Parallel Software (MK book, §2.4)

- Issues of writing software for parallel systems

ex) MIMD
- typical shared-memory program:
  - start a process → forks threads
- typical distributed-memory program:
  - start multiple processes

focus is on SPMD – single program/multiple data ↔ Running a different program on each core (MPMD)

ex) Task parallel

```
if (process_id/thread_id == 0)
    do this;
else
    do that;
```

Data parallel

```
if (process_id/thread_id == 0)
    operate on the 1st half of the array;
else
    operate on the 2nd half of the array;
```

SPMD model is suitable for implementing both data and task parallelism.

Task parallel program – dividing tasks among processes/threads.

Coordinating processes/threads:

1. Divide work among processes/threads

   Load balancing, minimum communication

2. Synchronization among processes/threads
3. Communication among processes/threads

- **Shared-memory system issues**

Dynamic threads – master thread forks a thread (joins to master thread) when needed;

vs.

→ high cost of thread fork/join (time-consuming operations)

Static threads – all are forked at the same time;

→ less efficient resource usage (idle thread holds resources)

Non-determinism: vague order, ex) which thread terminate earlier/later

ex)
```
my_val = compute_val (my_rank);
x += my_val;   // x is shared_var, my_val is local to each thread
```

one exec scenario:

time ↓	thread_0		thread_1
	x = 0	------	. . .
	my_val = 7	------	x = 0
	x = x + 7	------	my_val = 9
	store x = 7	------	x = x + 9
	. . .	------	store x = 9

finally, shared x = 9, which is different from the desired result (16).

→ problem: simultaneous updates of shared location.

critical section – shared location is updated

Critical section (C.S.) should be mutually exclusive;

How to implement? - using a mutex semaphore
- monitor
- busy-waiting (highly inefficient)

```
Lock
  C.S.
Unlock
```

C.S. is serialized among processes/threads

Thread safty – some built-in serial func.s malfunction in parallel program

- **Distributed-memory system issues**

  multiple processes, each on an independent processor

  message-passing API (ex, MPI)

  ref)
  It's possible to use distributed-mem API on shared-mem system;
  Programmer logically partitions shared-mem into private address spaces;

  send/receive functions

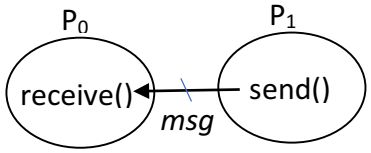
  ex)
```
char msg[100]; //local to each process
my_rank = get_rank( );
if (my_rank == 1)
{ msg = ". . . from process_1";
  send (msg, MSG_CHAR, 100, 0);
}
else if (my_rank == 0)
{ receive (msg, MSG_CHAR, 100, 1);
  cout<<. . . ;
}
```

  dest P_0

  type of msg

  source P_1

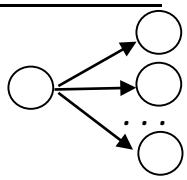
  SPMD: 2 proc's use same code, but different actions

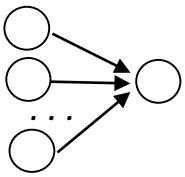


  functions: send( ) – 1. Sending proc is blocked until receiver is ready;
  or, 2. sending proc sends data to a buffer and continue op;

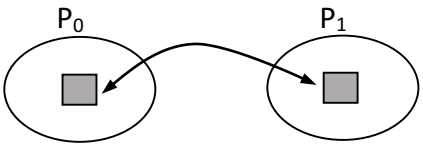
  receive( ) – receiving process is blocked until msg is received;

  Collective communication

  broadcast 

  reduction 

  One-sided communication (ex, remote mem access in DSM)

  not easy to use 

  P_0 updates P_1's mem with P_0's data
  or, P_0 updates P_0's mem with P_1's data

Programming hybrid systems – e.g., cluster of multicore processors

It's possible to use both MPI and OpenMP, but difficult to implement;

→ using only MPI for both inter- and intra-node communication.

Input and Output (MK book, §2.5)

C/C++ with OpenMP, MPI, Pthread

Multiple processes/threads → non-determinism on I/O (stdin, stdout, stderr)

ex) P_0 P_1 P_2 . . . P_n

only 1 process should be responsible for output

Assumptions/suggestions/rules:

- in distributed-mem programs – only process_0 will access stdin;
- in shared-mem programs – only the master thread (Thread_0) will access stdin;
- in both distributed-/shared-mem programs,
  all processes/threads can access stdout/stderr;
  → non-deterministic
  → Sol: only 1 process/thread should be used for stdout/stderr
- only a single process/thread attempts to access any single file;
  each process/thread can open its own private file for r/w, but no two processes/threads will open the same file
- debug output should include the rank (id) of process/thread;

Performance (MK book, §2.6)

ideal

$$T\_parallel = \frac{T\_serial}{P}$$ //linear speedup (ideal), where P is number of processors

overheads: increases as # of processes/threads increases

shared-mem – critical sections are executed in serial (with mutex);

distr-mem – data transition across IN (slower than local mem access);

ideal

ideal (linear) speedup $S = \frac{T\_serial}{T\_parallel} = \frac{T\_serial}{\frac{T\_serial}{P}} = P$

in reality,

as P increases, S becomes smaller;

as P increases, $\frac{S}{P}$ (efficiency of parallel program) becomes smaller;

efficiency (speedup per processor) $E = \frac{S}{P} = \frac{\frac{T\_serial}{T\_parallel}}{P} = \frac{T\_serial}{P * T\_parallel} = \frac{1}{P}(\frac{Ts}{Tp})$

effective

effective $T\_parallel = (\frac{T\_serial}{P}) + T\_overhead$

effective speedup $S = \frac{T\_serial}{effective\ T\_parallel} = p * (\underbrace{\frac{Ts}{Ts + P * To}}_{< 1})$

ex) Table 2.4

P	1	2	4	8	16	cores/processors
S	1.0	1.9	3.6	6.5	10.8	
E	1.0	0.95	0.9	0.81	0.68	

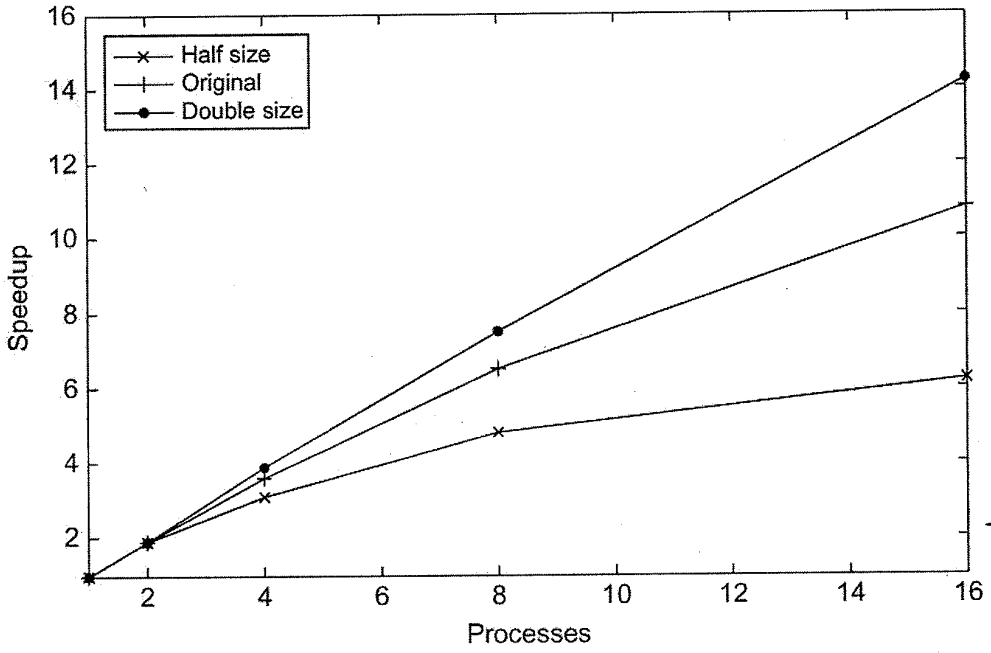


Fig 2.18 Speedups of parallel program on different problem sizes

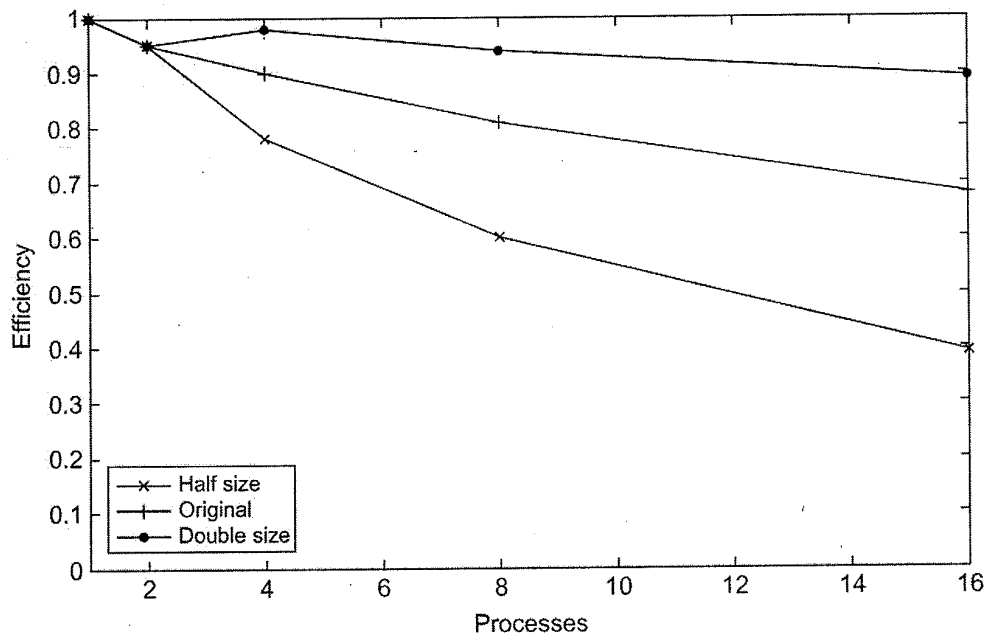


Fig 2.19 Efficiencies of parallel program on different problem sizes

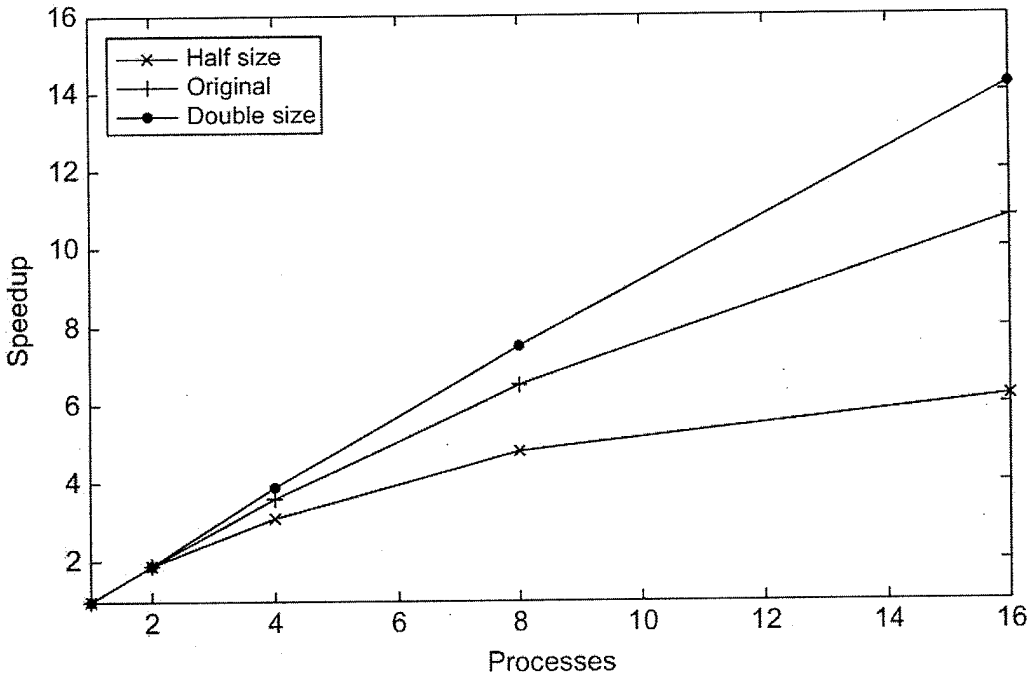


Fig 2.18 Speedups of parallel program on different problem sizes

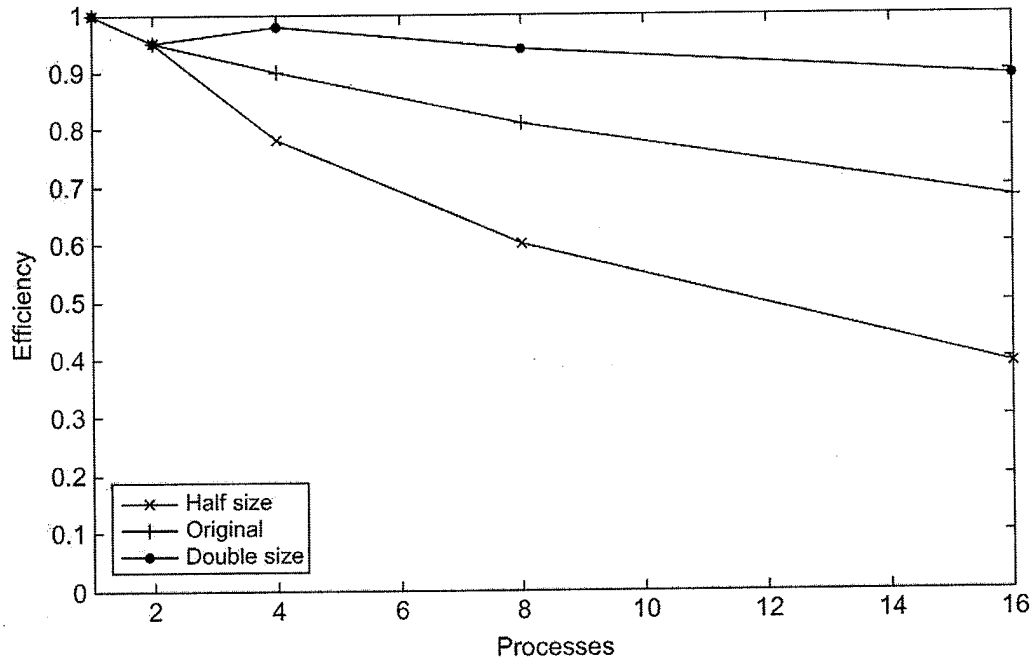


Fig 2.19 Efficiencies of parallel program on different problem sizes

Time checking (measuring parallel program run time)

Elapsed time = CPU time + I/O time *//$>time for elapsed time*

Run time of parallel program – varies at different run

wall-clock time from start to finish of any interested part

ex) double start, finish;

```
. . . .
start = Get_current_time( );
. . . .
. . . .
finish = Get_current_time( );
cout<<finish – start;
```

get wall-clock time

ex) MPI:
MPI_Wtime( );
OpenMP:
Omp_get_wtime( );

Global time checking for multiple processes/threads:

ex) shared double global_elapsed;
private double my_start, my_finish, my_elapsed;
barrier( ); //synchronize all processes/threads
my_start = Get_current_time( );
. . . .
. . . .
my_finish = Get_current_time( );
my_elapsed = my_finish – my_start; //for each local process/thread
//find the maximum across all processes/threads
global_elapsed = Global_max(my_elapsed); //updates shared location
if (my_rank == 0) //if my_elapsed > current_max
cout<<global_elapsed;

Reported run time should exclude I/O time.

Maximum speedup from parallelization

Amdahl's law

$$\text{Speedup} = \frac{1}{(1-Fe)+(\frac{Fe}{Se})}$$

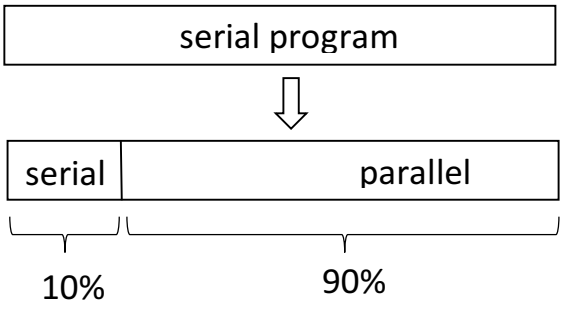
ex) 10% of a task is serial, 90% of the task is parallelizable;

using P processors, ideal (linear) speedup = P

→ maximum speedup $S = \frac{1}{(1-0.9)+(\frac{0.9}{P})} = \frac{1}{0.1 + \underbrace{(\frac{0.9}{P})}} = \frac{1}{0.1} = 10$

when P gets larger => 0

analysis:



T_serial = 20 sec (assume)

T_par = 0.9 * ($\frac{Ts}{P}$) + 0.1 * Ts

= $\frac{18}{P}$ + 2 sec

→ $S = \frac{Ts}{Tp} = \frac{20}{\frac{18}{P} + 2}$

as P gets larger, $\frac{18}{P}$ => 0

so, $S \leq \frac{20}{2} = 10$

Scalability

Assume that n (problem size) and P (# of processes/threads) yields efficiency E.

If we increase
- n → k * n
- P → k * P

and this yields same efficiency E, then the parallel program is scalable;

more specifically, the parallel program is weakly scalable.

strongly scalable:

only p → k * P (regardless of the problem size n) yields same efficiency E

weakly scalable:

both p → k*P and n → k*n yields same efficiency E
(same increasing factor k for P and n)

ex) problem size = n

T_s = n

T_p = ($\frac{n}{P}$) + 1

→ E = $\frac{S}{P}$ = $\frac{\frac{Ts}{Tp}}{P}$ = $\frac{Ts}{P*Tp}$ = $\frac{n}{P\,(\frac{n}{P}+1)}$ = $\frac{n}{n+P}$

ref)

Ideal Tp = $\frac{Ts}{P}$

Ideal S = $\frac{Ts}{Tp}$ = P

Ideal E = $\frac{S}{P}$ = 1

if we increase P → k*P,

E' = $\frac{n}{n+K*P}$ ≠ E --- program is not strongly scalable

if we increase P → K*P and n → k*n,

E' = $\frac{k*n}{k*n+k*P}$ = $\frac{n}{n+P}$ = E --- program is weakly scalable

Parallel program design (MK book, §2.7)

Design steps:

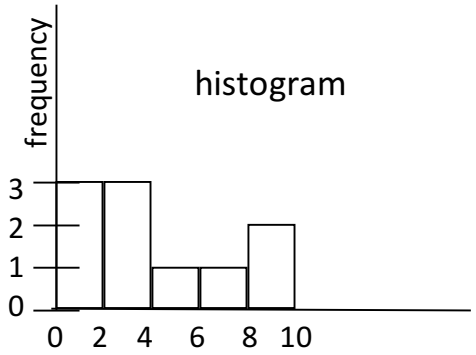
1. partitioning – divide computation into small tasks;
2. communication – among the small tasks;
3. aggregation – combine small tasks into a bigger task;

   → reduces communication cost
4. mapping – assign tasks to processes/threads with minimum communication and load-balancing way.

ex) computation for making a histogram

ex) 1, 3, 4, 2, 3, 5, 7, 10, 2, 9

bin#	range	count
[0]	1~2	3
[1]	3~4	3
[2]	5~6	1
[3]	7~8	1
[4]	9~10	2

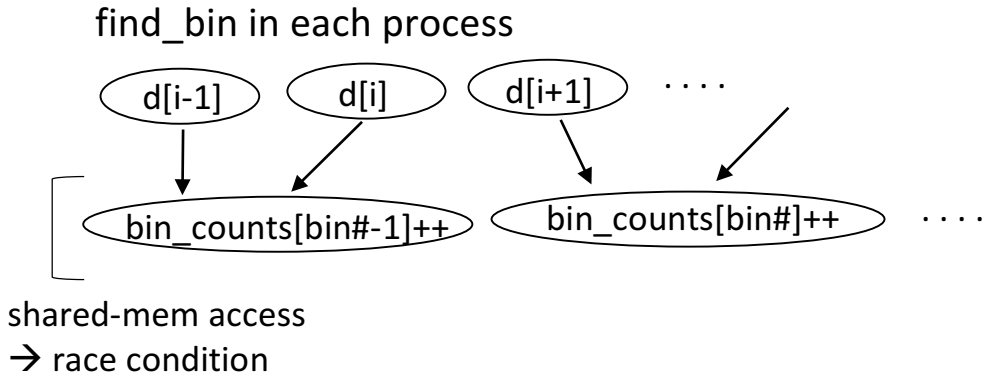


serial algorithm:

```
for each input data element i,
{  determine the bin (it is belong to);  // find_bin( i ) → bin# (index)
   increment the count of that bin;      // bin_counts[bin#]++;
}
```

parallel schemes

1. Shared bin-count way:



Processes/threads update shared-memory location (bin_counts[ ])

→ race condition: the shared location is C.S. and should be protected by mutex.

2. Local-sum way: -- data parallel on distributed-mem system

Input data is divided by the number of processes/threads;

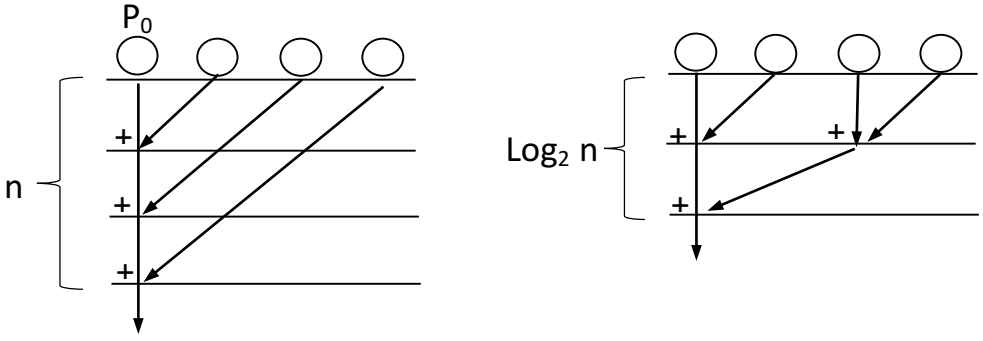
each process implements the serial algorithm on the part of the data;

then, local bin_counts are combined into the global one.

(1) by a process/thread

(2) by tree-reduction way

both needs communication (send/receive data)



Writing and running parallel programs

- Small shared-memory system:

  1 op sys schedules threads on the cores/processors;

- large shared-memory system:

  batch scheduler – user requests # of cores, etc.

- distributed-memory system (e.g., clustered system):

  host computer allocates nodes among users;

  or, batch (job queue), check-out way, etc.

Assumptions used in this course:

Homogeneous MIMD system

SPMD programming

At most one process/thread on a processor/core