Parallel Computing

AN-NAJAH UNIVERSITY
ADNAN SALMAN

Why parallel computing

- Until 2002 the performance of microprocessor increased by 50%
- After 2002 the performance increased by only 20%
- 2005: major manufacturers
  - Increasing performance lay in parallelism.
  - Multiple complete processors on a single integrated circuit
- Software has to take advantage of multiprocessors
  - No magic : a sequential code running on a multicore machine will perform as running on a single core machine

Why parallel computing ?

- Why do we care?
- Why can’t continue to develop faster single processor systems?
- Why can’t automatically convert serial programs into **parallel programs**?

Increasing performance

- Key of advancing many field
  - Science : decoding genome project
  - Internet : fast searches
  - Entertainment: computer games
- Climate modeling : the atmosphere, the oceans, solid land, and the ice caps at the poles
- Protein folding : misfolded proteins Parkinson's, and Alzheimer's
- Drug discovery
- Energy research
- Data analysis : particle and astrophysics, dna sequencing, etc

Why parallel systems

- Single processor performance increases with transistors density
  - size decreases → speed increases → power consumption increases → heat increases → unreliable integrated circuit
- Air-cooled IC reached the limits of dissipating heat
  - Impossible to continue to increase the speed of IC.
  - Can continue to increase transistor density
- Improve our existence, there is an almost moral imperative to continue to increase computational power.
- Integrated circuit industry continue to exist.

Why write a parallel program?

- Programs that have been written for single-core systems cannot exploit the presence of multiple cores.
- We can run multiple instances of a program on a multicore system
  - Not always useful (game or simulation)
- Parallelize the serial code
- Write a translational program to parallelize the serial code
  - limited success -- matrix multiplication
  - An efficient parallel implementation may need a new algorithm.

Why write a parallel program?

- Example: compute n values and add them together
  - Serial code:

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

  - Parallel code: p cores ($p << n$), each core can form a partial sum of $n = p$ 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;
}
```

Why write a parallel program?

- Assume n = 24 and p = 8

```
 x = 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

Why write a parallel program?

- Cores computing their values of my_sum
- When done, form a global sum by sending their results to a “master” core, that add their results
  - master = core 0: finds the sum by adding 8+19+7+15+7+ 13+12+14

```
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;
}
```

Why write a parallel program?

- A better way to do the global sum
  - Pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, and so on
  - If there is 1000 core === 999 additions compared to 10 additions
- Unlikely a translation program would “discover” the second global sum.

Writing parallel programs

- Partitioning the work to be done among the cores
- Two approaches
  - **task-parallelism**

    partition the tasks carried out in solving the problem among the cores.
  - **data-parallelism**

    partition the data used among the cores
- **Example**
  - **Is the previous example data or task parallelism**

Writing parallel programs

- When the cores can work independently, writing a parallel program is not as writing serial code
- When the cores need to coordinate their work, it gets more complex.
- Load balancing among the cores, all cores should have roughly same amount of work, so they finish together.
- Synchronization, force cores to wait each other at some point.

Writing parallel programs

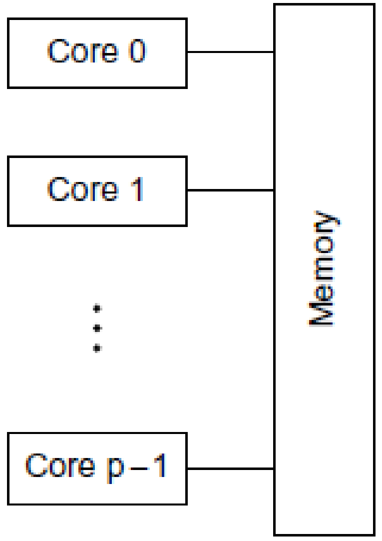
- Parallel programs are usually written using extensions to languages such as C and C++, **or Fortran**
- Explicit instructions for parallelism:
  - core 0 executes task 0,
  - core 1 executes task 1, . . . , all cores synchronize, . . . , and so on
- Higher level languages—but they tend to sacrifice performance in order to make program development somewhat easier.

Writing parallel programs

- MPI – message passing interface, a library extension to C/C++ and Fortran, distributed memory
- OpenMp: compiler directives, shared memory
- Pthreads : library extension , shared memory

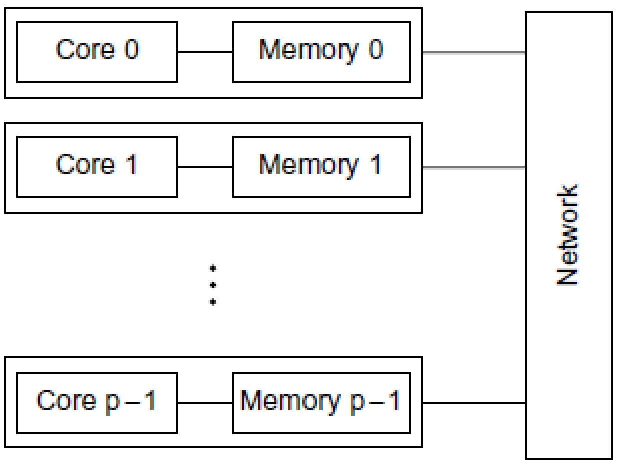
Writing parallel programs

- Shared memory
  - The cores share access to the computer's memory
  - Each core can read and write each memory location.
  - Coordinate the cores through shared-memory locations
  - Pthreads and OpenMP designed for programming shared memory. They provide mechanism for accessing shared memory.



Writing parallel programs

- Distributed memory
  - Each core has its own, private memory
  - Cores must communicate explicitly by sending and receiving messages across a network.
  - MPI designed for programming distributed-memory systems. It provides mechanisms for sending messages.



Concurrent, parallel, distributed

- Concurrent computing: multiple tasks can be *in progress* at any instant.
- Parallel computing: multiple tasks *cooperate closely* to solve a problem.
- Distributed computing: a program may need to cooperate with other programs to solve a problem (loosely coupled)
- Parallel and distributed are concurrent
- What is the difference

Message Passing Interface

- MIMD: Multiple Instruction Multiple Data
  - At any time, different processors may be executing different instructions on different pieces of data.
  - Can be either shared-memory or distributed-memory
- MPI – Message Passing Interface
- A process: A program running on one core-memory pair
- Two processes running on two cores can communicate through messages
  - One calls a send function and the other calls a receive
- Multiple processes can also communicate globally using collective functions