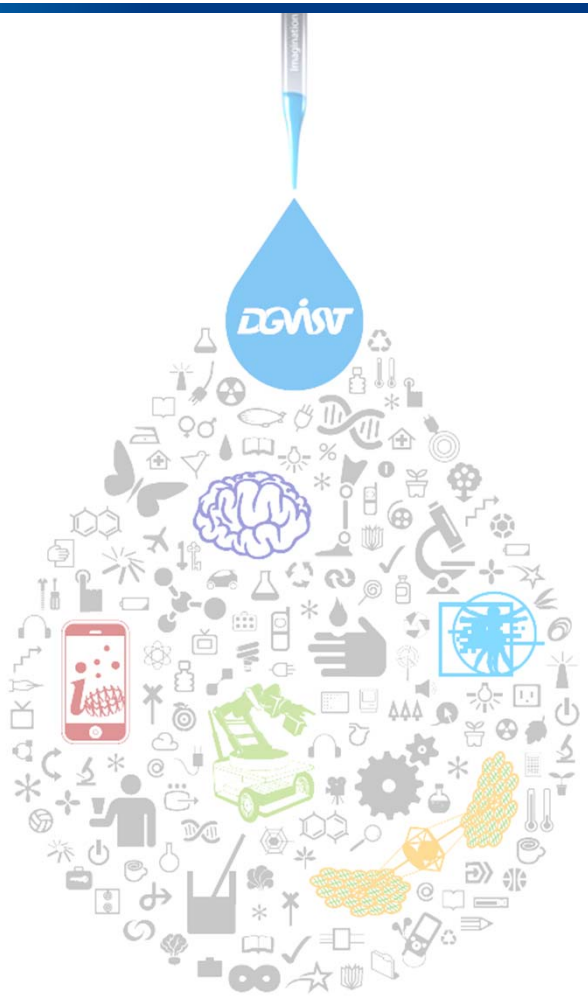
InfoLab

IC621: Distributed and Parallel Computing

Lecture 01 : Basic I



Min-Soo Kim

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Why Parallel Computing?

- **Performance of microprocessors had increased, on average, 50% per year, from 1986 to 2002**
  - simply wait for the next generation of microprocessors in order to obtain increased performance from an application program

- **However, single-processor performance improvement has slowed to about 20% per year since 2002**

- **So, manufacturers start to put multiple complete processors on a single integrated circuit**
  - simply adding more processors will not magically improve the performance of the vast majority of serial programs

Question 1

- **Q: Aren't single processor systems fast enough? 20% per year is still a pretty significant improvement**

- **A: We need ever-increasing performance**
  - *Climate modeling*
    - to better understand climate change
  - *Protein folding*
    - to study configurations of complex molecules such as proteins
    - e.g., misfolded proteins may be involved in diseases (e.g., Parkinson's)
  - *Drug discovery*
    - to devise alternative treatments by careful analysis of the genomes of the individuals for whom the known treatment is ineffective
  - *Energy research*
    - to program much more detailed models of technologies such as wind turbines, solar cells, and batteries
  - *Data analysis*
    - *to analyze* tremendous amounts of data
    - e.g., genomics, particle colliders, medical imaging, astronomical research

Question 2

- **Q: Why can't manufacturers continue to develop faster single processor systems? Why build parallel systems?**

- **A: Power consumption & heat problems**
  - As the speed of transistors increases, their power consumption also increases
  - Air-cooled integrated circuits are reaching the limits of their ability to dissipate heat
  - Therefore, it is becoming impossible to continue to increase the speed of integrated circuits

  - However, the increase in transistor density *can* continue through parallelism (i.e., multicore processors)

Question 3

- **Q: Why can't we write programs that will automatically convert serial programs into parallel programs?**

- **A: Inherently difficult problem**
  - researchers have had very limited success writing programs that convert serial programs in languages into parallel programs
  - example 1: computing *n* values and adding them together

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

  - we cannot simply continue to write serial programs, and we must write parallel programs that exploit the power of multiple processors

Approach 1 (for Example 1)

- **Each core calculates a local sum**

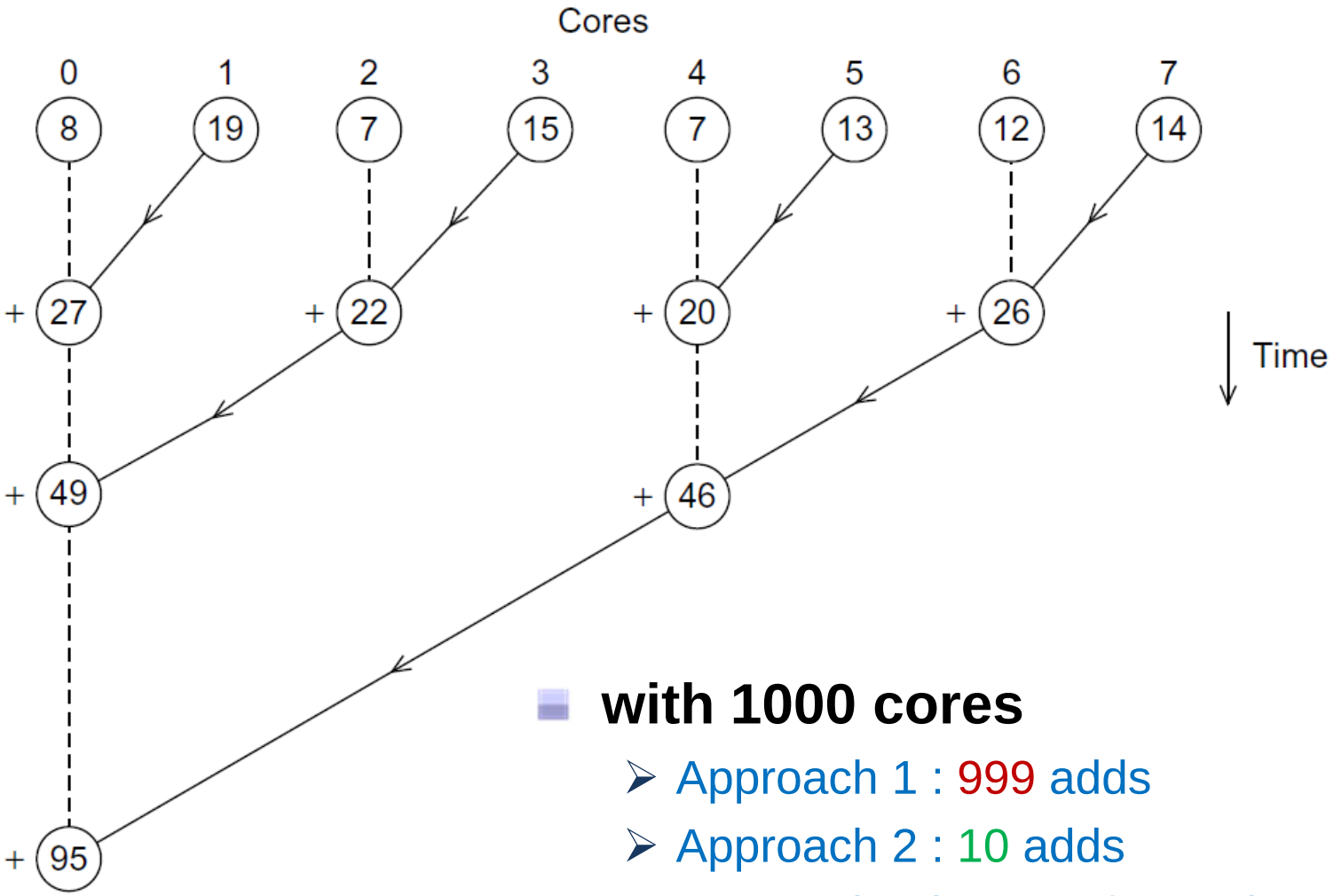
```
my_sum = 0;
my_first_i = . . . ;
my_last_i = . . . ;
for (my_i = my_first_i; my_i < my_last_i; my_i++) {
   my_x = Compute_next_value(. . .);
   my_sum += my_x;
}
```

- **Master core computes a global sum**
- **Example**

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

Approach 2 (for Example 1)



- **with 1000 cores**
  - Approach 1 : 999 adds
  - Approach 2 : 10 adds
  - Approach 2 is X100 faster than Approach 1

How Do We Write Parallel Programs?

- **Basic idea : *partitioning* the work into the cores**
  - data-parallelism : partition the data used among the cores
    - each core carries out more or less similar operations on its part of the data
    - e.g., the first part of Example 1
  - task-parallelism : partition the various tasks carried out among the cores
    - e.g., the second part of Example 1

- **Load balancing**
  - we want the cores all to be assigned roughly the same number of values to compute
  - if one core has to compute most of the values, then the other cores will finish much sooner than the heavily loaded core
    - so, their computational power will be wasted

Synchronization

- suppose that instead of computing the values to be added, the values are read from `stdin`

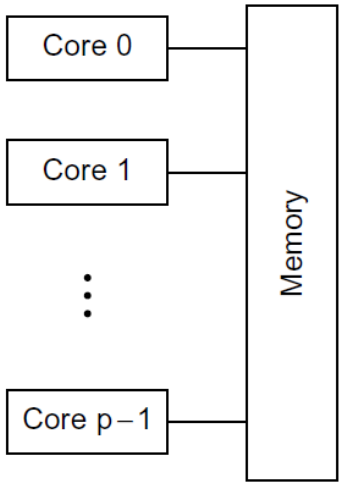
```
if (I'm the master core)
   for (my_i = 0; my_i < n; my_i++)
      scanf("%lf", &x[my_i]);
```

- problem : other cores might start computing their partial sums before the master is done initializing x
  - cores are not automatically synchronized in most systems
  - each core works at its own pace
- need to add in a point of synchronization between the initialization of x and the computation of the partial sums
  - each core will wait in the function Synchronize cores, in particular, until the master core has entered this function

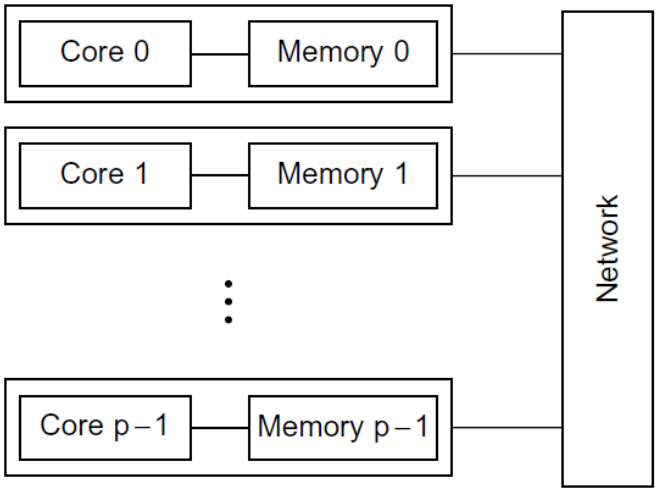
```
Synchronize_cores();
```

What We Will Be Doing?

- **Parallel programming on shared-memory systems**
  - POSIX threads (Pthreads)
  - OpenMP
  - CUDA



- **Parallel programming on distributed systems**
  - Message-Passing Interface (MPI)
  - MapReduce framework (Hadoop)



Concurrent, Parallel, Distributed

- **No complete agreement on the distinction among the terms**
  - many people consider shared-memory programs to be “parallel” and distributed-memory programs to be “distributed.”
- **Concurrent computing**
  - a program is one in which multiple tasks can be in progress at any instant
- **Parallel computing**
  - a program is one in which multiple tasks cooperate closely to solve a problem (*tightly coupled*)
- **Distributed computing**
  - a program may need to cooperate with other programs to solve a problem (*loosely coupled*)

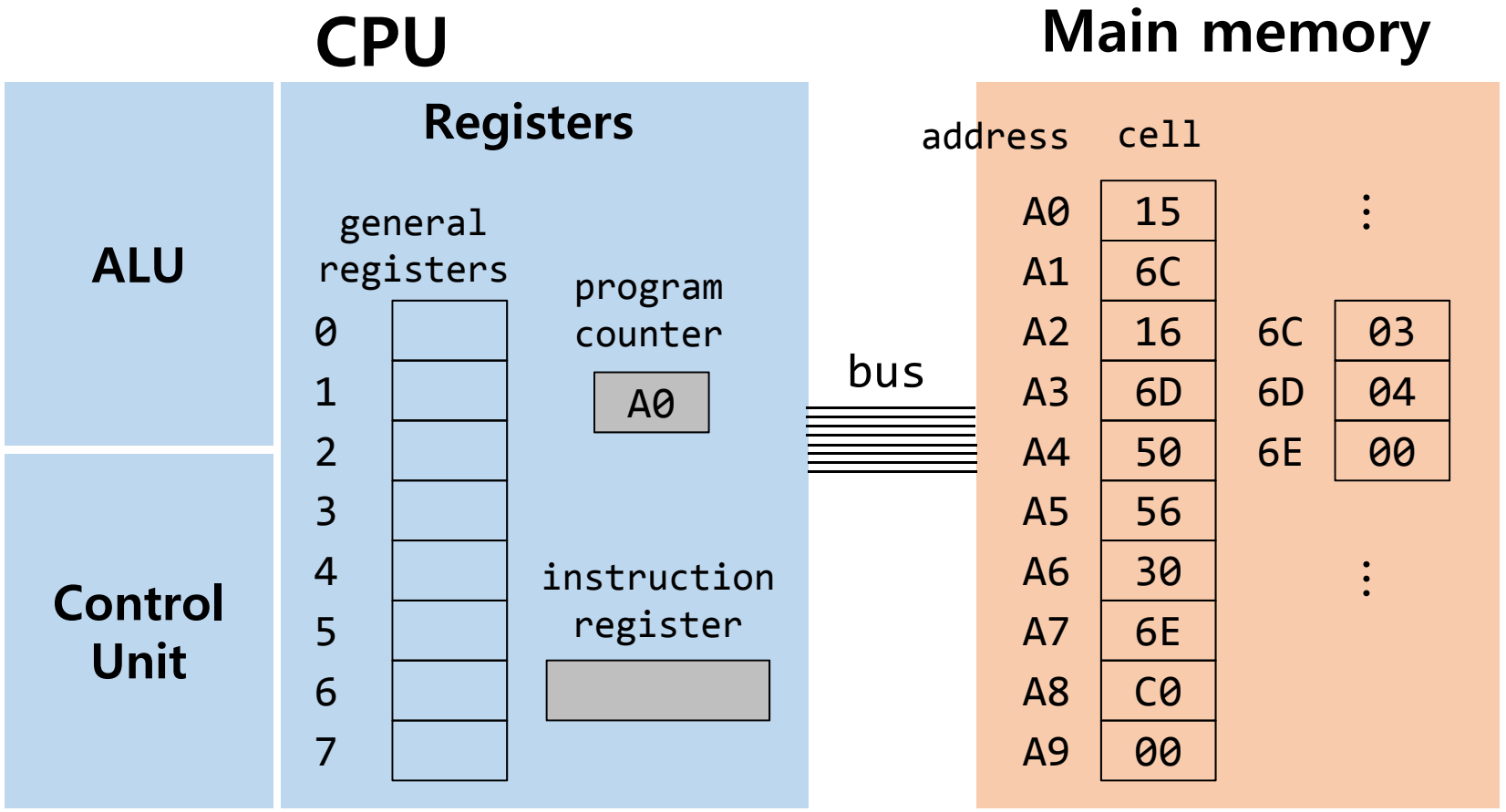
Parallel Hardware and Software

- **Background**
  - von Neumann architecture
  - processes, multitasking, and threads
- **Modification to von Neumann model**
  - caching
  - virtual memory
  - instruction-level parallelism
- **Parallel hardware**
  - SIMD
  - MIMD
  - interconnection networks
- **Parallel software**

Von Neumann architecture

- **Classical von Neumann architecture consists of**
  - main memory
  - Central Processing Unit (CPU) (or processor, or core)
  - interconnection between memory and CPU
- **Main memory**
  - consists of collection of locations (a location has its own address)
  - store both instructions and data
- **CPU**
  - control unit
    - decide which instructions in a program should be executed
  - Arithmetic and Logic Unit (ALU)
    - execute the actual instructions
  - registers
    - program counter: special register for the address of the next instruction

- **von Neumann bottleneck: separation of memory and CPU**
  - interconnection determines the rate at which instructions and data can be accessed
  - CPUs can execute instructions more than 100X faster than they can fetch items from main memory



Processes, Multitasking, and Threads

- **Process: an instance of a computer program that is being executed**
  - Executable machine language program
  - Block of memory
    - executable code, call stack, heap, …
  - Descriptors of resources that OS has allocated to the process
    - e.g., file descriptors
  - Security information
    - information specifying which HW and SW resources the process can access
  - State of the process
    - whether the process is ready to run or is waiting on some resource
    - content of the registers
    - information about the process' memory

Multitasking

- OS provides support for the apparent simultaneous execution of multiple programs
- each process runs for a small interval of time, called a time slice (a few milliseconds)
  - so, it is possible even on a system with a single core
- if a process needs to wait for a resource, it will block
  - stop executing the process such that OS can run another process
  - e.g., needs to read data from external storage

Threading

- mechanism to divide a program into more or less independent tasks
  - property that when one thread is blocked, another thread can be run
- switching between threads is much faster than that between processes
- threads can use the same executable, memory, and I/O devices
  - c.f., program counters, call stacks

InfoLab

Thank you!

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology