# Performance Analysis

Jingke Li

Portland State University



## Metrics and Approaches

Prove that parallel algorithm it.

Two commonly used performance metrics for parallel programs:

Execution time — direct reflection of program performance Absolute value

Speedup — direct reflection of improvement over sequential version Improvement

*Program Performance Analysis:*

- Static analysis and modeling — Analyze performance by examining program against an abstract performance model.
  - *"The program runs in $O(N)$ time with $O(N)$ processors."*, or more specifically, *"We can model the performance of the program by the function: $T(N, P) = (N^2 + N)/P + 100$."* // not for all applications.
- Extrapolation from actual performance data.
  - *"The program running on parallel machine X achieved the following speedup: 9.8 with 10 processors; 19.2 with 20 processors, and 28.7 with 30 processors. We hence conclude that the program achieves linear speedup."* Downside → how to connect current performance to general form.



Execution time doesn't always show all results needed.

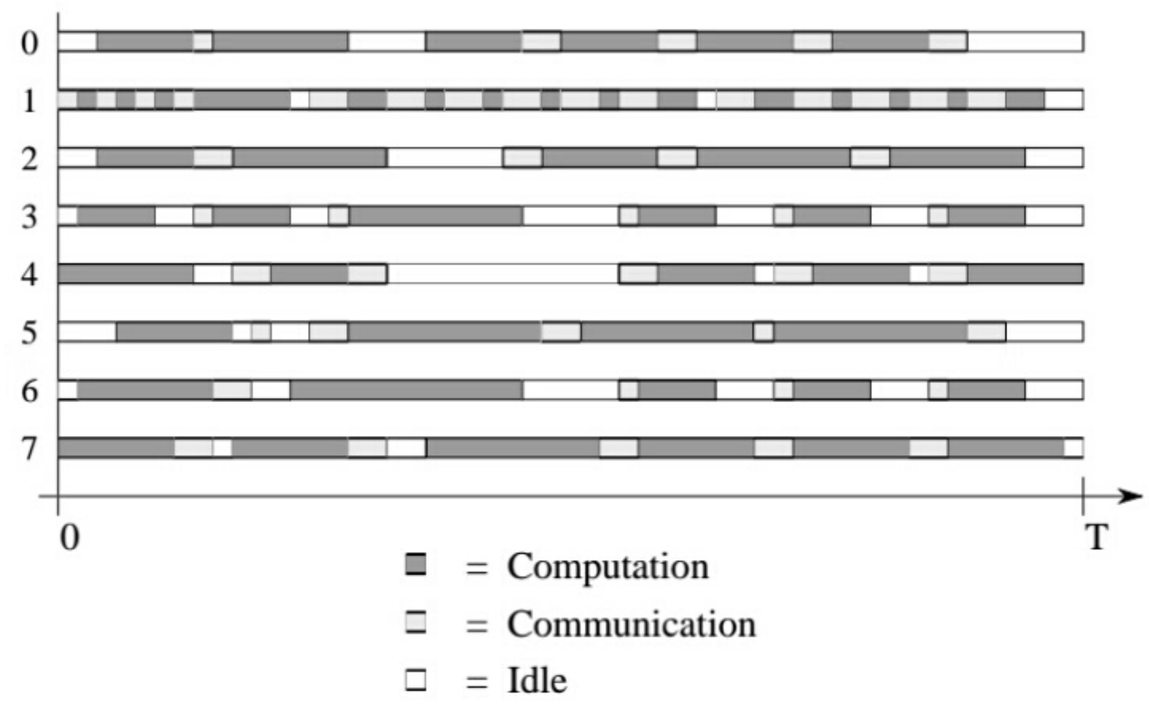
Theoretical analysis helps to avoid or predict general performance bounds.

For many appli

What defines execution time? → total time of all cpus or just the total time.

## Execution Time

For a parallel program, there can be multiple processes running on multiple processors.




How should execution time be defined?



## Execution Time (cont.)

Three choices:

- *Maximum* elapsed time: $T = \max_{i=0}^{P-1} T^i$
- *Average* elapsed time: $T = \frac{1}{P} \sum_{i=0}^{P-1} T^i$
- *Total* elapsed time: $T = \max_{i=0}^{P-1} t^i_{\text{end}} - \min_{i=0}^{P-1} t^i_{\text{start}}$

For users, the total elapsed time is often the most useful. But it is not always directly measurable.

## Measuring Time in C/Linux

Multiple timing routines are available through C libraries or Linux.

- time() — returns the current time in seconds. // not granular enough

```
#include <time.h>
time_t time(time_t *tval);
```

  // coarse grained

  - The return value represents the elapsed seconds since the Epoch, which is defined as 00:00 1/1/1970, GMT.
  - If `tval` is not `NULL`, the return value is also stored in `*tval`.

- clock() — returns the CPU time used by the calling process.

```
#include <time.h>
clock_t clock(void);
```

// of all cpus invded!

  - The CPU time is represented in terms of number of clock ticks. To convert it to seconds, divide it by `CLOCKS_PER_SEC`, which is typically set at $1,000,000$.



## Measuring Time in C/Linux (cont.)

- gettimeofday() — returns the real time at microsecond scale. // not used

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, void *tz);
```

struct
→ seconds
→ microseconds

  - `struct timeval` contains two members, `tv_sec` and `tv_usec`, whose values are seconds and microseconds since the `EPOCH`, respectively.
  - The parameter `tz` is obsolete, and should be set to `NULL`.

- clock_gettime() — returns the real time at nanosecond scale. // not available on all systems

```
#include <sys/time.h>
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

  - This function is a successor to `gettimeofday()`.
  - The parameter `clk_id` can be set to report either real time or CPU time.
  - `struct timespec` contains two members, `tp_sec` and `tp_nsec`, whose values are seconds and nanoseconds since the `EPOCH`, respectively.



Inconsistency in naming is a problem.

Linux

Gives another angle.

## Measuring Time in C/Linux (cont.)

- times() — returns both the CPU time (for the caller and all its terminated children) and the clock time, in number of clock ticks.

```
#include <sys/times.h>
clock_t time(struct tms *buffer);
```

  - struct tms contains the following members:

```
clock_t tms_utime;  /* user CPU time */
clock_t tms_stime;  /* system CPU time */
clock_t tms_cutime; /* user time of children */
clock_t tms_cstime; /* system time of children */
```

  - To convert a clock_t value to seconds, divide the value by the constant CLK_TCK, which typically is set to be 100.



## An Example

```
#include <sys/time.h>   /* for gettimeofday */
#include <time.h>       /* for clock_gettime */

int main() {
  struct timeval  start1, end1;
  struct timespec start2, end2, start3, end3;

  gettimeofday(&start1, NULL);
  sleep(1);
  gettimeofday(&end1, NULL);
  clock_gettime(CLOCK_REALTIME, &start2);
  sleep(1);
  clock_gettime(CLOCK_REALTIME, &end2);
  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start3);
  sleep(1);
  clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end3);

  printf("From gettimeofday:  sec=%ld, micro-sec=%ld\n",
         end1.tv_sec - start1.tv_sec, end1.tv_usec - start1.tv_usec);
  printf("From clock_gettime: sec=%ld, nano-nsec=%ld\n",
         end2.tv_sec - start2.tv_sec, end2.tv_nsec - start2.tv_nsec);
  printf("Elapsed CPU time:   sec=%ld, nano-sec=%ld\n",
         end3.tv_sec - start3.tv_sec, end3.tv_nsec - start3.tv_nsec);
}
```

## Measuring Time in MPI

// These are just wrappers around system implementations.

- MPI provides a wall-clock timer,

```
double MPI_Wtime(void);
```

  - It returns a double precision value that represents the number of seconds that have elapsed since some point in the past.

- A related function

```
double MPI_Wtick(void);
```

  - returns the precision of the timer.



## MPI Timing Example

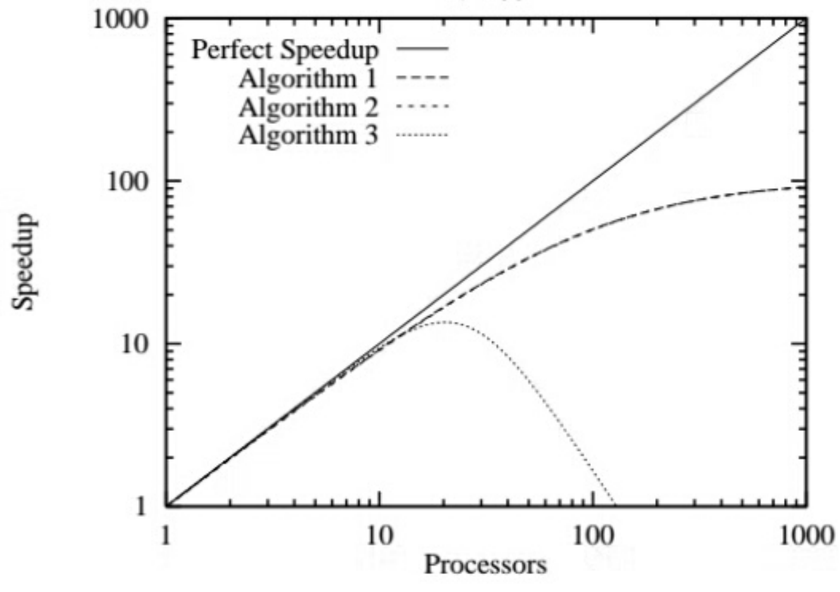
```
#include <mpi.h>

int main(int argc, char **argv) {
  int rank;
  double start, end, res;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  start = MPI_Wtime();
  sleep(1);
  end = MPI_Wtime();
  res = MPI_Wtick();
  printf("From P%d: %e seconds (resolution: %e)\n",
         rank, end - start, res);
  MPI_Finalize();
  return 0;
}
```

```
linux> mpirun -n 2 mpitime
From P0: 1.000105e+00 seconds (resolution: 1.000000e-06)
From P1: 1.000135e+00 seconds (resolution: 1.000000e-06)
```

## Speedup

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

- $T_s$ — Execution time using a single processor
- $T_p$ — Execution time using $N$ processors




*A Simple Observation:* $S \leq N$

↳ in theory but not in practice

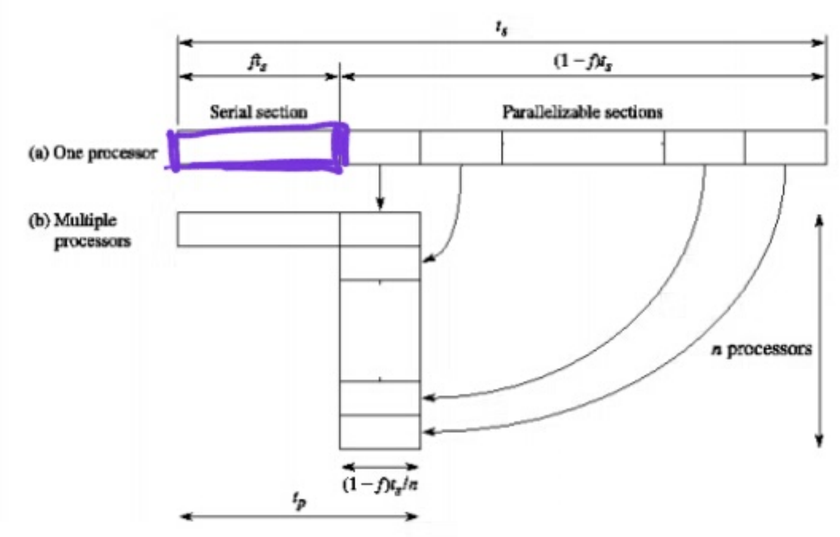


## Amdahl's Law

*The speedup of a program is upper bounded by the reciprocal of the sequential fraction of the program.* (The bigger the fraction, the smaller the speedup.)

For a given program, let $f$ be the sequential fraction, then

- $fT_s$ is the time required on sequential portion
- $(1-f)T_s$ is the time required on parallelizable portion




$$S = \frac{T_s}{T_p} = \frac{T_s}{fT_s + (1-f)T_s/N} = \frac{1}{f + (1-f)/N} \rightarrow \frac{1}{f}$$

↳ tends limit with $N \to \infty$

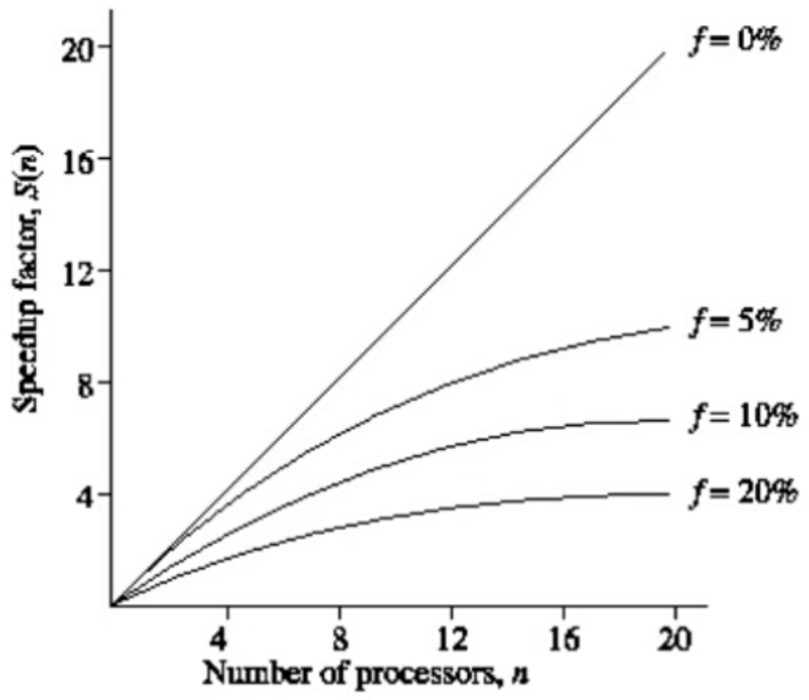


Flaw: wrt B sequentialized version of

Amdahls law is very pessimistic it basically tells us that x100 is the best speedup possible.
However it was later discovered that real speedup can be > x1000

## Amdahl's Law Example

*E.g.* If the sequential component of a program is 5%, then the maximum speedup that can be achieved is 20.






## Problems with Amdahl's Law

- Concepts are not precise. → Depends on problem size.
  - Both "sequential parts" and "parallelizable parts" can be moving targets. parts Dynamically change.
- Fixed problem size is an unrealistic assumption. As problem size gets bigger sequential parts gets smaller.
  - Smaller problems are usually run on larger machines.
- Communication and synchronization overhead is not included.
  ↳ too optimistic, do not include overhead.

Various proposals.
Amdahl's Law is still good

## Fixed-Size Speedup → Amdahl's Law.

The speedup model described above can be called *fixed-size* speedup, since it apply to the case when the *same* program is run on 1 and $P$ processors.

$$S_{FS}(P) = \frac{T_s(N_0)}{T_p(N_0)}$$ → problem size

*Example:* Given a program with following features:

$$\textit{Time:} \quad T_s(N) = 1,000,000 + 1,000N + 24N^2 \ (\mu s)$$
$$T_p(N) = 1,500,000 + 1,050N/P + 24N^2/P \ (\mu s)$$
$$\textit{Space:} \quad S_s(N) = 100,000 + 200N \ (\text{bytes})$$
$$S_p(N) = 125,000 + 200N/P \ (\text{bytes})$$

Compute the fixed-size speedup for $N_0 = 1,000$ → in this case works fine

*Answer:* $S_{FS} = \frac{T_s(N_0)}{T_p(N_0)} = \frac{26}{1.5+25.05/P} \to \frac{26}{1.5} = 17.33$ → best speedup for problem size $N_0 = 1000$



## Other Speedup Models

One problem with is model is that when the program size is really big, one may not be able to run it on a single processor. To overcome this problem, two other speedup models are proposed.

- *Fixed-Memory* Speedup (*Scaled* Speedup) — Memory capacity of each node
  Each processor runs a sub-program of equal size.

  $$S_{FM}(P) = \frac{T_s(PN_0)}{T_p(PN_0)}$$ (where $\frac{N}{P} = N_0$ is a constant)

  PN0 → too big for sequential machine
  → run first  P×N0 → total size in message passing Prodeg

- *Fixed-Time* Speedup —
  Each processor runs for a fixed amount of time.

  $$S_{FT}(P) = \frac{T_s(N_p)}{T_p(N_p)}$$ (where $T_p(N_p)$ is a constant)

  Same time as it takes $T_s$ (but doesn't matter) can be any time interval

Symmetry to the first proposal



Many systems measured by government so what is the speedup in some interval of time (for example can buy only 1 now).

## Back to the Example

*Time:* $T_s(N) = 1,000,000 + 1,000N + 24N^2 \ \ (\mu s)$

$T_p(N) = 1,500,000 + 1,050N/P + 24N^2/P \ \ (\mu s)$

*Space:* $S_s(N) = 100,000 + 200N \ \ (\text{bytes})$

$S_p(N) = 125,000 + 200N/P \ \ (\text{bytes})$

Compute (1) scaled speedup for $N/P = N_0$, and (2) fixed-time speedup for $T_p = T_s(N_0)$. (Recall $N_0 = 1,000$.)

*Answer:*

$S_{FM} = \frac{T_s(PN_0)}{T_p(PN_0)} = \frac{1+P+24P^2}{2.55+24P}$ (unbounded!)

$S_{FT} = \frac{T_s(N_p)}{T_p(N_p)}$, where $T_p(N_p) = T_s(N_0) = 26 \cdot 10^6$, or

$N_p = \frac{-1,050/P + \sqrt{(1050/P)^2 - 4\cdot(-24.5\cdot 10^6)\cdot 24/P}}{2\cdot(24/P)}$

Still unbounded but not as steep as $S_{FM}$



## Result Comparison

| $P$ | Fixed Size Spdup | Fixed Memory $T_P$ | Fixed Memory Spdup | Fixed Time $N_p$ | Fixed Time Space | Fixed Time Spdup |
|---|---|---|---|---|---|---|
| 1 | 0.98 | 26.6 | 0.98 | 989 | 323 kB | 0.98 |
| 2 | 1.85 | 50.6 | 1.96 | 1,407 | 266 kB | 1.92 |
| 4 | 3.35 | 98.6 | 3.95 | 1,999 | 225 kB | 3.80 |
| 8 | 5.61 | 194.6 | 7.94 | 2,836 | 196 kB | 7.57 |
| 16 | 8.48 | 386.6 | 15.94 | 4,020 | 175 kB | 15.11 |
| 32 | 11.39 | 770.6 | 31.94 | 5,694 | 160 kB | 30.18 |
| 64 | 13.75 | 1,538.6 | 63.94 | 8,061 | 150 kB | 60.33 |
| 128 | 15.33 | 3,074.6 | 127.94 | 11,409 | 143 kB | 120.63 |
| 256 | 16.27 | 6,146.6 | 255.94 | 16,144 | 138 kB | 241.24 |
| 512 | 16.78 | 12,290.6 | 511.94 | 22,840 | 134 kB | 482.46 |
| 1,024 | 17.06 | 24,578.6 | 1,023.94 | 32,310 | 131 kB | 964.90 |

26 seconds

Time to run



Three different measurements give three different angles.

## Other Related Concepts

tries to capture workload

- *Efficiency:* $\boxed{E = \frac{T_s}{T_p N}}$ $(\times 100\%)$ i.e single machine resources vs multi. N -> number of resources

  Efficiency is typically given as a percentage. It indicates the fraction of the time that the processors are being used on the computation.

- *Cost:* $\boxed{C = T_p N}$ Looking at pieces of the same measurement at different angle

  With this concept, one can talk about *cost-optimal* (parallel) algorithms, for which the cost to solve a problem is proportional to the execution time on a single processor.



## Performance Analysis Steps

Parallel programs require overhead.

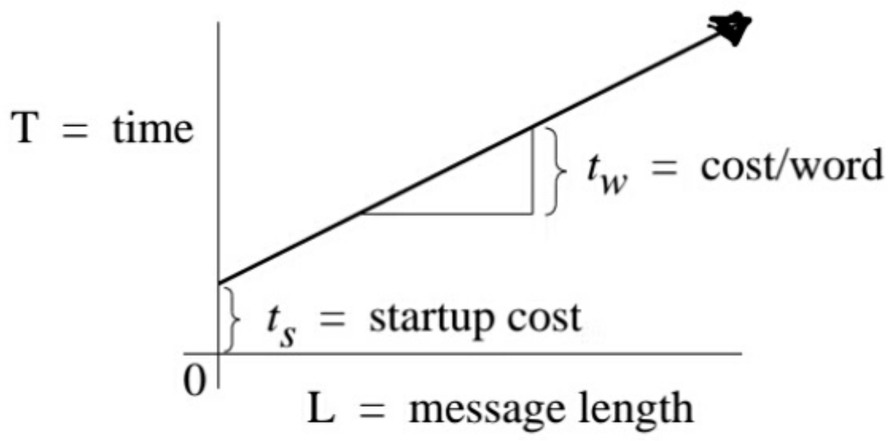
1. Develop performance models:
   - For computation time, use same technique as algorithm complexity analysis.
   - For communication time, need to have performance models for basic communication patterns.
2. Fit data to models:
   - Given a set of observed performance data points and a (hypothetic) performance function, use a statistic method to find the parameters in the function using statistic method.
   - An example of static method is the *least-square fit* method — Find parameters for function $f$, such that $\sum_i (\text{obs}(i) - f(i))^2$ is minimized.

To tune performance need to run!

Cost to send and receive

## A Cost Model for One-to-One Communication

Collision, time sharing, blocking etc.




$$t_{one} = t_s + t_w N$$

$t_s$ -> header message (20bytes), $t_w$ - Linear to the size of the message not the distance. all messages < 100 bytes takes the same time as 100 bytes



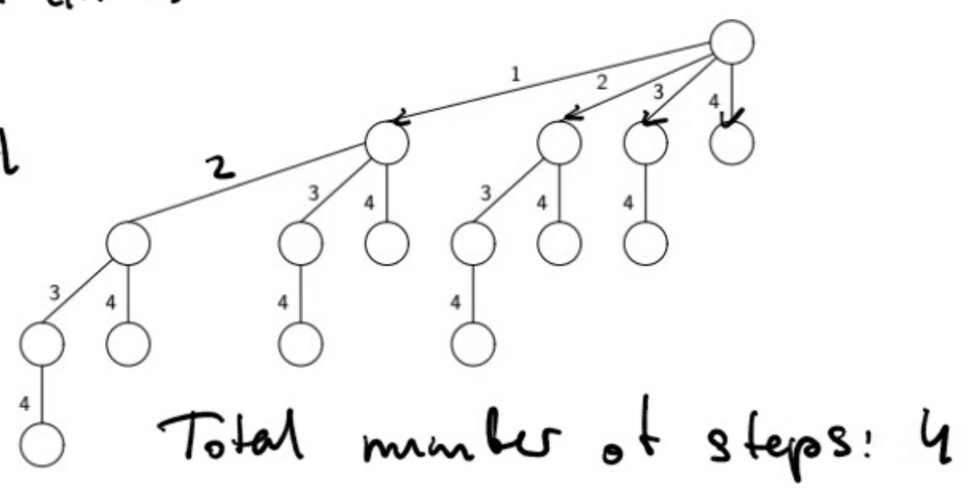
## Analysis on Collective Communication Routines

Although the implementation details of a collective communication is topology dependent, its abstract message pattern stays the same.

For example, for one-to-all broadcast, the pattern is a *binomial tree*:

Each node has 4 links

Growth pattern follows binomial sequence.




Total number of steps: 4

Thus, the implementations of one-to-all broadcast on different topologies can be viewed as the embedding of a binomial tree in those topologies.



Analysis depends greatly on implementation

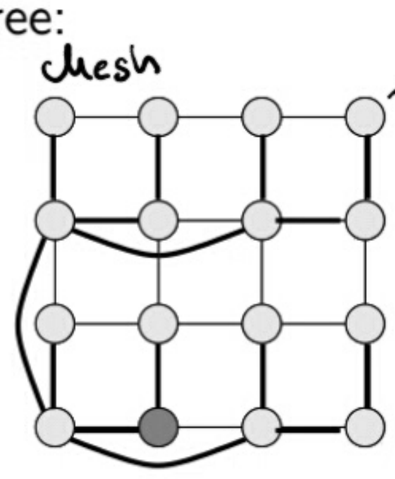
What is the best time? log(n)? → ~~send to~~ each node sends two messages. however not possible to send 2 messages simultaniously.

Hypercube is expensive 4 physical links per node

## Analysis on Collective Communication Routines (cont.)

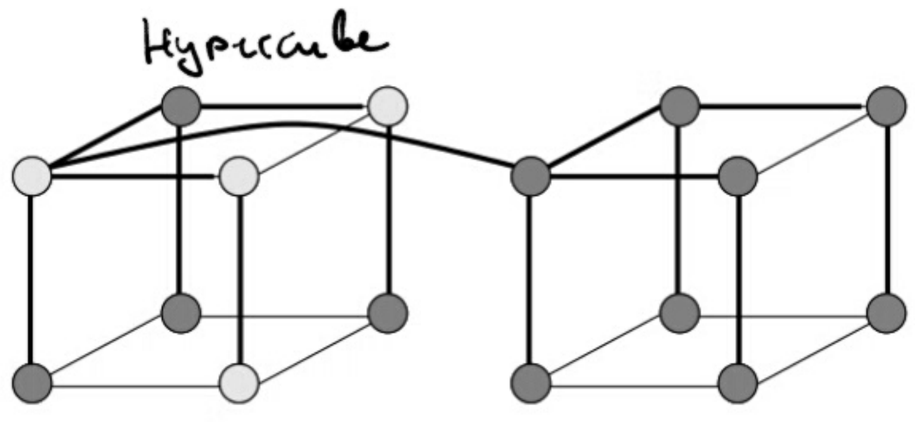
Embedding of a binomial tree:

Distance is not an issue




Embed Bynomial tree into a physical network

No matter what the physical Layout Bynomial tree can be embedded into it






Remember Performance of Each.

## Cost Models for Collective Communications

Each individual step has the same cost as single send/receive.

- *Broadcast and Reduction*: $\boxed{t_{bc} = t_{re} = (t_s + t_w m) \log p}$

  It involves $\log p$ steps of (concurrent) point-to-point message transfers.

- *Scatter and Gather*: $\boxed{t_{sc} = t_{ga} = t_s \log p + t_w m(p-1)}$

  Bynomial is not the best solution, order of (P)

  one→all || one→many

  These communication differ from broadcast and reduction in that the message size starts out at $m(p-1)/2$, and gets halved in each subsequent step (or vise versa). message size is important

  Bynomial

- *All-to-All*:
  - With concurrent one-to-alls: $\boxed{t_{aa} = t_s \log p + t_w m(p-1)}$
  - With circular shifts: $\boxed{t_{aa} = (t_s + t_w m)(p-1)}$

Complexity (All-All) → same order as scatter and gather



All to all implementation: 1. ~~Each~~ one→all for each node

2. circular shift (prefered)

## Analyzed vs. Observed Performance

Sometimes, the observed execution times are greater than predicted by a model, resulting in a worse than expected speedup. Some of the possible causes are:

- Load imbalances
- Replicated computation
- Tool/algorithm mismatch
- Competition for bandwidth
- Overly simplified models

Analyzed performance doesnt account for all of these

Occasionally the opposite can occur, i.e. the observed speedup is *greater* than predicted, sometime even greater than linear. What could be the causes?

- Cache effects
- Search algorithms

In general speedup is bounded by the number of processes (theoretically)

In real world it is possible



1. Less memory traffic (Cache effects)
2. Search algorithm
   Serial → search entire space

   Parallel → partition among processes
   hence might return immediately if one of the processes finds the value