Performance Analysis

Jingke Li

Portland State University

Metrics and Approaches

Two commonly used performance metrics for parallel programs:

Execution time — direct reflection of program performance

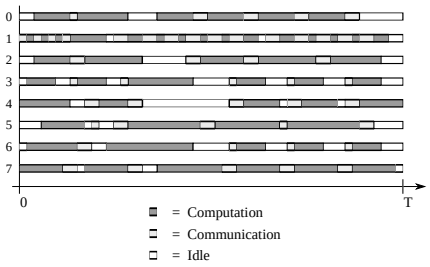
Speedup — direct reflection of improvement over sequential version

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: $T(N, P) = (N^2 + N)/P + 100$."*
- 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."*

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.

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

  - 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);
```

  - 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.

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

  - `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.

```
#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.

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

- 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

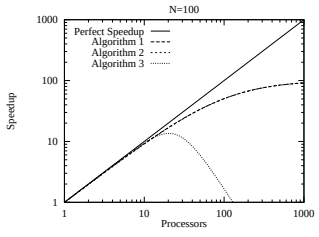
```
#include <mpi.h>

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



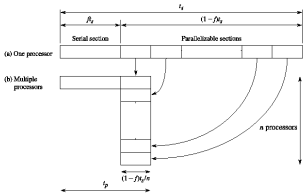
A Simple Observation: $S \leq N$

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

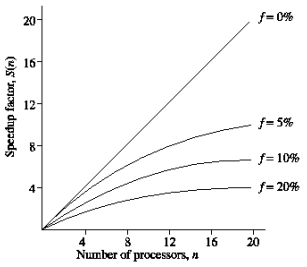
- fT_s is the time required on sequential portion
- $(1-f)T_s$ is the time required on parallelizable portion



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

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.
  - Both "sequential parts" and "parallelizable parts" can be moving targets.
- Fixed problem size is an unrealistic assumption.
  - Smaller problems are usually run on larger machines.
- Communication and synchronization overhead is not included.

Fixed-Size Speedup

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.

$$\boxed{S_{FS}(P) = \frac{T_s(N_0)}{T_p(N_0)}}$$

Example: Given a program with following features:

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 the fixed-size speedup for $N_0 = 1,000$

Answer: $S_{FS} = \frac{T_s(N_0)}{T_p(N_0)} = \frac{26}{1.5+25.05/P} \rightarrow \frac{26}{1.5} = 17.33$

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) —
  Each processor runs a sub-program of equal size.

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

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

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

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

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

Result Comparison

P	Fixed Size	Fixed Memory		Fixed Time		
	Spdup	T_P	Spdup	N_p	Space	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

Other Related Concepts

- *Efficiency:* $\boxed{E = \frac{T_s}{T_p N}}$ $(\times 100\%)$

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

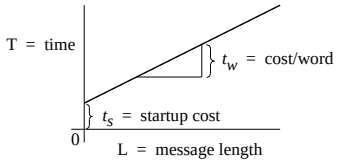
- *Cost:* $\boxed{C = T_p N}$

  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

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.

A Cost Model for One-to-One Communication

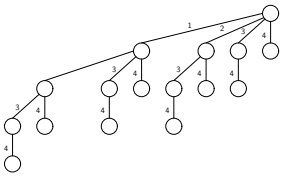


$\boxed{t_{one} = t_s + t_w N}$

Analysis on Collective Communication Routines

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

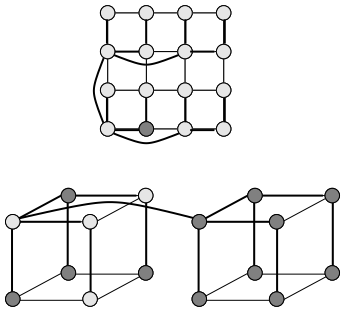
For example, for one-to-all broadcast, the pattern is a *binomial tree*:



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 on Collective Communication Routines (cont.)

Embedding of a binomial tree:



Cost Models for Collective Communications

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

  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).

- *All-to-All*:

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

  – With circular shifts: $\boxed{t_{aa} = (t_s + t_w m)(p-1)}$

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

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