Parallel Computing: How to Write Parallel Programs

Pacheco textbook Chapter 1

Tao Yang, UCSB CS140, 2014

Outline

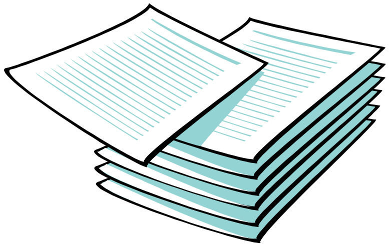
- **How do we write parallel programs?**
  - Rewrite serial programs so that they're parallel.
- **Task and data partitioning/mapping**
  - Examples
- **What we'll be doing.**

How do we write parallel programs?

- **Manage task and data parallelism**
- **Task parallelism**
  - Partition computations as tasks carried out solving the problem among the cores.

- **Data parallelism**
  - Partition the data used in solving the problem among the cores.
  - Each core carries out similar operations on it’s part of the data.

Application Example: Grading



Grading an exam with 15 questions

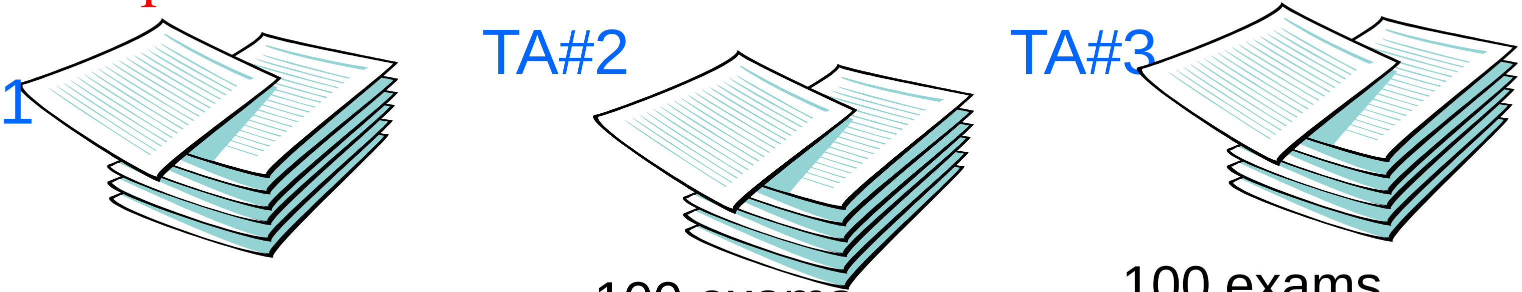
300 exams

Resource: 3 TAs

How to process grading in parallel?

Two options in division of work – data parallelism

Option 1: Data parallelism



TA#1

100 exams

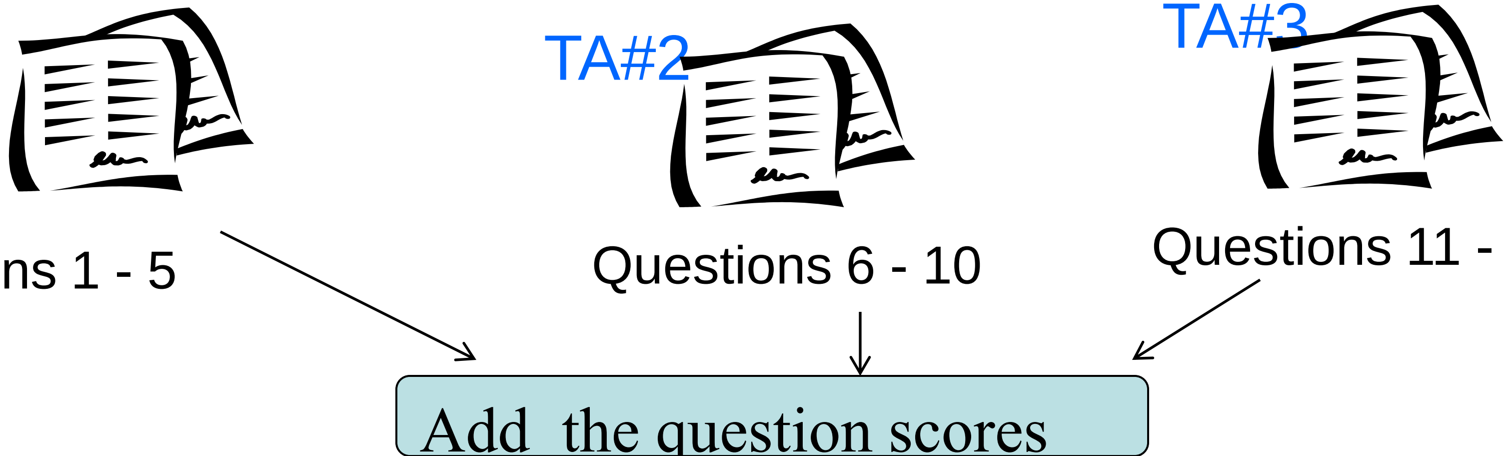
TA#2

100 exams

TA#3

100 exams

Option 2: Task parallelism



TA#1

Questions 1 - 5

TA#2

Questions 6 - 10

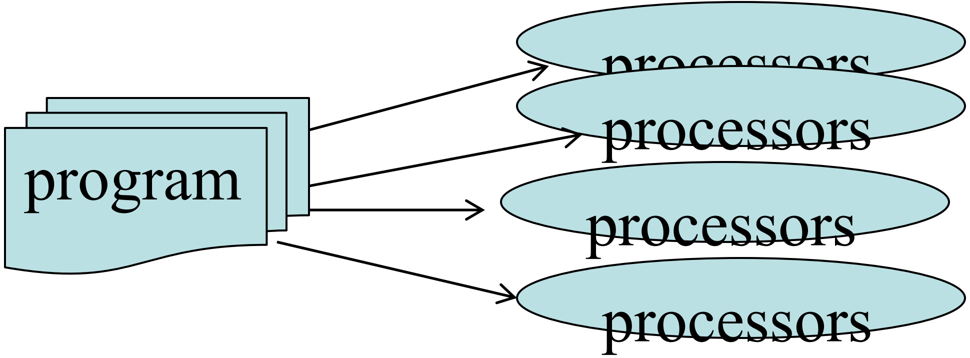
TA#3

Questions 11 - 15

Add the question scores

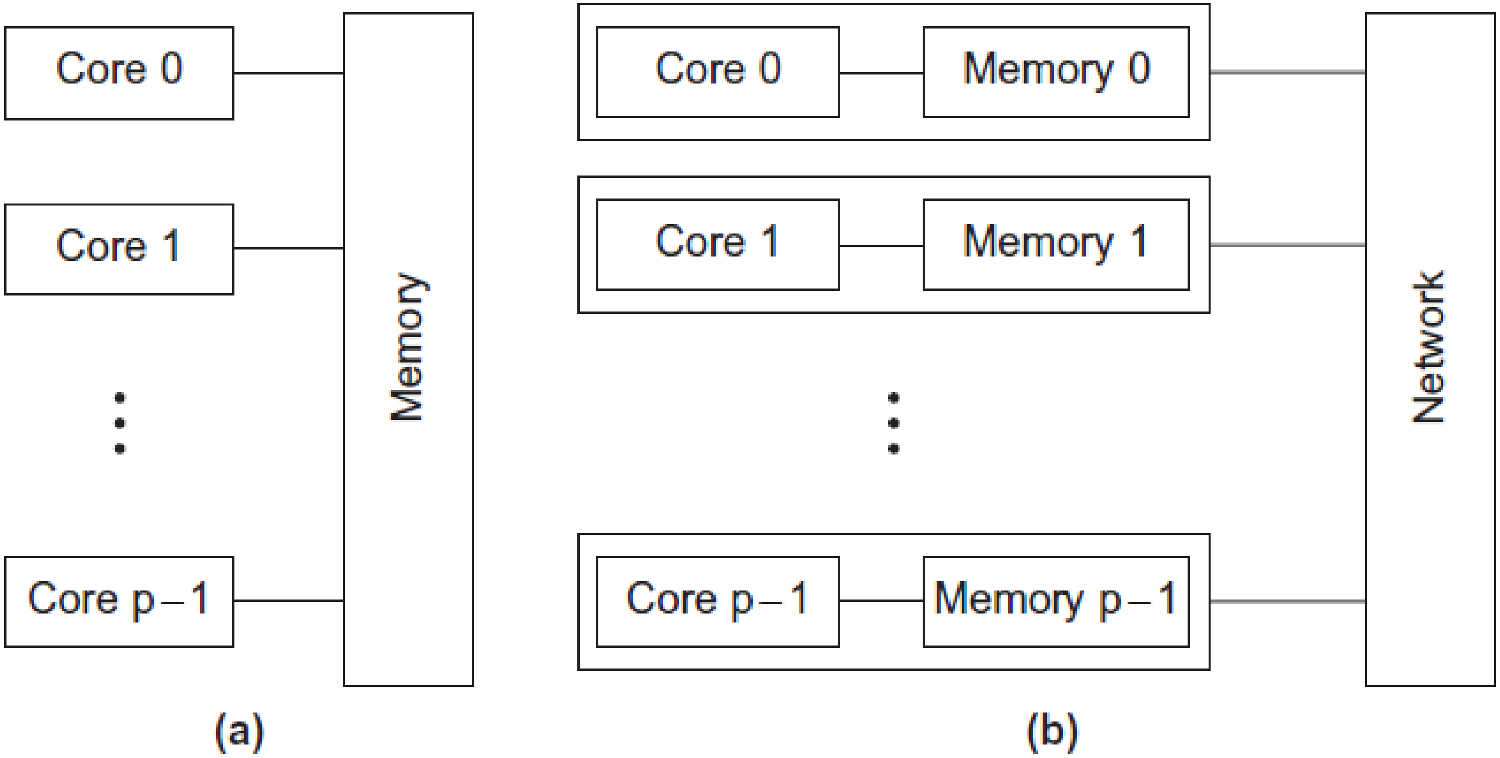
Division of Work: Partitioning/mapping for task and data

- **Task partitioning/mapping**
    - Divide code into a set of tasks
    - Map tasks to parallel processing units (processor cores, machines)



- **Data partitioning/mapping**
    - Divide data into a set of items for tasks to process
    - For a distributed architecture, map data items to physical machines

Type of parallel systems



Shared-memory

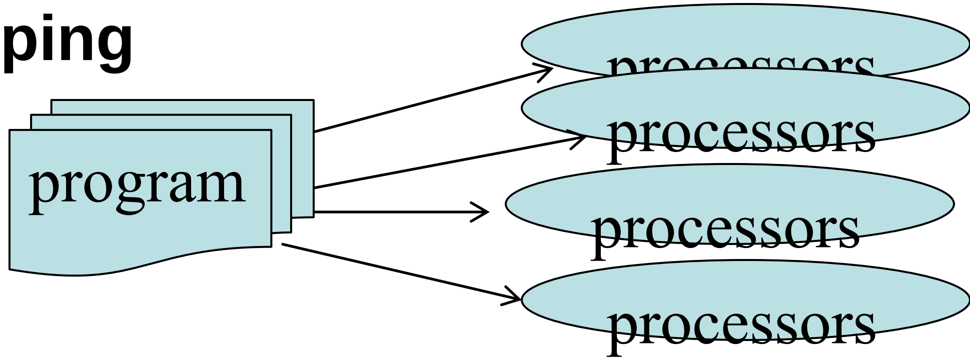
Distributed-memory

Type of parallel systems

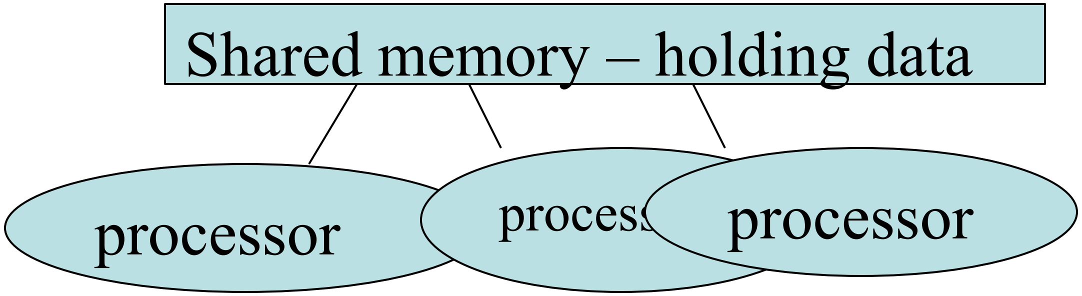
- **Shared-memory**
  - The cores can share access to the computer's memory.
  - Coordinate the cores by having them examine and update shared memory locations.
- **Distributed-memory**
  - Each machine has its own, private memory.
  - Machines must communicate explicitly by sending messages across a network.

Shared memory programming is easier

- **Task partitioning and mapping**
  - Required



- **Explicit data partitioning/mapping is not required because of shared memory**
  - Partitioning may be necessary for performance optimization



Parallel programming style

- **SPMD – single program multiple data**
  - Write one program, works for different data streams
  - Computation is distributed among processors, code is executed based on a predetermined schedule.
  - Each processor executes the same program but operates on different data based on processor identification.
- **Master/slaves: One control process is called the master (or host).**
  - There are a number of slaves working for this master.
  - These slaves can be coded using an SPMD style.

Generic Parallel Code Structure of SPMD

- **Processors/processes are numbered as 0, 1, 2, …**
  - Each processor executes the same program with a unique processor ID.
    - Differentiate the role of programs by their IDs
  - Assume two library functions
    - mynode() – returns processor ID of the program executed on one processor.
    - noproc() - returns # of processors used
- **Sequential code example:**
  - For i = 0 to n-1

    code for iteration I

Generic code structure of a process/processor

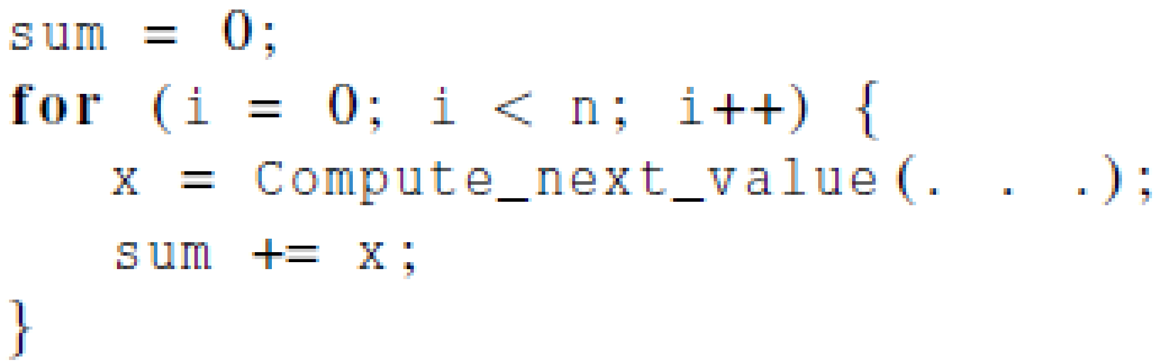
- **my_rank= mynode().   p=noproc();**
  - Detect who I am.

- **Scope the range of computation performed in this processor based on my_rank and p values.**
  - For example, given n iterations in a sequential code
    - My_first_i =   first iteration to handle
    - My_last_i =   last iteration to handle

- **Perform computation tasks under the derived scope.**

Example: Sequential program

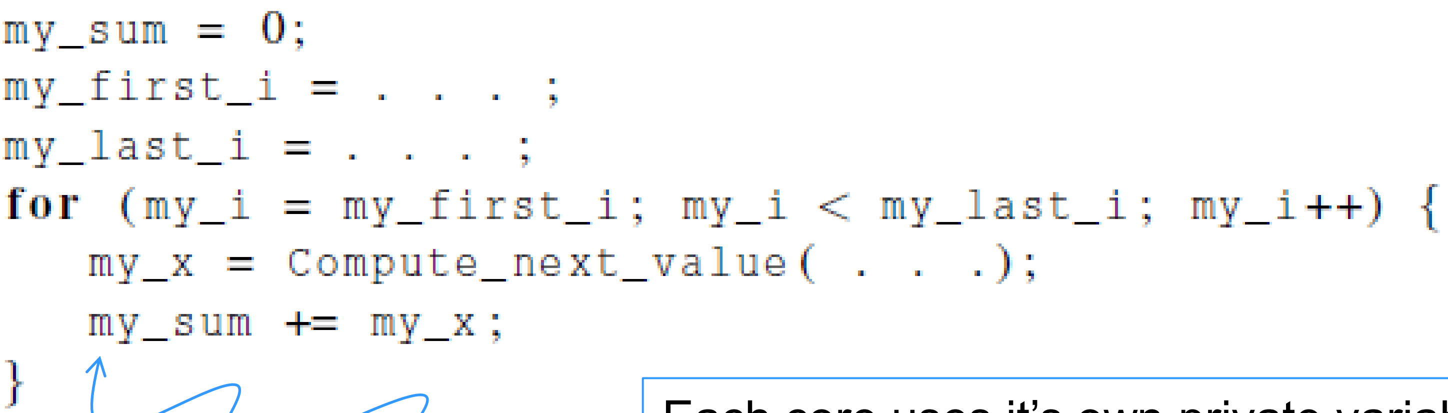
- Compute n values and add them together.
- Serial solution:



```
sum = 0;
for (i = 0; i < n; i++) {
   x = Compute_next_value(. . .);
   sum += x;
}
```

Example of Parallel code

- **We have p cores, p much smaller than n.**
- **Code for each core**
  - performs a partial sum of approximately n/p values.



```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value( . . .);
   my_sum += my_x;
}
```

Each core uses it’s own private variables and executes this block of code independently of the other cores.

Example with a sample data input

- **Private variable my_sum contains the sum of the values computed by its calls to Compute_next_value.**
  - Ex., 8 cores, n = 24, then the calls to Compute_next_value return for 8 parallel tasks:

1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

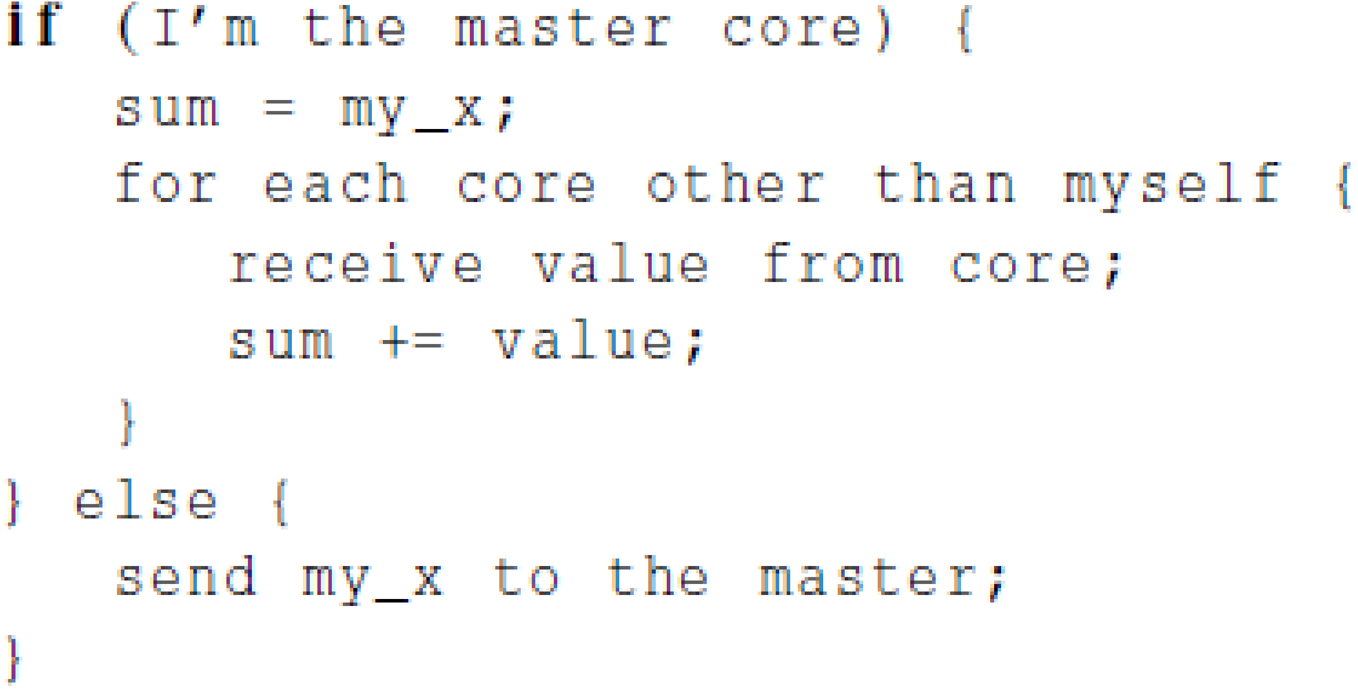
- **Once all the cores are done computing their private my_sum, they form a global sum by sending results to a designated "master" core which adds the final result.**

Coordination of task parallelism from master

<u>Code semantic.</u>

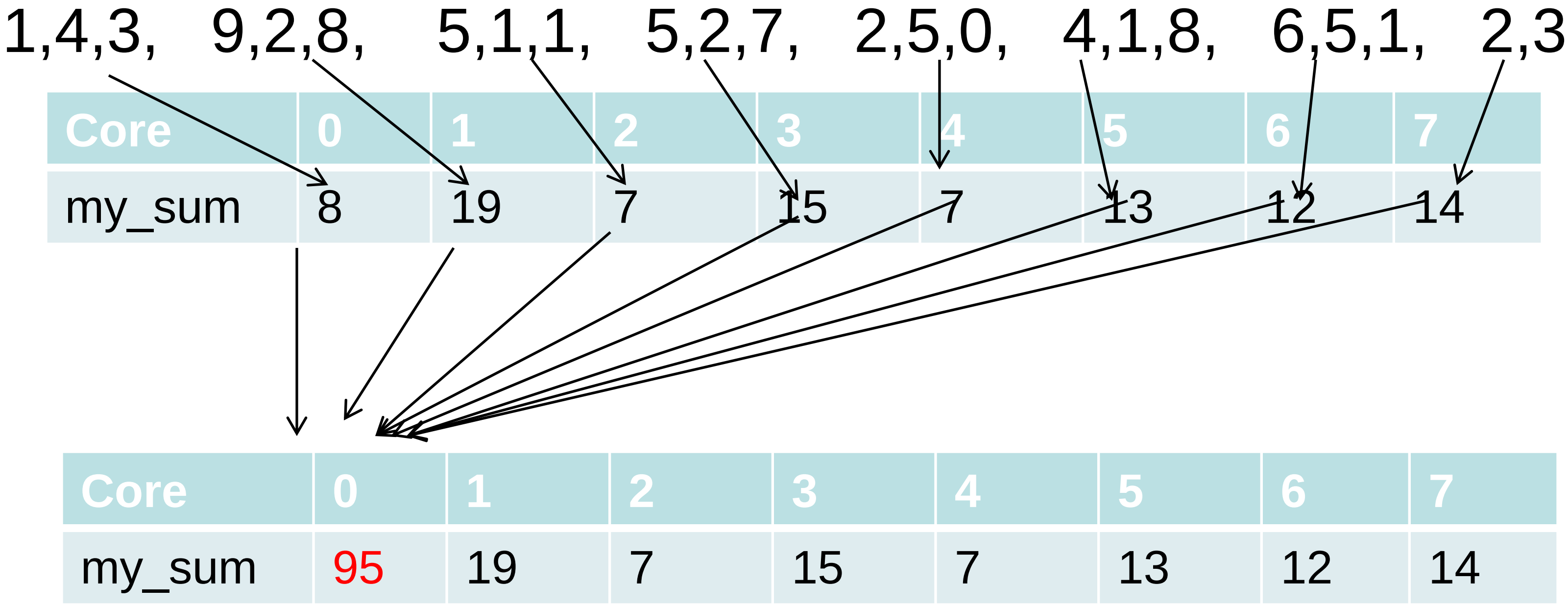
If my_rank==0

1) Receive
2) Accumulate



```
if (I'm the master core) {
    sum = my_x;
    for each core other than myself {
        receive value from core;
        sum += value;
    }
} else {
    send my_x to the master;
}
```

Example flow with sample data input



1,4,3, 9,2,8, 5,1,1, 5,2,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9

Core	0	1	2	3	4	5	6	7
my_sum	8	19	7	15	7	13	12	14

Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

Global sum

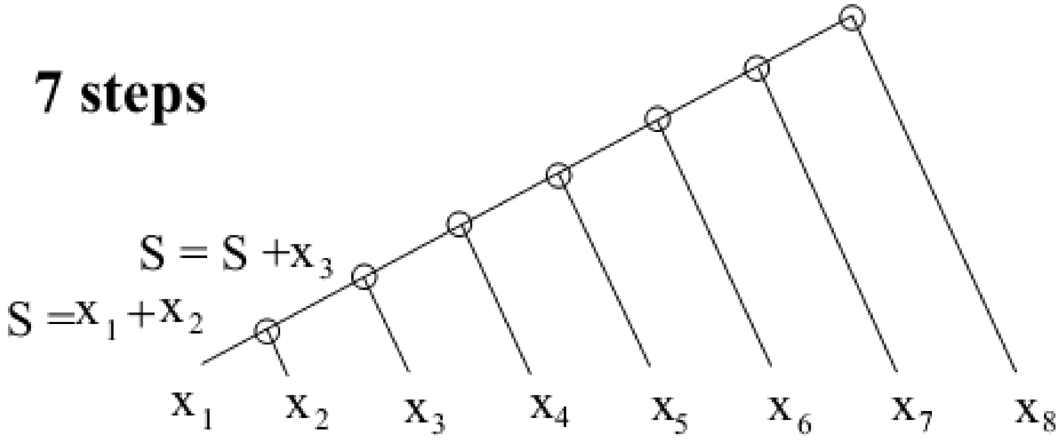
8 + 19 + 7 + 15 + 7 + 13 + 12 + 14 = 95

Weakness

Core 0 does all of the work to accumulate sequentially.

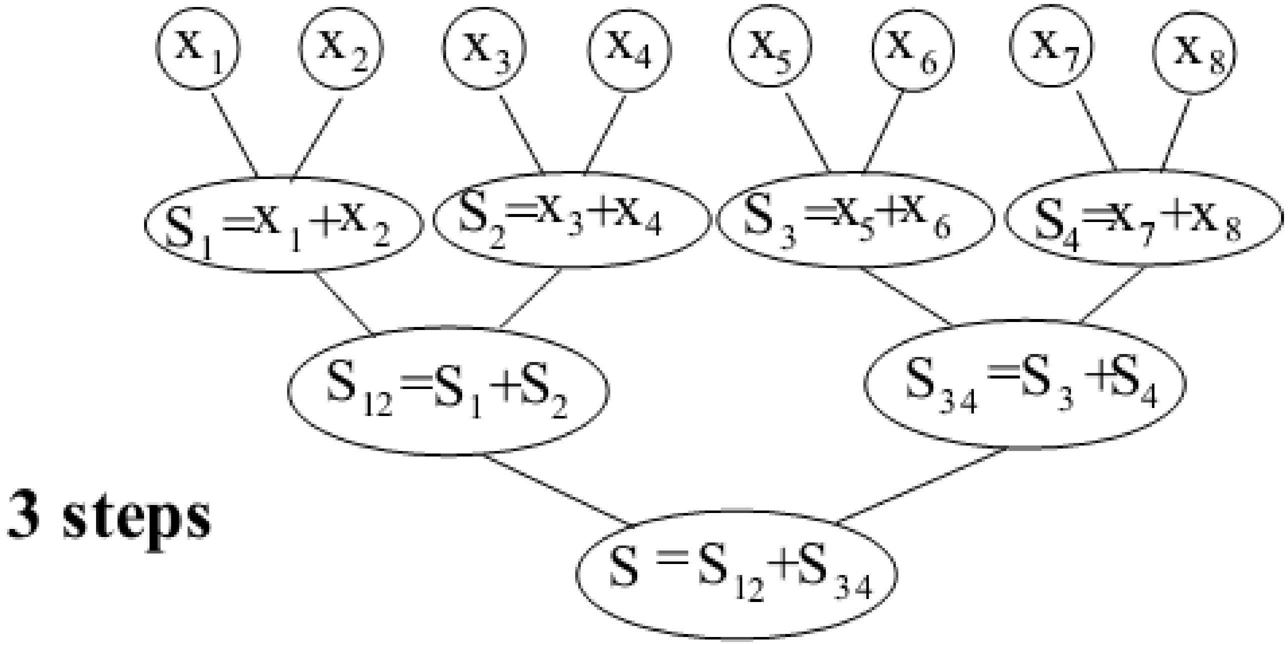
Core	0	1	2	3	4	5	6	7
my_sum	95	19	7	15	7	13	12	14

$$S = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8$$



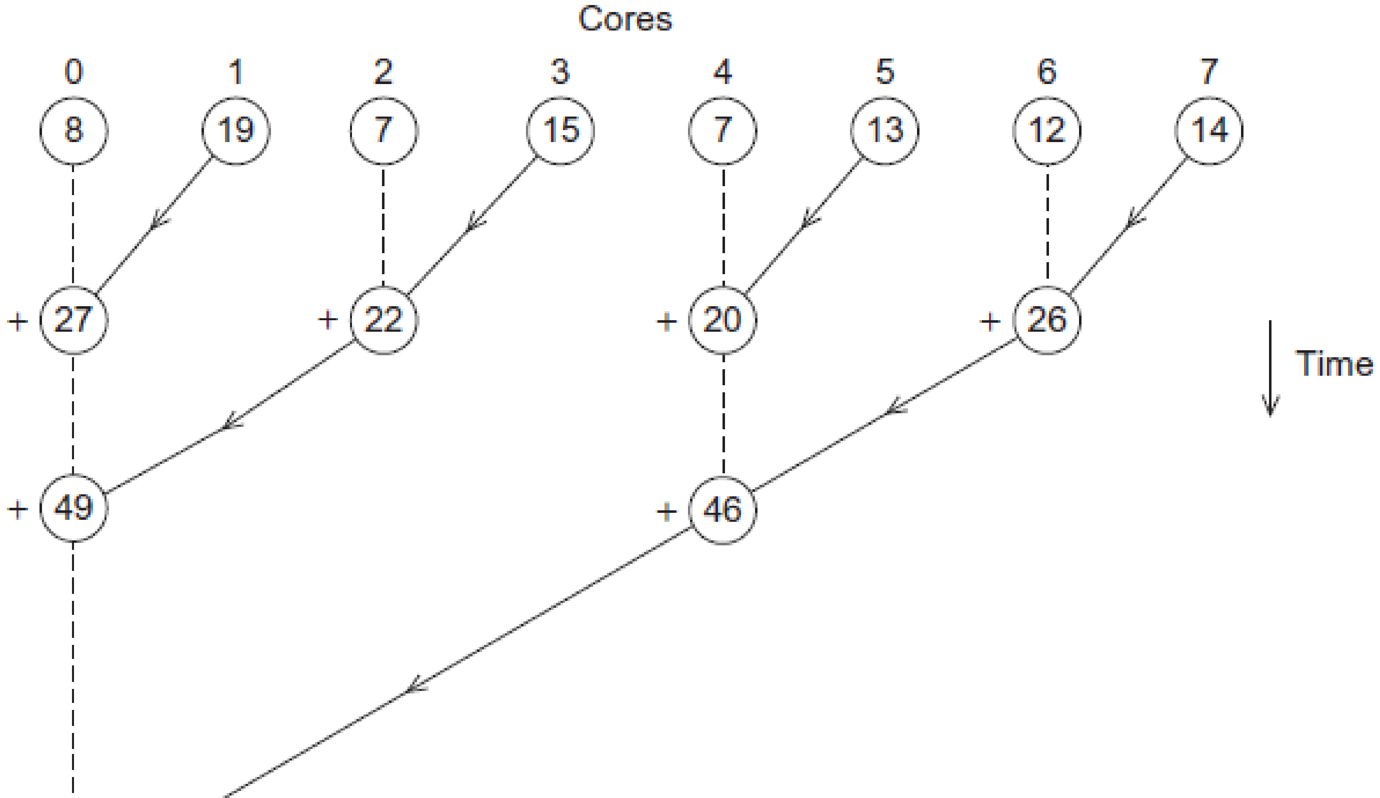
Tree summation for parallel addition

Tree-based accumulation



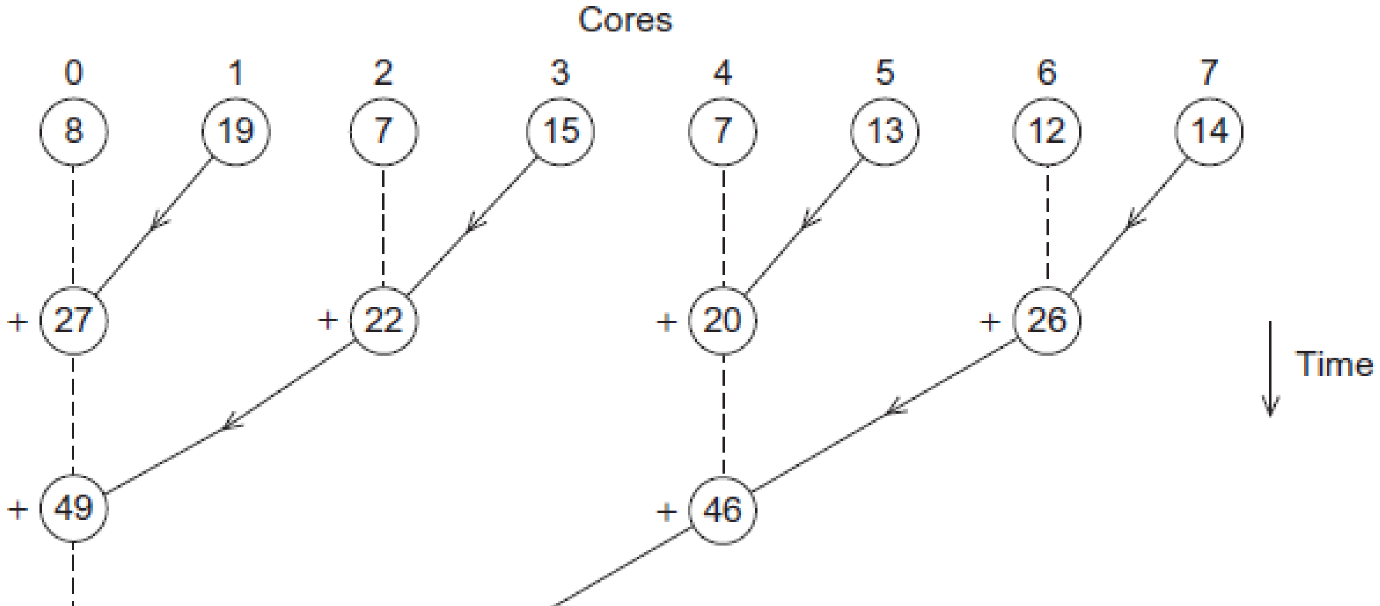
More implementation details for tree-based parallel accumulation

- **Who is responsible for parallel partial accumulation?**
  - Work with odd and even numbered pairs of cores.
    - core 0 adds its result with core 1's result.
    - Core 2 adds its result with core 3's result. etc.

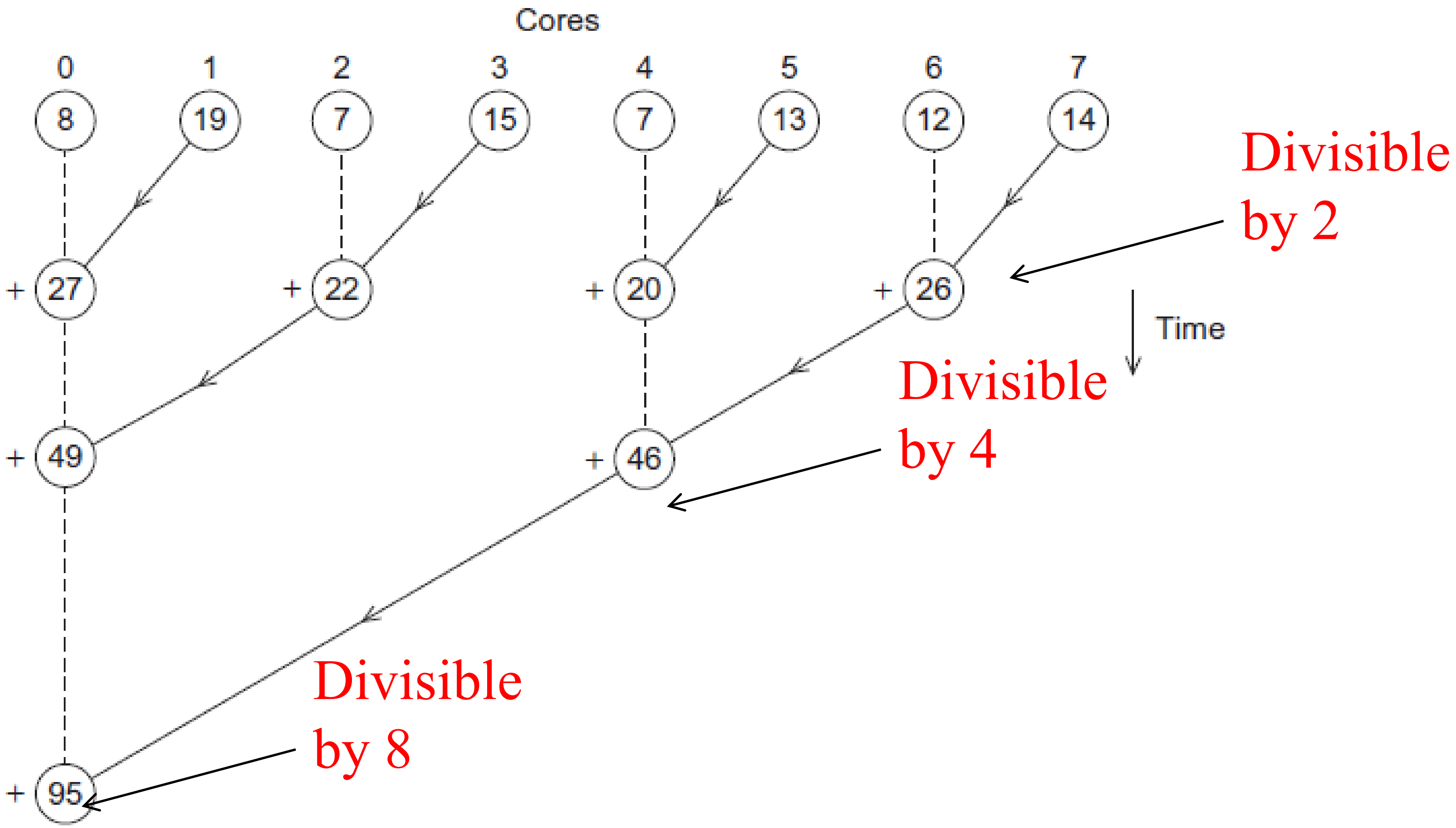


Parallel Accumulation (cont.)

- **Repeat the process now with only the evenly ranked cores.**
  - Core 0 adds result from core 2.
  - Core 4 adds result from core 6, etc.
- **Now cores divisible by 4 repeat the process, and so forth, until core 0 has the final result.**



Multiple cores forming a global sum



Divisible by 2

Divisible by 4

Divisible by 8

Analysis

- **In the first example, the master core performs 7 receives and 7 additions.**

- **In the second example, the master core performs 3 receives and 3 additions.**

- **The improvement is more than a factor of 2!**

Analysis (cont.)

- **The difference is more dramatic with a larger number of cores.**
- **If we have 1000 cores:**
  - The first example would require the master to perform 999 receives and 999 additions.
  - The second example would only require 10 receives and 10 additions.

- **That's an improvement of almost a factor of 100!**

Coordination and Overhead

- **Coordination is needed among parallel tasks**
  - Communication – one or more cores send their current partial sums to another core.
    - How to communicate?
  - Load balancing – share the work evenly among the cores so that one is not heavily loaded.
  - Synchronization – because each core works at its own pace, make sure cores do not get too far ahead of the rest.
- **Pay attentions to overhead of coordination**
  - Is it worthy to add 10 numbers in 5 machines in parallel?
  - Aggregation of small tasks is useful

What we'll be doing

- **Learning to write programs that are explicitly parallel.**
- **Using three different extensions to C/C++.**
  - Message-Passing Interface (MPI)
  - Posix Threads (Pthreads)
  - OpenMP if time permits
- **I/O-intensive parallel data processing**
  - Mapreduce/Hadoop with Java

Terminology

- **Concurrent computing – a program is one in which multiple tasks can be <u>in progress</u> at any instant.**
- **Parallel computing – a program is one in which multiple tasks <u>cooperate closely</u> to solve a problem**
- **Distributed computing – a program may need to cooperate with other programs to solve a problem.**

Concluding Remarks

- **Task/data partitioning/mapping is essential for writing parallel programs.**
- **Parallelism management involves coordination of cores/machines.**
- **Parallel programs are usually very complex and therefore, require sound program techniques and development.**
  - Automatic parallelization is difficult.