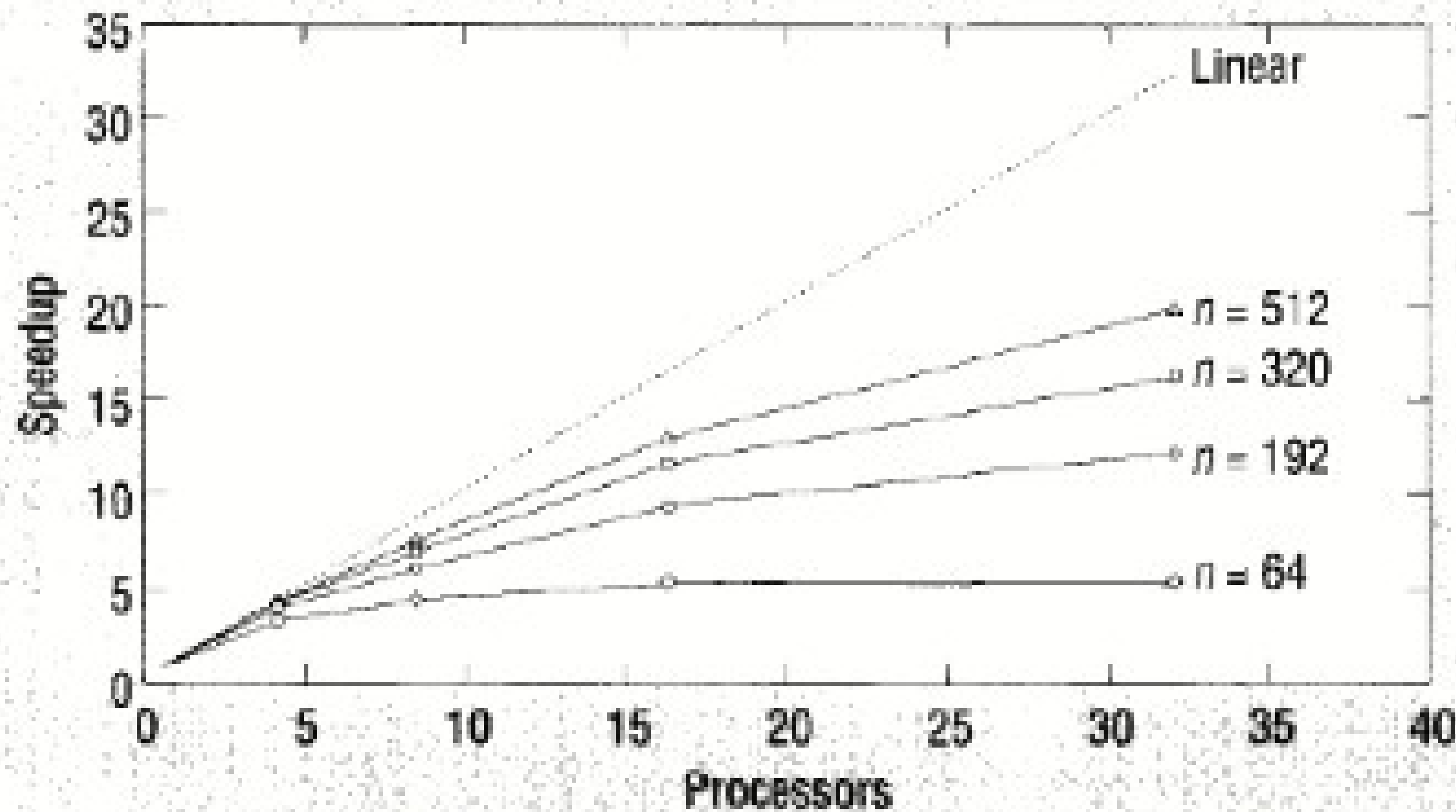
Isoefficiency analysis

- V2 typos fixed in matrix vector multiply

Measuring the parallel scalability of algorithms

- One of many parallel performance metrics
- Allows us to determine scalability with respect to machine parameters
  - number of processors and their speed
  - communication patterns, bandwidth and startup
  - Give us a way of computing
    - the relative scalability of two algorithms
    - how much work needs to be increased when the number of processors is increased to maintain the same efficiency

Amdahl's law reviewed



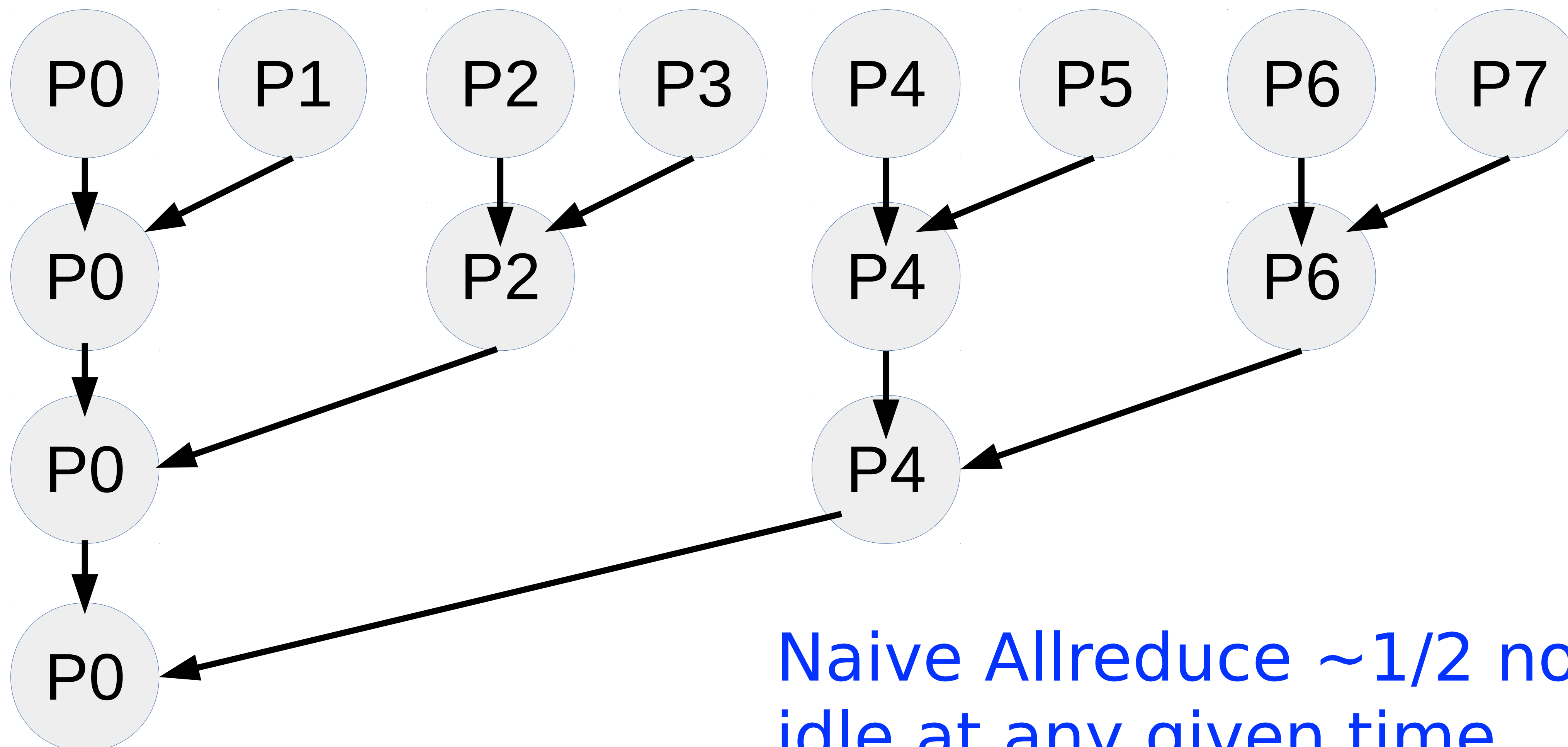
As number of processors increase, serial overheads reduce efficiency

As problem size increases, efficiency returns

Efficiency of adding *n* numbers on an ancient machine
P=4 gives ε of .80 with 64 numbers
P=8 gives ε of .80 with 192 numbers
P=16 gives ε of .80 with 512 numbers (4X processors, 8X data)

Motivating example

- Consider a program that does O(n) work
- Also assume the overhead is $O(\log_2 p)$, i.e. it does a reduction
- The *total overhead,* i.e. the amount of time processors are sitting idle or doing work associated with parallelism instead of the basic problem, is $O(p \log_2 p)$



Naive Allreduce ~1/2 nodes are idle at any given time

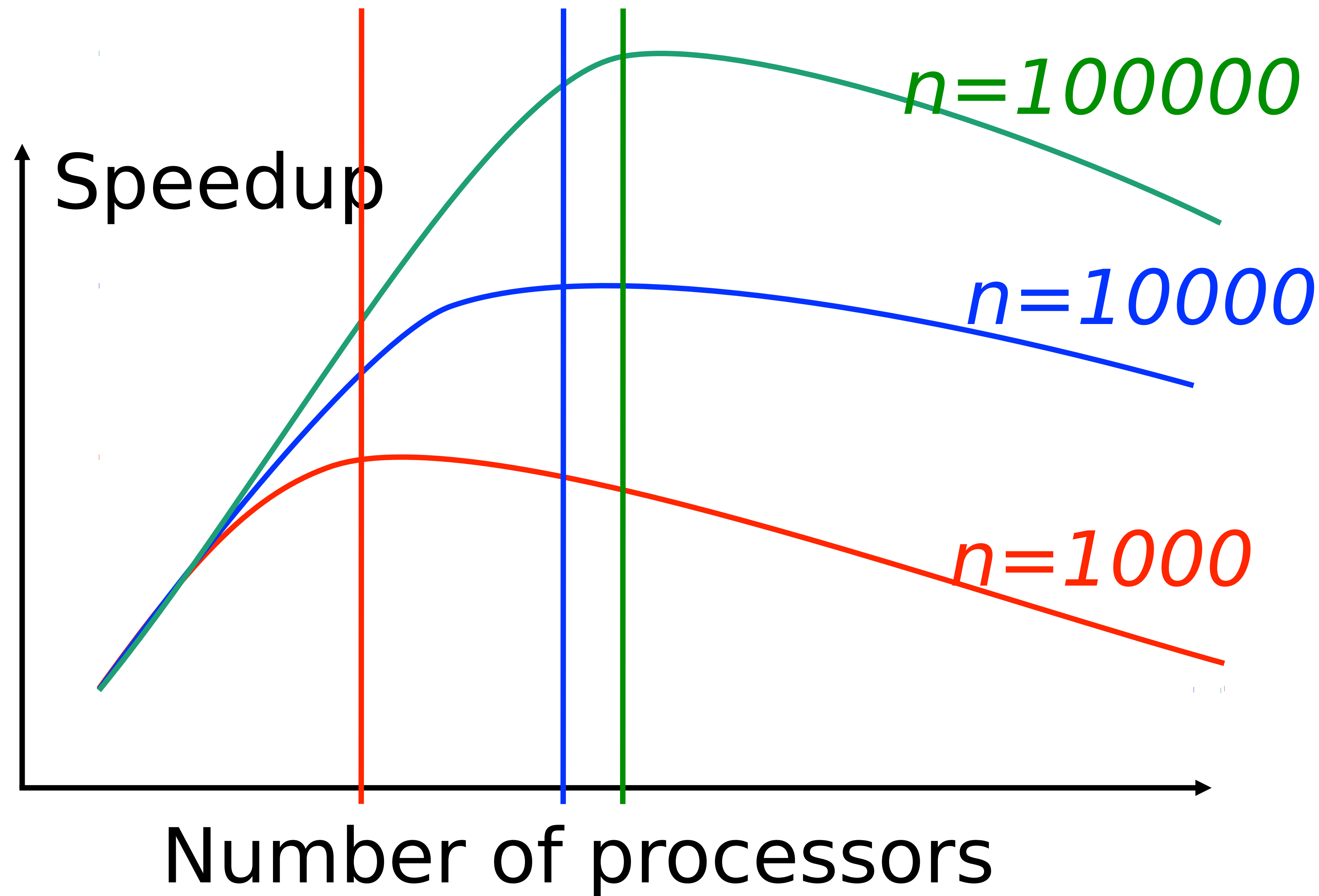
Data to maintain efficiency

P	P $\log_2$ P	Data needed per processor
2	2	1 GB
4	8	2 GB
8	24	3 GB
16	64	4

- As number of processors increase, serial overheads reduce efficiency
- As problem size increases,efficiency returns

Isoefficiency analysis allows us to analyze the rate at which the data size must grow to mask parallel overheads to determine if a computation is scalable.

Amdahl Effect both increases speedup and move the knee of the curve to the right



Total overhead T_O is the time spent

- Any work that is not part of the serial form of the program
  - Communication
  - Idle time because of waiting for some processor that is executing serial code
  - Idle time waiting for data from another processor
  - . . .

Efficiency revisited

- Total time spent on all processors is the original sequential execution [*best sequential implementation*] time plus the parallel overhead

$PT_p = T_1 + T_O$ (1)

- The time it takes the program to run in parallel is the total time spent on all processors divided by the number of processors. This is true because T_O includes the time processors are waiting for something to happen in a parallel execution.

$T_p = (T_1 + T_O)/P$ (1), which can be written $T_1 = P\,T_p - T_O$ (1a)

- Speedup S is as before (T_1/T_P) , or by substituting (1, 1a) above, we get:

$S = (P\,T_p - T_O) / ((T_1 + T_O)/P) = (\mathbf{P^2 T_p} - PT_O)/(T_1 + T_O)$. Using 1 we get

$= (\mathbf{P(T_1 + T_O)} - PT_O)/(T_1 + T_O) = P\,(T_1 + T_O - T_O)/(T_1 + T_O$

$= P\,T_1 / (T_1 + T_O)$

Efficiency revisited

- With speedup being

$S = T_1/T_P = (P\ T_1)/(T_1 + T_O)$

- Efficiency can be computed using the previous definition of the ratio of S to P as:

$E = S/P = ((PT_1)/(T_1 + T_O))/P = T_1/(T_1 + T_O)$

$= 1/(1+ T_O/T1)$ (2)

Efficiency as a function of work, data size and overhead

- Let

T_1 be the single processor time

W be the amount of work in units of work)

t_c be the time to perform each unit of work

- Then $T_1 = W \cdot t_c$
- T_O is the total overhead, i.e. time spent doing parallel stuff but not the original work
- Then *efficiency* can be rewritten as

(see Eqn. 2, previous page)

$E = 1/(1 + T_O/T1)$

$$E = \frac{1}{1 + \frac{T_o}{W t_c}}$$

Some insights into Amdahl's law and the Amdahl effect can be gleaned from this

- Efficiency is $E = \frac{1}{1 + \frac{T_o}{W t_c}}$
- For the same problem size on more processors, W is constant and T_O is growing. **Thus efficiency decreases.**
- Let $\theta(W)$ be some function that grow at the same or faster rate than W, i.e. $\theta(W)$ is an upper bound
- As *P* increases, T_O will grow faster, the same, or slower than $\theta(W)$
    - If faster, system has limited scalability
    - If slower or the same, system is very scalable, can grow work the same or slower than processor growth

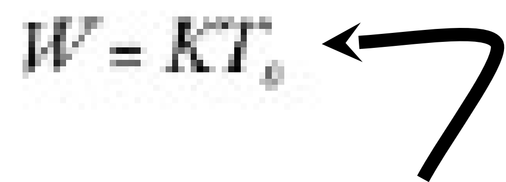
The relationship of work and constant efficiency

$$E = \frac{1}{1 + \frac{T_o}{W t_c}}$$

$$\frac{T_o}{W} = t_c\left(\frac{1-E}{E}\right)$$

$$W = \frac{1}{t_c}\left(\frac{E}{1-E}\right)T_o$$

If $K = E/(t_c(1 - E))$ is a constant that depends on the efficiency, then we can reduce the last equation to



$$W = KT_o$$

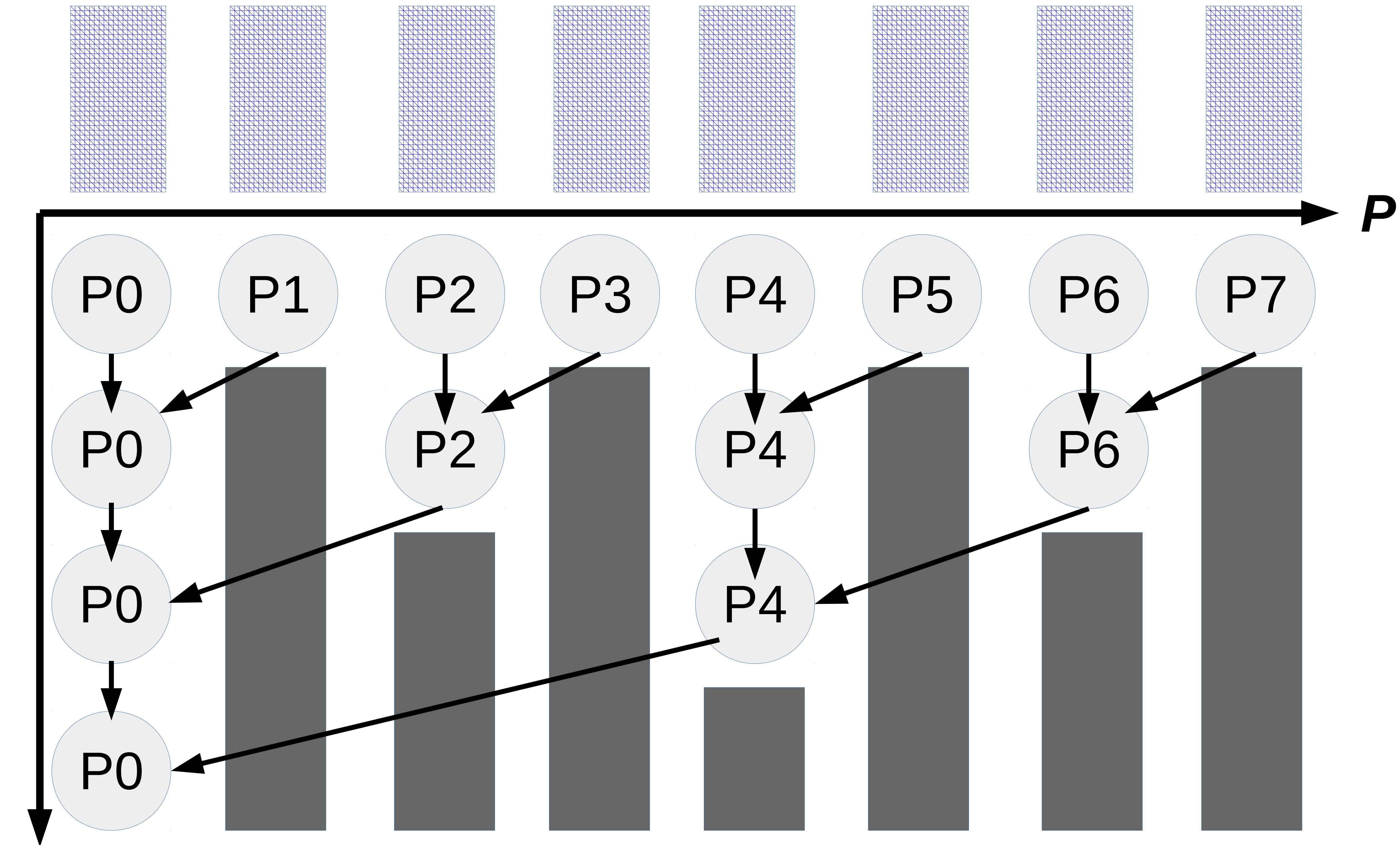
This relationship holds when the efficiency is constant

Will use algebraic manipulations to (eventually) represent W as a function of P. This indicates how W must grow as the number of processors grows to maintain the same efficiency.

Isoefficiency review

- The goal of isoefficiency analysis is co determine how fast work needs to increase to allow the efficiency to stay constant
- First step: divide the time needed to perform a parallel calculation (T_p) into the *sequential time* and the *total overhead* T_O.
    - $T_p = (T_1 + T_O)/P; \; P\,T_p = T_1 + T_O$

$$T_p = (T_1 + T_O)/P$$



Sum of all blue (hatched) times is T1. Sum of all gray is T0 (plus communication time)

Let's cast efficiency (E) in terms of T_1 and T_O *so we can see how* T_1, T_O *and E are related.*

- With speedup being

$S = T_1/T_P = (P\ T_1)/(T_1 + T_O)$

- Efficiency can be computed using the previous definition of the ratio of S to P as:

$E = S/P = ((PT_1)/(T_1 + T_O))/P = T_1/(T_1 + T_O)$

$\quad = 1/(1+ T_O/T_1)$.

Now look at how E is related to the work (W) in T_1

- $E = S/P = ((PT_1)/(T_1 + T_O))/P = T_1/(T_1 + T_O)$

  $= 1/(1 + T_O/T_1)$.
- Note that T_1 is the number of operations times the amount of time to perform an operation, i.e., $t_C * W$
- Then $E = 1/(1 + T_O/T_1) = 1/(1 + T_O/(t_C * W))$ or

$$E = \frac{1}{1 + \frac{T_o}{W t_c}}$$

Solve for W in terms of E and T_O

$$E = \frac{1}{1 + \frac{T_o}{W t_c}}$$

$$\frac{T_o}{W} = t_c\left(\frac{1-E}{E}\right)$$

$$W = \frac{1}{t_c}\left(\frac{E}{1-E}\right)T_o$$

Do the algebra, combine constants, and we have the Isoefficiency relationship.

For efficiency to be a constant, W must be equal to the overhead times a constant, i.e., *W* must grow proportionally to the overhead T_O

If we can solve for KT_O we can find out how fast W needs to grow to maintain constant efficiency with a larger number of processors.

If $K = E/(t_c(1-E))$ is a constant that depends on the efficiency, then we can reduce the last equation to

$$W = KT_o$$

What if T_O is negative?

- *Superlinear* speedups can lead to negative values for T_O
- Appears to cause work to need to decrease
- Causes of *superlinear* speedup
  - increased memory size in NUMA and hierarchical (i.e. caching) memory systems
  - Search based problems
- We assume $T_O \geq 0$

Simple case superlinear speedup -- linear scan search for 9

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

To find an element takes O(pos) steps

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

To find an element takes O(pos - offset) steps

Doubling processors leads to a speedup of ~9

Cache effects

- Moving data into cache is a hidden work item
- As P grows larger, total cache available also grows larger
- If P grows faster than data size, eventually all data fits in cache
- Thus cache misses due to capacity vanish, and the work associated with those misses vanishes, and the parallel program is doing less work, enabling superlinear speedups if everything else is highly efficient

There are no magical causes of superlinear speedup

- In the early days of parallel computing it was thought by some that there might be something else going one with parallel executions
- All cases of superlinear speedup observed to date can be explained by a reduction in overall work in solving the problem

How to define the problem size or amount of work

- Given an n x n matrix multiply, what is the problem size?
  - Commonly called n
- How about adding two n x n matrices?
  - Could call it the same n
- How about adding to vectors of length n?
  - Could also call the problem size n
- Yet one involves n^3 work, one n^2 work, and one n work

Same name, different amounts of work - this causes problems

- The goal of isoefficiency is to see how work should scale to maintain efficiency
- Let $W=n$ for matrix multiply, matrix add and vector addition
- Let all three (for the sake of this example, *even though not true*) have a similar T_0 that grows linearly with P
- Doubling n would lead to 8 times more operations for matrix multiply, 4 times for matrix add, and 2 times for vector add
- Intuitively the vector add seems to be right, since number of operations and work (W) seem to be the same thing, not data size.
- We will normalize W in terms of *operations in the best sequential algorithm*, not some other metric of problem size

Isoefficiency of adding n numbers

- $n-1$ operations in sequential algorithm -- asymptotically is n and we will use n, and $T_1 = n \cdot t_c$
- Let each add take one unit of time, and each communication take one unit of time
- On P processors, n/P operations + $\log_2 P$ communication steps + one add operation at each communication step
- $T_P = n/P + 2 \log_2 P$
- $T_O = P (2 \log_2 P)$ since each processor is either doing this or waiting for this to finish on other processors
- $S = T_1 / T_P = n/(n/P + 2 \log_2 P)$
- $E = S / P = n / (n + 2 P \log_2 P)$

Isoefficiency analysis of adding *n* numbers

- From slide 12, $W = K\,T_O$ if same efficiency is to be maintained
- $T_O = P\,(2 \log_2 P)$ from the previous slide, then

  $W = 2\,K\,P \log_2 P$

 and ignoring constants give an isoefficiency function of

  $\theta(P \log_2 P)$

- If the number of processors is increased to P', then the work must be increased not by P'/P, but by

$(P' \log_2 P') / (P \log_2 P)$

- Thus going from 4 to 16 processors requires having

$(16 \log_2 16) / (4 \log_2 4)$ or 8X as much work, spread over $4X$ as many processors, or 2X more work/processor. Since data size grows proportional to work, we need 2X more data per processor!

More complicated T_O

- Consider $T_O = P^{3/2} + P^{3/4}W^{3/4}$, and $W = K\ T_O$, then

$W = P^{3/2} + P^{3/4}W^{3/4}$ (again, ignoring constant K)

- Difficult to solve for W in terms of P
- Note that we need ratio of W and T_O to remain fixed for E (efficiency) to remain fixed
- Problem will scale well if no term of T_O grows faster than W
- Thus we can examine terms independently

$$W = P^{3/2} + P^{3/4}W^{3/4}$$

- Solve for first term, i.e. $W = KP^{3/2} = \theta(P^{3/2})$
- Solve for second term, i.e.

$W = K\,P^{3/4}W^{3/4}$

$W^{1/4} = K\,P^{3/4}$

$W = K^4\,P^3 = \theta(P^3)$

- If problem size grows at least as fast as $\theta(P^{3/2})$ and $\theta(P^3)$ then efficiency will not decrease as P increases.
- Thus the isoefficiency function for the system is $\theta(P^3)$

Cost optimality

- Parallel system is *cost-optimal* if product of $PT_P \propto W$, i.e, is not growing faster than W
  - Stated differently, the system is cost proportional to the execution time of the fastest known sequential algorithm on a single processor.
- Because $PT_p = T_1 + T_O$, then $T_1 + T_O \propto W$
- Since $T_1 = Wt_c$, we have $Wt_c + T_O \propto W$ and therefore $W \propto T_O$.

- Suggests a parallel system is cost optimal if its overhead function and problem size are of the same order of magnitude, i.e. have same order of complexity.
- **Conforming to the isoefficiency relationship keeps a system cost-optimal as it is scaled up**

How small can an isoefficiency function be?

- Let a problem contain W basic operations
- Let problem size grow slower than $\theta(P)$
- As P grows, eventually $P > W$
- At this time efficiency E must drop because there will be processors doing no work
- Thus, problem size must grow at least by $\theta(P)$ for the problem to scale
  - $\theta(P)$ is the lower bound on the isoefficiency function
  - $\theta(P)$ is the isoefficiency function of an ideally scalable system

Degree of concurrency $C(W)$

- Lower bound of $\theta(P)$ for some algorithm is imposed by the algorithm's *degree of concurrency*
- If $\theta(P)$ is an algorithm's degree of concurrency, at most $\theta(P)$ processors can be used to solve the problem
- Example: Gaussian elimination has $\theta(n^3)$ amount of computation, but ...
  - n variables must be eliminated one after the other (sequentially)
  - n^2 work per variable
  - thus at most n^2 processors can ever be effectively be used at a time.

Degree of concurrency, cont.

- If $W=\theta(n^3)$ for this problem, degree of concurrency is $\theta(W^{2/3})$
- Given a problem of size W, at most $\theta(W^{2/3})$ processors can be used
  - For P processors, need $\theta(P^{3/2})$ work ($W^{2/3} = P$)
  - Thus, because of concurrency, isoefficiency function for this operation is $\theta(P^{3/2})$
- If algorithm's degree of concurrency is $<\theta(W)$, then
  - isoefficiency function due to concurrency is worse than $\theta(P)$
  - In these cases, isoefficiency function is the max of the isoefficiency functions due to concurrency, communication, and other overheads

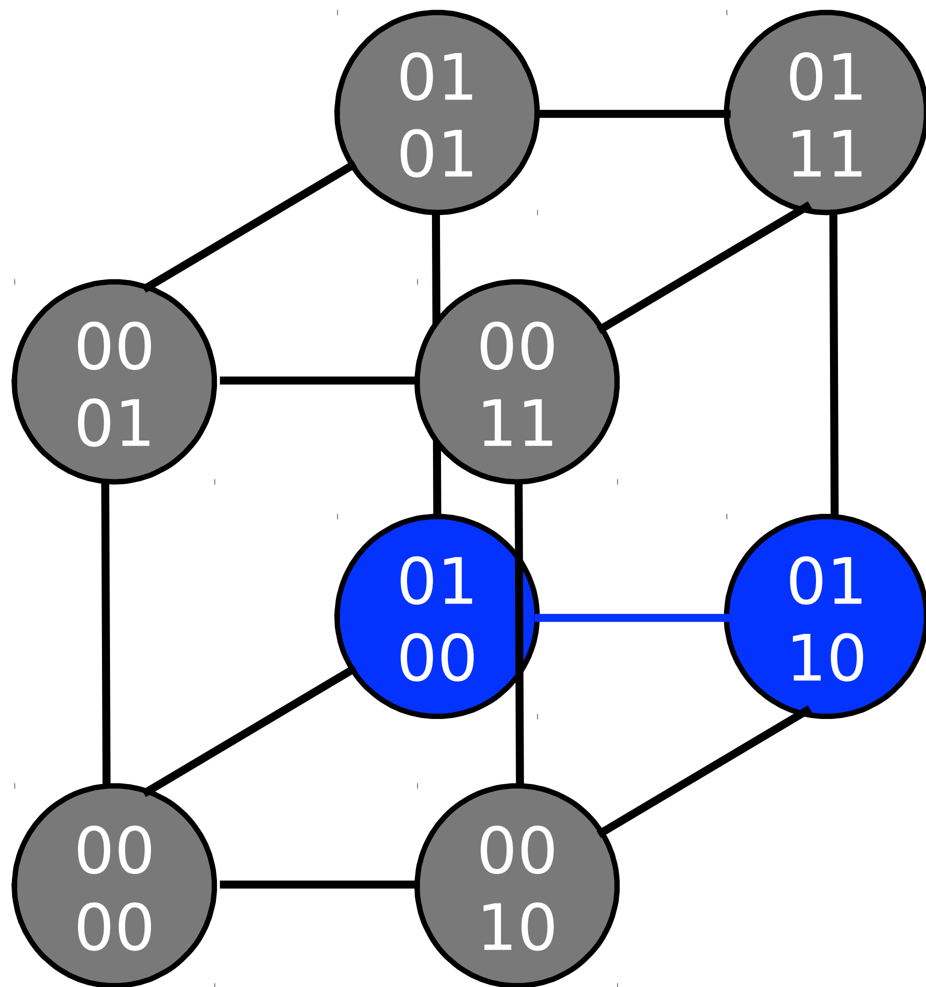
Hypercubes: short aside

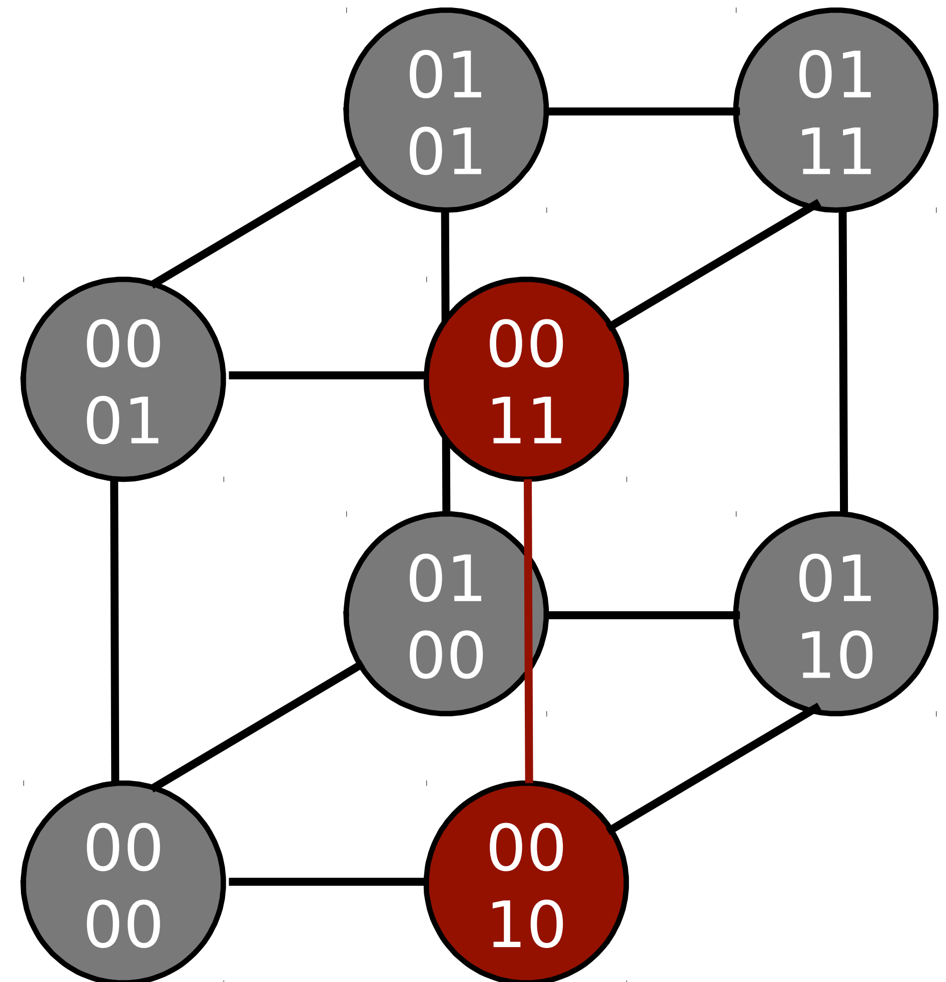
- Since hypercubes are mentioned in Grama’s paper, let's talk about them for a few minutes.
- Hypercubes were first developed as part of the Cosmic Cube project at CalTech (Seitz and Fox). Commercial version came out as the Intel iPSC, with Cleve Moler as one of the designers.
- Cleve Moler went on to found Matlab, Jeff Fox now at IU CS, Seitz won 2011 IEEE Computer Society Seymour Cray Computer Engineering Award
- The original *Cosmic Cube* was a plot device used in Marvel Comics

Hypercube

- Direct topology (one switch node/processor)
- 2 x 2 x … x 2 mesh
- Number of nodes a power of 2, denoted k
- Node addresses 0, 1, …, 2^k-1
- Node i connected to k nodes whose addresses differ from i in exactly one bit position

Hypercube labeling



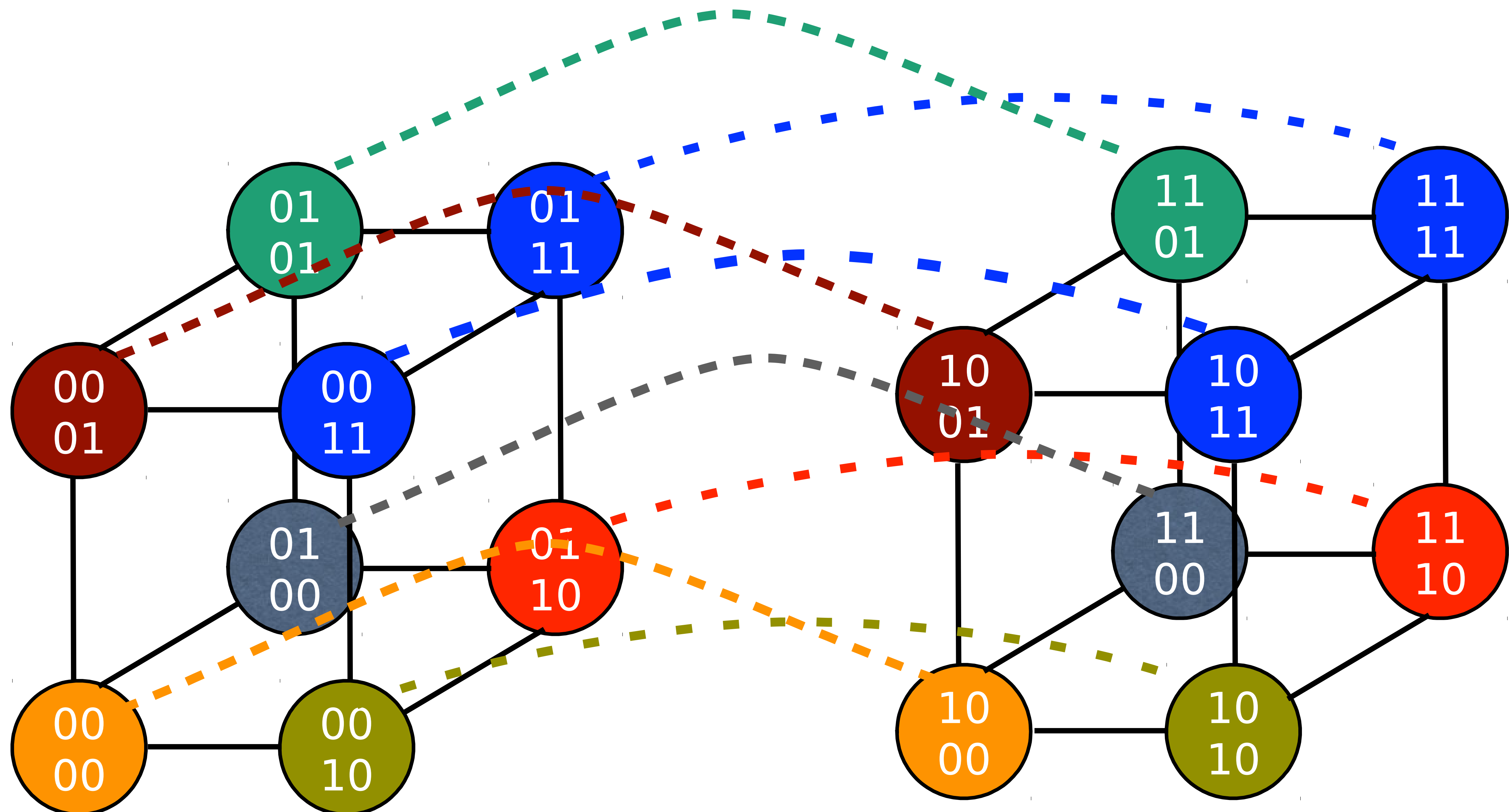


Pairs of adjacent nodes differ by 1 bit in their label -- result of gray code numbering

Hypercube labeling

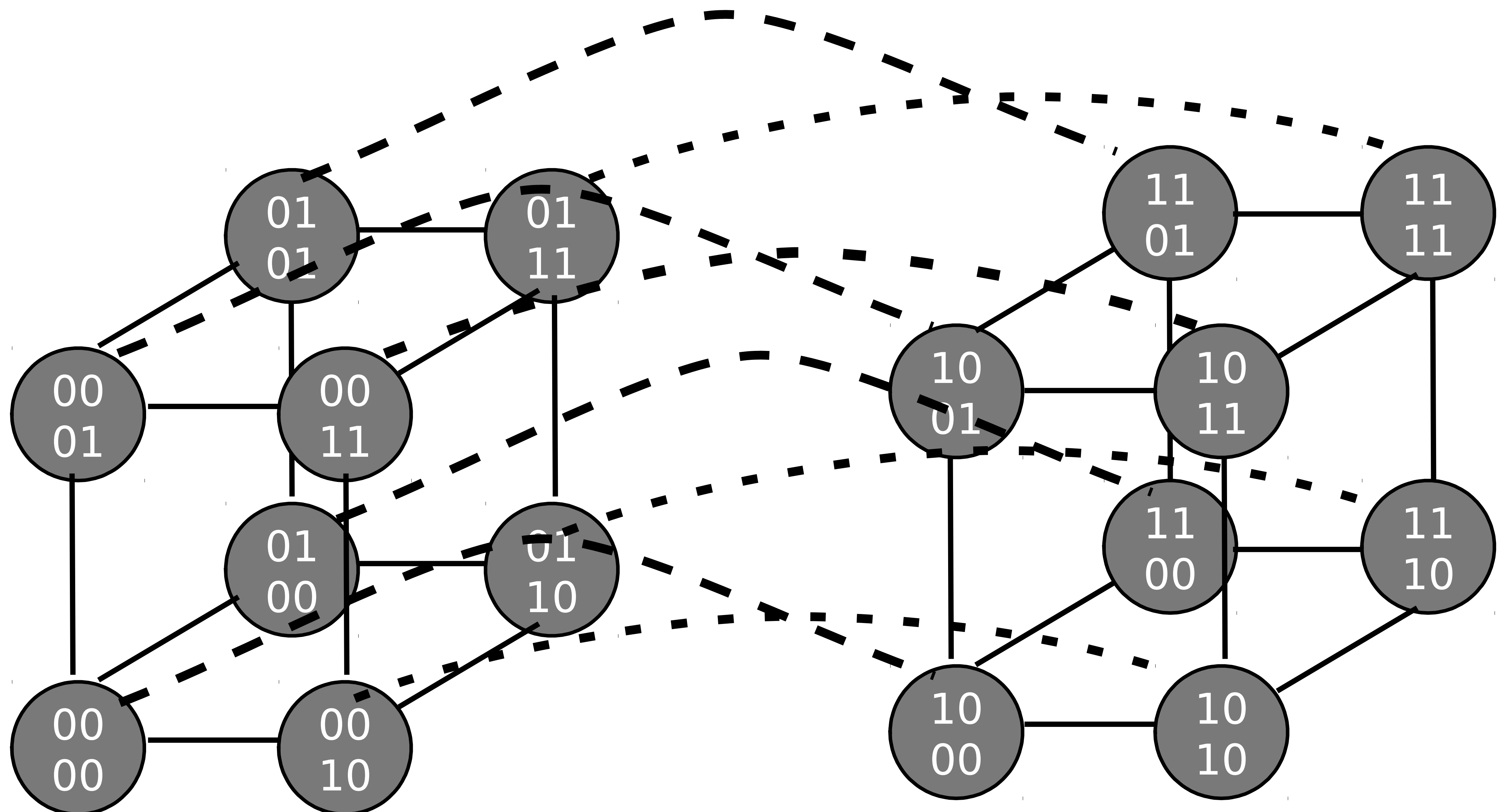
A large hypercube is made up of smaller hypercubes.

1. Add 1 (high-order) bit to labels
2. Make bit 1 for one small hypercube, 0 for the other
3. Add edge to nodes whose labels differ in one bit



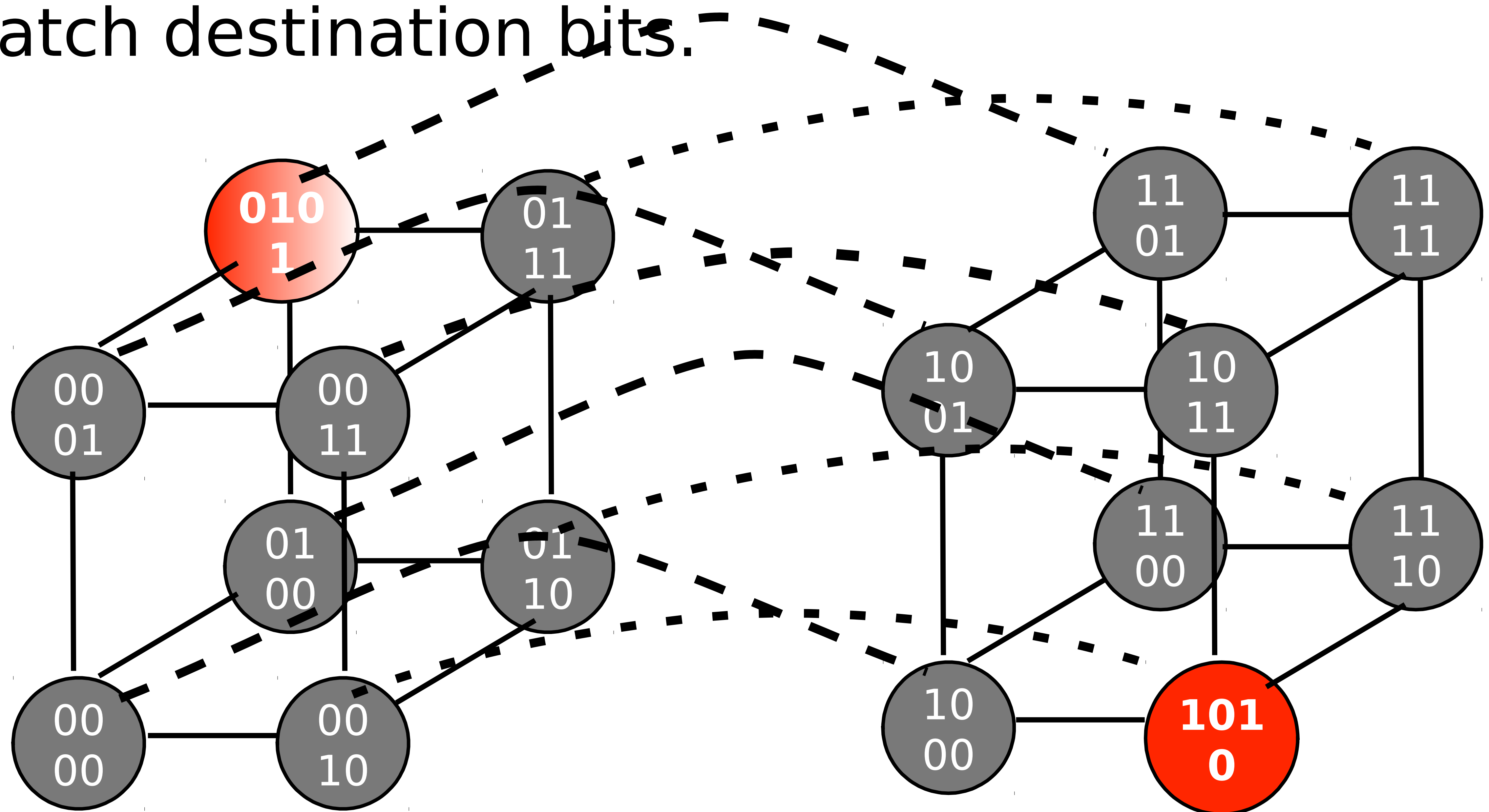
Labeling leads to routing

Given a source a destination label, always move one bit closer to the destination label with each hop.

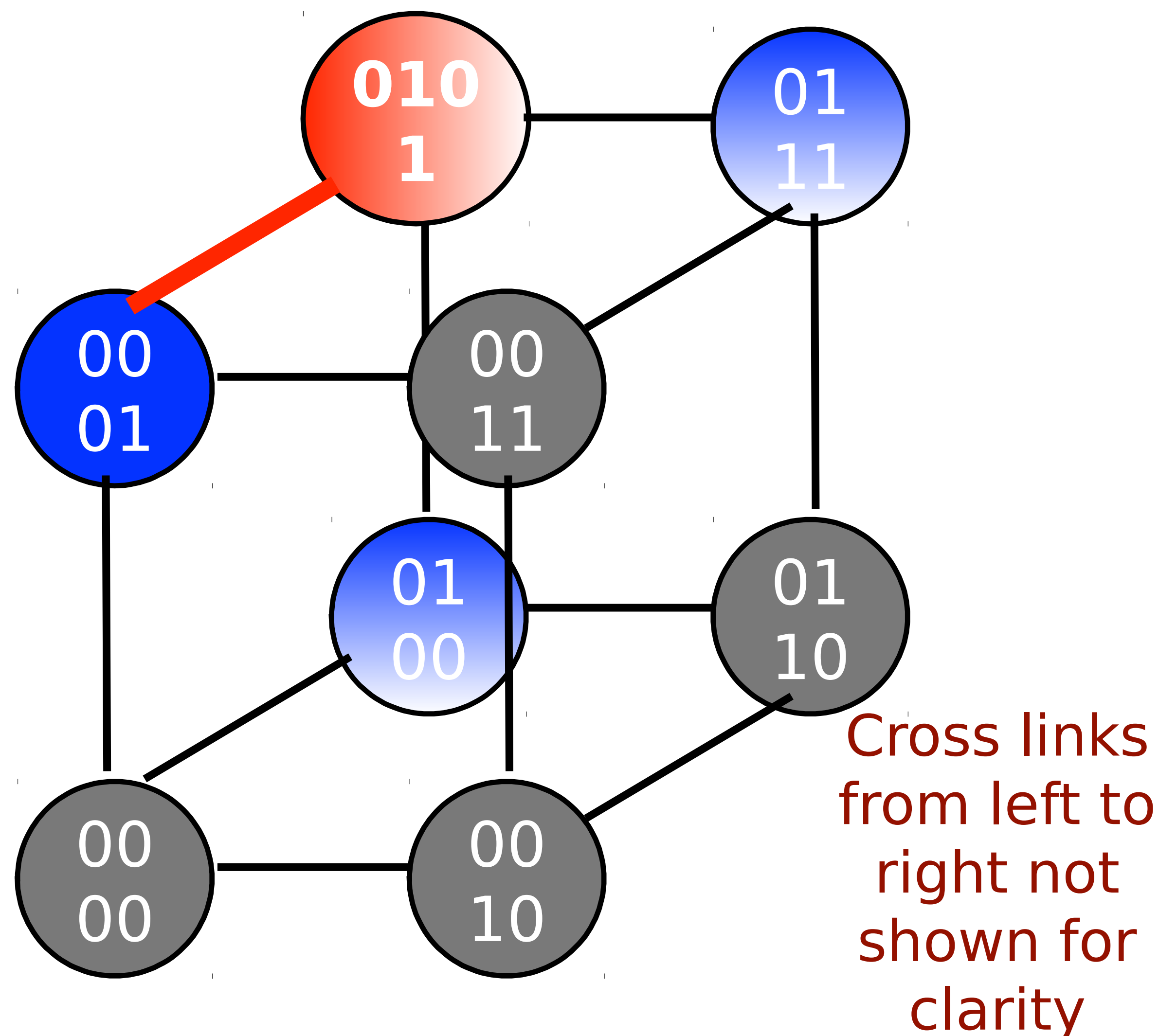


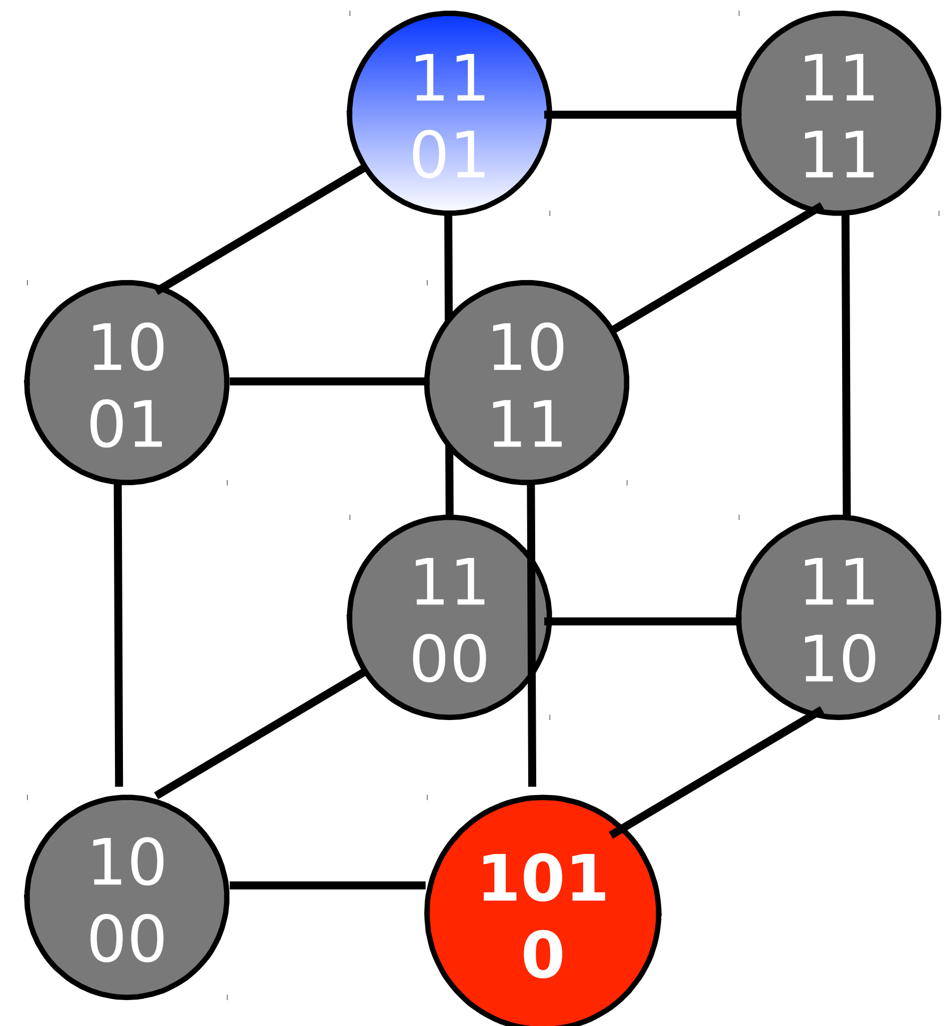
Labeling leads to routing

Go from 0101 to 1010, want to change source 0’s to 1’s, and 1’s to 0’s, i.e., change source bits to match destination bits.



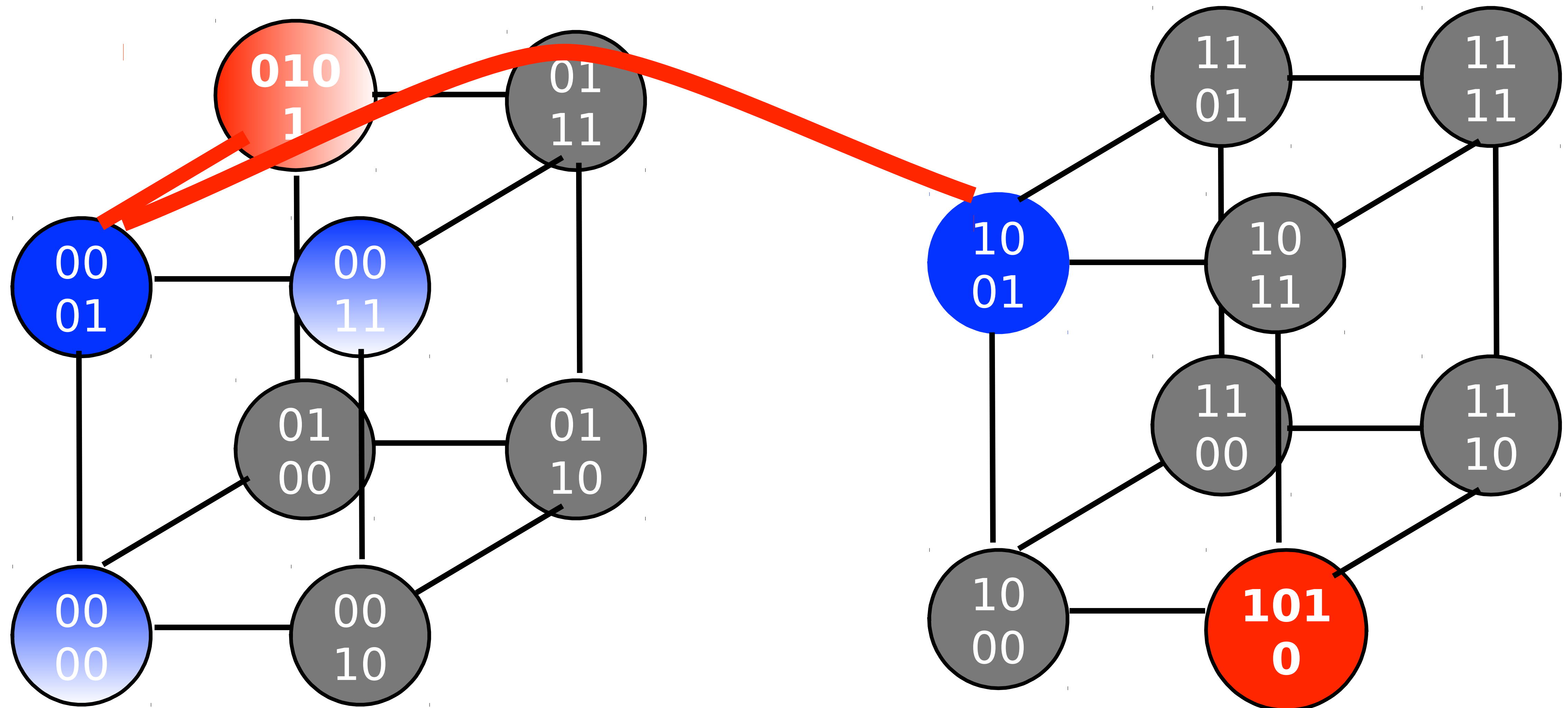
Go from 0**1**01 to 0**0**01, on the way to 1010. Note that since every bit needs to change, and every bit link changes one bit, we have four choices. In general, B choices, where B is the number of bits to change.



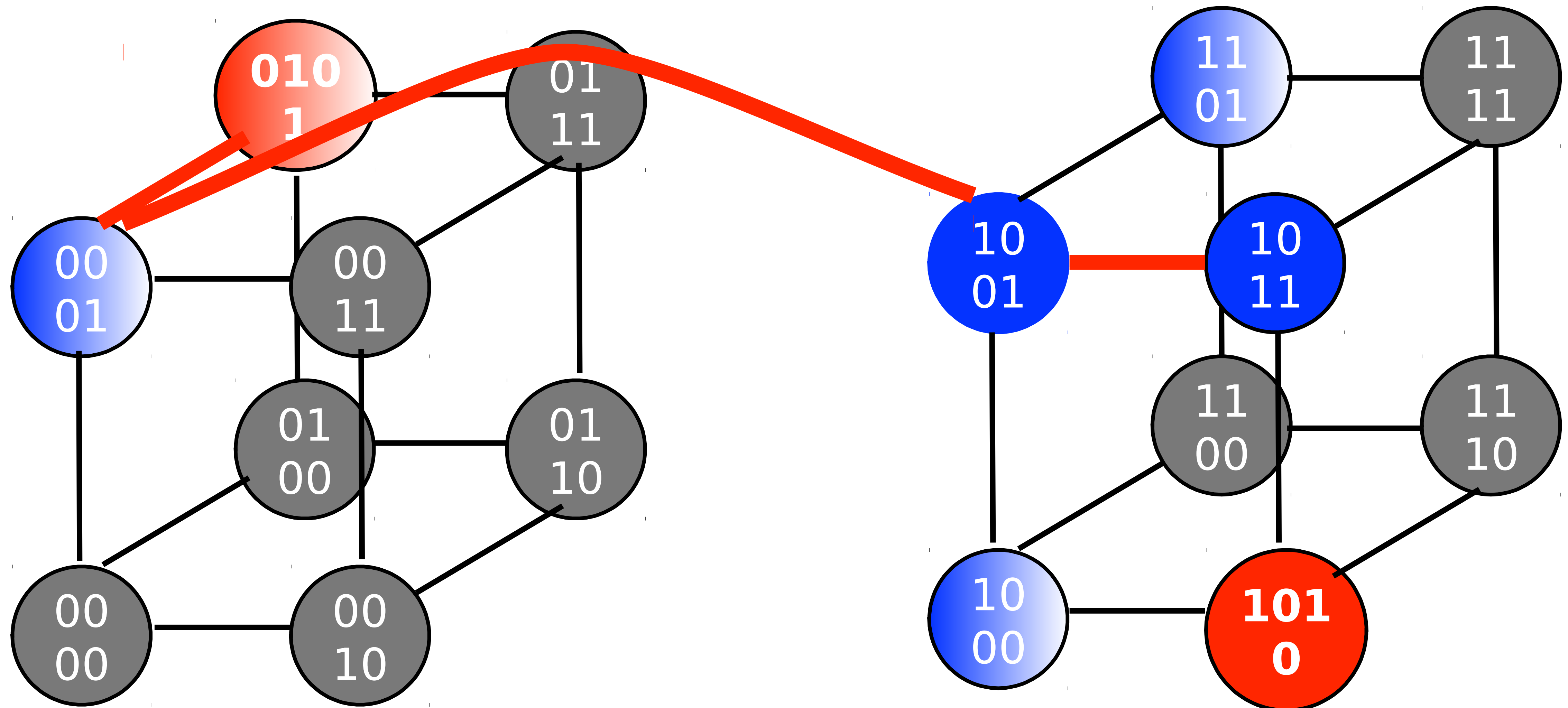


Labeling leads to routing

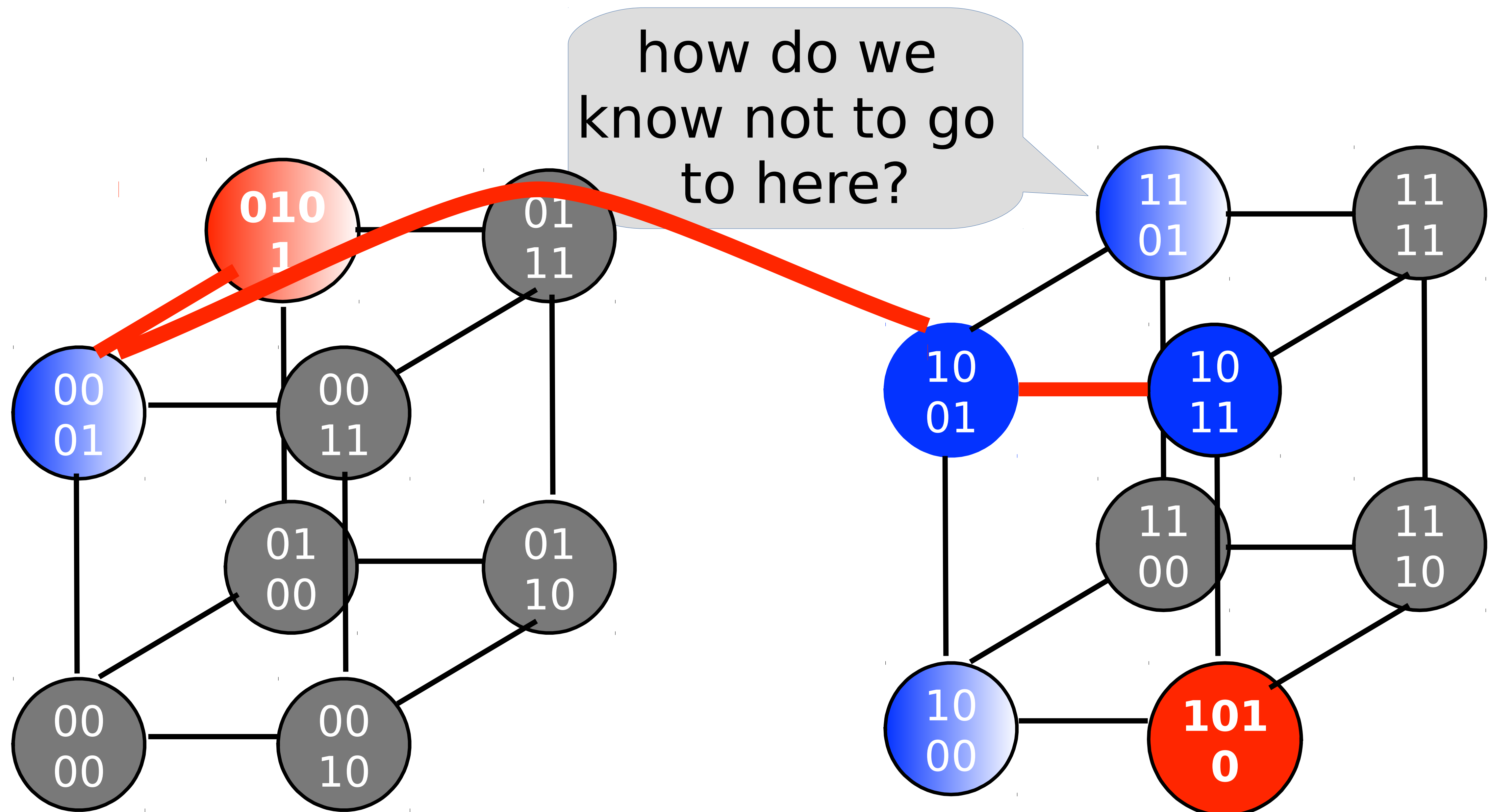
At **0**001, on route from 0101 to 1010. Three bits differ, three choices, pick one (**1**001)



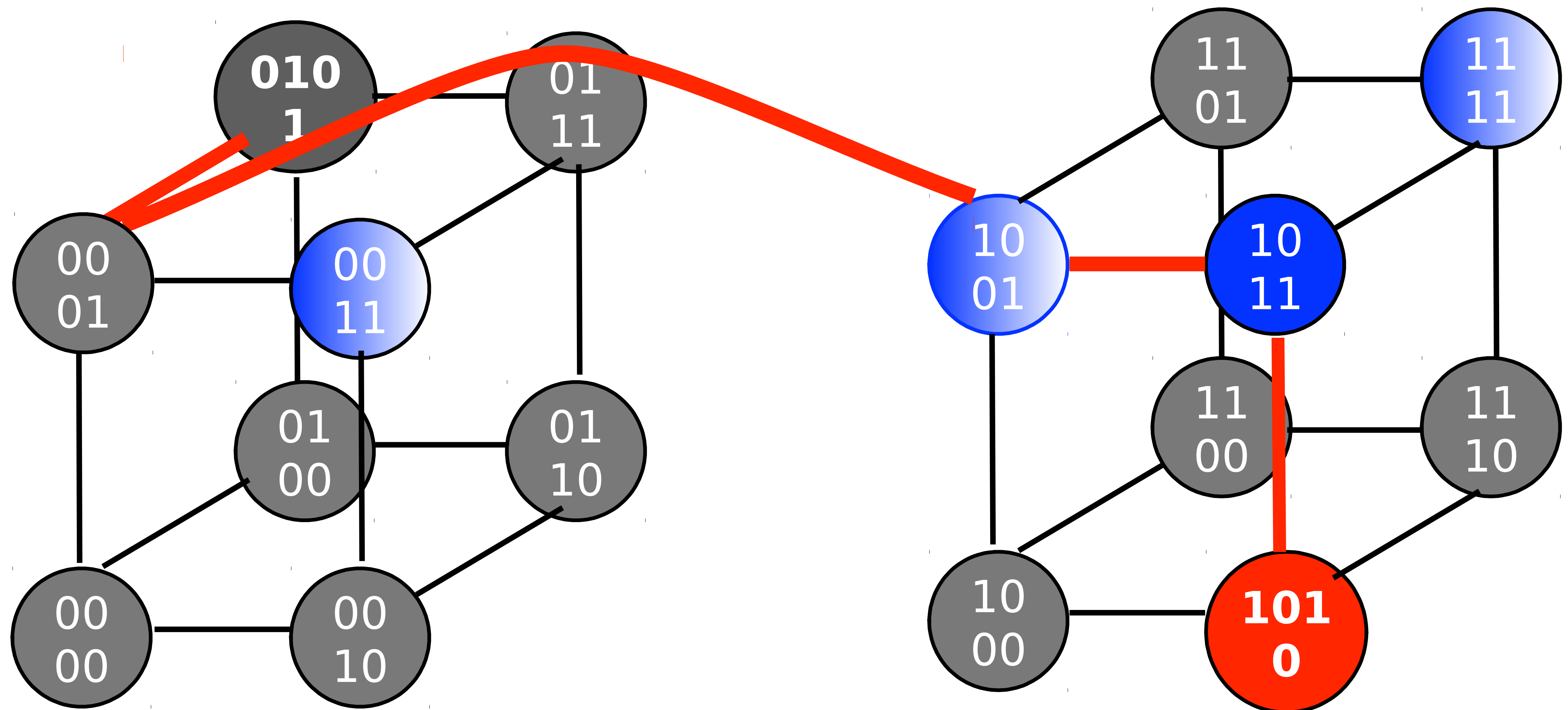
At **1001**, on route from 0101 to 1010. Two bits differ, two choices, pick one (10**1**1)



At **1001**, on route from 0101 to 1010. Two bits differ, two choices, pick one (10**1**1)



At 101**1**, on route from 0101 to 1010. One bit differs, only one choice, pick it (101**0**)

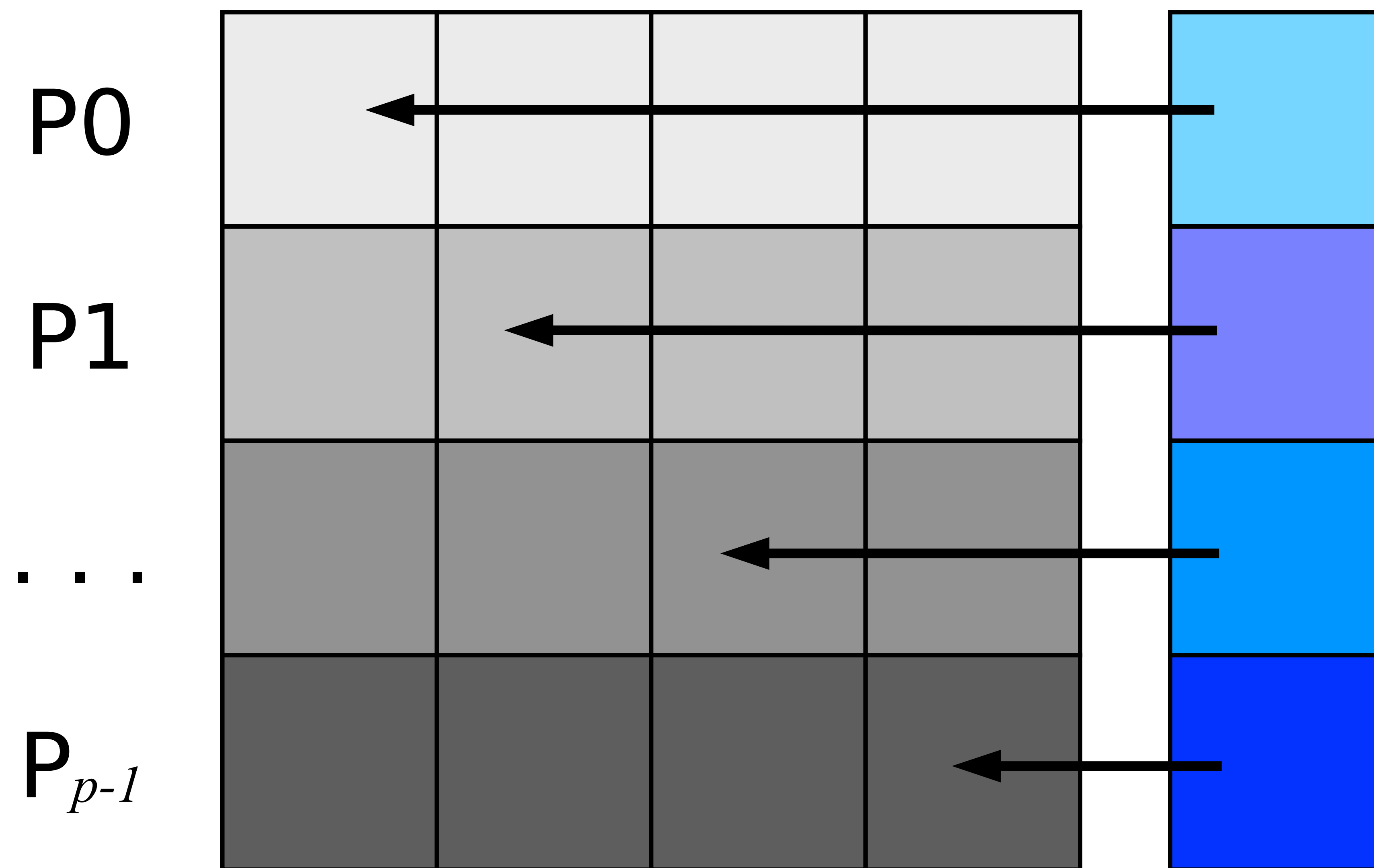


Comparing matrix vector algorithms - sequential alg.

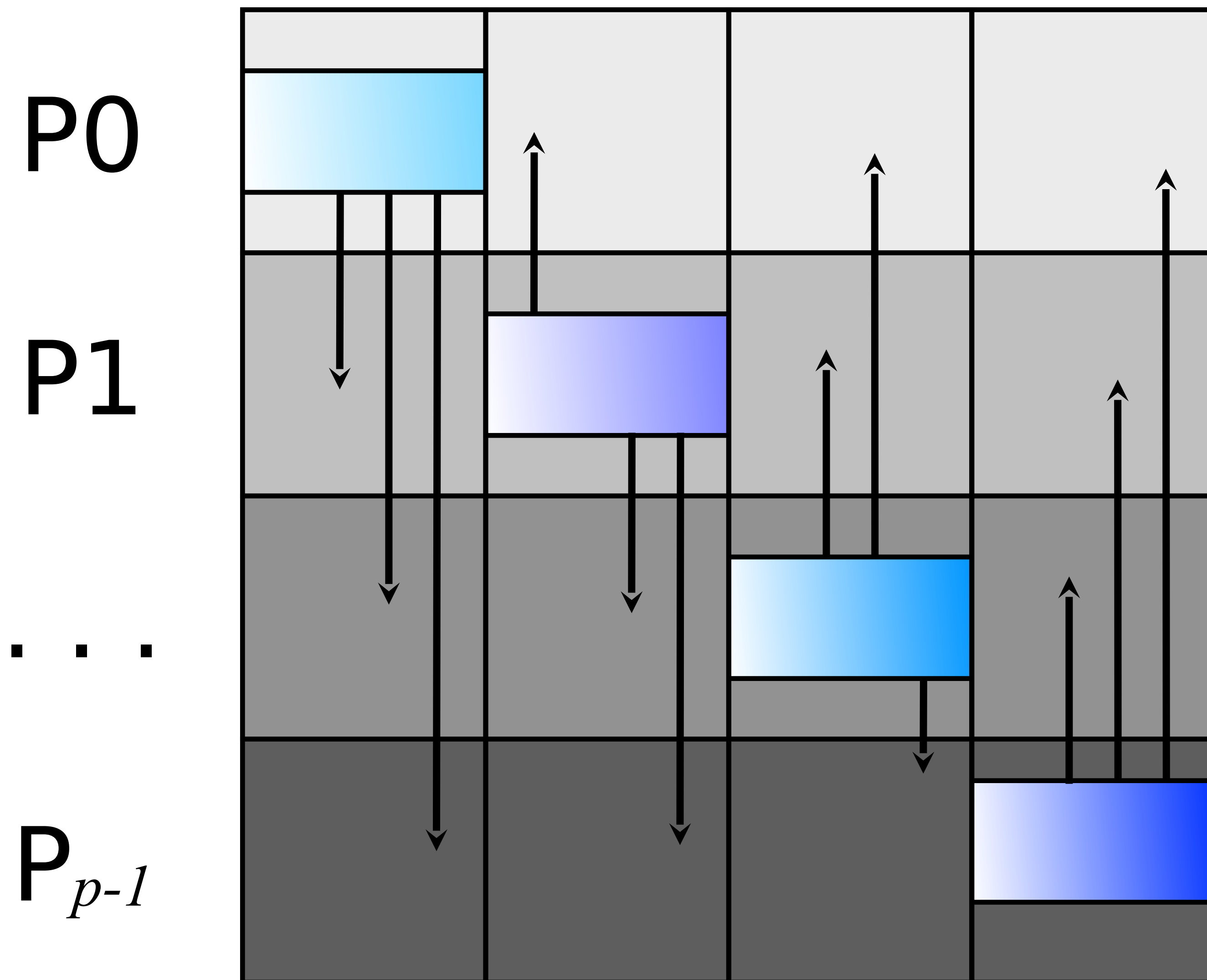
- Consider matrix vector multiply, i.e. an n x n matrix times an n x 1 matrix
- Number of basic operations (W) is n^2, with t_c the time for a single floating multiply-add
- Sequential time is $n^2 t_c$, i.e.

$$T_1 = n^2 t_c$$

With striped, data starts out like this, e.g., from reading matrix and vector from disks



Every process sends its n/P elements of the vector to every other process



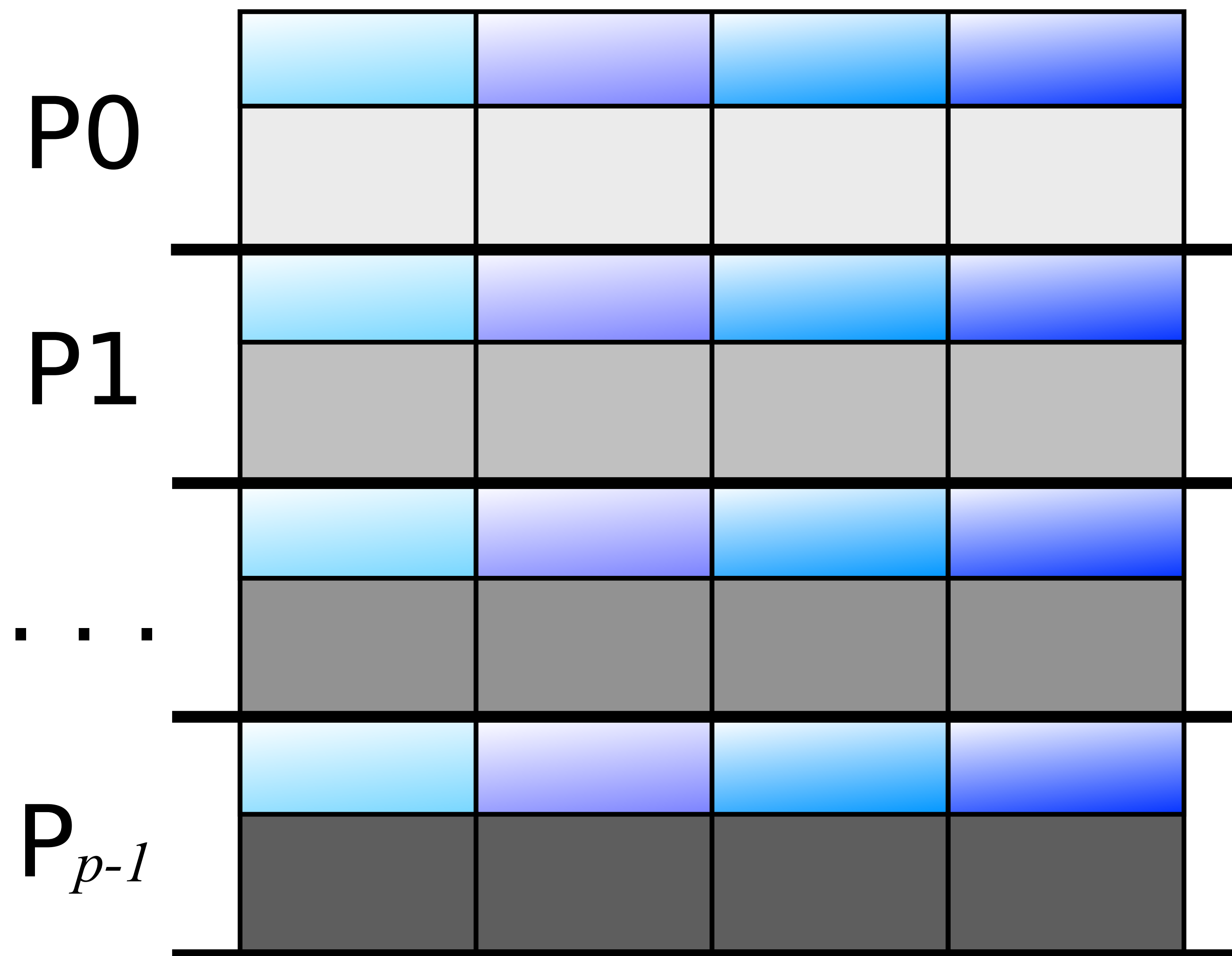
Do this with an all-to-all broadcast

$t_s \log P + t_w n(P-1)/P$

startup time (t_s is startup time of the network

comm. time -- t_w is the time to send one word

After communication every process has a copy of the vector



Row-striped parallel alg.

- n/p matrix rows and vector elements to each processor
- Costs:
  - all-to-all broadcast of vector elements so that each processor has a copy:

$$t_s \log P + t_w n(P-1)/P$$

or, as P grows large, simply

$$t_s \log P + t_w n$$

where t_s is startup time, t_w is per-word transfer time

Row-striped parallel alg.

- n/p matrix rows and vector elements to each processor
  - Each node does $t_c\, n^2/p$ work multiplying n/p rows times the vector
  - $T_P = t_c\, n^2/P + t_s \log P + t_w\, n$

Using the relation $T_O = P\, T_P - T_1$, we get $T_O = t_s\, P \log P + t_w\, n\, P$

$T_O = \mathrm{P}(t_c\, n^2/P + t_s \log P + t_w\, n) - n^2\, t_c$

$= \cancel{t_c\, n^2} + t_s\, P \log P + t_w\, P\, n - \cancel{n^2\, t_c}$

$= t_s\, P \log P + t_w\, n\, P$

Isoefficiency relationship

- $T_O = t_s P \log P + t_w n P$
- Balance the first term of T_O by rewriting $W = K\, T_O$ using only first term $T_O = \boldsymbol{t_s P \log P}$ to get $W = K\, t_s P \log P$
- Balancing the second term of T_O ($t_w n P$ due to per-word transfer time) against the problem size W and in terms of P we get

$n^2 = K\, t_w n P$

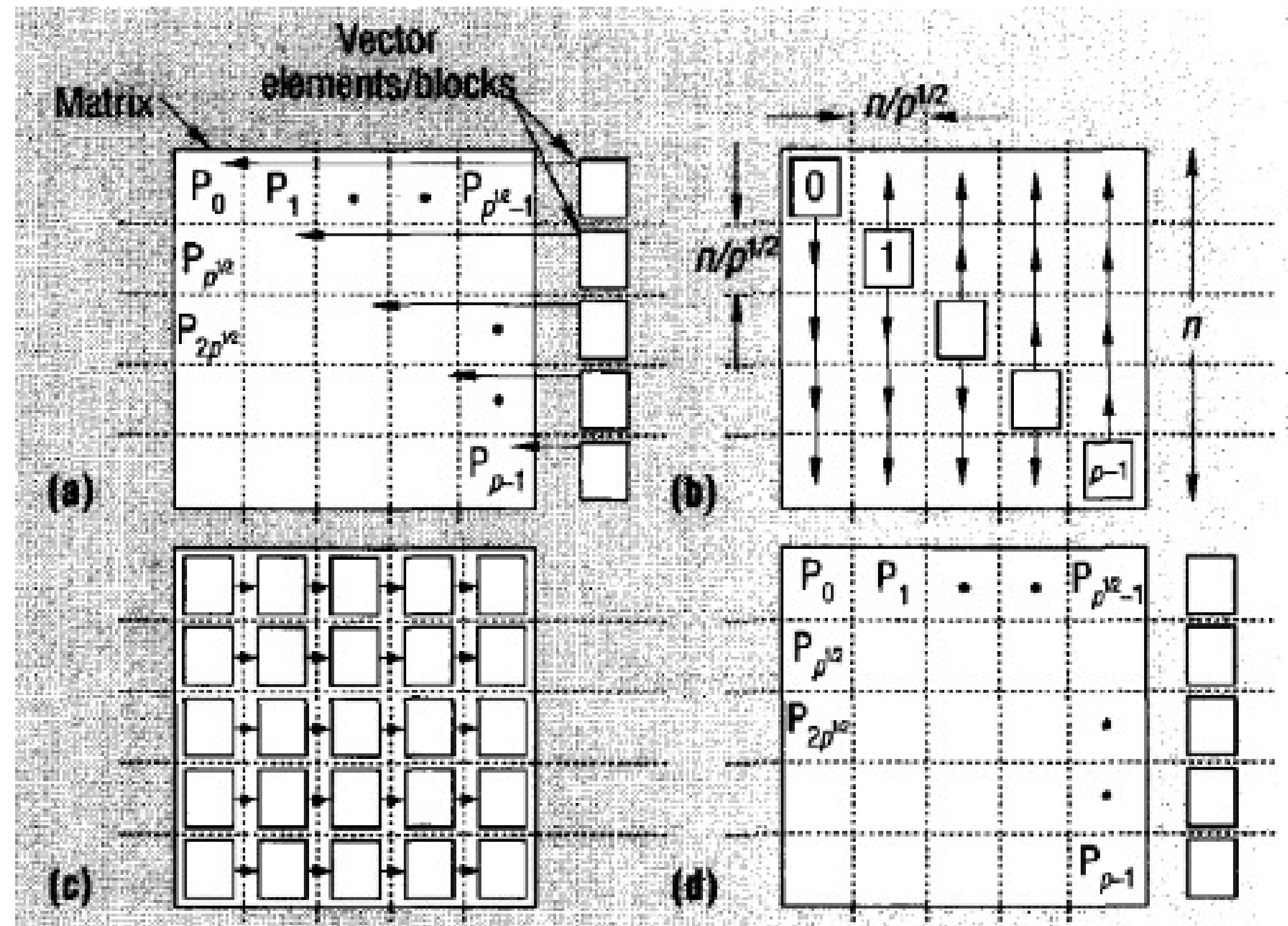
$n = K\, t_w P$ (solve for n in terms of K and t_w (constants) and P)

$W = n^2 = K^2 t_w^2 P^2$

- To maintain efficiency, work must increase proportional to P^2

Checkerboard partitioning - data is originally in the last processor of each column

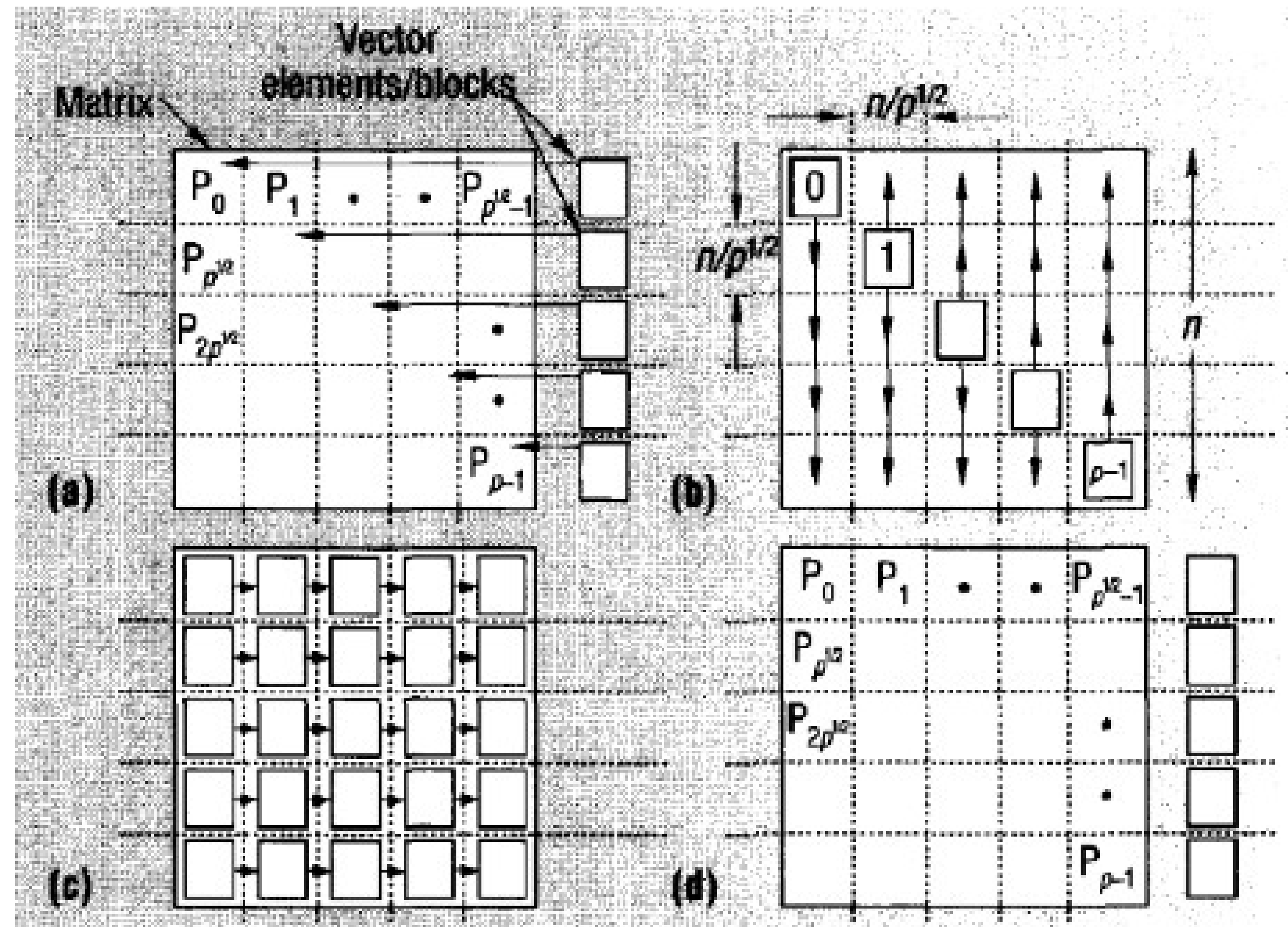
- Divide data into $n/\sqrt{p}$ x $n/\sqrt{p}$ squares and place on the last column of processes
- Each process w/data sends it to the diagonal of its row (a)
- Column-wise one-to-all broadcast of $n/\sqrt{p}$ elements (b)



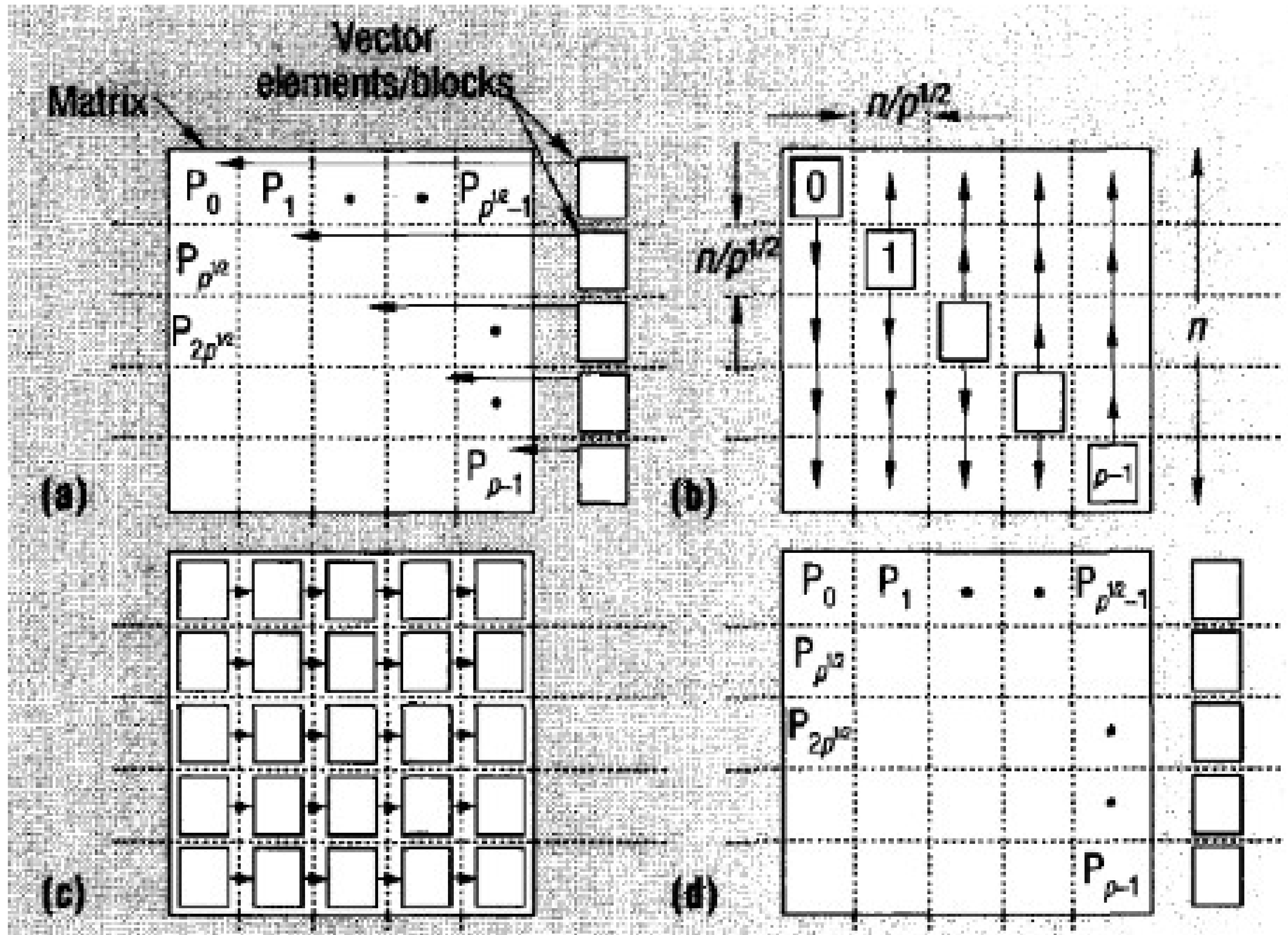
Checkerboard partitioning - data is originally in the last processor of each column

- Each processor performs n^2/p multiplications, and locally adds $n/\sqrt{p}$ sets of products. (c)
- $n/\sqrt{p}$ partial sums to be accumulated along each row (c)

State at end of computation (d)



Checkerboard partitioning - data is originally in the last processor of each column



Checkerboard partitioning analysis

1. Divide data into $n/\sqrt{p}$ x $n/\sqrt{p}$ squares, send along rows

$t_s + t_w(n/\sqrt{p}) \log \sqrt{p}$

2. Column-wise one-to-all broadcast of $n/\sqrt{p}$ elements

takes $(t_s + t_w n/\sqrt{p}) \log \sqrt{p}$ time on a hypercube with store-and-forward routing, *or* $t_s \log \sqrt{p} + t_w n/\sqrt{p} \log \sqrt{p}$ time.

3. Adding the numbers: Each processor performs n^2/p multiplications, and locally adds $n/\sqrt{p}$ sets of products. takes $t_c\, n^2/p$ time

4. $n/\sqrt{p}$ partial sums to be accumulated along each row (a reduction)

also takes takes $(t_s + t_w n/\sqrt{p}) \log \sqrt{p}$ time on a hypercube with store-and-forward routing using a reduction

5. total parallel time is

$T_P = t_c(n^2/p) + t_s + 2\, t_s \log \sqrt{p} + 3\, t_w\, (n/\sqrt{p}) \log \sqrt{p}$

Simplify

- Can approximate

$T_P = t_c(n^2/p) + t_s + 2\ t_s \log \sqrt{p} + 3\ t_w\ (n/\sqrt{p}) \log \sqrt{p}$

with (substituting $(\log p)/2$ for $\log \sqrt{p}$), ignoring non-p terms

$T_P = t_c(n^2/P) + t_s \log p + (3/2)\ t_w\ (n/\sqrt{p}) \log p$

- will use this expression to find isoefficiency, in particular, using $pT_P = T_0 + T_1$, we find

$T_O = pT_P - T_1$ or

serial work

$T_O = \cancel{t_c n^2} + t_s\, p \log p + (3/2)\ t_w\ (n \sqrt{p}) \log p - \cancel{t_c n^2}$

- and thus $T_O = t_s\, p \log p + (3/2)\ t_w\ (n \sqrt{p}) \log p$

Simplify and analyze

$T_O = t_s \, p \log p + (3/2) \, t_w \, n\sqrt{p} \log p$

- Solve for isoefficiency resulting from the t_w term

Equate each term of T_O with the problem size W in terms of P and constants

$n^2 t_c = K \, (3/2) \, t_w \, n\sqrt{p} \log p$

$n = K \, (3/2) \, (t_w/t_c) \, \sqrt{p} \log p$

$W = n^2 = K \, (9/4) \, (t^2_w/t^2_c) \, p \log^2 p$

constants for a given problem and machine

- The isoefficiency due to t_w is $\theta(p \log^2 p)$
- This is also overall isoefficiency, since it dominates the $\theta(p \log p)$ term involving t_s

What we can conclude

- For the striped model

  $W = n^2 = K^2 t_w^2 P^2$

  and to maintain efficiency, work must increase proportional to P^2
- For the checkerboard model,

  $\theta(p \log^2 p)$ and $p \log^2 p < P^2$
- Therefore, the checkerboard model will scale better than the striped model
- The fundamental reason for this is that the communication is over a smaller number of processors

Isoefficiency and concurrency

- Some algorithms with low overhead also have limited concurrency
- This has a negative effect on isoefficiency, as we will see from Dijkstra's all-pairs shortest-path algorithm
- One instance of Dijkstra's algorithm computes the shortest distance between a single node **s** and all other nodes

Edgar Dijkstra

- Dutch computer scientist, eventually worked to UT Austin, didn't particularly like computers, considered fairly cranky (but very smart and dedicated to teaching) by those who worked with him.

The job [of operating or using a computer] was actually beyond the electronic technology of the day, and, as a result, the question of how to get and keep the physical equipment more or less in working condition became in the early days the all-overriding concern. As a result, the topic became —primarily in the USA— prematurely known as "computer science" —which, actually is like referring to surgery as "knife science"— and it was firmly implanted in people's minds that computing science is about machines and their peripheral equipment. Quod non [Latin: "Which is not true"]

“And I don’t need to waste my time with a computer just because I’m a computer scientist. [Medical researchers are not required to suffer from the diseases they investigate.]” EWD 1305

Edgar Dijkstra

I think anthropomorphism is worst of all. I have now seen programs "trying to do things", "wanting to do things", "believing things to be true", "knowing things" etc. Don't be so naive as to believe that this use of language is harmless. It invites the programmer to identify himself with the execution of the program and almost forces upon him the use of operational semantics.

Edgar Dijkstra

We could, for instance, begin with cleaning up our language by no longer calling a bug a bug but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, viz. with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it disguises that the error is the programmer's own creation... My next linguistically suggestion is more rigorous. It is to fight the "if-this-guy-wants-to-talk-to-that-guy" syndrome: never refer to parts of programs or pieces of equipment in anthropomorphic terminology...

EMD books, Dijkstra font

reasoning steps needed to keep the design
under strict intellectual control. What is
needed to achieve this goal, I can only
describe as improving one's mathematical
skills, where I use mathematics in the
sense of "the art and science of effective
reasoning". As a matter of fact, the challenges
of designing high-quality programs and of
designing high-quality proofs are very
similar, so similar that I am no longer able
to distinguish between the two: I see no
meaningful difference between programming
methodology and mathematical methodology

AaBbCcDdEeFfGgHhIi
JjKkLlMmNnOoPpQqRr
SsTtUuVvWwXxYyZz
1234567890

EMD books, Dijkstra font

I came across a comment on Reddit by someone that had Dijkstra as a professor. Here's what it said:

I've always had horrible handwriting. When I was a computer science student I was in a class taught by Edsger Dijkstra. During the class he asked us to occasionally turn in our notes, because he wanted to see what we thought was important.

The final was an oral final and after going through a few questions to his satisfaction he said "You seem competent, but your handwriting is horrible…" The remaining 30 mins of my final exam by Dijkstra was me writing phrases repeatedly on a pad of paper while he said, 'no, you need to round the o's a bit more, the A is misformed, etc…'.. https://joshldavis.com/2013/05/20/the-path-to-dijkstras-handwriting/

Contributions

- Dijkstra's algorithm
- DJP algorithm
- First implementation of *ALGOL 60*
- Structured programming
- Semaphore
- THE multiprogramming system
- Multithreaded programming
- Concurrent programming
- Principles of distributed computing
- Mutual exclusion
- Call stack
- Fault-tolerant systems
  - Self-stabilizing distributed systems
  - Deadly embrace
  - Shunting-yard algorithm
  - Banker's algorithm
  - Dining philosophers problem
  - Predicate transformer semantics
  - Guarded Command Language
  - Weakest precondition calculus
  - Smoothsort
  - Separation of concerns
  - Software architecture[1]

Structured programming

- Created the phrase *structured programming*
- His March, 1968 letter to the Communications of the ACM, entitled *Go To Statement Considered Harmful* was a major turning point in structured programming
- By the early 1970s, structured programming was firmly engrained in practice

Dijkstra's algorithm

// d_i is the distance from d_s to d_i
// V is the set of N vertices
// T is the set of unprocessed nodes

1.procedure sequential_dijkstra
2.$d_s = 0$
3.$d_i = \infty$, $i \neq s, i \in V$
4.$T=V$
5.for $i=0$ *to* $N-1$
6. find $v_m \in T$ with minimum d_m
7. for each edge (v_m, v_t) with $v_t \in T$
8. if $(d_t > d_m +$ length$((v_m, v_t)))$ then
9. $d_t = d_m +$ length$((v_m, v_t))$
10. $T = T - v_m$

at each step *i*, finds shortest paths from v_s to nodes of length *i*

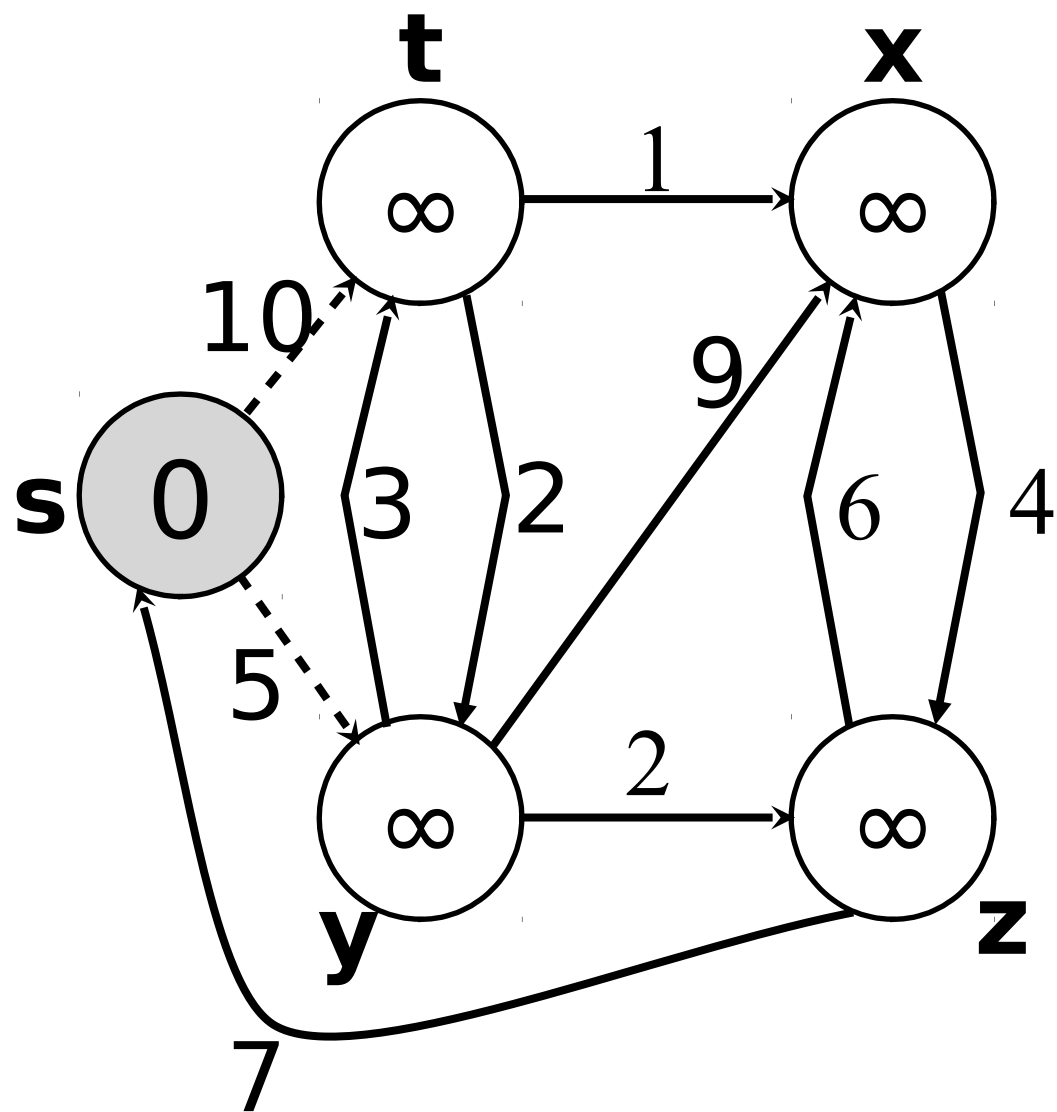
To find the shortest path from a vertex *s* to all other vertices

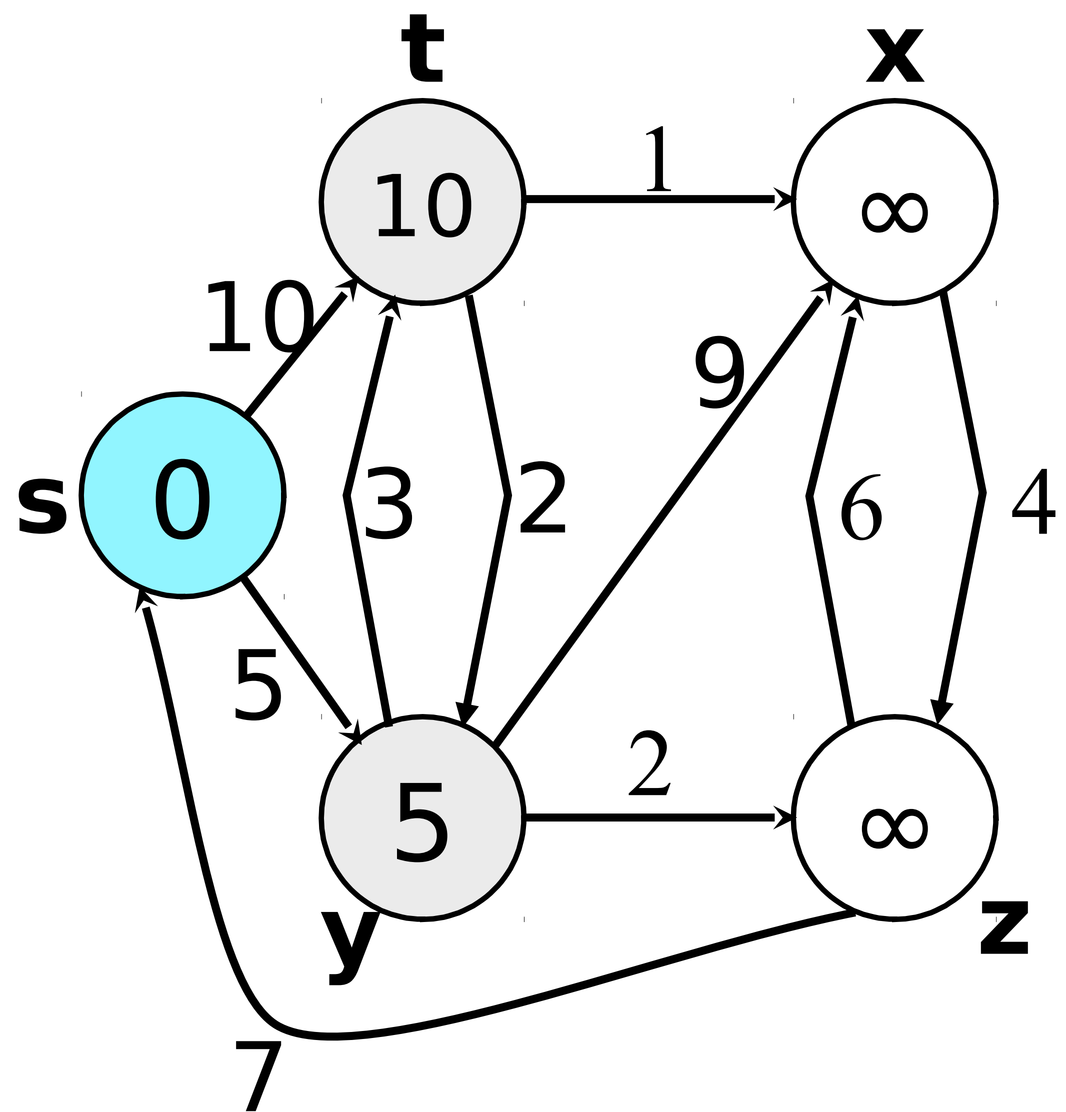
At each step

pick the node to be processed (a member of T) v_m that is closest to s (this is v_m on the first iteration)

for every other node v_t that is to be processed
see if there is a edge from v_m to v_t that leads
to a shorter distance from *s* to v_t

remove v_m from the set *T* of unprocessed nodes

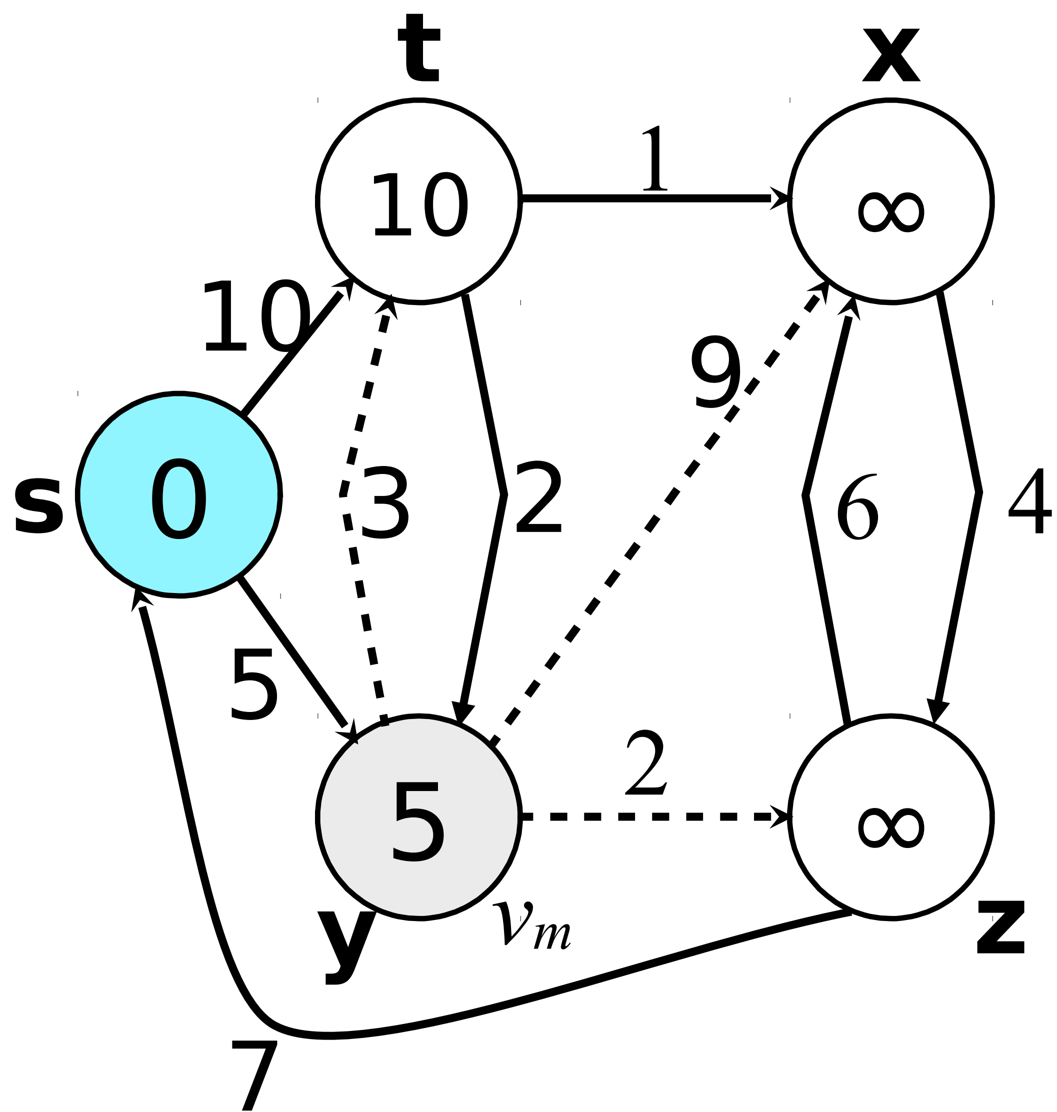


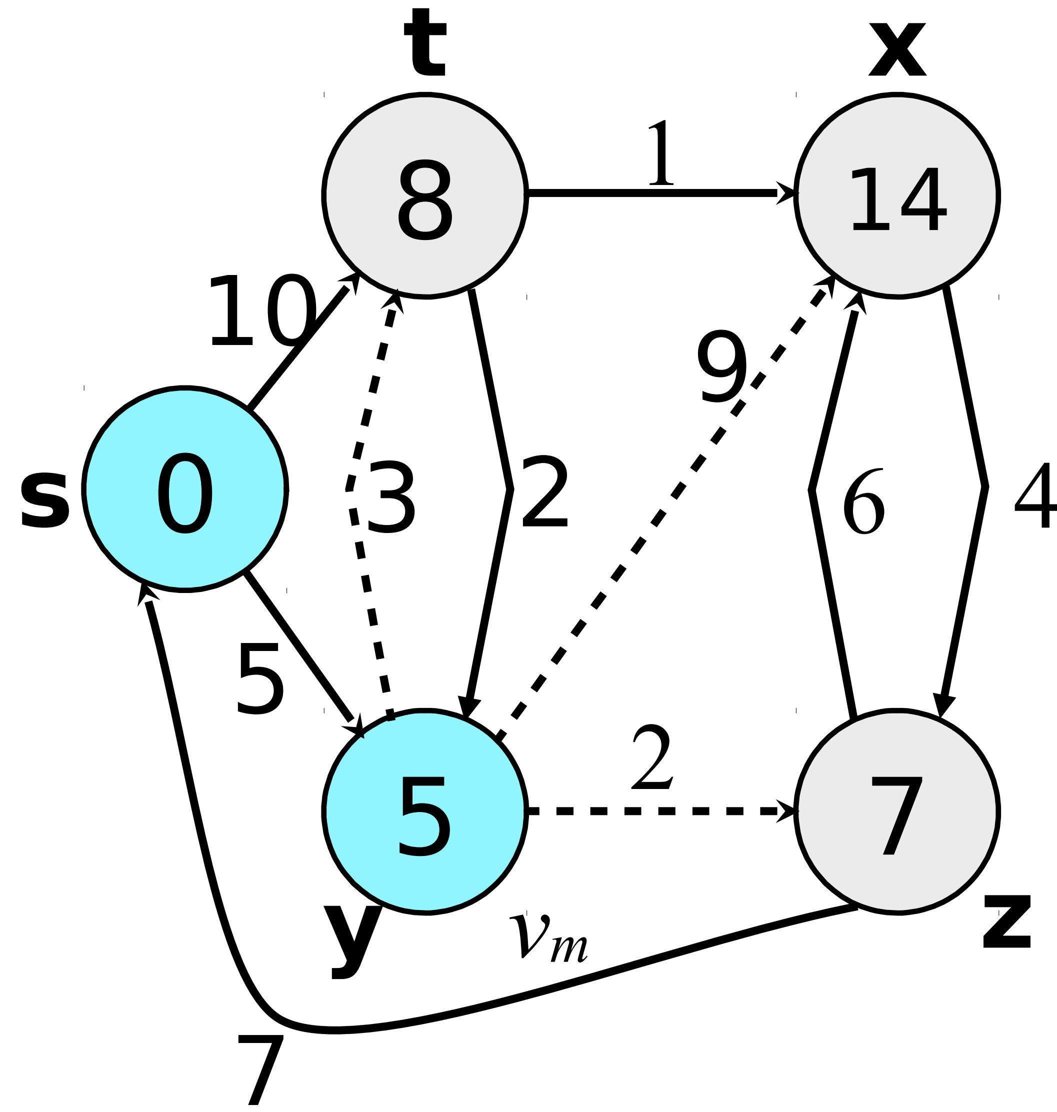


step 1

$v_m = \mathbf{s}$

$d_m = 0$

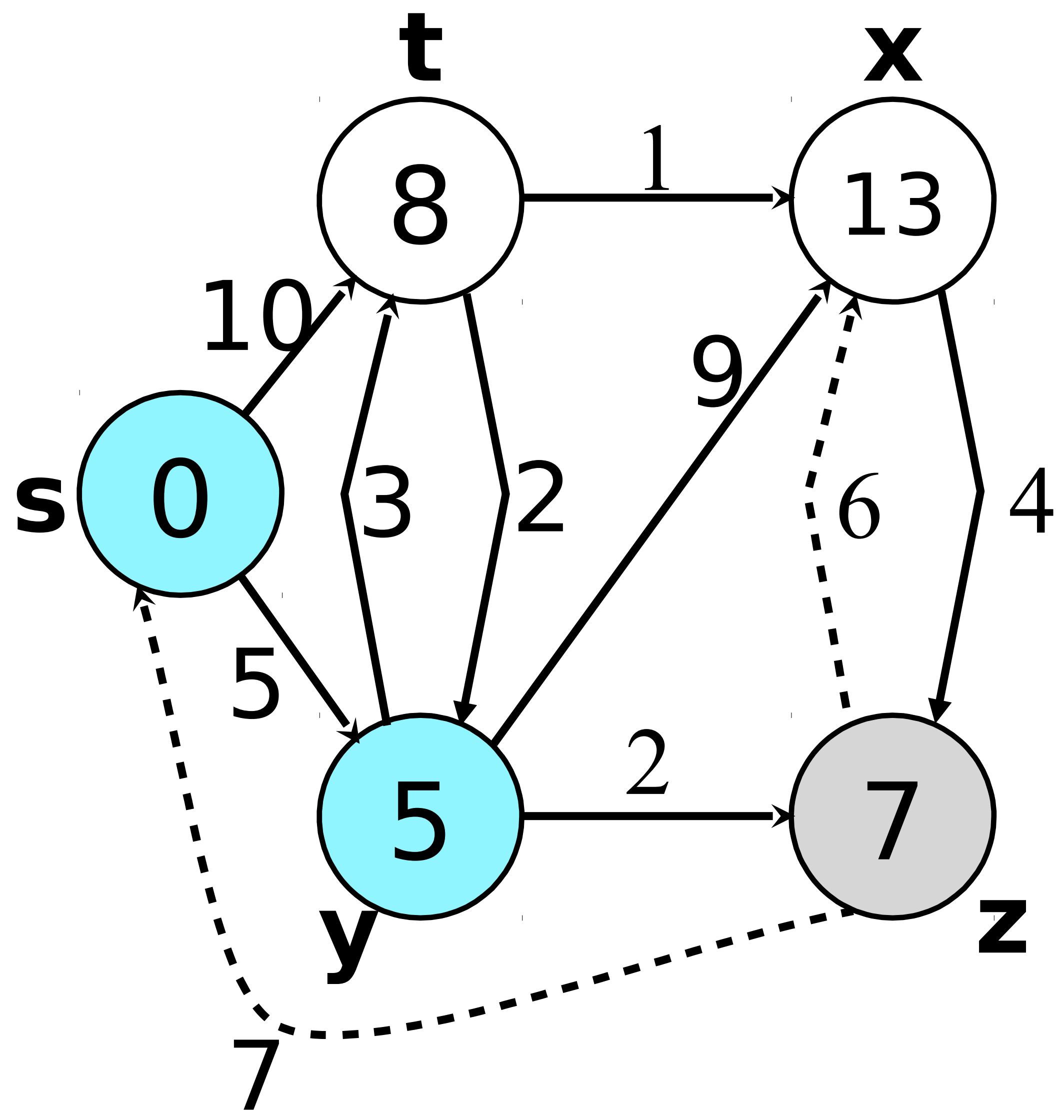




step 2

$v_m = \mathbf{y}$

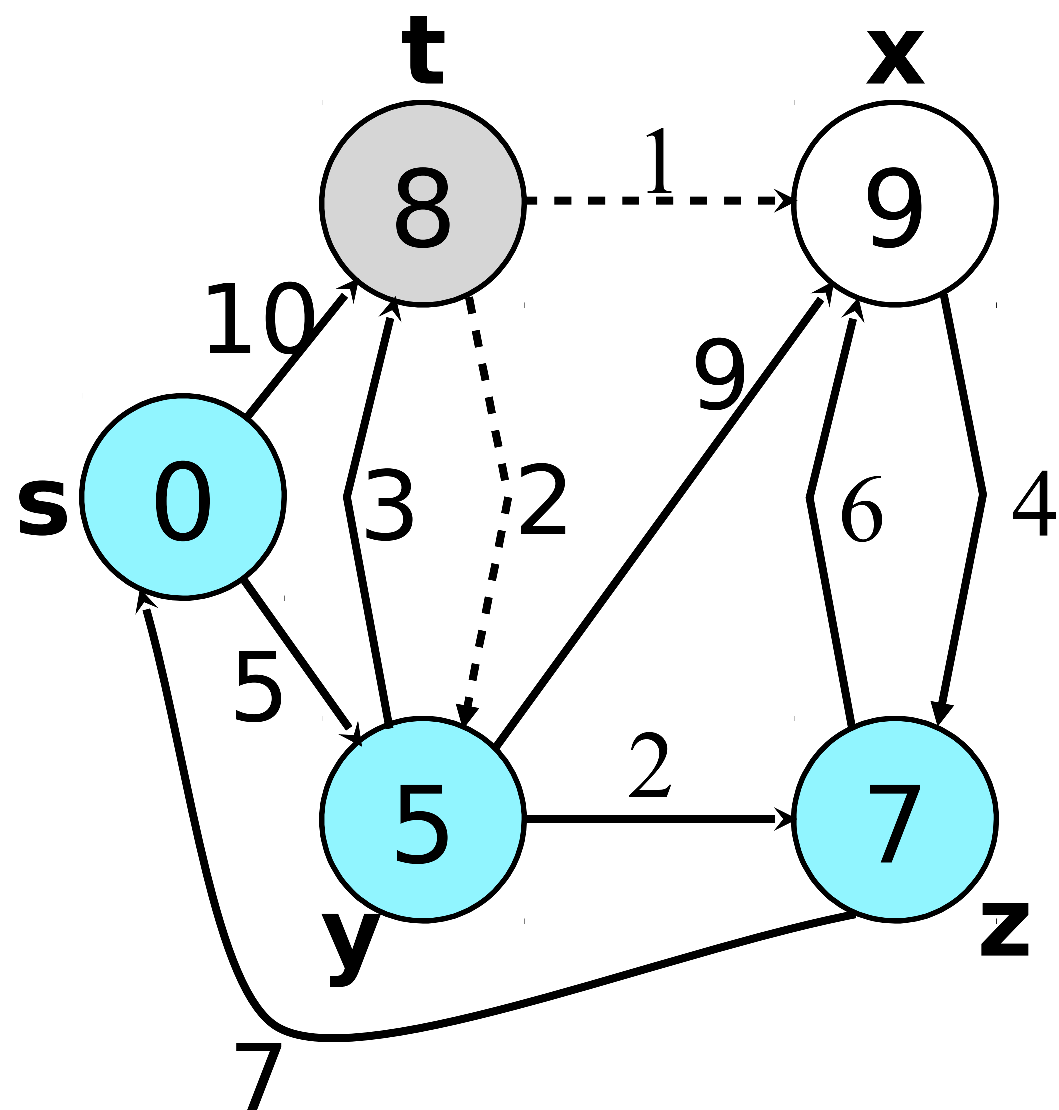
$d_y = 5$



step 3

$v_m = \mathbf{z}$

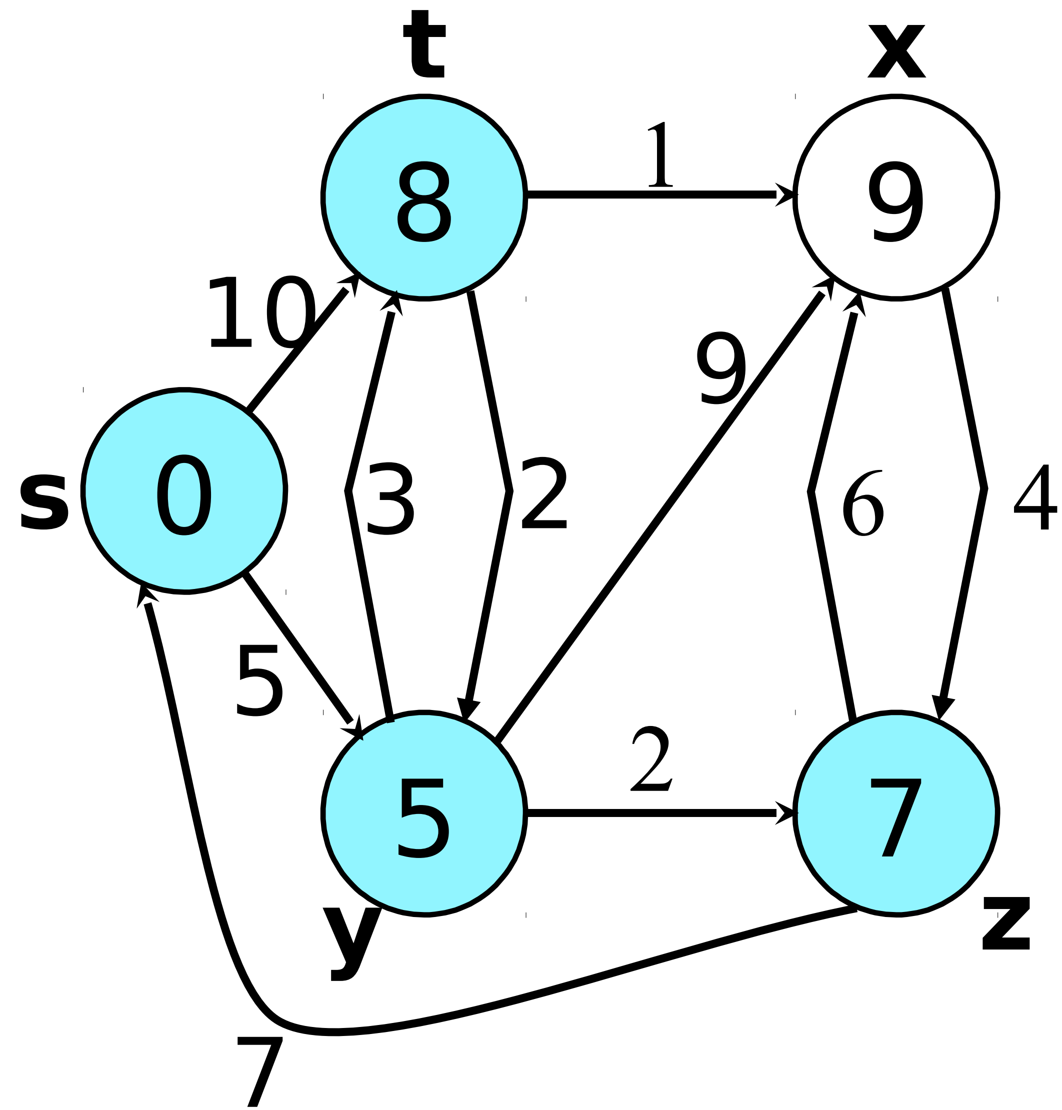
$d_z = 7$



step 4

$v_m = \mathbf{t}$

$d_t = 8$



step 5

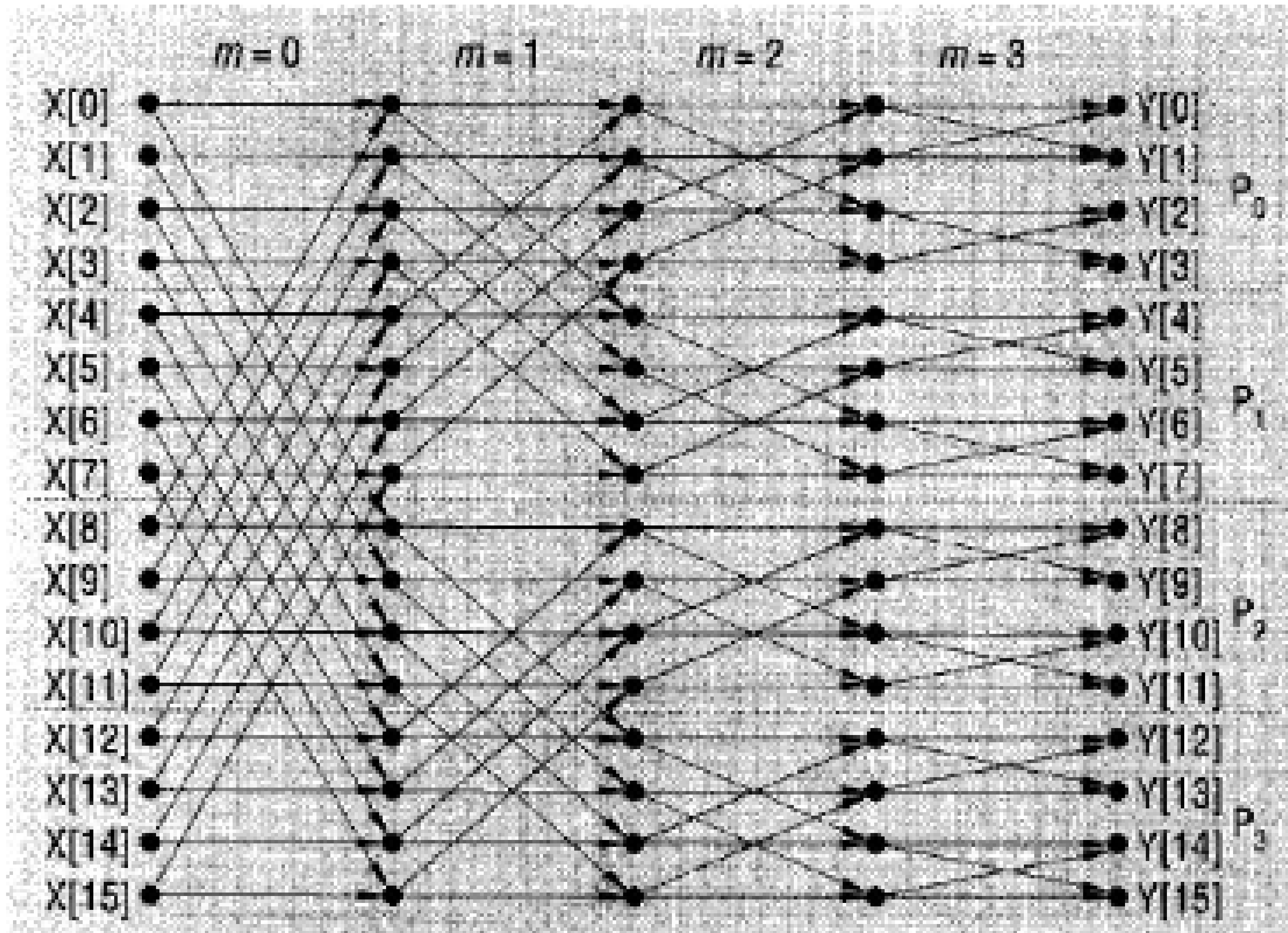
$v_m = \mathbf{t}$

$d_m = 8$

A parallel Dijkstra's algorithm for all paths

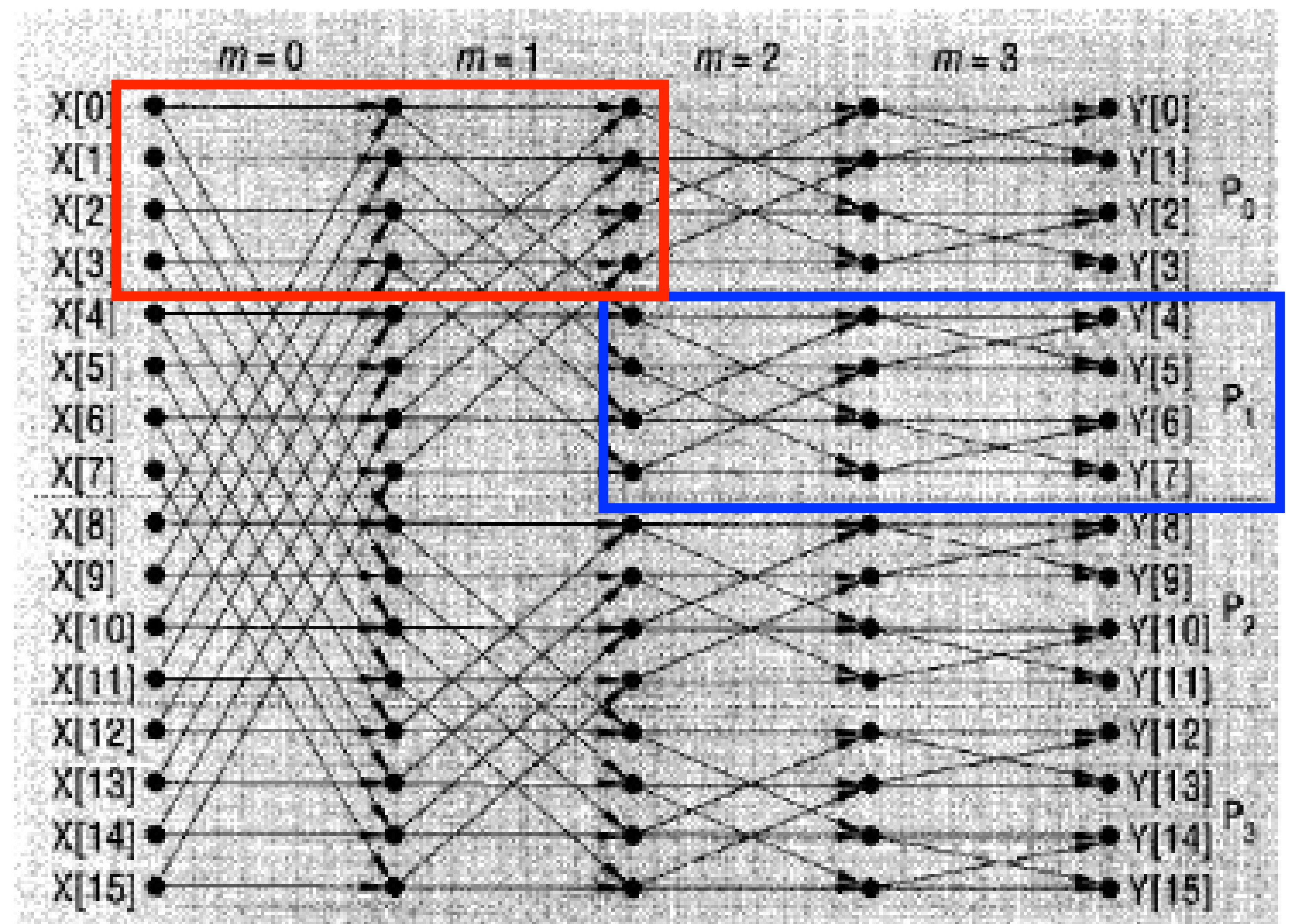
- Replicate the graph *N* times (*N* is the number of vertices), with each processor getting *N/P* vertices to treat as *s* vertices, i.e., N/P vertices to find shortest paths from it to other vertices
- Each node computes the shortest distances from the *N/P* vertices it owns to all other N vertices
- No communication needed
- Seems like the perfect algorithm, but it isn't
- $O(N^3)$ work, but only $O(N)$ parallelism
- W is $\theta(N^3)$, $P=N$, W must grow as $\theta(P^3)$ to scale and the isoefficiency is high

Cooley-Tukey FFT Algorithm – the iso-efficiency relationship depends on the machine parameters for bandwidth and operation time



Machine specific parameters

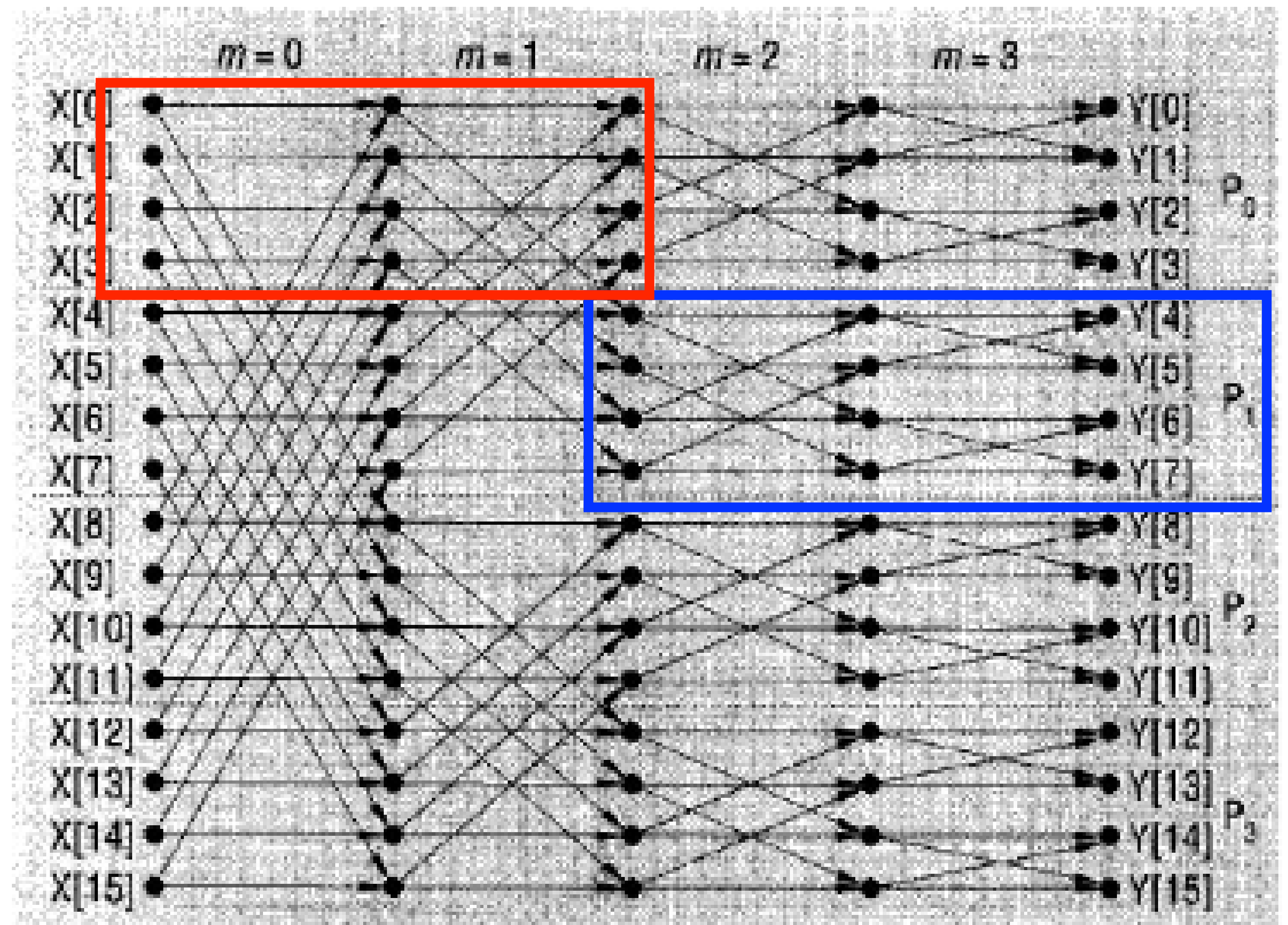
- Sequential complexity is $\theta(n \log n)$
- Parallel version based on the *binary exchange* method for a *d*-dimensional ($P=2^d$) hypercube
- partition vectors into block of n/p contiguous elements, $n=2^r$
- 1 block of 2^{r-d} elements assigned per processor



r = log2 16, r = 4
4 dimensional hypercube, d = 2

Machine specific parameters

- vector elements on different processors combined during first d iterations, pairs on the same processors combined in the last r-d iterations
- interprocessor communication in only $d = \log P$ of the $r = log\ n$ iterations
- Each communication exchanges n/P words
- Communication time is $(t_s + t_w\ n/P) \log P$



- During each iteration a processor updates n/P elements of vector r
- Let each complex multiply take time t_c

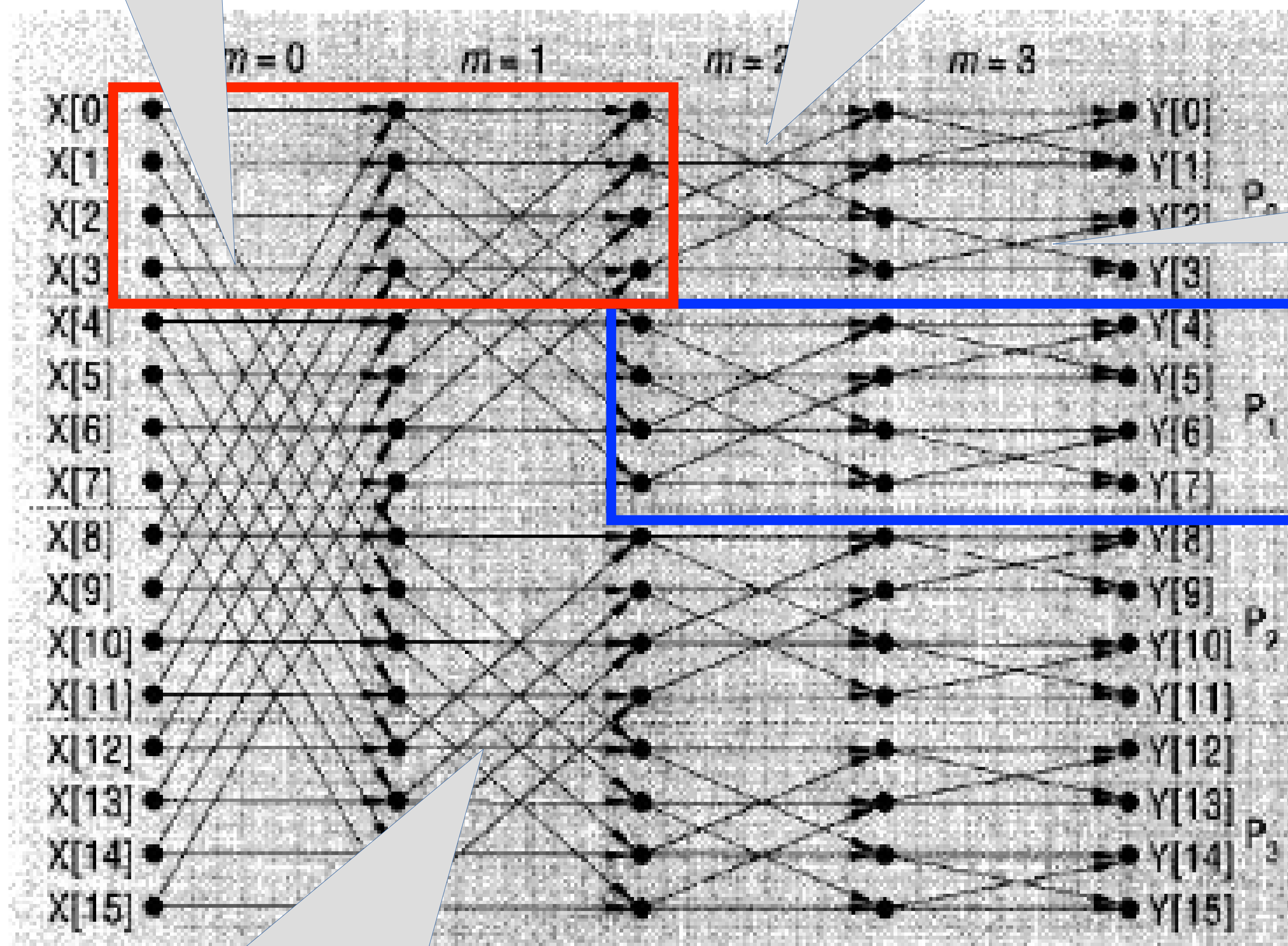
Machine specific parameters

Always talk to adjacent node in a hypercube

differ in high order bit

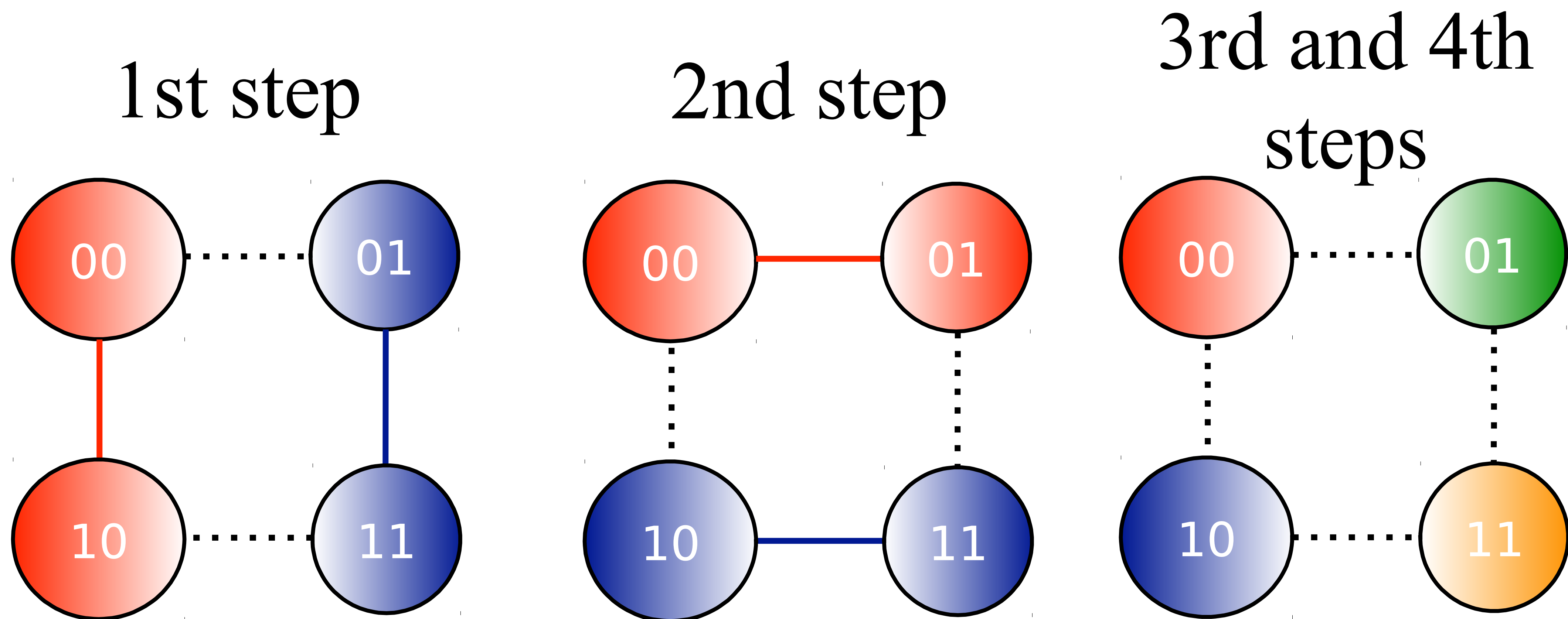
differ in next to low order bit

differ in low order bit



differ in next to high order bit

- On a hypercube communicating nodes are *always* adjacent, i.e. a single hop to communicate
- Allows each communication to happen in time $t_s + t_w\, n/P$ time
- With d communicating steps, hypercube will communicate over each adjacent edge during computation



Parallel execution time

- $T_P = t_c(n/P) \log n + t_s \log P + t_w (n/P) \log P$

Computation time

startup times for log p communications

startup times for log p communications

- $T_O = P(t_s + t_w n/P) \log P = t_s P \log P + t_w n \log P$
- $W = n \log n$

Solve for different terms

- First term (t_s), $W \propto P\, t_s \log P$, isoefficiency function is $P \log P$
- Second term, $n \log n = K\, t_w\, n \log P$

$\log n = K t_w \log P$

$n = P^{K(t_w/t_e)}$

$n\, log\, n = K\, t_w\, P^{K t_w} \log P$

Substituting for K

$W = E/(1-E)\ (t_w/t_c) P^{E/(1-E)(t_w/t_c)} \log P$

Isoefficiency a function of E

- $W = E/(1-E)\ (t_w/t_c) P^{E/(1-E)(t_w/t_c)} \log P$ (from the previous slide)
- Consider if exponent of P, $t_w E/(t_c\,(1-E)) < 1$

W grows slower than $P \log P$

Overall isoefficiency is $\theta(P \log P)$ (from t_s term the previous page)

- Consider if $t_w E/(t_c\,(1-E)) > 1$

isoefficiency a function of relative values of

$E/(1-E),\ t_w,\ t_c$

- Consider if $t_w E/(t_c\,(1-E)) = 1$

Isoefficiency is $P \log P$, a lower threshold for a hypercube

effect of $t_w E/(t_c(1\text{-}E))$ on isoefficiency

- if $t_w = t_c$, isoefficiency is $W=E/(1\text{-}E)P^{E/(1\text{-}E)}\log P$
- Now for $E/(1\text{-}E) \leq 1$, $E \leq 0.5$

isoefficiency is $\theta(P \log P)$

- For $E/(1\text{-}E)$, $E \geq 0.5$

If $E=0.9$, $E/(1\text{-}E)=9$, isoefficiency is $P^9 \log P$

- Effect of t_w and t_c: let's make the bandwidth lower

if $t_w=2t_c$ then the *threshold* efficiency is 0.333

Isoefficiency for $E=0.333$ is $\theta(P \log P)$

Isoefficiency for $E=0.5$ is $\theta(P^2 \log P)$ and for

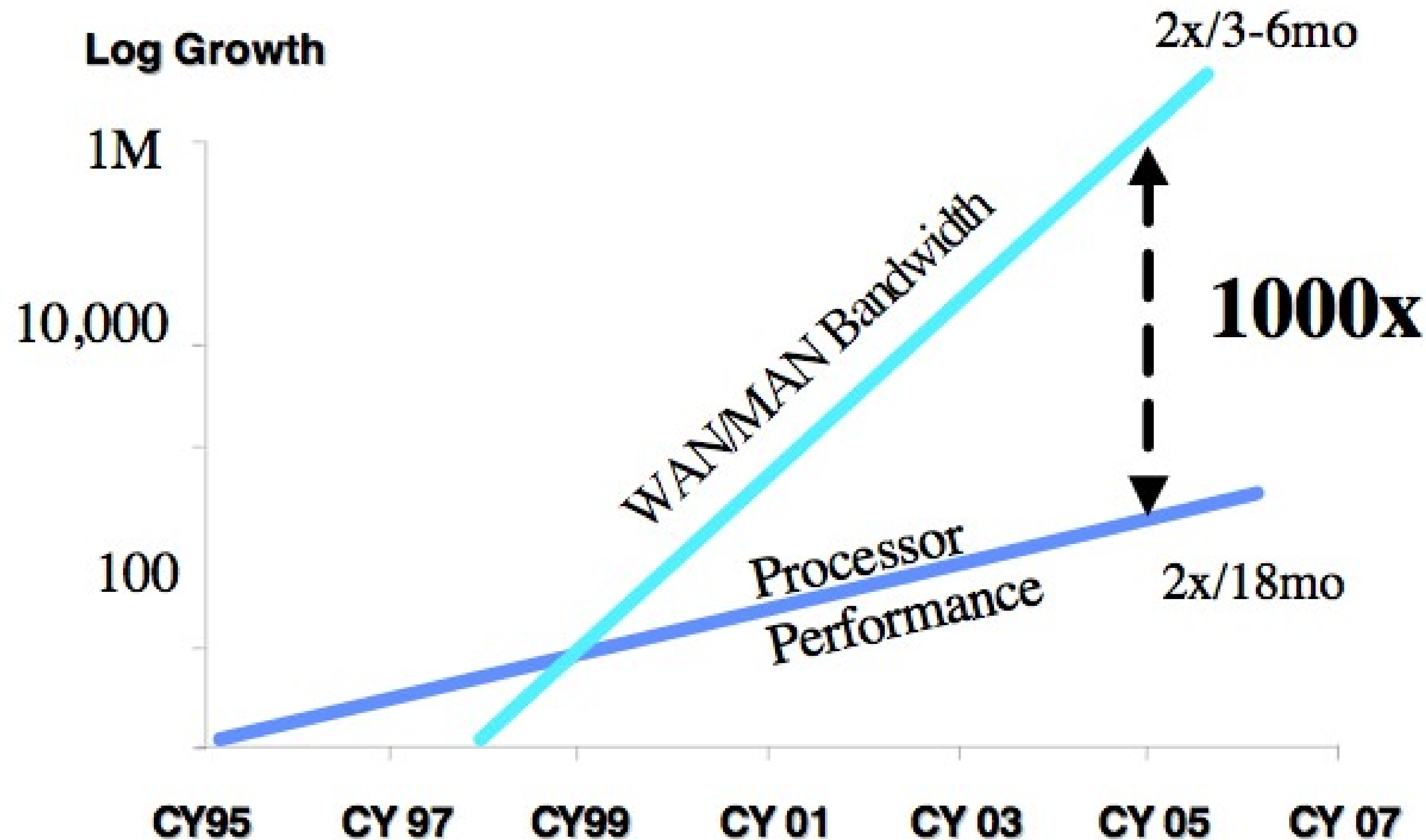
$E=0.9$ is $\theta(p^{18} \log p)$ $(t_w E/(t_c(1\text{-}E)) = 2E(1\text{-}E) = 1.8/0.1)$

- What can we conclude from this?

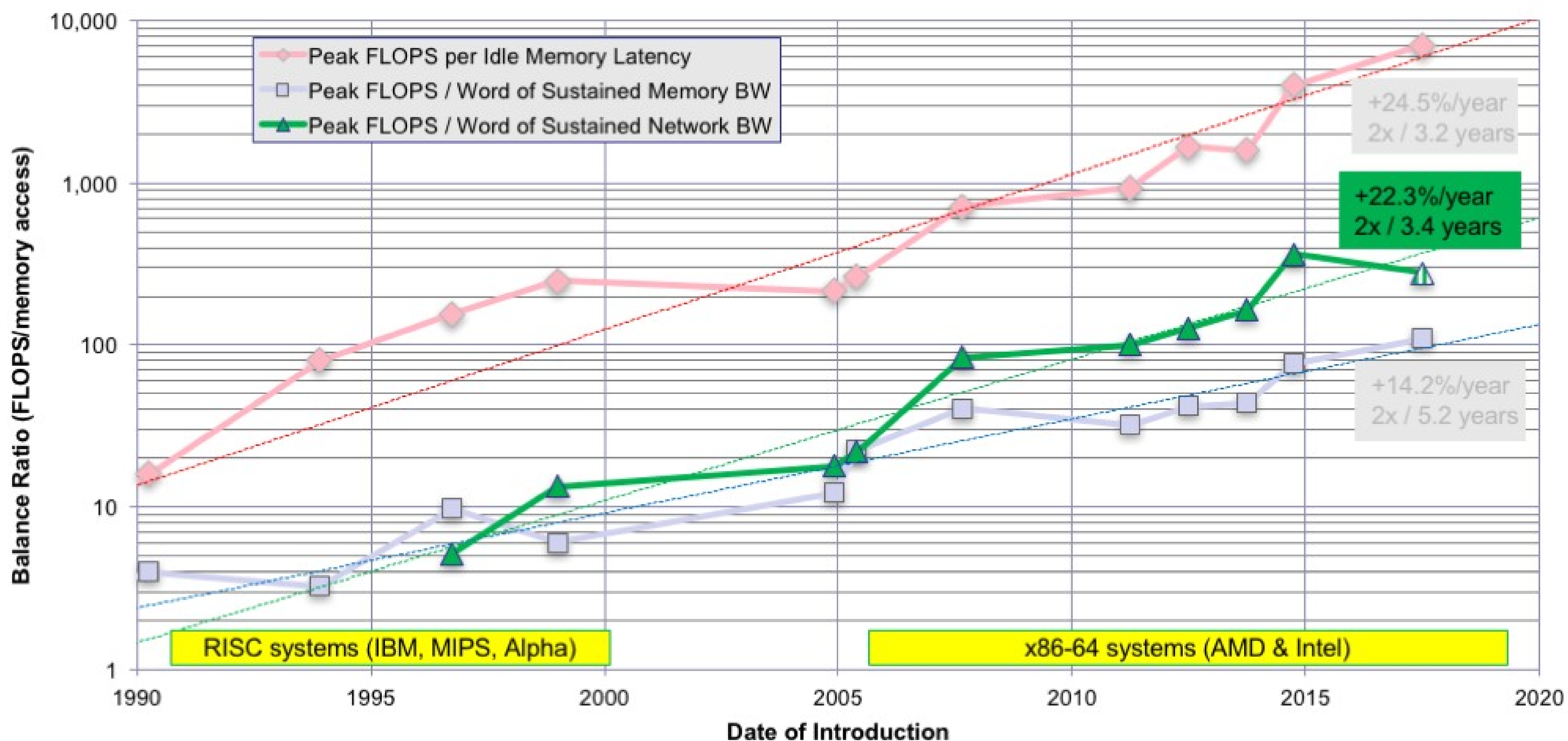
Conclusions for FFT

- Balance of bandwidth and CPU is important for this problem - scalability is good on a balanced system
- Making bandwidth higher helps
- Increasing CPU performance without increasing bandwidth reduces scalability
- On modern systems . . .

From a talk by Horst Simon



Interconnect Bandwidth is Falling Behind at a comparable rate



FFT is unique in this property

- **But**, the ratios of t_w and t_c can be high
- May result in t_c term being important in small machine sizes, and the t_w or t_s terms dominating for larger machines
- Again, need to apply intelligence, and again, using isoefficiency gives insights into what is required to have an app scale

Summary

- Data structure contention also must be considered if it is the dominating term
- In summary:
  - Want to increase problem size to maintain efficiency
  - Must have enough memory to hold larger problem size
  - Rate of growth of problem size is a limit on the number of processors we can run on
  - Thus rate of growth of problem size is a limit on how scalable the algorithm is *if we want to maintain constant efficiency*
  - Isoefficiency functions provide a way of determining the rate of growth of the problem size